Part IV. Advanced Design and Analysis Techniques
Part IV. Advanced Design and Analysis Techniques

- Chapter 14.9 Exhaustive search
- Chapter 15. Dynamic programming
- Chapter 16. Greedy algorithms
Chapter 14.9. Exhaustive Search

Chapter 14.9. Exhaustive Search (not in the text!)
Chapter 14.9. Exhaustive Search (not in the text!)

To enumerate all possible solutions to the problem instance
Chapter 14.9. Exhaustive Search

Chapter 14.9. Exhaustive Search (not in the text!)

To enumerate all possible solutions to the problem instance

How?
Chapter 14.9. Exhaustive Search

To enumerate all possible solutions to the problem instance

How?

- systematic examining all solutions
Chapter 14.9. Exhaustive Search

To enumerate all possible solutions to the problem instance

How?

• systematic examining all solutions

• without repeating solutions that have been examined
Chapter 14.9. Exhaustive Search

To enumerate all possible solutions to the problem instance

How?

- systematic examining all solutions
- without repeating solutions that have been examined
- stop when a satisfactory solution is found
Chapter 14.9. Exhaustive Search

First, we need to be able to count total number of “things” to be enumerated.
Chapter 14.9. Exhaustive Search

First, we need to be able to count total number of “things” to be enumerated.

• without missing one (correctness)
Chapter 14.9. Exhaustive Search

First, we need to be able to count total number of “things” to be enumerated.

- without missing one (correctness)
- without over-counting (efficiency)
Chapter 14.9. Exhaustive Search

First, we need to be able to count total number of “things” to be enumerated.

- without missing one (correctness)
- without over-counting (efficiency)
- A sophisticated counting often has recursive solution.
Chapter 14.9. Exhaustive Search

Examples of counting:

1. The total number of permutations of \(n\) items is \(P(n) = n \times P(n-1)\) with base case \(P(1) = 1\).

2. The total number of ways to choose \(k\) from \(n\) items is \(\binom{n}{k} = \frac{n \times (n-1) \times \ldots \times (n-k+1)}{k!}\) or, alternatively, \(\binom{n}{k} = \frac{n \times (n-1) \times \ldots \times (n-k+1)}{k!} \times \frac{(n-k)!}{(n-k)!} = \frac{n!}{(n-k)!k!}\) with base cases: (?).
Chapter 14.9. Exhaustive Search

Examples of counting:

(1) total number of permutations of $(1, 2, \ldots, n)$ is

$$P(n) =$$
Chapter 14.9. Exhaustive Search

Examples of counting:

(1) total number of permutations of \((1, 2, \ldots, n)\) is

\[ P(n) = n \times P(n - 1) \]
Chapter 14.9. Exhaustive Search

Examples of counting:

(1) total number of permutations of \((1, 2, \ldots, n)\) is

\[
P(n) = n \times P(n-1)
\]

with base case \(P(1) = 1\).
Chapter 14.9. Exhaustive Search

Examples of counting:

(1) total number of permutations of \((1, 2, \ldots, n)\) is

\[
P(n) = n \times P(n - 1)
\]

with base case \(P(1) = 1\).

\[
P(n) =
\]
Examples of counting:

(1) total number of permutations of \(1, 2, \ldots, n\) is

\[ P(n) = n \times P(n - 1) \]

with base case \(P(1) = 1\).

\[ P(n) = n \times P(n - 1) \]
Chapter 14.9. Exhaustive Search

Examples of counting:

(1) total number of permutations of \((1, 2, \ldots, n)\) is

\[
P(n) = n \times P(n - 1)
\]

with base case \(P(1) = 1\).

\[
P(n) = n \times P(n - 1) = n
\]
Chapter 14.9. Exhaustive Search

Examples of counting:

(1) total number of permutations of $(1, 2, \ldots, n)$ is

$$P(n) = n \times P(n - 1)$$

with base case $P(1) = 1$.

$$P(n) = n \times P(n - 1) = n \times (n - 1) \times P(n - 2)$$
Chapter 14.9. Exhaustive Search

Examples of counting:

(1) total number of permutations of \((1, 2, \ldots, n)\) is

\[
P(n) = n \times P(n - 1)
\]

with base case \(P(1) = 1\).

\[
P(n) = n \times P(n - 1) = n \times (n - 1) \times P(n - 2) = n \times (n - 1) \times \ldots \times 2 \times 1 = n!
\]
Examples of counting:

1. total number of permutations of \((1, 2, \ldots, n)\) is

\[
P(n) = n \times P(n - 1)
\]

with base case \(P(1) = 1\).

\[
P(n) = n \times P(n - 1) = n \times (n - 1) \times P(n - 2) = n \times (n - 1) \times \ldots 2 \times 1 = n!
\]
Chapter 14.9. Exhaustive Search

Examples of counting:

(1) total number of permutations of \((1, 2, \ldots, n)\) is

\[ P(n) = n \times P(n - 1) \]

with base case \(P(1) = 1\).

\[ P(n) = n \times P(n - 1) = n \times (n - 1) \times P(n - 2) = n \times (n - 1) \times \ldots 2 \times 1 = n! \]

(2) total number of ways to choose \(k\) from \(n\) items is

\[ \binom{n}{k} \]
Examples of counting:

(1) total number of permutations of \((1, 2, \ldots, n)\) is

\[ P(n) = n \times P(n - 1) \]

with base case \(P(1) = 1\).

\[ P(n) = n \times P(n - 1) = n \times (n - 1) \times P(n - 2) = n \times (n - 1) \times \ldots \times 2 \times 1 = n! \]

(2) total number of ways to choose \(k\) from \(n\) items is

\[ \binom{n}{k} = n(n - 1) \ldots (n - k + 1) \]
Chapter 14.9. Exhaustive Search

Examples of counting:

(1) total number of permutations of \((1, 2, \ldots, n)\) is

\[
P(n) = n \times P(n - 1)
\]

with base case \(P(1) = 1\).

\[
P(n) = n \times P(n - 1) = n \times (n - 1) \times P(n - 2) = n \times (n - 1) \times \ldots 2 \times 1 = n!
\]

(2) total number of ways to choose \(k\) from \(n\) items is

\[
\binom{n}{k} = \frac{n(n - 1) \ldots (n - k + 1)}{k!} = 
\]
Chapter 14.9. Exhaustive Search

Examples of counting:

(1) total number of permutations of \((1, 2, \ldots, n)\) is

\[ P(n) = n \times P(n - 1) \]

with base case \(P(1) = 1\).

\[ P(n) = n \times P(n - 1) = n \times (n - 1) \times P(n - 2) = n \times (n - 1) \times \ldots \times 2 \times 1 = n! \]

(2) total number of ways to choose \(k\) from \(n\) items is

\[ \binom{n}{k} = \frac{n(n-1)\ldots(n-k+1)}{k!} = \frac{n!}{(n-k)! \times k!} \]
Chapter 14.9. Exhaustive Search

Examples of counting:

(1) total number of permutations of \((1, 2, \ldots, n)\) is

\[ P(n) = n \times P(n - 1) \]

with base case \(P(1) = 1\).

\[ P(n) = n \times P(n - 1) = n \times (n - 1) \times P(n - 2) = n \times (n - 1) \times \ldots 2 \times 1 = n! \]

(2) total number of ways to choose \(k\) from \(n\) items is

\[ \binom{n}{k} = \frac{n(n - 1) \ldots (n - k + 1)}{k!} = \frac{n!}{(n - k)! \times k!} \]

or, alternatively,

\[ \binom{n}{k} \]
Examples of counting:

(1) total number of permutations of \((1, 2, \ldots, n)\) is

\[
P(n) = n \times P(n - 1)
\]

with base case \(P(1) = 1\).

\[
P(n) = n \times P(n - 1) = n \times (n - 1) \times P(n - 2) = n \times (n - 1) \times \ldots 2 \times 1 = n!
\]

(2) total number of ways to choose \(k\) from \(n\) items is

\[
\binom{n}{k} = \frac{n(n - 1) \ldots (n - k + 1)}{k!} = \frac{n!}{(n - k)! \times k!}
\]

or, alternatively,

\[
\binom{n}{k} = \binom{n - 1}{k}
\]
Chapter 14.9. Exhaustive Search

Examples of counting:

1. total number of permutations of \((1, 2, \ldots, n)\) is

\[
P(n) = n \times P(n-1)
\]

with base case \(P(1) = 1\). 

\[
P(n) = n \times P(n-1) = n \times (n-1) \times P(n-2) = n \times (n-1) \times \ldots 2 \times 1 = n!
\]

2. total number of ways to choose \(k\) from \(n\) items is

\[
\binom{n}{k} = \frac{n(n-1) \ldots (n-k+1)}{k!} = \frac{n!}{(n-k)! \times k!}
\]

or, alternatively,

\[
\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}
\]
Chapter 14.9. Exhaustive Search

Examples of counting:

(1) total number of permutations of \((1, 2, \ldots, n)\) is

\[ P(n) = n \times P(n - 1) \]

with base case \(P(1) = 1\).

\[ P(n) = n \times P(n - 1) = n \times (n - 1) \times P(n - 2) = n \times (n - 1) \times \ldots 2 \times 1 = n! \]

(2) total number of ways to choose \(k\) from \(n\) items is

\[ \binom{n}{k} = \frac{n(n - 1) \ldots (n - k + 1)}{k!} = \frac{n!}{(n - k)! \times k!} \]

or, alternatively,

\[ \binom{n}{k} = \binom{n - 1}{k} + \binom{n - 1}{k - 1} \]
Chapter 14.9. Exhaustive Search

Examples of counting:

1. total number of permutations of \((1, 2, \ldots, n)\) is

\[
P(n) = n \times P(n-1)
\]

with base case \(P(1) = 1\).

\[
P(n) = n \times P(n-1) = n \times (n-1) \times P(n-2) = n \times (n-1) \times \ldots \times 2 \times 1 = n!
\]

2. total number of ways to choose \(k\) from \(n\) items is

\[
\binom{n}{k} = \frac{n(n-1) \ldots (n-k+1)}{k!} = \frac{n!}{(n-k)! \times k!}
\]

or, alternatively,

\[
\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}
\]

with base cases: (??)
Example: Boolean Formula Satisfiability problem (SAT)
Example: Boolean Formula Satisfiability problem (SAT)

**INPUT:** boolean formula \( f(x_1, x_2, \ldots, x_n) \),

**OUTPUT:** "yes" if and only if \( f(x_1, x_2, \ldots, x_n) \) is satisfiable.
Example: Boolean Formula Satisfiability problem (SAT)

**INPUT:** boolean formula $f(x_1, x_2, \ldots, x_n)$,

**OUTPUT:** "yes" if and only if $f(x_1, x_2, \ldots, x_n)$ is satisfiable.

$f(x_1, x_2, \ldots, x_n)$ is **satisfiable** if there is an **assignment** to boolean variables $x_i \in \{T, F\}$, $i = 1, 2, \ldots, n$. 

"yes" if and only if $f(x_1, x_2, \ldots, x_n)$ is satisfiable.
Example: Boolean Formula Satisfiability problem (SAT)

**INPUT:** boolean formula \( f(x_1, x_2, \ldots, x_n) \),
**OUTPUT:** "yes" if and only if \( f(x_1, x_2, \ldots, x_n) \) is satisfiable.

\( f(x_1, x_2, \ldots, x_n) \) is **satisfiable** if there is an **assignment** to boolean variables \( x_i \in \{T, F\}, i = 1, 2, \ldots, n \), such that \( f \) is evaluated to \( T \).
Chapter 14.9. Exhaustive Search

Example: Boolean Formula Satisfiability problem (SAT)

**Input:** boolean formula \( f(x_1, x_2, \ldots, x_n) \),

**Output:** "yes" if and only if \( f(x_1, x_2, \ldots, x_n) \) is satisfiable.

\( f(x_1, x_2, \ldots, x_n) \) is **satisfiable** if there is an **assignment** to boolean variables \( x_i \in \{T, F\}, \ i = 1, 2, \ldots, n \), such that \( f \) is evaluated to \( T \).

e.g.,
Example: Boolean Formula Satisfiability problem (SAT)

**INPUT:** boolean formula $f(x_1, x_2, \ldots, x_n)$,
**OUTPUT:** "yes" if and only if $f(x_1, x_2, \ldots, x_n)$ is satisfiable.

$f(x_1, x_2, \ldots, x_n)$ is **satisfiable** if there is an **assignment** to boolean variables $x_i \in \{T, F\}$, $i = 1, 2, \ldots, n$, such that $f$ is evaluated to $T$.

e.g,

$$f(x_1, x_2, x_3) = (x_1 \lor x_2 \lor \neg x_3) \land (\neg x_1 \lor x_3)$$ is satisfiable
Chapter 14.9. Exhaustive Search

Example: Boolean Formula Satisfiability problem (SAT)

INPUT: boolean formula \( f(x_1, x_2, \ldots, x_n) \),
OUTPUT: "yes" if and only if \( f(x_1, x_2, \ldots, x_n) \) is satisfiable.

\( f(x_1, x_2, \ldots, x_n) \) is satisfiable if there is an assignment to boolean variables 
\( x_i \in \{T, F\}, \; i = 1, 2, \ldots, n \), such that \( f \) is evaluated to T.

e.g.,
\[
\begin{align*}
f(x_1, x_2, x_3) &= (x_1 \lor x_2 \lor \neg x_3) \land (\neg x_1 \lor x_3) \text{ is satisfiable} \\
g(x_1, x_2) &= (x_1 \lor \neg x_2) \land (\neg x_1 \lor x_2) \land (x_1 \lor x_2) \land (\neg x_1 \lor \neg x_2) \text{ is not!}
\end{align*}
\]
Chapter 14.9. Exhaustive Search

Use exhaustive search to solve the SAT problem.
Use exhaustive search to solve the SAT problem.

**Input:** boolean formula $f(x_1, x_2, \ldots, x_n)$,

**Output:** "yes" if and only if $f(x_1, x_2, \ldots, x_n)$ is satisfiable.
Chapter 14.9. Exhaustive Search

Use exhaustive search to solve the SAT problem.

**INPUT:** boolean formula $f(x_1, x_2, \ldots, x_n)$,

**OUTPUT:** "yes" if and only if $f(x_1, x_2, \ldots, x_n)$ is satisfiable.

How?
Chapter 14.9. Exhaustive Search

Use exhaustive search to solve the SAT problem.

**INPUT:** boolean formula $f(x_1, x_2, \ldots, x_n)$,

**OUTPUT:** "yes" if and only if $f(x_1, x_2, \ldots, x_n)$ is satisfiable.

How? What will you exhaustively search on?
Use exhaustive search to solve the SAT problem.

**INPUT:** boolean formula \( f(x_1, x_2, \ldots, x_n) \),

**OUTPUT:** ”yes” if and only if \( f(x_1, x_2, \ldots, x_n) \) is satisfiable.

How? What will you exhaustively search on?

- **Enumerate all combinations of T and F for** \( x_1, \ldots, x_n \).
Chapter 14.9. Exhaustive Search

Use exhaustive search to solve the SAT problem.

**Input**: boolean formula $f(x_1, x_2, \ldots, x_n)$,

**Output**: "yes" if and only if $f(x_1, x_2, \ldots, x_n)$ is satisfiable.

How? What will you exhaustively search on?

- **Enumerate all combinations of T and F for $x_1, \ldots, x_n$.**
- Can you solve it with a recursive algorithm?
Chapter 14.9. Exhaustive Search

Use exhaustive search to solve the SAT problem.

**INPUT:** boolean formula $f(x_1, x_2, \ldots, x_n)$,

**OUTPUT:** "yes" if and only if $f(x_1, x_2, \ldots, x_n)$ is satisfiable.

How? What will you exhaustively search on?

- **Enumerate all combinations of T and F for $x_1, \ldots, x_n$.**
- Can you solve it with a recursive algorithm?
- Can you solve it with an iterative algorithm?
Solve SAT problem with a recursive algorithm:

- what data will the recursion be applied to? boolean formula $f(x_1, \ldots, x_n)$
- what is the terminating (base) case? $n=0$, formula without variables
- what is the recursive case?
  
  $f(x_1, \ldots, x_{n-1}, x_n) = f(x_1, \ldots, x_{n-1}, T) \lor f(x_1, \ldots, x_{n-1}, F)$

  $g(x_1, \ldots, x_{n-1}) = \Rightarrow h(x_1, \ldots, x_{n-1})$
Chapter 14.9. Exhaustive Search

Solve SAT problem with a recursive algorithm:

- what data will the recursion be applied to?
Chapter 14.9. Exhaustive Search

Solve SAT problem with a recursive algorithm:

- what data will the recursion be applied to?

  boolean formula \( f(x_1, \ldots, x_n) \)
Chapter 14.9. Exhaustive Search

Solve SAT problem with a recursive algorithm:

- what data will the recursion be applied to?
  
  boolean formula \( f(x_1, \ldots, x_n) \)

- what is the terminating (base) case?
  
  \[ f(x_1, \ldots, x_{n-1}, T) = f(x_1, \ldots, x_{n-1}, F) = \implies g(x_1, \ldots, x_{n-1}) = h(x_1, \ldots, x_{n-1}) \]
Chapter 14.9. Exhaustive Search

Solve SAT problem with a recursive algorithm:

• what data will the recursion be applied to?
  boolean formula \( f(x_1, \ldots, x_n) \)

• what is the terminating (base) case?
  \( n=0 \), formula without variables
Chapter 14.9. Exhaustive Search

Solve SAT problem with a recursive algorithm:

- what data will the recursion be applied to?
  
  boolean formula  \( f(x_1, \ldots, x_n) \)

- what is the terminating (base) case?
  
  \( n=0, \) formula without variables

- what is the recursive case?
  
  \[ f(x_1, \ldots, x_{n-1}, x_n) = \]

Chapter 14.9. Exhaustive Search

Solve SAT problem with a recursive algorithm:

- what data will the recursion be applied to?
  boolean formula $f(x_1, \ldots, x_n)$

- what is the terminating (base) case?
  $n=0$, formula without variables

- what is the recursive case?
  
  
  $f(x_1, \ldots, x_{n-1}, x_n) = f(x_1, \ldots, x_{n-1}, T) \lor f(x_1, \ldots, x_{n-1}, F)$
Solve SAT problem with a recursive algorithm:

- what data will the recursion be applied to?
  boolean formula \( f(x_1, \ldots, x_n) \)

- what is the terminating (base) case?
  \( n=0 \), formula without variables

- what is the recursive case?

\[
f(x_1, \ldots, x_{n-1}, x_n) = f(x_1, \ldots, x_{n-1}, T) \lor f(x_1, \ldots, x_{n-1}, F) \\
f(x_1, \ldots, x_{n-1}, T)
\]
Chapter 14.9. Exhaustive Search

Solve SAT problem with a recursive algorithm:

- what data will the recursion be applied to?
  
  boolean formula $f(x_1, \ldots, x_n)$

- what is the terminating (base) case?
  
  $n=0$, formula without variables

- what is the recursive case?

  
  $$
  f(x_1, \ldots, x_{n-1}, x_n) = \begin{cases} 
  f(x_1, \ldots, x_{n-1}, T) \lor f(x_1, \ldots, x_{n-1}, F) 
  
  f(x_1, \ldots, x_{n-1}, T) \implies g(x_1, \ldots, x_{n-1})
  
  \end{cases}
  $$
Chapter 14.9. Exhaustive Search

Solve SAT problem with a recursive algorithm:

- what data will the recursion be applied to?
  
  boolean formula \( f(x_1, \ldots, x_n) \)

- what is the terminating (base) case?
  
  \( n=0 \), formula without variables

- what is the recursive case?

  \[
  f(x_1, \ldots, x_{n-1}, x_n) = f(x_1, \ldots, x_{n-1}, T) \lor f(x_1, \ldots, x_{n-1}, F)
  \]

  \[
  f(x_1, \ldots, x_{n-1}, T) \implies g(x_1, \ldots, x_{n-1})
  \]

  \[
  f(x_1, \ldots, x_{n-1}, F)
  \]
Chapter 14.9. Exhaustive Search

Solve SAT problem with a recursive algorithm:

- what data will the recursion be applied to?
  boolean formula $f(x_1, \ldots, x_n)$

- what is the terminating (base) case?
  $n=0$, formula without variables

- what is the recursive case?

  $$f(x_1, \ldots, x_{n-1}, x_n) = f(x_1, \ldots, x_{n-1}, T) \lor f(x_1, \ldots, x_{n-1}, F)$$

  $f(x_1, \ldots, x_{n-1}, T) \implies g(x_1, \ldots, x_{n-1})$
  $f(x_1, \ldots, x_{n-1}, F) \implies h(x_1, \ldots, x_{n-1})$
Chapter 14.9. Exhaustive Search

Algorithm SAT Solver($f(x_1, \ldots, x_{n-1}, x_n)$)
Chapter 14.9. Exhaustive Search

Algorithm SAT Solver\( (f(x_1, \ldots, x_{n-1}, x_n)) \)

1. if \( n = 0 \), return \( f \);
Algorithm \textsc{SAT Solver}(f(x_1, \ldots, x_{n-1}, x_n))

1. \textbf{if} $n = 0$, \textbf{return} $(f)$;
2. \textbf{else} $g(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, T)$
Chapter 14.9. Exhaustive Search

Algorithm SAT SOLVER($f(x_1, \ldots, x_{n-1}, x_n)$)
1. if $n = 0$, return ($f$);
2. else $g(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, T)$
3. $h(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, F)$
Algorithm SAT SOLVER(\(f(x_1, \ldots, x_{n-1}, x_n)\))

1. if \(n = 0\), return \(f\);
2. else \(g(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, T)\)
3. \(h(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, F)\)
4. return (SAT SOLVER(\(g(x_1, \ldots, x_{n-1})\)) \(\lor\) SAT SOLVER(\(h(x_1, \ldots, x_{n-1})\)))
Chapter 14.9. Exhaustive Search

Algorithm SAT Solver\(f(x_1, \ldots, x_{n-1}, x_n)\)

1. if \(n = 0\), return \(f\);
2. else \(g(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, \text{T})\)
3. \(h(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, \text{F})\)
4. return \(\text{SAT Solver}(g(x_1, \ldots, x_{n-1})) \lor \text{SAT Solver}(h(x_1, \ldots, x_{n-1}))\)

Does this algorithm exhaustively search all assignments to the variables?
Chapter 14.9. Exhaustive Search

Algorithm SAT Solver($f(x_1, \ldots, x_{n-1}, x_n)$)

1. if $n = 0$, return ($f$);
2. else $g(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, T)$
3. $h(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, F)$
4. return (SAT Solver($g(x_1, \ldots, x_{n-1})$) $\lor$ SAT Solver($h(x_1, \ldots, x_{n-1})$))

Does this algorithm exhaustively search all assignments to the variables?

- draw a search tree based on the algorithm.
Chapter 14.9. Exhaustive Search

Algorithm \textsc{Sat Solver}(f(x_1, \ldots, x_{n-1}, x_n))

1. \textbf{if} \ n = 0, \ \textbf{return} (f);
2. \textbf{else} \ g(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, T)
3. \ \ h(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, F)
4. \ \ \ \textbf{return} (\textsc{Sat Solver}(g(x_1, \ldots, x_{n-1})) \lor \textsc{Sat Solver}(h(x_1, \ldots, x_{n-1})))

Does this algorithm exhaustively search all assignments to the variables?

- draw a \textit{search tree} based on the algorithm.
- what does the tree look like?
Chapter 14.9. Exhaustive Search

Algorithm SAT Solver\(f(x_1, \ldots , x_{n-1}, x_n)\)

1. \textbf{if } n = 0, \textbf{return } (f);
2. \textbf{else } \quad g(x_1, \ldots , x_{n-1}) = f(x_1, \ldots , x_{n-1}, T)
3. \quad h(x_1, \ldots , x_{n-1}) = f(x_1, \ldots , x_{n-1}, F)
4. \textbf{return } (\text{SAT Solver}(g(x_1, \ldots , x_{n-1})) \lor \\
\text{SAT Solver}(h(x_1, \ldots , x_{n-1})))

Does this algorithm exhaustively search all assignments to the variables?

- draw a \textit{search tree} based on the algorithm.
- what does the tree look like?
- what does each path mean?
Algorithm $\text{SAT SOLVER}(f(x_1, \ldots, x_{n-1}, x_n))$

1. if $n = 0$, return $(f)$;
2. else $g(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, T)$
3. $h(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, F)$
4. return $(\text{SAT SOLVER}(g(x_1, \ldots, x_{n-1})) \lor \text{SAT SOLVER}(h(x_1, \ldots, x_{n-1})))$

Does this algorithm exhaustively search all assignments to the variables?

- draw a *search tree* based on the algorithm.
- what does the tree look like?
- what does each path mean? how many paths?
Chapter 14.9. Exhaustive Search

Algorithm SAT Solver($f(x_1, \ldots, x_{n-1}, x_n)$)

1. if $n = 0$, return ($f$);
2. else $g(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, T)$
3. $h(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, F)$
4. return (SAT Solver($g(x_1, \ldots, x_{n-1})$) ∨
   SAT Solver($h(x_1, \ldots, x_{n-1})$))

Does this algorithm exhaustively search all assignments to the variables?

- draw a search tree based on the algorithm.
- what does the tree look like?
- what does each path mean? how many paths?
- time?
Chapter 14.9. Exhaustive Search

Algorithm $\text{SAT Solver}(f(x_1, \ldots, x_{n-1}, x_n))$

1. if $n = 0$, return $(f)$;
2. else $g(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, T)$
3. $h(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, F)$
4. return $(\text{SAT Solver}(g(x_1, \ldots, x_{n-1})) \lor \text{SAT Solver}(h(x_1, \ldots, x_{n-1})))$

Does this algorithm exhaustively search all assignments to the variables?

- draw a search tree based on the algorithm.
- what does the tree look like?
- what does each path mean? how many paths?
- time? $T(n) = 2T(n-1) + cn$, $T(0) = c$, 

$T(n) = \Theta(2^n)$
Chapter 14.9. Exhaustive Search

Algorithm SAT Solver \( f(x_1, \ldots, x_{n-1}, x_n) \)

1. if \( n = 0 \), return \( f \);
2. else \( g(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, T) \)
3. \( h(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, F) \)
4. return (SAT Solver \( g(x_1, \ldots, x_{n-1}) \)) \( \lor \) (SAT Solver \( h(x_1, \ldots, x_{n-1}) \))

Does this algorithm exhaustively search all assignments to the variables?

- draw a search tree based on the algorithm.
- what does the tree look like?
- what does each path mean? how many paths?
- time? \( T(n) = 2T(n-1) + cn, T(0) = c, \implies T(n) = \Theta(2^n) \)
Chapter 14.9. Exhaustive Search

Solve SAT problem with iterative algorithms
Chapter 14.9. Exhaustive Search

Solve SAT problem with iterative algorithms

- How? what to iterate on?
Chapter 14.9. Exhaustive Search

Solve SAT problem with iterative algorithms

• How? what to iterate on?
  assignments
Chapter 14.9. Exhaustive Search

Solve SAT problem with iterative algorithms

- How? what to iterate on?
  assignments
- what is the initial value?

\[ x_1 = F, x_2 = F, \ldots, x_n = F, \]

or simply \((F, F, \ldots, F)\)

always flip the last bit.
Chapter 14.9. Exhaustive Search

Solve SAT problem with iterative algorithms

- How? what to iterate on?
  - assignments
- what is the initial value?
  
  \[ x_1 = F, x_2 = F, \ldots, x_n = F, \]
Chapter 14.9. Exhaustive Search

Solve SAT problem with iterative algorithms

- How? what to iterate on?
  - assignments
- what is the initial value?
  - $x_1 = F, x_2 = F, \ldots, x_n = F$, or simply $(F, F, \ldots, F)$
Chapter 14.9. Exhaustive Search

Solve SAT problem with iterative algorithms

• How? what to iterate on?

  assignments

• what is the initial value?

  \[ x_1 = F, x_2 = F, \ldots, x_n = F, \text{ or simply } (F, F, \ldots, F) \]

• what to increment

  \[ \ldots, F, T, \ldots, T \rightarrow \ldots, T, F, \ldots, F \]
  always flip the last bit.
Chapter 14.9. Exhaustive Search

Solve SAT problem with iterative algorithms

• How? what to iterate on?

  assignments

• what is the initial value?

  \[ x_1 = F, x_2 = F, \ldots, x_n = F, \text{ or simply } (F, F, \ldots, F) \]

• what to increment

  \((\ldots, F, T, \ldots, T)\)
Chapter 14.9. Exhaustive Search

Solve SAT problem with iterative algorithms

- How? what to iterate on?
  
  assignments

- what is the initial value?

  \[ x_1 = F, x_2 = F, \ldots, x_n = F, \text{ or simply } (F, F, \ldots, F) \]

- what to increment

  \( (\ldots, F, T, \ldots, T) \longrightarrow (\ldots, T, F, \ldots, F) \)
Chapter 14.9. Exhaustive Search

Solve SAT problem with iterative algorithms

• How? what to iterate on?

  assignments

• what is the initial value?

  \[ x_1 = F, x_2 = F, \ldots, x_n = F, \text{ or simply } (F, F, \ldots, F) \]

• what to increment

  \[
  (\ldots, F, T, \ldots, T) \quad \rightarrow \quad (\ldots, T, F, \ldots, F)
  \]
  always flip the last bit.
Chapter 14.9. Exhaustive Search

Algorithm SAT Solver-Enum

1. for $\langle x_1, \ldots, x_n \rangle = \langle F, \ldots, F \rangle$ to $\langle T, \ldots, T \rangle$
2. $V = \text{Evaluate}(f, x_1, \ldots, x_n)$
3. if $V = T$, return $(T)$
4. return $(F)$

• for loop can be implemented by encoding vectors $\langle F, \ldots, F \rangle$, $\ldots$, $\langle T, \ldots, T \rangle$ with binary numbers then further with integers

• a decoding process is needed to converting integers back to vectors
Chapter 14.9. Exhaustive Search

Algorithm $\text{SAT Solver-Enum}(f(x_1, \ldots, x_{n-1}, x_n))$
Chapter 14.9. Exhaustive Search

Algorithm SAT_SOLVER_ENUM\( (f(x_1, \ldots, x_{n-1}, x_n)) \)

1. for \( \langle x_1, \ldots, x_n \rangle = \langle F, \ldots, F \rangle \) to \( \langle T, \ldots, T \rangle \)
Chapter 14.9. Exhaustive Search

Algorithm $\text{SAT SOLVER-ENUM}(f(x_1, \ldots, x_{n-1}, x_n))$

1. $\textbf{for } \langle x_1, \ldots, x_n \rangle = \langle F, \ldots, F \rangle \textbf{ to } \langle T, \ldots, T \rangle$
2. $V = Evaluate(f, x_1, \ldots, x_n)$
Chapter 14.9. Exhaustive Search

Algorithm $\text{SAT Solver-Enum}(f(x_1, \ldots, x_{n-1}, x_n))$

1. $\text{for } \langle x_1, \ldots, x_n \rangle = \langle F, \ldots, F \rangle \text{ to } \langle T, \ldots, T \rangle$
2. $V = \text{Evaluate}(f, x_1, \ldots, x_n)$
3. $\text{if } V = T, \text{ return } (T)$
Algorithm SAT Solver-Enum($f(x_1, \ldots, x_{n-1}, x_n)$)

1. for $\langle x_1, \ldots, x_n \rangle = \langle F, \ldots, F \rangle$ to $\langle T, \ldots, T \rangle$
2. $V = Evaluate(f, x_1, \ldots, x_n)$
3. if $V = T$, return (T)
4. return (F)
Algorithm \textsc{SAT Solver-Enum}(f(x_1, \ldots, x_{n-1}, x_n))

1. \textbf{for} \ (x_1, \ldots, x_n) = \langle F, \ldots, F \rangle \ \textbf{to} \ \langle T, \ldots, T \rangle \\
2. \quad V = \text{Evaluate}(f, x_1, \ldots, x_n) \\
3. \quad \textbf{if} \ V = T, \ \textbf{return} \ (T) \\
4. \quad \textbf{return} \ (F)

- \textbf{for} loop can be implemented by encoding vectors \langle F, \ldots, F \rangle, \\
  \ldots, \langle T, \ldots, T \rangle \ \text{with}
Chapter 14.9. Exhaustive Search

Algorithm \texttt{SAT Solver-Enum}(f(x_1, \ldots, x_{n-1}, x_n))

1. \textbf{for} \(\langle x_1, \ldots, x_n \rangle = \langle F, \ldots, F \rangle \ \textbf{to} \ \langle T, \ldots, T \rangle\)
2. \hspace{0.5cm} \(V = \text{Evaluate}(f, x_1, \ldots, x_n)\)
3. \hspace{0.5cm} \textbf{if} \(V = T\), \textbf{return} \(T\)
4. \hspace{0.5cm} \textbf{return} \(F\)

- \textbf{for} loop can be implemented by encoding vectors \(\langle F, \ldots, F \rangle, \ldots, \langle T, \ldots, T \rangle\) with binary numbers then
Chapter 14.9. Exhaustive Search

Algorithm SAT Solver-Enum\((f(x_1, \ldots, x_{n-1}, x_n))\)

1. \textbf{for} \(\langle x_1, \ldots, x_n \rangle = \langle F, \ldots, F \rangle\) \textbf{to} \(\langle T, \ldots, T \rangle\)
2. \(V = \text{Evaluate}(f, x_1, \ldots, x_n)\)
3. \textbf{if} \(V = T\), return \(T\)
4. return \(F\)

- \textbf{for} loop can be implemented by encoding vectors \(\langle F, \ldots, F \rangle, \ldots, \langle T, \ldots, T \rangle\) with binary numbers then further with integers
Chapter 14.9. Exhaustive Search

Algorithm SAT Solver-Enum($f(x_1, \ldots, x_{n-1}, x_n)$)

1. for $\langle x_1, \ldots, x_n \rangle = \langle F, \ldots, F \rangle$ to $\langle T, \ldots, T \rangle$
2. $V = \text{Evaluate}(f, x_1, \ldots, x_n)$
3. if $V = T$, return ($T$)
4. return ($F$)

- for loop can be implemented by encoding vectors $\langle F, \ldots, F \rangle$, $\ldots$, $\langle T, \ldots, T \rangle$ with binary numbers then further with integers

- a decoding process is needed to converting integers back to vectors
Chapter 14.9. Exhaustive Search

Iterative exhaustive search seems to be more convenient. Another example: Travel Salesman Problem (TSP).

Related problem: Hamiltonian Cycle.

Input: a graph \( G = (V, E) \)

Output: yes if and only if \( G \) contains a Hamiltonian cycle (Hamiltonian path is a cycle going through every vertex exactly once).

How to enumerate all cycles and validate?

• enumerate all permutations of \( (1, 2, \ldots, n) \)
• how to encode these permutations as integers?
Iterative exhaustive search seems to be more convenient
Iterative exhaustive search seems to be more convenient

Another example: Travel Salesman Problem (TSP)
Chapter 14.9. Exhaustive Search

Iterative exhaustive search seems to be more convenient

Another example: Travel Salesman Problem (TSP)

Related problem: Hamiltonian Cycle
Chapter 14.9. Exhaustive Search

Iterative exhaustive search seems to be more convenient

Another example: Travel Salesman Problem (TSP)

Related problem: Hamiltonian Cycle

**Input:** a graph $G = (V, E)$

**Output:** yes if and only if $G$ contains a Hamiltonian cycle

(Hamiltonian path is a cycle going through every vertex exactly once.)
Iterative exhaustive search seems to be more convenient

Another example: Travel Salesman Problem (TSP)

Related problem: Hamiltonian Cycle

**Input**: a graph $G = (V, E)$

**Output**: yes if and only if $G$ contains a Hamiltonian cycle

(Hamiltonian path is a cycle going through every vertex exactly once.)

How to enumerate all cycles and validate?
Iterative exhaustive search seems to be more convenient

Another example: Travel Salesman Problem (TSP)

Related problem: Hamiltonian Cycle

**Input:** a graph $G = (V, E)$

**Output:** yes if and only if $G$ contains a Hamiltonian cycle

(Hamiltonian path is a cycle going through every vertex exactly once.)

How to enumerate all cycles and validate?

- enumerate all permutations of $(12\ldots n)$
Iterative exhaustive search seems to be more convenient

Another example: Travel Salesman Problem (TSP)

Related problem: Hamiltonian Cycle

**Input:** a graph $G = (V, E)$

**Output:** yes if and only if $G$ contains a Hamiltonian cycle

(Hamiltonian path is a cycle going through every vertex exactly once.)

How to enumerate all cycles and validate?

- enumerate all permutations of $(12\ldots n)$
- how to encode these permutations as integers?
Chapter 14.9. Exhaustive Search

Exhaustive search could be non-trivial.
Exhaustive search could be non-trivial

Maximum Independent Set

**Input**: a graph \( G = (V, E) \)

**Output**: a subset \( I \subseteq V \) such that

1. \( \forall u, v \in I, (u, v) \not\in E \)
2. \( |I| \) is the maximum.

• trivial exhaustive search: check every subset of \( V \) and verify
  
  Use \( n \)-binary bits to encode a subset; totally \( 2^n \) subsets

• non-trivial: use a search tree, achieving a better time upper bound, taking advantage of the independent set
Chapter 14.9. Exhaustive Search

Exhaustive search could be non-trivial

Maximum Independent Set

**Input:** a graph $G = (V, E)$

**Output:** a subset $I \subseteq V$ such that

1. $\forall u, v \in I, (u, v) \notin E$, 

Exhaustive search could be non-trivial

Maximum Independent Set

\textbf{Input:} a graph \( G = (V, E) \)

\textbf{Output:} a subset \( I \subseteq V \) such that

1. \( \forall u, v \in I, (u, v) \notin E \), and

2. \( |I| \) is the maximum.
Chapter 14.9. Exhaustive Search

Exhaustive search could be non-trivial

Maximum Independent Set

**Input:** a graph $G = (V, E)$

**Output:** a subset $I \subseteq V$ such that

1. $\forall u, v \in I, (u, v) \notin E$, and
2. $|I|$ is the maximum.

- trivial exhaustive search: check every subset of $V$ and verify
Chapter 14.9. Exhaustive Search

Exhaustive search could be non-trivial

Maximum Independent Set

**Input:** a graph $G = (V, E)$

**Output:** a subset $I \subseteq V$ such that

1. $\forall u, v \in I$, $(u, v) \notin E$, and
2. $|I|$ is the maximum.

- trivial exhaustive search: check every subset of $V$ and verify
  - use $n$-binary bits to encode a subset; totally $2^n$ subsets
Exhaustive search could be non-trivial

Maximum Independent Set

**Input:** a graph $G = (V, E)$

**Output:** a subset $I \subseteq V$ such that

1. $\forall u, v \in I$, $(u, v) \notin E$, and
2. $|I|$ is the maximum.

- trivial exhaustive search: check every subset of $V$ and verify
  - use $n$-binary bits to encode a subset; totally $2^n$ subsets
- non-trivial: use a search tree, achieving a better time upper bound.
Exhaustive search could be non-trivial

Maximum Independent Set

**Input:** a graph $G = (V, E)$

**Output:** a subset $I \subseteq V$ such that

1. $\forall u, v \in I, (u, v) \notin E$, and
2. $|I|$ is the maximum.

- **trivial exhaustive search:** check every subset of $V$ and verify
  use $n$-binary bits to encode a subset; totally $2^n$ subsets

- **non-trivial:** use a search tree, achieving a better time upper bound.
  taking advantage of the independent set
Chapter 14.9. Exhaustive Search

The algorithm follows a logical search tree

• given a graph $G$, it picks an arbitrary vertex $v$ from $G$;
• exhaustively, there are two cases to consider:
  1. to include $v$ in the independent set;
  2. to exclude $v$ from the independent set;
• resulting in two subgraphs $G_1$ and $G_2$ to be recursively considered,
  1. $G_1$ is the result of $G$ after $v$ and all its neighbors are removed;
  2. $G_2$ is the result of $G$ after $v$ is removed.
• the algorithm terminates when the considered graph is empty.
Chapter 14.9. Exhaustive Search

The algorithm follows a logical search tree

- given a graph $G$, it picks an arbitrary vertex $v$ from $G$;
The algorithm follows a logical search tree

- given a graph $G$, it picks an arbitrary vertex $v$ from $G$;
- exhaustively, there are two cases to consider:
  1. to include $v$ in the independent set;
  2. to exclude $v$ from the independent set;

resulting in two subgraphs $G_1$ and $G_2$ to be recursively considered,

1. $G_1$ is the result of $G$ after $v$ and all its neighbors are removed;
2. $G_2$ is the result of $G$ after $v$ is removed.

the algorithm terminates when the considered graph is empty.
Chapter 14.9. Exhaustive Search

The algorithm follows a logical search tree

- given a graph $G$, it picks an arbitrary vertex $v$ from $G$;
- exhaustively, there are two cases to consider:
  1. to include $v$ in the independent set;
  2. to exclude $v$ from the independent set.

  - $G_1$ is the result of $G$ after $v$ and all its neighbors are removed;
  - $G_2$ is the result of $G$ after $v$ is removed.

  The algorithm terminates when the considered graph is empty.
Chapter 14.9. Exhaustive Search

The algorithm follows a logical search tree

- given a graph $G$, it picks an arbitrary vertex $v$ from $G$;
- exhaustively, there are two cases to consider:
  1. to include $v$ in the independent set;
  2. to exclude $v$ from the independent set;
Chapter 14.9. Exhaustive Search

The algorithm follows a logical search tree

- given a graph $G$, it picks an arbitrary vertex $v$ from $G$;
- exhaustively, there are two cases to consider:
  1. to include $v$ in the independent set;
  2. to exclude $v$ from the independent set;
- resulting in two subgraphs $G_1$ and $G_2$ to be recursively considered,
Chapter 14.9. Exhaustive Search

The algorithm follows a logical search tree

- given a graph $G$, it picks an arbitrary vertex $v$ from $G$;
- exhaustively, there are two cases to consider:
  1. to include $v$ in the independent set;
  2. to exclude $v$ from the independent set;
- resulting in two subgraphs $G_1$ and $G_2$ to be recursively considered,
  1. $G_1$ is the result of $G$ after $v$
Chapter 14.9. Exhaustive Search

The algorithm follows a logical search tree

- given a graph \( G \), it picks an arbitrary vertex \( v \) from \( G \);
- exhaustively, there are two cases to consider:
  1. to include \( v \) in the independent set;
  2. to exclude \( v \) from the independent set;

- resulting in two subgraphs \( G_1 \) and \( G_2 \) to be recursively considered,
  1. \( G_1 \) is the result of \( G \) after \( v \) and all its neighbors are removed;
Chapter 14.9 Exhaustive Search

The algorithm follows a logical search tree

- given a graph $G$, it picks an arbitrary vertex $v$ from $G$;
- exhaustively, there are two cases to consider:
  1. to include $v$ in the independent set;
  2. to exclude $v$ from the independent set;
- resulting in two subgraphs $G_1$ and $G_2$ to be recursively considered,
  1. $G_1$ is the result of $G$ after $v$ and all its neighbors are removed;
  2. $G_2$ is the result of $G$ after $v$ is removed.
The algorithm follows a logical search tree

- given a graph $G$, it picks an arbitrary vertex $v$ from $G$;
- exhaustively, there are two cases to consider:
  1. to include $v$ in the independent set;
  2. to exclude $v$ from the independent set;
- resulting in two subgraphs $G_1$ and $G_2$ to be recursively considered,
  1. $G_1$ is the result of $G$ after $v$ and all its neighbors are removed;
  2. $G_2$ is the result of $G$ after $v$ is removed.
- the algorithm terminates when the considered graph is empty.
Chapter 14.9. Exhaustive Search

Algorithm $\text{MaxIndSet} \ (G)$
**Chapter 14.9. Exhaustive Search**

**Algorithm MaxIndSet** ($G$)

1. **if** $G = \emptyset$ **return** ($\emptyset$)
Algorithm \textbf{MaxIndSet} \((G)\)

1. \textbf{if} \( G = \emptyset \) \textbf{return} \((\emptyset)\)
2. \textbf{else} pick an arbitrary vertex \( v \) in \( G \)
Chapter 14.9. Exhaustive Search

Algorithm $\text{MaxIndSet} \ (G)$

1. if $G = \emptyset$ return $(\emptyset)$
2. else pick an arbitrary vertex $v$ in $G$
3. let $G_1$ be $G$ with $v$ and all its neighbors removed

$T(n) = T(n-1-m) + T(n-1) + cn^2$ where $m$ is the number of neighbors of $v$'s
Algorithm \textsc{MaxIndSet} \((G)\)

1. \textbf{if} \quad G = \emptyset \quad \textbf{return} \quad (\emptyset)
2. \textbf{else} \quad \text{pick an arbitrary vertex} \; v \; \text{in} \; G
3. \quad \text{let} \; G_1 \; \text{be} \; G \; \text{with} \; v \; \text{and all its neighbors removed}
4. \quad I_1 = \{v\} \cup \textsc{MaxIndSet} \; (G_1)
Algorithm **MaxIndSet** ($G$)

1. if $G = \emptyset$ return $(\emptyset)$
2. else pick an arbitrary vertex $v$ in $G$
3. let $G_1$ be $G$ with $v$ and all its neighbors removed
4. $I_1 = \{v\} \cup \text{MaxIndSet}(G_1)$
5. let $G_2$ be $G$ with $v$ removed

• the algorithm is a search tree
• the time complexity:

$$T(n) = T(n-1-m) + T(n-1) + cn^2$$

where $m$ is the number of neighbors of $v$'s
Algorithm $\text{MaxIndSet} \ (G)$

1. if $G = \emptyset$ return $(\emptyset)$
2. else pick an arbitrary vertex $v$ in $G$
3. let $G_1$ be $G$ with $v$ and all its neighbors removed
4. $I_1 = \{v\} \cup \text{MaxIndSet} \ (G_1)$
5. let $G_2$ be $G$ with $v$ removed
6. $I_2 = \text{MaxIndSet} \ (G_2)$
Algorithm $\text{MaxIndSet} (G)$

1. \textbf{if} $G = \emptyset$ return $(\emptyset)$
2. \textbf{else} pick an arbitrary vertex $v$ in $G$
3. let $G_1$ be $G$ with $v$ and all its neighbors removed
4. $I_1 = \{v\} \cup \text{MaxIndSet} (G_1)$
5. let $G_2$ be $G$ with $v$ removed
6. $I_2 = \text{MaxIndSet} (G_2)$
7. \textbf{if} $|I_1| \geq |I_2|$ return $(I_1)$
8. return $(I_2)$
Algorithm $\text{MaxIndSet} \ (G)$

1. if $G = \emptyset$ return $\emptyset$
2. else pick an arbitrary vertex $v$ in $G$
3. let $G_1$ be $G$ with $v$ and all its neighbors removed
4. $I_1 = \{v\} \cup \text{MaxIndSet} \ (G_1)$
5. let $G_2$ be $G$ with $v$ removed
6. $I_2 = \text{MaxIndSet} \ (G_2)$
7. if $|I_1| \geq |I_2|$ return $I_1$
8. else return $I_2$
Algorithm **MaxIndSet** \((G)\)

1. **if** \(G = \emptyset\) **return** \((\emptyset)\)
2. **else** pick an arbitrary vertex \(v\) in \(G\)
3. let \(G_1\) be \(G\) with \(v\) and all its neighbors removed
4. \(I_1 = \{v\} \cup \text{MaxIndSet} (G_1)\)
5. let \(G_2\) be \(G\) with \(v\) removed
6. \(I_2 = \text{MaxIndSet} (G_2)\)
7. **if** \(|I_1| \geq |I_2|\) **return** \((I_1)\)
8. **else** **return** \((I_2)\)

- the algorithm is a search tree
Algorithm $\text{MaxIndSet} \ (G)$

1. if $G = \emptyset$ return ($\emptyset$)
2. else pick an arbitrary vertex $v$ in $G$
3. let $G_1$ be $G$ with $v$ and all its neighbors removed
4. $I_1 = \{v\} \cup \text{MaxIndSet} \ (G_1)$
5. let $G_2$ be $G$ with $v$ removed
6. $I_2 = \text{MaxIndSet} \ (G_2)$
7. if $|I_1| \geq |I_2|$ return ($I_1$)
8. else return ($I_2$)

- the algorithm is a search tree
- the time complexity:
Chapter 14.9. Exhaustive Search

Algorithm \textbf{MaxIndSet} \((G)\)

1. \textbf{if} \(G = \emptyset\) \textbf{return} \((\emptyset)\)
2. \textbf{else} pick an arbitrary vertex \(v\) in \(G\)
3. let \(G_1\) be \(G\) with \(v\) and all its neighbors removed
4. \(I_1 = \{v\} \cup \text{MaxIndSet} \((G_1)\)\)
5. let \(G_2\) be \(G\) with \(v\) removed
6. \(I_2 = \text{MaxIndSet} \((G_2)\)\)
7. \textbf{if} \(|I_1| \geq |I_2|\) \textbf{return} \((I_1)\)
8. \textbf{else} \textbf{return} \((I_2)\)

- the algorithm is a search tree
- the time complexity: \(T(|G|) = cn^2 + T(|G_1|) + T(|G_2|)\)
Algorithm MaxIndSet \((G)\)

1. \textbf{if} \(G = \emptyset\) \textbf{return} \((\emptyset)\)
2. \textbf{else} pick an arbitrary vertex \(v\) in \(G\)
3. let \(G_1\) be \(G\) with \(v\) and all its neighbors removed
4. \(I_1 = \{v\} \cup \text{MaxIndSet} \left(G_1\right)\)
5. let \(G_2\) be \(G\) with \(v\) removed
6. \(I_2 = \text{MaxIndSet} \left(G_2\right)\)
7. \textbf{if} \(|I_1| \geq |I_2|\) \textbf{return} \((I_1)\)
8. \textbf{else} \textbf{return} \((I_2)\)

- the algorithm is a search tree
- the time complexity: \(T(|G|) = cn^2 + T(|G_1|) + T(|G_2|)\)
  \[T(n) = T(n - 1 - m) + T(n - 1) + cn^2\]
  where \(m\) is the number of neighbors of \(v\)’s
Chapter 14.9. Exhaustive Search

\[ T(n) = T(n - 1 - m) + T(n - 1) + cn^2 \]
Chapter 14.9. Exhaustive Search

\[ T(n) = T(n - 1 - m) + T(n - 1) + cn^2 \]

- \( m \geq 0, \)
Chapter 14.9. Exhaustive Search

\[ T(n) = T(n - 1 - m) + T(n - 1) + cn^2 \]

- \( m \geq 0 \), \( T(n) \leq T(n - 1) + T(n - 1) + cn^2 \),
Chapter 14.9. Exhaustive Search

\[ T(n) = T(n - 1 - m) + T(n - 1) + cn^2 \]

- \( m \geq 0, T(n) \leq T(n - 1) + T(n - 1) + cn^2 \), \( \implies T(n) = O(2^n) \)
Chapter 14.9. Exhaustive Search

\[ T(n) = T(n - 1 - m) + T(n - 1) + cn^2 \]

- \( m \geq 0, \ T(n) \leq T(n - 1) + T(n - 1) + cn^2, \implies T(n) = O(2^n) \)
- Can we guarantee \( m \geq 1 \)
Chapter 14.9. Exhaustive Search

\[ T(n) = T(n - 1 - m) + T(n - 1) + cn^2 \]

- \( m \geq 0, \ T(n) \leq T(n - 1) + T(n - 1) + cn^2, \implies T(n) = O(2^n) \)
- Can we guarantee \( m \geq 1 \) so we have 
  \[ T(n) \leq T(n - 2) + T(n - 1) + cn^2, \]
Chapter 14.9. Exhaustive Search

\[ T(n) = T(n - 1 - m) + T(n - 1) + cn^2 \]

- If \( m \geq 0 \), then \( T(n) \leq T(n - 1) + T(n - 1) + cn^2 \), \( \implies T(n) = O(2^n) \)
- Can we guarantee \( m \geq 1 \) so we have \( T(n) \leq T(n - 2) + T(n - 1) + cn^2 \), \( \implies T(n) = O(1.6181^n) \)
Chapter 14.9. Exhaustive Search

\[ T(n) = T(n - 1 - m) + T(n - 1) + cn^2 \]

- \( m \geq 0, \quad T(n) \leq T(n - 1) + T(n - 1) + cn^2, \quad \implies \quad T(n) = O(2^n) \)
- Can we guarantee \( m \geq 1 \) so we have
  \[ T(n) \leq T(n - 2) + T(n - 1) + cn^2, \quad \implies \quad T(n) = O(1.6181^n) \]
- Or even better, to guarantee \( m \geq 2 \)?
Chapter 14.9. Exhaustive Search

\[ T(n) = T(n - 1 - m) + T(n - 1) + cn^2 \]

- \( m \geq 0, \) \( T(n) \leq T(n - 1) + T(n - 1) + cn^2, \implies T(n) = O(2^n) \)
- Can we guarantee \( m \geq 1 \) so we have
  \( T(n) \leq T(n - 2) + T(n - 1) + cn^2, \implies T(n) = O(1.6181^n) \)
- Or even better, to guarantee \( m \geq 2? \) if we can,
  \( T(n) \leq T(n - 3) + T(n - 1) + cn^2, \)
Chapter 14.9. Exhaustive Search

\[ T(n) = T(n - 1 - m) + T(n - 1) + cn^2 \]

- \( m \geq 0, \ T(n) \leq T(n - 1) + T(n - 1) + cn^2, \implies T(n) = O(2^n) \)
- Can we guarantee \( m \geq 1 \) so we have  
  \( T(n) \leq T(n - 2) + T(n - 1) + cn^2, \implies T(n) = O(1.6181^n) \)
- Or even better, to guarantee \( m \geq 2 \)? if we can,  
  \( T(n) \leq T(n - 3) + T(n - 1) + cn^2, \implies T(n) = O(1.5^n) \)
Chapter 14.9. Exhaustive Search

\[ T(n) = T(n - 1 - m) + T(n - 1) + cn^2 \]

- \( m \geq 0, \ T(n) \leq T(n - 1) + T(n - 1) + cn^2, \implies T(n) = O(2^n) \)

- Can we guarantee \( m \geq 1 \) so we have
  \[ T(n) \leq T(n - 2) + T(n - 1) + cn^2, \implies T(n) = O(1.6181^n) \]

- Or even better, to guarantee \( m \geq 2 \)? if we can,
  \[ T(n) \leq T(n - 3) + T(n - 1) + cn^2, \implies T(n) = O(1.5^n) \]

use the substitution method to prove \( T(n) = O(1.5^n) \).
Chapter 14.9. Exhaustive Search

\[ T(n) = T(n - 1 - m) + T(n - 1) + cn^2 \]

- \( m \geq 0 \), \( T(n) \leq T(n - 1) + T(n - 1) + cn^2 \), \( \implies T(n) = O(2^n) \)

- Can we guarantee \( m \geq 1 \) so we have
  \( T(n) \leq T(n - 2) + T(n - 1) + cn^2 \), \( \implies T(n) = O(1.6181^n) \)

- Or even better, to guarantee \( m \geq 2 \)? if we can,
  \( T(n) \leq T(n - 3) + T(n - 1) + cn^2 \), \( \implies T(n) = O(1.5^n) \)

use the substitution method to prove \( T(n) = O(1.5^n) \).

- Can we guarantee \( m \geq 3 \)?
Chapter 14.9. Exhaustive Search

\[ T(n) = T(n - 1 - m) + T(n - 1) + cn^2 \]

- \( m \geq 0 \), \( T(n) \leq T(n - 1) + T(n - 1) + cn^2 \), \( \implies T(n) = O(2^n) \)
- Can we guarantee \( m \geq 1 \) so we have
  \( T(n) \leq T(n - 2) + T(n - 1) + cn^2 \), \( \implies T(n) = O(1.6181^n) \)
- Or even better, to guarantee \( m \geq 2 \)? if we can,
  \( T(n) \leq T(n - 3) + T(n - 1) + cn^2 \), \( \implies T(n) = O(1.5^n) \)

use the substitution method to prove \( T(n) = O(1.5^n) \).

- Can we guarantee \( m \geq 3 \)? possible but a little more complicated.
Chapter 14.9. Exhaustive Search

Let $T(n) \leq T(n - 3) + T(n - 1) + cn^2$, with $T(1) = O(1)$
Chapter 14.9. Exhaustive Search

Let \( T(n) \leq T(n-3) + T(n-1) + cn^2 \), with \( T(1) = O(1) \)

**Claim:** \( T(n) = O(1.5^n) \)
Chapter 14.9. Exhaustive Search

Let $T(n) \leq T(n - 3) + T(n - 1) + cn^2$, with $T(1) = O(1)$

Claim: $T(n) = O(1.5^n)$

Proof (use the substitution method)
Chapter 14.9. Exhaustive Search

Let $T(n) \leq T(n-3) + T(n-1) + cn^2$, with $T(1) = O(1)$

Claim: $T(n) = O(1.5^n)$

Proof (use the substitution method)

Assume that $T(k) \leq 1.5^k$ for all $k < n$. 

...
Chapter 14.9. Exhaustive Search

Let $T(n) \leq T(n - 3) + T(n - 1) + cn^2$, with $T(1) = O(1)$

**Claim:** $T(n) = O(1.5^n)$

**Proof** (use the substitution method)

Assume that $T(k) \leq 1.5^k$ for all $k < n$. Then

$$T(n) \leq T(n - 3) + T(n - 1) + n^2$$
Let $T(n) \leq T(n - 3) + T(n - 1) + cn^2$, with $T(1) = O(1)$

**Claim:** $T(n) = O(1.5^n)$

**Proof (use the substitution method)**

Assume that $T(k) \leq 1.5^k$ for all $k < n$. Then

$$T(n) \leq T(n - 3) + T(n - 1) + n^2 \leq 1.5^{n-3} + 1.5^{n-1} + n^2$$
Chapter 14.9. Exhaustive Search

Let $T(n) \leq T(n - 3) + T(n - 1) + cn^2$, with $T(1) = O(1)$

Claim: $T(n) = O(1.5^n)$

Proof (use the substitution method)

Assume that $T(k) \leq 1.5^k$ for all $k < n$. Then

$$T(n) \leq T(n - 3) + T(n - 1) + n^2 \leq 1.5^{n-3} + 1.5^{n-1} + n^2$$

when $n > n_0$ ($n_0$ to be determined)
Chapter 14.9. Exhaustive Search

Let $T(n) \leq T(n - 3) + T(n - 1) + cn^2$, with $T(1) = O(1)$

**Claim:** $T(n) = O(1.5^n)$

**Proof** (use the substitution method)

Assume that $T(k) \leq 1.5^k$ for all $k < n$. Then

$$T(n) \leq T(n - 3) + T(n - 1) + n^2 \leq 1.5^{n-3} + 1.5^{n-1} + n^2$$

when $n > n_0$ ($n_0$ to be determined)

$$\leq 1.5^{n-3}(1 + 1.5^2 + \frac{n^2}{1.5^{n-3}})$$
Let $T(n) \leq T(n - 3) + T(n - 1) + cn^2$, with $T(1) = O(1)$

**Claim:** $T(n) = O(1.5^n)$

**Proof (use the substitution method)**

Assume that $T(k) \leq 1.5^k$ for all $k < n$. Then

$$T(n) \leq T(n - 3) + T(n - 1) + n^2 \leq 1.5^{n-3} + 1.5^{n-1} + n^2$$

when $n > n_0$ ($n_0$ to be determined)

$$\leq 1.5^{n-3}(1 + 1.5^2 + \frac{n^2}{1.5^{n-3}}) \leq 1.5^{n-3}(1 + 1.5^2 + 0.1)$$
Chapter 14.9. Exhaustive Search

Let $T(n) \leq T(n-3) + T(n-1) + cn^2$, with $T(1) = O(1)$

Claim: $T(n) = O(1.5^n)$

Proof (use the substitution method)

Assume that $T(k) \leq 1.5^k$ for all $k < n$. Then

$$T(n) \leq T(n-3) + T(n-1) + n^2 \leq 1.5^{n-3} + 1.5^{n-1} + n^2$$

when $n > n_0$ ($n_0$ to be determined)

$$\leq 1.5^{n-3}(1 + 1.5^2 + \frac{n^2}{1.5^{n-3}}) \leq 1.5^{n-3}(1 + 1.5^2 + 0.1) = 1.5^{n-3}(1 + 2.25 + 0.1)$$
Chapter 14.9. Exhaustive Search

Let $T(n) \leq T(n - 3) + T(n - 1) + cn^2$, with $T(1) = O(1)$

**Claim:** $T(n) = O(1.5^n)$

**Proof** (use the substitution method)

Assume that $T(k) \leq 1.5^k$ for all $k < n$. Then

$$T(n) \leq T(n - 3) + T(n - 1) + n^2 \leq 1.5^{n-3} + 1.5^{n-1} + n^2$$

when $n > n_0$ ($n_0$ to be determined)

$$\leq 1.5^{n-3}(1 + 1.5^2 + \frac{n^2}{1.5^{n-3}}) \leq 1.5^{n-3}(1 + 1.5^2 + 0.1) = 1.5^{n-3}(1 + 2.25 + 0.1)$$

$$= 1.5^{n-3} \times 3.35$$
Let $T(n) \leq T(n - 3) + T(n - 1) + cn^2$, with $T(1) = O(1)$

**Claim:** $T(n) = O(1.5^n)$

**Proof** (use the substitution method)

Assume that $T(k) \leq 1.5^k$ for all $k < n$. Then

$$T(n) \leq T(n - 3) + T(n - 1) + n^2 \leq 1.5^{n-3} + 1.5^{n-1} + n^2$$

when $n > n_0$ ($n_0$ to be determined)

$$\leq 1.5^{n-3}(1 + 1.5^2 + \frac{n^2}{1.5^{n-3}}) \leq 1.5^{n-3}(1 + 1.5^2 + 0.1) = 1.5^{n-3}(1 + 2.25 + 0.1)$$

$$= 1.5^{n-3} \times 3.35 \leq 1.5^{n-3} \times 3.375$$
Chapter 14.9. Exhaustive Search

Let $T(n) \leq T(n - 3) + T(n - 1) + cn^2$, with $T(1) = O(1)$

**Claim:** $T(n) = O(1.5^n)$

**Proof** (use the substitution method)

Assume that $T(k) \leq 1.5^k$ for all $k < n$. Then

$$T(n) \leq T(n - 3) + T(n - 1) + n^2 \leq 1.5^{n-3} + 1.5^{n-1} + n^2$$

when $n > n_0$ ($n_0$ to be determined)

$$\leq 1.5^{n-3}(1 + 1.5^2 + \frac{n^2}{1.5^{n-3}}) \leq 1.5^{n-3}(1 + 1.5^2 + 0.1) = 1.5^{n-3}(1 + 2.25 + 0.1)$$

$$= 1.5^{n-3} \times 3.35 \leq 1.5^{n-3} \times 3.375 = 1.5^{n-3} \times 1.5^3$$
Chapter 14.9. Exhaustive Search

Let \( T(n) \leq T(n - 3) + T(n - 1) + cn^2 \), with \( T(1) = O(1) \)

**Claim:** \( T(n) = O(1.5^n) \)

**Proof** (use the substitution method)

Assume that \( T(k) \leq 1.5^k \) for all \( k < n \). Then

\[
T(n) \leq T(n - 3) + T(n - 1) + n^2 \leq 1.5^{n-3} + 1.5^{n-1} + n^2
\]

when \( n > n_0 \) (\( n_0 \) to be determined)

\[
\leq 1.5^{n-3}(1 + 1.5^2 + \frac{n^2}{1.5^{n-3}}) \leq 1.5^{n-3}(1 + 1.5^2 + 0.1) = 1.5^{n-3}(1 + 2.25 + 0.1)
\]

\[
= 1.5^{n-3} \times 3.35 \leq 1.5^{n-3} \times 3.375 = 1.5^{n-3} \times 1.5^3 = 1.5^n
\]
Chapter 14.9. Exhaustive Search

Let $T(n) \leq T(n - 3) + T(n - 1) + cn^2$, with $T(1) = O(1)$

**Claim:** $T(n) = O(1.5^n)$

**Proof (use the substitution method)**

Assume that $T(k) \leq 1.5^k$ for all $k < n$. Then

$$T(n) \leq T(n - 3) + T(n - 1) + n^2 \leq 1.5^{n-3} + 1.5^{n-1} + n^2$$

when $n > n_0$ ($n_0$ to be determined)

$$\leq 1.5^{n-3}(1 + 1.5^2 + \frac{n^2}{1.5^{n-3}}) \leq 1.5^{n-3}(1 + 1.5^2 + 0.1) = 1.5^{n-3}(1 + 2.25 + 0.1)$$

$$= 1.5^{n-3} \times 3.35 \leq 1.5^{n-3} \times 3.375 = 1.5^{n-3} \times 1.5^3 = 1.5^n$$

Now we decide $n_0$:

$$\frac{n^2}{1.5^{n-3}} \leq 0.1$$
Chapter 14.9. Exhaustive Search

Let $T(n) \leq T(n - 3) + T(n - 1) + cn^2$, with $T(1) = O(1)$

**Claim:** $T(n) = O(1.5^n)$

**Proof** (use the substitution method)

Assume that $T(k) \leq 1.5^k$ for all $k < n$. Then

$$T(n) \leq T(n - 3) + T(n - 1) + n^2 \leq 1.5^{n-3} + 1.5^{n-1} + n^2$$

when $n > n_0$ ($n_0$ to be determined)

$$\leq 1.5^{n-3}(1 + 1.5^2 + \frac{n^2}{1.5^{n-3}}) \leq 1.5^{n-3}(1 + 1.5^2 + 0.1) = 1.5^{n-3}(1 + 2.25 + 0.1)$$

$$= 1.5^{n-3} \times 3.35 \leq 1.5^{n-3} \times 3.375 = 1.5^{n-3} \times 1.5^3 = 1.5^n$$

Now we decide $n_0$:

$$\frac{n^2}{1.5^{n-3}} \leq 0.1 \implies n^2 \leq 0.1 \times 1.5^{n-3}$$
Chapter 14.9. Exhaustive Search

Let $T(n) \leq T(n - 3) + T(n - 1) + cn^2$, with $T(1) = O(1)$

Claim: $T(n) = O(1.5^n)$

Proof (use the substitution method)

Assume that $T(k) \leq 1.5^k$ for all $k < n$. Then

$$T(n) \leq T(n - 3) + T(n - 1) + n^2 \leq 1.5^{n-3} + 1.5^{n-1} + n^2$$

when $n > n_0$ ($n_0$ to be determined)

$$\leq 1.5^{n-3}(1 + 1.5^2 + \frac{n^2}{1.5^{n-3}}) \leq 1.5^{n-3}(1 + 1.5^2 + 0.1) = 1.5^{n-3}(1 + 2.25 + 0.1)$$

$$= 1.5^{n-3} \times 3.35 \leq 1.5^{n-3} \times 3.375 = 1.5^{n-3} \times 1.5^3 = 1.5^n$$

Now we decide $n_0$:

$$\frac{n^2}{1.5^{n-3}} \leq 0.1 \implies n^2 \leq 0.1 \times 1.5^{n-3} \text{ holds when roughly } n \geq n_0 = 29$$
Chapter 14.9. Exhaustive Search

- Algorithms for SAT and \texttt{MAXINDSET} run in exponential time $O(2^n)$ or $O(\gamma^n)$ for $1 < \gamma < 2$. 
Chapter 14.9. Exhaustive Search

- Algorithms for SAT and \texttt{MAXINDSET} run in exponential time $O(2^n)$ or $O(\gamma^n)$ for $1 < \gamma < 2$

- search tree (solution search space) is large, \textit{inherently} large
Chapter 14.9. Exhaustive Search

- Algorithms for SAT and $\text{MAXINDSET}$ run in exponential time $O(2^n)$ or $O(\gamma^n)$ for $1 < \gamma < 2$
- Search tree (solution search space) is large, inherently large
- Search tree does not have obvious overlapping subproblems,
Chapter 14.9. Exhaustive Search

- Algorithms for SAT and MAXINDSET run in exponential time $O(2^n)$ or $O(\gamma^n)$ for $1 < \gamma < 2$
- Search tree (solution search space) is large, inherently large
- Search tree does not have obvious overlapping subproblems, which otherwise would incur dynamic programming approaches.
Chapter 15. Dynamic Programming

Chapter 15. Dynamic Programming
Chapter 15. Dynamic Programming

• we will deal with optimization problems
we will deal with optimization problems

the output solution satisfies certain desired optimality
• we will deal with optimization problems
  the output solution satisfies certain desired optimality
• such problems can be solved with divide-and-conquer
Chapter 15. Dynamic Programming

• we will deal with optimization problems
  the output solution satisfies certain desired optimality
• such problems can be solved with divide-and-conquer
  direct, top-down recursive approaches would incur high time complexity
Chapter 15. Dynamic Programming

- we will deal with optimization problems
  the output solution satisfies certain desired optimality
- such problems can be solved with divide-and-conquer
  direct, top-down recursive approaches would incur high time complexity
- there are overlapping subproblems
Chapter 15. Dynamic Programming

• we will deal with optimization problems
  the output solution satisfies certain desired optimality
• such problems can be solved with divide-and-conquer
  direct, top-down recursive approaches would incur high time complexity
• there are overlapping subproblems
  bottom-up approaches avoid re-computation for subproblems
Chapter 15. Dynamic Programming

Will discuss the following examples:
Chapter 15. Dynamic Programming

Will discuss the following examples:

- assembly-line scheduling
Will discuss the following examples:

- assembly-line scheduling
- casino dice problem and decoding algorithm
Chapter 15. Dynamic Programming

Will discuss the following examples:

• assembly-line scheduling
• casino dice problem and decoding algorithm
• probabilistic finite automata, Viterbi’s decoding algorithm
Chapter 15. Dynamic Programming

Will discuss the following examples:

- assembly-line scheduling
- casino dice problem and decoding algorithm
- probabilistic finite automata, Viterbi’s decoding algorithm
- matrix-chain multiplication
Chapter 15. Dynamic Programming

Will discuss the following examples:

- assembly-line scheduling
- casino dice problem and decoding algorithm
- probabilistic finite automata, Viterbi’s decoding algorithm
- matrix-chain multiplication
- longest common subsequence
Chapter 15. Dynamic Programming

Will discuss the following examples:

- assembly-line scheduling
- casino dice problem and decoding algorithm
- probabilistic finite automata, Viterbi’s decoding algorithm
- matrix-chain multiplication
- longest common subsequence
- molecular sequence comparison/alignment
Chapter 15. Dynamic Programming

Will discuss the following examples:

- assembly-line scheduling
- casino dice problem and decoding algorithm
- probabilistic finite automata, Viterbi’s decoding algorithm
- matrix-chain multiplication
- longest common subsequence
- molecular sequence comparison/alignment
- molecular structure prediction
Chapter 15. Dynamic Programming

Revisit the problem of computing the $n$th element of Fibonacci sequence
Chapter 15. Dynamic Programming

Revisit the problem of computing the \( n \)th element of Fibonacci sequence

\[
F(n) = F(n - 1) + F(n - 2), \ F(1) = F(2) = 1.
\]
Chapter 15. Dynamic Programming

Revisit the problem of computing the $n$th element of Fibonacci sequence

$$F(n) = F(n - 1) + F(n - 2), \quad F(1) = F(2) = 1.$$ 

- it divides problems into two subproblems
Revisit the problem of computing the $n$th element of Fibonacci sequence

$$F(n) = F(n-1) + F(n-2), \ F(1) = F(2) = 1.$$ 

- it divides problems into two subproblems
- there are overlapping subproblems
Chapter 15. Dynamic Programming

Revisit the problem of computing the $n$th element of Fibonacci sequence
\[ F(n) = F(n - 1) + F(n - 2), \quad F(1) = F(2) = 1. \]

- it divides problems into two subproblems
- there are overlapping subproblems
- direct top-down search would incur an exponential time
Chapter 15. Dynamic Programming

Revisit the problem of computing the \( n \)th element of Fibonacci sequence

\[ F(n) = F(n - 1) + F(n - 2), \ F(1) = F(2) = 1. \]

- it divides problems into two subproblems
- there are overlapping subproblems
- direct top-down search would incur an exponential time
- bottom-up computation is much faster
Revisit the problem of computing the $n$th element of Fibonacci sequence

$$F(n) = F(n-1) + F(n-2), \quad F(1) = F(2) = 1.$$ 

- it divides problems into two subproblems
- there are overlapping subproblems
- direct top-down search would incur an exponential time
- bottom-up computation is much faster

compute table $T[1..n]$ from left to right, by looking up values that have already been computed
Chapter 15. Dynamic Programming

Assembly-line scheduling

• \( S_{i,j} \): the \( j \)th station on line \( i \), \( i = 1, 2, \ldots, n \).
• \( a_{i,j} \) is the assembly time at station \( S_{i,j} \).
• \( t_{i,j} \) is the time for transferring lines from station \( S_{i,j} \).
• \( e_1 \) and \( e_2 \) are entering time costs.
• \( x_1 \) and \( x_2 \) are exiting time costs.

To determine the fastest way through a factory, there are \( 2^n \) possible ways.
Chapter 15. Dynamic Programming

Assembly-line scheduling

- $S_{i,j}$: the $j$th station on line $i$, $i = 1, 2$, $j = 1, 2, \cdots, n$. 

![Diagram of assembly-line scheduling](image-url)
Assembly-line scheduling

- $S_{i,j}$: the $j$th station on line $i$, $i = 1, 2$, $j = 1, 2, \cdots, n$.
- $a_{i,j}$ is the assembly time at station $S_{i,j}$
Assembly-line scheduling

- $S_{i,j}$: the $j$th station on line $i$, $i = 1, 2$, $j = 1, 2, \ldots, n$.
- $a_{i,j}$ is the assembly time at station $S_{i,j}$
- $t_{i,j}$ is time for transferring lines from station $S_{i,j}$
Chapter 15. Dynamic Programming

Assembly-line scheduling

- $S_{i,j}$: the $j$th station on line $i$, $i = 1, 2$, $j = 1, 2, \cdots, n$.
- $a_{i,j}$ is the assembly time at station $S_{i,j}$
- $t_{i,j}$ is time for transferring lines from station $S_{i,j}$
- $e_1$ and $e_2$ are entering time costs

To determine the fastest way through a factory there are $2^n$ possible ways.
Chapter 15. Dynamic Programming

Assembly-line scheduling

- $S_{i,j}$: the $j$th station on line $i$, $i = 1, 2, j = 1, 2, \ldots, n$.
- $a_{i,j}$ is the assembly time at station $S_{i,j}$
- $t_{i,j}$ is time for transferring lines from station $S_{i,j}$
- $e_1$ and $e_2$ are entering time costs
- $x_1$ and $x_2$ are exiting time costs

To determine the fastest way through a factory

There are $2^n$ possible ways.
Chapter 15. Dynamic Programming

Assembly-line scheduling

- \( S_{i,j} \): the \( j \)th station on line \( i \), \( i = 1, 2 \), \( j = 1, 2, \ldots, n \).
- \( a_{i,j} \) is the assembly time at station \( S_{i,j} \)
- \( t_{i,j} \) is time for transferring lines from station \( S_{i,j} \)
- \( e_1 \) and \( e_2 \) are entering time costs
- \( x_1 \) and \( x_2 \) are exiting time costs

To determine the fastest way through a factory
Chapter 15. Dynamic Programming

Assembly-line scheduling

- $S_{i,j}$: the $j$th station on line $i$, $i = 1, 2$, $j = 1, 2, \ldots, n$.
- $a_{i,j}$ is the assembly time at station $S_{i,j}$
- $t_{i,j}$ is time for transferring lines from station $S_{i,j}$
- $e_1$ and $e_2$ are entering time costs
- $x_1$ and $x_2$ are exiting time costs

To determine the fastest way through a factory

There are $2^n$ possible ways.
Chapter 15. Dynamic Programming

Analyzing the problem:
Chapter 15. Dynamic Programming

Analyzing the problem:

The problem is to determine the fastest way through $S_{1,n}$ or $S_{2,n}$.
Chapter 15. Dynamic Programming

Analyzing the problem:

The problem is to determine the fastest way through $S_{1,n}$ or $S_{2,n}$

For example, the fastest way through $S_{1,3}$ is faster between
Chapter 15. Dynamic Programming

Analyzing the problem:

The problem is to determine the fastest way through $S_{1,n}$ or $S_{2,n}$

For example, the fastest way through $S_{1,3}$ is faster between

- the fastest way through $S_{1,2}$ then $S_{1,2} \rightarrow S_{1,3}$,
Chapter 15. Dynamic Programming

Analyzing the problem:

The problem is to determine the fastest way through $S_{1,n}$ or $S_{2,n}$.

For example, the fastest way through $S_{1,3}$ is faster between

- the fastest way through $S_{1,2}$ then $S_{1,2} \rightarrow S_{1,3}$,
- the fastest way through $S_{2,2}$ then $S_{2,2} \rightarrow S_{1,3}$. 
Chapter 15. Dynamic Programming

Analyzing the problem:

The problem is to determine the fastest way through $S_{1,n}$ or $S_{2,n}$.

For example, the fastest way through $S_{1,3}$ is faster between

- the fastest way through $S_{1,2}$ then $S_{1,2} \rightarrow S_{1,3}$,
- the fastest way through $S_{2,2}$ then $S_{2,2} \rightarrow S_{1,3}$. 
Chapter 15. Dynamic Programming

For general $j, 1 \leq j \leq n$, the fastest way through $S_{1,j}$ is faster between
Chapter 15. Dynamic Programming

For general $j, 1 \leq j \leq n$, the fastest way through $S_{1,j}$ is faster between

- the fastest way through $S_{1,j-1}$ then $S_{1,j-1} \rightarrow S_{1,j}$,
For general $j, 1 \leq j \leq n$, the fastest way through $S_{1,j}$ is faster between

- the fastest way through $S_{1,j-1}$ then $S_{1,j-1} \rightarrow S_{1,j}$,
- the fastest way through $S_{2,j-1}$ then $S_{2,j-1} \rightarrow S_{1,j}$.

Likewise, the fastest way through $S_{2,j}$ is faster between

- the fastest way through $S_{1,j-1}$ then $S_{1,j-1} \rightarrow S_{2,j}$,
- the fastest way through $S_{2,j-1}$ then $S_{2,j-1} \rightarrow S_{2,j}$.
Chapter 15. Dynamic Programming

For general \( j, 1 \leq j \leq n \), the fastest way through \( S_{1,j} \) is faster between

- the fastest way through \( S_{1,j-1} \) then \( S_{1,j-1} \rightarrow S_{1,j} \),
- the fastest way through \( S_{2,j-1} \) then \( S_{2,j-1} \rightarrow S_{1,j} \).

Likewise, the fastest way through \( S_{2,j} \) is faster between
Chapter 15. Dynamic Programming

For general \( j, 1 \leq j \leq n \), the fastest way through \( S_{1,j} \) is faster between

- the fastest way through \( S_{1,j-1} \) then \( S_{1,j-1} \rightarrow S_{1,j} \),
- the fastest way through \( S_{2,j-1} \) then \( S_{2,j-1} \rightarrow S_{1,j} \).

Likewise, the fastest way through \( S_{2,j} \) is faster between

- the fastest way through \( S_{1,j-1} \) then \( S_{1,j-1} \rightarrow S_{2,j} \),
- the fastest way through \( S_{2,j-1} \) then \( S_{2,j-1} \rightarrow S_{2,j} \).
Chapter 15. Dynamic Programming

A dynamic programming approach:

1. Define the recursive solution to the problem:

   \( f_i(j) \) is the minimum time through station \( S_{i,j} \)
   for \( i = 1, 2 \) and \( j = 1, \ldots, n \).

2. The problem is to compute

   \[ \min \{ f_1(n) + x_1, f_2(n) + x_2 \} \]

   where

   \( f_1() \) and \( f_2() \) are defined recursively:

   \( f_1(j) = \min \{ f_1(j-1) + a_1, j, f_2(j-1) + t_2, j-1 + a_1, j \} \)

   \( f_2(j) = \min \{ f_2(j-1) + a_2, j, f_1(j-1) + t_1, j-1 + a_2, j \} \)

3. Base cases:

   \( f_1(1) = e_1 + a_1, 1 \) and \( f_2(1) = e_2 + a_2, 1 \)
Chapter 15. Dynamic Programming

A dynamic programming approach:

**step 1**: the above analysis
Chapter 15. Dynamic Programming

A dynamic programming approach:

**step 1**: the above analysis

**step 2**: formulate a recursive solution to the problem

- Define \( f_i(j) \) to be the minimum time through station \( S_{i,j} \), for \( i = 1, 2 \) and \( j = 1, \ldots, n \).

- The problem is to compute
  \[
  \min \{ f_1(n) + x_1, f_2(n) + x_2 \}
  \]

- Where \( f_1() \) and \( f_2() \) are defined recursively:
  \[
  f_1(j) = \min \{ f_1(j - 1) + a_{1,j}, f_2(j - 1) + t_{2,j-1} + a_{1,j} \}
  \]
  \[
  f_2(j) = \min \{ f_2(j - 1) + a_{2,j}, f_1(j - 1) + t_{1,j-1} + a_{2,j} \}
  \]

- Base cases:
  \( f_1(1) = e_1 + a_{1,1} \) and \( f_2(1) = e_2 + a_{2,1} \).
A dynamic programming approach:

**step 1**: the above analysis

**step 2**: formulate a recursive solution to the problem

define $f_i(j)$ to be the minimum time *through* station $S_{i,j}$, for $i = 1, 2$ and $j = 1, \ldots, n$. 
Chapter 15. Dynamic Programming

A dynamic programming approach:

**step 1**: the above analysis

**step 2**: formulate a recursive solution to the problem

- define \( f_i(j) \) to be the minimum time through station \( S_{i,j} \), for \( i = 1, 2 \) and \( j = 1, \ldots, n \).
- the problem is to compute \( \min\{f_1(n) + x_1, f_2(n) + x_2\} \)
Chapter 15. Dynamic Programming

A dynamic programming approach:

step 1: the above analysis

step 2: formulate a recursive solution to the problem

define $f_i(j)$ to be the minimum time through station $S_{i,j}$, for $i = 1, 2$ and $j = 1, \cdots, n$.

- the problem is to compute $\min\{f_1(n) + x_1, f_2(n) + x_2\}$

- where $f_1()$ and $f_2()$ are defined recursively:
A dynamic programming approach:

**step 1:** the above analysis

**step 2:** formulate a recursive solution to the problem

- define \( f_i(j) \) to be the minimum time through station \( S_{i,j} \), for \( i = 1, 2 \) and \( j = 1, \ldots, n \).

- the problem is to compute \( \min \{ f_1(n) + x_1, f_2(n) + x_2 \} \)

- where \( f_1() \) and \( f_2() \) are defined recursively:

\[
(1) f_1(j) = \min\{ f_1(j - 1) + a_{1,j}, f_2(j - 1) + t_{2,j-1} + a_{1,j} \}
\]
A dynamic programming approach:

**step 1**: the above analysis

**step 2**: formulate a recursive solution to the problem

define $f_i(j)$ to be the minimum time through station $S_{i,j}$, for $i = 1, 2$ and $j = 1, \cdots, n$.

- the problem is to compute $\min\{f_1(n) + x_1, f_2(n) + x_2\}$

- where $f_1()$ and $f_2()$ are defined recursively:

  (1) $f_1(j) = \min\{f_1(j - 1) + a_{1,j}, f_2(j - 1) + t_{2,j-1} + a_{1,j}\}$

  (2) $f_2(j) = \min\{f_2(j - 1) + a_{2,j}, f_1(j - 1) + t_{1,j-1} + a_{2,j}\}$
Chapter 15. Dynamic Programming

A dynamic programming approach:

**step 1**: the above analysis

**step 2**: formulate a recursive solution to the problem

define $f_i(j)$ to be the minimum time through station $S_{i,j}$, for $i = 1, 2$ and $j = 1, \cdots, n$.

- the problem is to compute $\min\{f_1(n) + x_1, f_2(n) + x_2\}$

- where $f_1()$ and $f_2()$ are defined recursively:

  1. $f_1(j) = \min\{f_1(j - 1) + a_{1,j}, f_2(j - 1) + t_{2,j-1} + a_{1,j}\}$
  2. $f_2(j) = \min\{f_2(j - 1) + a_{2,j}, f_1(j - 1) + t_{1,j-1} + a_{2,j}\}$

- base cases $f_1(1) = e_1 + a_{1,1}$ and $f_2(1) = e_2 + a_{2,1}$
Chapter 15. Dynamic Programming

Data dependency:
Chapter 15. Dynamic Programming

**step 3**: compute $f_1(j)$ and $f_2(j)$, for all $j = 1, 2, \ldots, n$. 
Chapter 15. Dynamic Programming

**step 3:** compute $f_1(j)$ and $f_2(j)$, for all $j = 1, 2, \ldots, n$.

to fill out a $2 \times n$ table, from left to right

\[
\begin{array}{cccccc}
  f_1(1) & f_1(2) & f_1(3) & \ldots & f_1(n) \\
  f_2(1) & f_2(2) & f_2(3) & \ldots & f_2(n)
\end{array}
\]

- using the recurrences to compute; and
Chapter 15. Dynamic Programming

**step 3**: compute $f_1(j)$ and $f_2(j)$, for all $j = 1, 2, \ldots, n$.

to fill out a $2 \times n$ table, from left to right

\begin{center}
\begin{tabular}{|c|c|c|c|c|c|}
\hline
$f_1(1)$ & $f_1(2)$ & $f_1(3)$ & $\ldots$ & $f_1(n)$ \\
\hline
$f_2(1)$ & $f_2(2)$ & $f_2(3)$ & $\ldots$ & $f_2(n)$ \\
\hline
\end{tabular}
\end{center}

• using the recurrences to compute; and
• looking up already-computed values, where
Chapter 15. Dynamic Programming

**step 3:** compute $f_1(j)$ and $f_2(j)$, for all $j = 1, 2, \ldots, n$.

to fill out a $2 \times n$ table, from left to right

\[
\begin{array}{cccccc}
  f_1(1) & f_1(2) & f_1(3) & \ldots & f_1(n) \\
  f_2(1) & f_2(2) & f_2(3) & \ldots & f_2(n) \\
\end{array}
\]

- using the recurrences to compute; and
- looking up already-computed values, where

  (0) base cases $f_1(1) = e_1 + a_{1,1}$ and $f_2(1) = e_2 + a_{2,1}$
Chapter 15. Dynamic Programming

**step 3**: compute $f_1(j)$ and $f_2(j)$, for all $j = 1, 2, \ldots, n$.

to fill out a $2 \times n$ table, from left to right

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1(1)$</td>
<td>$f_1(2)$</td>
<td>$f_1(3)$</td>
<td>$\ldots$</td>
<td>$f_1(n)$</td>
</tr>
<tr>
<td>$f_2(1)$</td>
<td>$f_2(2)$</td>
<td>$f_2(3)$</td>
<td>$\ldots$</td>
<td>$f_2(n)$</td>
</tr>
</tbody>
</table>

- using the recurrences to compute; and
- looking up already-computed values, where

(0) base cases $f_1(1) = e_1 + a_{1,1}$ and $f_2(1) = e_2 + a_{2,1}$

(1) $f_1(j) = \min\{f_1(j - 1) + a_{1,j}, f_2(j - 1) + t_{2,j-1} + a_{1,j}\}$
Chapter 15. Dynamic Programming

**step 3**: compute \( f_1(j) \) and \( f_2(j) \), for all \( j = 1, 2, \ldots, n \).

To fill out a \( 2 \times n \) table, from left to right

\[
\begin{array}{cccccccc}
  f_1(1) & f_1(2) & f_1(3) & \ldots & f_1(n) \\
  f_2(1) & f_2(2) & f_2(3) & \ldots & f_2(n) \\
\end{array}
\]

- using the recurrences to compute; and
- looking up already-computed values, where

(0) base cases \( f_1(1) = e_1 + a_{1,1} \) and \( f_2(1) = e_2 + a_{2,1} \)

(1) \( f_1(j) = \min\{f_1(j - 1) + a_{1,j}, f_2(j - 1) + t_{2,j-1} + a_{1,j}\} \)

(2) \( f_2(j) = \min\{f_2(j - 1) + a_{2,j}, f_1(j - 1) + t_{1,j-1} + a_{2,j}\} \)
step 3: compute $f_1(j)$ and $f_2(j)$, for all $j = 1, 2, \ldots, n$.

to fill out a $2 \times n$ table, from left to right

<table>
<thead>
<tr>
<th>$f_1(1)$</th>
<th>$f_1(2)$</th>
<th>$f_1(3)$</th>
<th>\ldots</th>
<th>$f_1(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_2(1)$</td>
<td>$f_2(2)$</td>
<td>$f_2(3)$</td>
<td>\ldots</td>
<td>$f_2(n)$</td>
</tr>
</tbody>
</table>

- using the recurrences to compute; and
- looking up already-computed values, where

(0) base cases $f_1(1) = e_1 + a_{1,1}$ and $f_2(1) = e_2 + a_{2,1}$
(1) $f_1(j) = \min\{f_1(j - 1) + a_{1,j}, f_2(j - 1) + t_{2,j-1} + a_{1,j}\}$
(2) $f_2(j) = \min\{f_2(j - 1) + a_{2,j}, f_1(j - 1) + t_{1,j-1} + a_{2,j}\}$

- the fastest time is $f^* = \min\{f_1(n) + x_1, f_2(n) + x_2\}$
Chapter 15. Dynamic Programming

Example
Chapter 15. Dynamic Programming

Example

\[
\begin{array}{ccccccc}
  j & 1 & 2 & 3 & 4 & 5 & 6 \\
  f_1 & 9 & 18 & 20 & 24 & 32 & 36 \\
  f_2 & 12 & 16 & 22 & 26 & 31 & 38 \\
\end{array}
\]

Min finish time
\[ f^* = 36 + 3 = 39 \]
Algorithm ASSEMBLYLINE\((a, t, e, x, n)\)
Algorithm \textsc{AssemblyLine}(a, t, e, x, n)
1. \hspace{1em} M[1, 1] = e_1 + a_{1,1}
Chapter 15. Dynamic Programming

Algorithm ASSEMBLYLINE(a, t, e, x, n)
1.   $M[1, 1] = e_1 + a_{1,1}$
2.   $M[2, 1] = e_2 + a_{2,1}$
Chapter 15. Dynamic Programming

Algorithm \texttt{ASSEMBLYLINE}(a, t, e, x, n)
1. \hspace{1em} M[1, 1] = e_1 + a_{1,1} \\
2. \hspace{1em} M[2, 1] = e_2 + a_{2,1} \\
3. \hspace{1em} \text{for j=2 to n}
Algorithm \textsc{AssembleLine}(a, t, e, x, n)
1. \quad M[1, 1] = e_1 + a_{1,1}
2. \quad M[2, 1] = e_2 + a_{2,1}
3. \quad \text{for } j=2 \text{ to } n
4. \quad \text{if } M[1, j-1] + a_{1,j} \leq M[2, j-1] + t_{2,j-1} + a_{1,j}
5. \quad \text{else}
6. \quad \text{return } (\min\{f_1(n) + x_1, f_2(n) + x_2\})
Algorithm ASSEMBLYLINE\((a, t, e, x, n)\)

1. \(M[1, 1] = e_1 + a_{1,1}\)
2. \(M[2, 1] = e_2 + a_{2,1}\)
3. \(\text{for } j=2 \text{ to } n\)
4. \(\text{if } M[1, j-1] + a_{1,j} \leq M[2, j-1] + t_{2,j-1} + a_{1,j}\)
5. \(M[1, j] = M[1, j-1] + a_{1,j}\)
Algorithm ASSEMBLYLINE(a, t, e, x, n)
1. $M[1, 1] = e_1 + a_{1,1}$
2. $M[2, 1] = e_2 + a_{2,1}$
3. for $j = 2$ to $n$
4. \quad if $M[1, j - 1] + a_{1,j} \leq M[2, j - 1] + t_{2,j-1} + a_{1,j}$
5. \quad $M[1, j] = M[1, j - 1] + a_{1,j}$
6. \quad $P[1, j] = 1$
7. \quad else
8. \quad $M[2, j] = M[2, j - 1] + t_{1,j-1} + a_{2,j}$
9. \quad $P[2, j] = 2$
10. return $(\min\{f_1(n) + x_1, f_2(n) + x_2\})$
Chapter 15. Dynamic Programming

Algorithm ASSEMBLYLINE\((a, t, e, x, n)\)

1. \(M[1, 1] = e_1 + a_{1,1}\)
2. \(M[2, 1] = e_2 + a_{2,1}\)
3. \textbf{for} \(j = 2\) \textbf{to} \(n\) \textbf{do}
4. \hspace{1em} \textbf{if} \(M[1, j - 1] + a_{1,j} \leq M[2, j - 1] + t_{2,j-1} + a_{1,j}\)
5. \hspace{1em} \(M[1, j] = M[1, j - 1] + a_{1,j}\)
6. \hspace{1em} \(P[1, j] = 1\)
7. \hspace{1em} \textbf{else} \(M[1, j] = M[2, j - 1] + t_{2,j-1} + a_{1,j}\)

\textbf{return} \(\min\{f_1(n) + x_1, f_2(n) + x_2\}\)
Algorithm \textsc{AssemblyLine}(a, t, e, x, n)

1. \(M[1, 1] = e_1 + a_{1,1}\)
2. \(M[2, 1] = e_2 + a_{2,1}\)
3. \textbf{for} \(j = 2\) \textbf{to} \(n\)
4. \quad \textbf{if} \(M[1, j - 1] + a_{1,j} \leq M[2, j - 1] + t_{2,j-1} + a_{1,j}\)
5. \quad \quad \(M[1, j] = M[1, j - 1] + a_{1,j}\)
6. \quad \quad \(P[1, j] = 1\)
7. \quad \textbf{else} \(M[1, j] = M[2, j - 1] + t_{2,j-1} + a_{1,j}\)
8. \quad \quad \(P[1, j] = 2\)
9. \textbf{return} \((\min\{f_1(n) + x_1, f_2(n) + x_2\})\)
Algorithm ASSEMBLYLINE\((a, t, e, x, n)\)
1. \(M[1, 1] = e_1 + a_{1,1}\)
2. \(M[2, 1] = e_2 + a_{2,1}\)
3. for \(j=2\) to \(n\)
4. if \(M[1, j - 1] + a_{1,j} \leq M[2, j - 1] + t_{2,j-1} + a_{1,j}\)
5. \(M[1, j] = M[1, j - 1] + a_{1,j}\)
6. \(P[1, j] = 1\)
7. else \(M[1, j] = M[2, j - 1] + t_{2,j-1} + a_{1,j}\)
8. \(P[1, j] = 2\)
9. if \(M[2, j - 1] + a_{2,j} \leq M[1, j - 1] + t_{1,j-1} + a_{2,j}\)
Algorithm `ASSEMBLYLINE(a, t, e, x, n)`

1. $M[1, 1] = e_1 + a_{1,1}$
2. $M[2, 1] = e_2 + a_{2,1}$
3. for $j = 2$ to $n$
   4. if $M[1, j - 1] + a_{1,j} \leq M[2, j - 1] + t_{2,j-1} + a_{1,j}$
   5. $M[1, j] = M[1, j - 1] + a_{1,j}$
   6. $P[1, j] = 1$
   7. else $M[1, j] = M[2, j - 1] + t_{2,j-1} + a_{1,j}$
   8. $P[1, j] = 2$
9. if $M[2, j - 1] + a_{2,j} \leq M[1, j - 1] + t_{1,j-1} + a_{2,j}$
Chapter 15. Dynamic Programming

Algorithm \textsc{AssemblyLine}(a, t, e, x, n)

1. \[ M[1, 1] = e_1 + a_{1, 1} \]
2. \[ M[2, 1] = e_2 + a_{2, 1} \]
3. \textbf{for} \ j=2 \ \textbf{to} \ n
4. \textbf{if} \ M[1, j-1] + a_{1,j} \leq M[2, j-1] + t_{2,j-1} + a_{1,j}
5. \[ M[1, j] = M[1, j-1] + a_{1,j} \]
6. \[ P[1, j] = 1 \]
7. \textbf{else} \[ M[1, j] = M[2, j-1] + t_{2,j-1} + a_{1,j} \]
8. \[ P[1, j] = 2 \]
9. \textbf{if} \ M[2, j-1] + a_{2,j} \leq M[1, j-1] + t_{1,j-1} + a_{2,j}
10. \[ M[2, j] = M[2, j-1] + a_{2,j} \]
11. \[ P[2, j] = 2 \]
12. \quad \textbf{return} \ \{ \min \{ f_1(n) + x_1, f_2(n) + x_2 \} \}
Chapter 15. Dynamic Programming

Algorithm `ASSEMBLYLINE(a, t, e, x, n)`
1. $M[1, 1] = e_1 + a_{1,1}$
2. $M[2, 1] = e_2 + a_{2,1}$
3. for $j=2$ to $n$
4. \hspace{1em} if $M[1, j - 1] + a_{1,j} \leq M[2, j - 1] + t_{2,j-1} + a_{1,j}$
5. \hspace{2em} $M[1, j] = M[1, j - 1] + a_{1,j}$
6. \hspace{2em} $P[1, j] = 1$
7. \hspace{1em} else $M[1, j] = M[2, j - 1] + t_{2,j-1} + a_{1,j}$
8. \hspace{2em} $P[1, j] = 2$
9. \hspace{1em} if $M[2, j - 1] + a_{2,j} \leq M[1, j - 1] + t_{1,j-1} + a_{2,j}$
10. \hspace{2em} $M[2, j] = M[2, j - 1] + a_{2,j}$
11. \hspace{2em} $P[2, j] = 2$
12. \hspace{1em} else $M[2, j] = M[1, j - 1] + t_{1,j-1} + a_{2,j}$
13. return $(\min\{f_1(n) + x_1, f_2(n) + x_2\})$
Chapter 15. Dynamic Programming

Algorithm ASSEMBLYLINE\((a, t, e, x, n)\)
1. \(M[1, 1] = e_1 + a_{1,1}\)
2. \(M[2, 1] = e_2 + a_{2,1}\)
3. for \(j=2\) to \(n\)
4. \(\text{if } M[1, j - 1] + a_{1,j} \leq M[2, j - 1] + t_{2,j-1} + a_{1,j}\)
5. \(M[1, j] = M[1, j - 1] + a_{1,j}\)
6. \(P[1, j] = 1\)
7. \(\text{else } M[1, j] = M[2, j - 1] + t_{2,j-1} + a_{1,j}\)
8. \(P[1, j] = 2\)
9. \(\text{if } M[2, j - 1] + a_{2,j} \leq M[1, j - 1] + t_{1,j-1} + a_{2,j}\)
10. \(M[2, j] = M[2, j - 1] + a_{2,j}\)
11. \(P[2, j] = 2\)
12. \(\text{else } M[2, j] = M[1, j - 1] + t_{1,j-1} + a_{2,j}\)
13. \(P[2, j] = 1\)

\[\text{return } (\min\{f_1(n) + x_1, f_2(n) + x_2\})\]
Algorithm ASSEMBLYLINE(a, t, e, x, n)
1. \[ M[1, 1] = e_1 + a_{1,1} \]
2. \[ M[2, 1] = e_2 + a_{2,1} \]
3. for \( j=2 \) to \( n \)
4. \[ \text{if } M[1, j-1] + a_{1,j} \leq M[2, j-1] + t_{2,j-1} + a_{1,j} \]
5. \[ M[1, j] = M[1, j-1] + a_{1,j} \]
6. \[ P[1, j] = 1 \]
7. \[ \text{else } M[1, j] = M[2, j-1] + t_{2,j-1} + a_{1,j} \]
8. \[ P[1, j] = 2 \]
9. \[ \text{if } M[2, j-1] + a_{2,j} \leq M[1, j-1] + t_{1,j-1} + a_{2,j} \]
10. \[ M[2, j] = M[2, j-1] + a_{2,j} \]
11. \[ P[2, j] = 2 \]
12. \[ \text{else } M[2, j] = M[1, j-1] + t_{1,j-1} + a_{2,j} \]
13. \[ P[2, j] = 1 \]
14. return \( (\min\{f_1(n) + x_1, f_2(n) + x_2\}) \)
step 4: get the fastest path, not just the faster time.
Chapter 15. Dynamic Programming

**step 4:** get the fastest path, not just the faster time.

Do we know the fastest path from the fastest time $f^*$?
Chapter 15. Dynamic Programming

**step 4**: get the fastest path, not just the faster time.

Do we know the fastest path from the fastest time $f^*$?

<table>
<thead>
<tr>
<th>$f_1(1)$</th>
<th>$f_1(2)$</th>
<th>$f_1(3)$</th>
<th>...</th>
<th>$f_1(n - 1)$</th>
<th>$f_1(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_2(1)$</td>
<td>$f_2(2)$</td>
<td>$f_2(3)$</td>
<td>...</td>
<td>$f_2(n - 1)$</td>
<td>$f_2(n)$</td>
</tr>
</tbody>
</table>
**step 4**: get the fastest path, not just the faster time.

Do we know the fastest path from the fastest time \( f^* \)?

<table>
<thead>
<tr>
<th>( f_1(1) )</th>
<th>( f_1(2) )</th>
<th>( f_1(3) )</th>
<th>( \ldots )</th>
<th>( f_1(n - 1) )</th>
<th>( f_1(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f_2(1) )</td>
<td>( f_2(2) )</td>
<td>( f_2(3) )</td>
<td>( \ldots )</td>
<td>( f_2(n - 1) )</td>
<td>( f_2(n) )</td>
</tr>
</tbody>
</table>

\[
f^* = \min\{f_1(n) + x_1, f_2(n) + x_2\}
\]

\[
f_1(j) = \min\{f_1(j - 1) + a_{1,j}, f_2(j - 1) + t_{2,j-1} + a_{1,j}\}
\]

\[
f_2(j) = \min\{f_2(j - 1) + a_{2,j}, f_1(j - 1) + t_{1,j-1} + a_{2,j}\}
\]
Algorithm PRINTPATHMAIN($M, P, f^*, x$)
1. \textbf{if} ($f^* - x_1$) = $M[1, n]$

2. $i = 1$
3. $else$
4. $i = 2$
5. $PrintPath(P, i, n$)

Algorithm PRINTPATH($P, i, j$)
1. \textbf{if} $j \geq 1$
2. $PrintPath(P, P[j], j - 1$)
3. \textbf{print} ($i, j$)
Algorithm PRINTPATHMAIN($M, P, f^*, x$)
1. \textbf{if} ($f^* - x_1) = M[1, n]$
2. \hspace{1em} $i = 1$
Algorithm \textsc{PrintPathMain}(M, P, f^*, x)
1. \textbf{if} \ (f^* - x_1) = M[1, n]
2. \hspace{1em} i = 1
3. \textbf{else} \ i = 2
Chapter 15. Dynamic Programming

Algorithm \textsc{PrintPathMain}(M, P, f^*, x)
1. \textbf{if} \ (f^* - x_1) = M[1,n]
2. \hspace{1em} i = 1
3. \textbf{else} \ i = 2
4. \textbf{PrintPath} \ (P, i, n)
Algorithm PrintPathMain\((M, P, f^*, x)\)
1. \(\textbf{if } (f^* - x_1) = M[1, n]\)
2. \(i = 1\)
3. \(\textbf{else } i = 2\)
4. \textbf{PrintPath}(P, i, n)

Algorithm PrintPath\((P, i, j)\)
1. \(\textbf{if } j \geq 1\)
Algorithm \textbf{PrintPathMain}(M, P, f^*, x)
1. \textbf{if} \ (f^* - x_1) = M[1, n]
2. \hspace{0.5cm} i = 1
3. \textbf{else} \ i = 2
4. \textbf{PrintPath} \ (P, i, n)

Algorithm \textbf{PrintPath} \ (P, i, j)
1. \textbf{if} \ j \geq 1
2. \hspace{0.5cm} \textbf{PrintPath} \ (P, P[i, j], j - 1)
Algorithm PrintPathMain\((M, P, f^*, x)\)
1.  \textbf{if} \ (f^* - x_1) = M[1, n]
2.  \hspace{1em} i = 1
3.  \hspace{2em} \textbf{else} \ i = 2
4.  \hspace{3em} \textbf{PrintPath} \ (P, i, n)

Algorithm \textbf{PrintPath} \ (P, i, j)
1.  \hspace{1em} \textbf{if} \ j \geq 1
2.  \hspace{2em} \textbf{PrintPath} \ (P, P[i, j], j - 1)
3.  \hspace{3em} \textbf{print} \ (i, j)
Summary on steps for dynamic programming solving the assembly line problem
Chapter 15. Dynamic Programming

Summary on steps for dynamic programming solving the assembly line problem

(1) the structure of optimal solution
Chapter 15. Dynamic Programming

Summary on steps for dynamic programming solving the assembly line problem

1. the structure of optimal solution
2. defining optimal cost recursively
Summary on steps for dynamic programming solving the assembly line problem

(1) the structure of optimal solution
(2) defining optimal cost recursively
(3) computing optimal cost (bottom-up approach)
Summary on steps for dynamic programming solving the assembly line problem

(1) the structure of optimal solution
(2) defining optimal cost recursively
(3) computing optimal cost (bottom-up approach)
(4) constructing optimal solution (traceback)
Chapter 15. Dynamic Programming

We consider a little more about the assembly line problem.

• can we still compute the fastest time? yes.
  so can the fastest path and the sequence of parts be computed!
• Given a sequence of parts produced, can we know the path? No, but we may predict such a path with a confidence.
We consider a little more about the assembly line problem.

Assume that each station $S_{i,j}$ produces a part $p_k$ with time cost $\tau_{i,j,k}$, where $k = 1, 2, 3$. 
We consider a little more about the assembly line problem.

Assume that each station $S_{i,j}$ produces a part $p_k$ with time cost $\tau_{i,j,k}$, where $k = 1, 2, 3$.

- can we still compute the fastest time?
Chapter 15. Dynamic Programming

We consider a little more about the assembly line problem. Assume that each station $S_{i,j}$ produces a part $p_k$ with time cost $\tau_{i,j,k}$, where $k = 1, 2, 3$.

- can we still compute the fastest time? yes.
Chapter 15. Dynamic Programming

We consider a little more about the assembly line problem.

Assume that each station $S_{i,j}$ produces a part $p_k$ with time cost $\tau_{i,j,k}$, where $k = 1, 2, 3$.

- can we still compute the fastest time? yes.

  so can the fastest path and the sequence of parts be computed!
We consider a little more about the assembly line problem.

Assume that each station $S_{i,j}$ produces a part $p_k$ with time cost $\tau_{i,j,k}$, where $k = 1, 2, 3$.

- can we still compute the fastest time? yes.
  so can the fastest path and the sequence of parts be computed!

- Given a sequence of parts produced, can we know the path?
Chapter 15. Dynamic Programming

We consider a little more about the assembly line problem. Assume that each station $S_{i,j}$ produces a part $p_k$ with time cost $\tau_{i,j,k}$, where $k = 1, 2, 3$.

- can we still compute the fastest time? yes.
  so can the fastest path and the sequence of parts be computed!

- Given a sequence of parts produced, can we know the path?
  No, but we may predict such a path with a confidence.
Chapter 15. Dynamic Programming

Example-2. Casino dice and decoding algorithm

The house rolls die \( n \) times: e.g., 5, 3, 2, 5, 6, 6, 1, ...

- not honest, switch between dice: fair (F) and loaded (L)
- a small chance to switch is small, a large chance to stay on the same die

Given the sequence of numbers, e.g., 5, 3, 2, 5, 6, 6, 1, what is the sequence of dices used most likely?
Chapter 15. Dynamic Programming

Example-2. Casino dice and decoding algorithm

The house rolls die $n$ times: e.g., 5, 3, 2, 5, 6, 6, 1, ... , 1
Chapter 15. Dynamic Programming

Example-2. Casino dice and decoding algorithm

The house rolls die $n$ times: e.g., 5, 3, 2, 5, 6, 6, 1, ... , 1

- not honest, switch between dice: fair (F) and loaded (L)
Chapter 15. Dynamic Programming

Example-2. Casino dice and decoding algorithm

The house rolls die $n$ times: e.g., 5, 3, 2, 5, 6, 6, 1, . . . , 1

- not honest, switch between dice: fair (F) and loaded (L)
- a small chance to switch is small, a large chance to stay on the same die
Chapter 15. Dynamic Programming

Example-2. Casino dice and decoding algorithm

The house rolls die \( n \) times: e.g., 5, 3, 2, 5, 6, 6, 1, \ldots, 1

- not honest, switch between dice: fair (F) and loaded (L)
- a small chance to switch is small, a large chance to stay on the same die

Given the sequence of numbers, e.g., 5, 3, 2, 5, 6, 6, 1, \ldots, 1, what is the sequence of dices used
Chapter 15. Dynamic Programming

Example-2. Casino dice and decoding algorithm

The house rolls die \( n \) times: e.g., 5, 3, 2, 5, 6, 6, 1, \ldots, 1

- not honest, switch between dice: fair (F) and loaded (L)

- a small chance to switch is small, a large chance to stay on the same die

Given the sequence of numbers, e.g., 5, 3, 2, 5, 6, 6, 1, \ldots, 1, what is the sequence of dices used most likely?
We compute the most probable path of dices to roll a given sequence of numbers $S = d_1 \ldots d_n$, where $d_i \in \{1, 2, \ldots, 6\}$.
Chapter 15. Dynamic Programming

We compute the most probable path of dices to roll a given sequence of numbers $S = d_1 \ldots d_n$, where $d_i \in \{1, 2, \ldots, 6\}$.
Chapter 15. Dynamic Programming

We compute the most probable path of dices to roll a given sequence of numbers $S = d_1 \ldots d_n$, where $d_i \in \{1, 2, \ldots, 6\}$

i.e., we compute the highest probability of a path of dice to roll $d_1 \ldots d_n$
We compute the most probable path of dices to roll a given sequence of numbers $S = d_1 \ldots d_n$, where $d_i \in \{1, 2, \ldots, 6\}$

i.e., we compute the highest probability of a path of dice to roll $d_1 \ldots d_n$

As in AssemblyLine problem, we compute the most probable path to roll $d_1 \ldots d_i$:
Chapter 15. Dynamic Programming

We compute the most probable path of dices to roll a given sequence of numbers $S = d_1 \ldots d_n$, where $d_i \in \{1, 2, \ldots, 6\}$

i.e., we compute the highest probability of a path of dice to roll $d_1 \ldots d_n$

As in AssemblyLine problem, we compute the most probable path to roll $d_1 \ldots d_i$:

- $p(S, i, F)$ for which the fair die $F$ is used in the last step
- $p(S, i, L)$ for which the loaded die $L$ is used in the last step
Chapter 15. Dynamic Programming

Both $p(S, i, F)$ and $p(S, i, L)$ have recursive solutions.
Chapter 15. Dynamic Programming

Both $p(S, i, F)$ and $p(S, i, L)$ have recursive solutions.

$$p(S, i, F) = \max\{p(S, i - 1, F) \times 0.95 \times \frac{1}{6}, p(S, i - 1, L) \times 0.1 \times \frac{1}{6}\}$$
Chapter 15. Dynamic Programming

Both \( p(S, i, F) \) and \( p(S, i, L) \) have recursive solutions.

\[
p(S, i, F) = \max \{ p(S, i - 1, F) \times 0.95 \times \frac{1}{6}, \ p(S, i - 1, L) \times 0.1 \times \frac{1}{6} \}
\]

\[
p(S, i, L) = \max \{ p(S, i - 1, L) \times 0.9 \times q, \ p(S, i - 1, F) \times 0.05 \times q \}
\]
Chapter 15. Dynamic Programming

Both $p(S, i, F)$ and $p(S, i, L)$ have recursive solutions.

$$p(S, i, F) = \max \{p(S, i - 1, F) \times 0.95 \times \frac{1}{6}, \ p(S, i - 1, L) \times 0.1 \times \frac{1}{6} \}$$

$$p(S, i, L) = \max \{p(S, i - 1, L) \times 0.9 \times q, \ p(S, i - 1, F) \times 0.05 \times q \}$$

where $q = \frac{1}{10}$ if $1 \leq d_i \leq 5$; $q = \frac{1}{2}$ if $d_i = 6$
Both \( p(S, i, F) \) and \( p(S, i, L) \) have recursive solutions.

\[
p(S, i, F) = \max\{p(S, i - 1, F) \times 0.95 \times \frac{1}{6}, \ p(S, i - 1, L) \times 0.1 \times \frac{1}{6}\}
\]

\[
p(S, i, L) = \max\{p(S, i - 1, L) \times 0.9 \times q, \ p(S, i - 1, F) \times 0.05 \times q\}
\]

where \( q = \frac{1}{10} \) if \( 1 \leq d_i \leq 5; \ q = \frac{1}{2} \) if \( d_i = 6 \)

Base cases: when \( i = \)
Both $p(S, i, F)$ and $p(S, i, L)$ have recursive solutions.

$$p(S, i, F) = \max\{p(S, i - 1, F) \times 0.95 \times \frac{1}{6}, \ p(S, i - 1, L) \times 0.1 \times \frac{1}{6}\}$$

$$p(S, i, L) = \max\{p(S, i - 1, L) \times 0.9 \times q, \ p(S, i - 1, F) \times 0.05 \times q\}$$

where $q = \frac{1}{10}$ if $1 \leq d_i \leq 5$; $q = \frac{1}{2}$ if $d_i = 6$

Base cases: when $i = 1$
Both \( p(S, i, F) \) and \( p(S, i, L) \) have recursive solutions.

\[
p(S, i, F) = \max \{ p(S, i - 1, F) \times 0.95 \times \frac{1}{6}, \ p(S, i - 1, L) \times 0.1 \times \frac{1}{6} \}
\]

\[
p(S, i, L) = \max \{ p(S, i - 1, L) \times 0.9 \times q, \ p(S, i - 1, F) \times 0.05 \times q \}
\]

where \( q = \frac{1}{10} \) if \( 1 \leq d_i \leq 5 \); \( q = \frac{1}{2} \) if \( d_i = 6 \)

Base cases: when \( i = 1 \)

\[
p(S, 1, F) = 0.5 \times \frac{1}{6}
\]

\[
p(S, 1, L) = 0.5 \times q
\]
Both $p(S, i, F)$ and $p(S, i, L)$ have recursive solutions.

$$p(S, i, F) = \max\{p(S, i - 1, F) \times 0.95 \times \frac{1}{6}, p(S, i - 1, L) \times 0.1 \times \frac{1}{6}\}$$

$$p(S, i, L) = \max\{p(S, i - 1, L) \times 0.9 \times q, p(S, i - 1, F) \times 0.05 \times q\}$$

where $q = \frac{1}{10}$ if $1 \leq d_i \leq 5$; $q = \frac{1}{2}$ if $d_i = 6$

Base cases: when $i = 1$

$$p(S, 1, F) = 0.5 \times \frac{1}{6}$$

$$p(S, 1, L) = 0.5 \times q$$

where $q = \frac{1}{10}$ if $1 \leq d_1 \leq 5$; $q = \frac{1}{2}$ if $d_1 = 6$
Chapter 15. Dynamic Programming

Both \( p(S, i, F) \) and \( p(S, i, L) \) have recursive solutions.

\[
p(S, i, F) = \max\{p(S, i - 1, F) \times 0.95 \times \frac{1}{6}, p(S, i - 1, L) \times 0.1 \times \frac{1}{6}\}
\]

\[
p(S, i, L) = \max\{p(S, i - 1, L) \times 0.9 \times q, p(S, i - 1, F) \times 0.05 \times q\}
\]

where \( q = \frac{1}{10} \) if \( 1 \leq d_i \leq 5; q = \frac{1}{2} \) if \( d_i = 6 \)

Base cases: when \( i = 1 \)

\[
p(S, 1, F) = 0.5 \times \frac{1}{6}
\]

\[
p(S, 1, L) = 0.5 \times q
\]

where \( q = \frac{1}{10} \) if \( 1 \leq d_1 \leq 5; q = \frac{1}{2} \) if \( d_1 = 6 \)

- design dynamic programming algorithm to compute \( \max\{P(S, n, F), P(S, n, L)\} \) given from \( d_1d_2 \ldots d_n \)
Both \( p(S, i, F) \) and \( p(S, i, L) \) have recursive solutions.

\[
p(S, i, F) = \max \{p(S, i - 1, F) \times 0.95 \times \frac{1}{6}, \ p(S, i - 1, L) \times 0.1 \times \frac{1}{6}\}
\]

\[
p(S, i, L) = \max \{p(S, i - 1, L) \times 0.9 \times q, \ p(S, i - 1, F) \times 0.05 \times q\}
\]

where \( q = \frac{1}{10} \) if \( 1 \leq d_i \leq 5 \); \( q = \frac{1}{2} \) if \( d_i = 6 \)

Base cases: when \( i = 1 \)

\[
p(S, 1, F) = 0.5 \times \frac{1}{6}
\]

\[
p(S, 1, L) = 0.5 \times q
\]

where \( q = \frac{1}{10} \) if \( 1 \leq d_1 \leq 5 \); \( q = \frac{1}{2} \) if \( d_1 = 6 \)

- design dynamic programming algorithm to compute \( \max \{P(S, n, F), P(S, n, L)\} \)
  given from \( d_1 d_2 \ldots d_n \)
- design a traceback process to print out the sequence of dices used.
Chapter 15. Dynamic Programming

Consider a sequence

... 1 3 2 4 6 6 6 4 1 6 5 6 6 2 4 2 1 2 3 5 ...

By computing the maximum probability to produce the sequence, we decode the dices used...

But besides the casino interest, is such algorithm meaningful in other applications? The answer is definitely YES!
Consider a sequence

... 1 3 2 4 6 6 6 4 1 6 5 6 6 2 4 2 1 2 3 5 ...
Consider a sequence

\[ \ldots 1 \ 3 \ 2 \ 4 \ 6 \ 6 \ 6 \ 4 \ 1 \ 6 \ 5 \ 6 \ 6 \ 2 \ 4 \ 2 \ 1 \ 2 \ 3 \ 5 \ldots \]

By computing the maximum probability to produce the sequence, we decode the dices used

\[ \ldots 1 \ 3 \ 2 \ 4 \ 6 \ 6 \ 6 \ 4 \ 1 \ 6 \ 5 \ 6 \ 6 \ 2 \ 4 \ 2 \ 1 \ 2 \ 3 \ 5 \ldots \]
Chapter 15. Dynamic Programming

Consider a sequence

\[ ... \quad 1 \quad 3 \quad 2 \quad 4 \quad 6 \quad 6 \quad 6 \quad 4 \quad 1 \quad 6 \quad 5 \quad 6 \quad 6 \quad 2 \quad 4 \quad 2 \quad 1 \quad 2 \quad 3 \quad 5 \ldots \]

By computing the maximum probability to produce the sequence, we decode the dices used

\[ ... \quad 1 \quad 3 \quad 2 \quad 4 \quad 6 \quad 6 \quad 6 \quad 4 \quad 1 \quad 6 \quad 5 \quad 6 \quad 6 \quad 2 \quad 4 \quad 2 \quad 1 \quad 2 \quad 3 \quad 5 \ldots \]

\[ ... \quad F \quad F \quad F \quad F \quad L \quad L \quad L \quad L \quad L \quad L \quad L \quad L \quad F \quad F \quad F \quad F \quad F \quad F \quad F \ldots \]
Chapter 15. Dynamic Programming

Consider a sequence

\[
\cdots 1 3 2 4 6 6 6 4 1 6 5 6 6 2 4 2 1 2 3 5 \cdots
\]

By computing the maximum probability to produce the sequence, we decode the dices used

\[
\cdots F F F F L L L L L L L L F F F F F F F \cdots
\]

But besides the casino interest, is such algorithm meaningful in other applications?
Consider a sequence

\[ \ldots \ 1 \ 3 \ 2 \ 4 \ 6 \ 6 \ 6 \ 6 \ 4 \ 1 \ 6 \ 5 \ 6 \ 6 \ 2 \ 4 \ 2 \ 1 \ 2 \ 3 \ 5 \ldots \]

By computing the maximum probability to produce the sequence, we decode the dices used

\[ \ldots \ 1 \ 3 \ 2 \ 4 \ 6 \ 6 \ 6 \ 6 \ 4 \ 1 \ 6 \ 5 \ 6 \ 6 \ 2 \ 4 \ 2 \ 1 \ 2 \ 3 \ 5 \ldots \]
\[ \ldots \ \text{F \ F \ F \ F \ L \ L \ L \ L \ L \ L \ L \ L \ F \ F \ F \ F \ F \ F \ F} \ldots \]

But besides the casino interest, is such algorithm meaningful in other applications?

The answer is definitely YES!
a segment of DNA sequence, is it meaningful?
### Chapter 15. Dynamic Programming

<table>
<thead>
<tr>
<th>AGGACCATAAAAAACTCAGTCAGTGAA</th>
<th>AAAACAAGTTATAAAAACCTTTCA</th>
</tr>
</thead>
<tbody>
<tr>
<td>TGTTCTGGCATCGATGAAGAAGAAGCGAG</td>
<td>GTAATGTGAATTCGAGGACTCAGTGAA</td>
</tr>
<tr>
<td>GAACGCACTTTTGCGCCCTTGGTATTTC</td>
<td>TGTTCGAGCGTCATTACCAACCCCTCAAG</td>
</tr>
<tr>
<td>TGGGCTCCTCCCTCCTCCACGGACGCCTT</td>
<td>GGTGCGCTCTTGCCCTCAAGCGTAGTACG</td>
</tr>
<tr>
<td>TTGGAGCGCACCGCGTCGCACCACCGGAA</td>
<td>TATTTCTCAAGGTTGACCTCGGATCAT</td>
</tr>
<tr>
<td>AAGGTAAGAACGATTTTCCTTCCCGCTG</td>
<td>CTGGGTGCTGGGTGTGGGTGGGTGGGTGGGT</td>
</tr>
<tr>
<td>TTGGCTTATCGCTTCCGGTGAGGGGACAT</td>
<td>TGGCCCGCGCTTAAGCTCGTGTCGGG</td>
</tr>
</tbody>
</table>
| CGCATCTGGGTTTTTTCGAGCGGCGT |}

If the highlighted segment is not random, it may be a gene.
Chapter 15. Dynamic Programming

Technically, the “linear model” is unfolded of a “more condensed model”.

A hidden Markov model (HMM)
Chapter 15. Dynamic Programming
Chapter 15. Dynamic Programming
Chapter 15. Dynamic Programming
Chapter 15. Dynamic Programming
Knapsack Problem

**Input**
- \( n \) items of sizes \( s_1, \ldots, s_n \) and values \( v_1, \ldots, v_n \);
- and a knapsack size \( B \).

**Output**
- A subset \( A \subseteq \{1, 2, \ldots, n\} \) that maximizes \( \sum_{i \in A} v_i \) with constraint \( \sum_{i \in A} s_i \leq B \).

**Analysis:**
For \( n \) items and knapsack size \( B \);
- examine the last item \( n \), then either put item \( n \) into the knapsack or discard it;
- That is:
  1. either increase value by \( v_n \), reduce available space from \( B \) to \( B - s_n \), reduce the number of items by 1;
  2. or simply reduce the number of items by 1.
- Both situations reduce the problem size to a subproblem.
Knapsack Problem

**INPUT:** $n$ items of sizes $s_1, \ldots, s_n$ and values $v_1, \ldots, v_n$; and a knapsack size $B$
Chapter 15. Dynamic Programming

Knapsack Problem

**INPUT:** \( n \) items of sizes \( s_1, \ldots, s_n \) and values \( v_1, \ldots, v_n \); and a knapsack size \( B \)

**OUTPUT:** a subset \( A \subseteq \{1, 2, \ldots, n\} \) that maximizes \( \sum_{i \in A} v_i \)
Chapter 15. Dynamic Programming

Knapsack Problem

**Input:** \( n \) items of sizes \( s_1, \ldots, s_n \) and values \( v_1, \ldots, v_n \); and a knapsack size \( B \)

**Output:** a subset \( A \subseteq \{1, 2, \ldots, n\} \) that maximizes \( \sum_{i \in A} v_i \)

with constraint \( \sum_{i \in A} s_i \leq B \)
Knapsack Problem

**INPUT:** $n$ items of sizes $s_1, \ldots, s_n$ and values $v_1, \ldots, v_n$; and a knapsack size $B$

**OUTPUT:** a subset $A \subseteq \{1, 2, \ldots, n\}$ that maximizes $\sum_{i \in A} v_i$

with constraint $\sum_{i \in A} s_i \leq B$

Analysis:
Chapter 15. Dynamic Programming

Knapsack Problem

**INPUT:** $n$ items of sizes $s_1, \ldots, s_n$ and values $v_1, \ldots, v_n$; and a knapsack size $B$

**OUTPUT:** a subset $A \subseteq \{1, 2, \ldots, n\}$ that maximizes $\sum_{i \in A} v_i$

with constraint $\sum_{i \in A} s_i \leq B$

Analysis:

For $n$ items and knapsack size $B$; examine the last item $n$, then
Chapter 15. Dynamic Programming

Knapsack Problem

**INPUT:** $n$ items of sizes $s_1, \ldots, s_n$ and values $v_1, \ldots, v_n$; and a knapsack size $B$

**OUTPUT:** a subset $A \subseteq \{1, 2, \ldots, n\}$ that maximizes $\sum_{i \in A} v_i$

with constraint $\sum_{i \in A} s_i \leq B$

Analysis:

For $n$ items and knapsack size $B$; examine the last item $n$, then

either put item $n$ into the knapsack or discard it;
**Chapter 15. Dynamic Programming**

---

**Knapsack Problem**

**Input:** \( n \) items of sizes \( s_1, \ldots, s_n \) and values \( v_1, \ldots, v_n \); and a knapsack size \( B \).

**Output:** a subset \( A \subseteq \{1, 2, \ldots, n\} \) that maximizes \( \sum_{i \in A} v_i \) with constraint \( \sum_{i \in A} s_i \leq B \).

**Analysis:**

For \( n \) items and knapsack size \( B \); examine the last item \( n \), then either put item \( n \) into the knapsack or discard it; That is
Chapter 15. Dynamic Programming

Knapsack Problem

**Input:** \( n \) items of sizes \( s_1, \ldots, s_n \) and values \( v_1, \ldots, v_n \); and a knapsack size \( B \)

**Output:** a subset \( A \subseteq \{1, 2, \ldots, n\} \) that maximizes \( \sum_{i \in A} v_i \)
with constraint \( \sum_{i \in A} s_i \leq B \)

Analysis:

For \( n \) items and knapsack size \( B \); examine the last item \( n \), then

either put item \( n \) into the knapsack or discard it; That is

(1) either increase value by \( v_n \),
Chapter 15. Dynamic Programming

Knapsack Problem

**INPUT:** $n$ items of sizes $s_1, \ldots, s_n$ and values $v_1, \ldots, v_n$; and a knapsack size $B$

**OUTPUT:** a subset $A \subseteq \{1, 2, \ldots, n\}$ that maximizes $\sum_{i \in A} v_i$

with constraint $\sum_{i \in A} s_i \leq B$

Analysis:

For $n$ items and knapsack size $B$; examine the last item $n$, then

- either **put item $n$ into the knapsack or discard it**; That is

  1. either **increase** value by $v_n$, **reduce** available space from $B$ to $B-s_n$, 

Knapsack Problem

**Input:** $n$ items of sizes $s_1, \ldots, s_n$ and values $v_1, \ldots, v_n$; and a knapsack size $B$.

**Output:** a subset $A \subseteq \{1, 2, \ldots, n\}$ that maximizes $\sum_{i \in A} v_i$ with constraint $\sum_{i \in A} s_i \leq B$.

**Analysis:**

For $n$ items and knapsack size $B$; examine the last item $n$, then either put item $n$ into the knapsack or discard it; That is

1. either **increase** value by $v_n$, **reduce** available space from $B$ to $B-s_n$, **reduce** the number of items by 1,
Chapter 15. Dynamic Programming

Knapsack Problem

**INPUT:** $n$ items of sizes $s_1, \ldots, s_n$ and values $v_1, \ldots, v_n$; and a knapsack size $B$

**OUTPUT:** a subset $A \subseteq \{1, 2, \ldots, n\}$ that maximizes $\sum_{i \in A} v_i$

with constraint $\sum_{i \in A} s_i \leq B$

Analysis:

For $n$ items and knapsack size $B$; examine the last item $n$, then

**either put item $n$ into the knapsack or discard it;** That is

1. either **increase** value by $v_n$, **reduce** available space from $B$ to $B-s_n$, **reduce** the number of items by 1, or
2. or simply **reduce** the number of items by 1
Chapter 15. Dynamic Programming

Knapsack Problem

**INPUT:** $n$ items of sizes $s_1, \ldots, s_n$ and values $v_1, \ldots, v_n$; and a knapsack size $B$

**OUTPUT:** a subset $A \subseteq \{1, 2, \ldots, n\}$ that maximizes $\sum_{i \in A} v_i$

with constraint $\sum_{i \in A} s_i \leq B$

Analysis:

For $n$ items and knapsack size $B$; examine the last item $n$, then

either put item $n$ into the knapsack or discard it; That is

(1) either increase value by $v_n$, reduce available space from $B$ to $B-s_n$, reduce the number of items by 1, or

(2) or simply reduce the number of items by 1

Both situations reduce the problem size to a subproblem.
In general, let \( \{1, 2, \ldots, k\} \) be the first \( k \) items, and \( X \) be the available space in the knapsack;
In general, let \( \{1, 2, \ldots, k\} \) be the first \( k \) items, and \( X \) be the available space in the knapsack;

If we define \( A(k, X) \) to be the subset of items drawn from \( \{1, 2, \ldots, k\} \) such that \( A \) maximizes the total value while fits into the space \( X \).
Chapter 15. Dynamic Programming

In general, let \( \{1, 2, \ldots, k\} \) be the first \( k \) items, and \( X \) be the available space in the knapsack;

If we define \( A(k, X) \) to be the subset of items drawn from \( \{1, 2, \ldots, k\} \) such that \( A \) maximizes the total value while fits into the space \( X \)

Then

\[
A(k, X) = \begin{cases} 
\text{either } & A(k - 1, X - s_k) \cup \{k\} & \text{value gained by } v_k \\
\text{or } & A(k - 1, X) & \text{value not gained}
\end{cases}
\]
In general, let \( \{1, 2, \ldots, k\} \) be the first \( k \) items, and \( X \) be the available space in the knapsack;

If we define \( A(k, X) \) to be the subset of items drawn from \( \{1, 2, \ldots, k\} \) such that \( A \) maximizes the total value while fits into the space \( X \)

Then

\[
A(k, X) = \begin{cases} 
\text{either } A(k - 1, X - s_k) \cup \{k\} & \text{value gained by } v_k \\
\text{or } A(k - 1, X) & \text{value not gained}
\end{cases}
\]

Define \( V(k, X) \) to be the maximum value of items drawn from \( \{1, 2, \ldots, k\} \) which fit into the space of \( X \), then

\[
V(k, X) = \max \left\{ \begin{array}{ll}
V(k - 1, X - s_k) + v_k & X \geq s_k \\
V(k - 1, X) &
\end{array} \right.
\]
Chapter 15. Dynamic Programming

- For every $k \leq n$ and every $X \leq B$. 

$$V(k, X) = \max \left\{ \begin{array}{ll} V(k - 1, X - s_k) + v_k & X \geq s_k \\ V(k - 1, X) & \end{array} \right.$$ 

base case,
Chapter 15. Dynamic Programming

• For every $k \leq n$ and every $X \leq B$.

$$V(k, X) = \max \begin{cases} 
V(k - 1, X - s_k) + v_k & X \geq s_k \\
V(k - 1, X) & \text{otherwise}
\end{cases}$$

base case,

$$V(0, X) = 0,$$
Chapter 15. Dynamic Programming

- For every \( k \leq n \) and every \( X \leq B \).

\[
V(k, X) = \max \left\{ \begin{array}{ll}
V(k-1, X - s_k) + v_k & X \geq s_k \\
V(k-1, X) & \end{array} \right.
\]

base case,

\[
V(0, X) = 0, \quad V(k, 0) = 0
\]
Chapter 15. Dynamic Programming

- For every $k \leq n$ and every $X \leq B$.

  \[
  V(k, X) = \max \begin{cases} 
  V(k - 1, X - s_k) + v_k & X \geq s_k \\
  V(k - 1, X) & \end{cases}
  \]

  base case,

  \[
  V(0, X) = 0, \quad V(k, 0) = 0
  \]

- What does the table look like?
For every $k \leq n$ and every $X \leq B$.

$$V(k, X) = \max \begin{cases} V(k - 1, X - s_k) + v_k & X \geq s_k \\ V(k - 1, X) & \end{cases}$$

base case,

$$V(0, X) = 0, \quad V(k, 0) = 0$$

What does the table look like? How to fill it out?
Chapter 15. Dynamic Programming

- For every \( k \leq n \) and every \( X \leq B \).

\[
V(k, X) = \max \left\{ \begin{array}{ll}
V(k-1, X - s_k) + v_k & X \geq s_k \\
V(k-1, X) & \text{base case,}
\end{array} \right.
\]

- What does the table look like? How to fill it out?
- How to traceback for \( A \), the optimal subset of items?
Chapter 15. Dynamic Programming

Example

The optimal knapsack should contain \{1,2\} = 7

\[
\begin{array}{c|cccccc}
\text{i} & 0 & 1 & 2 & 3 & 4 & 5 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 3 & 3 & 3 & 3 \\
2 & 0 & 0 & 3 & 3 & 3 & 3 \\
3 & 0 & 0 & 3 & 4 & 4 & 4 \\
4 & 0 & 0 & 3 & 4 & 5 & 5 \\
\end{array}
\]

\[
V(k, X) = \max \begin{cases} 
V(k - 1, X - s_k) + v_k & X \geq s_k \\
V(k - 1, X) & \end{cases}
\]

Items \((s_i, v_i)\)

1: (2, 3)
2: (3, 4)
3: (4, 5)
4: (5, 6)
Chapter 15. Dynamic Programming

Example

The optimal knapsack should contain \{1,2\} = 7

\begin{align*}
\begin{array}{|c|c|c|c|c|c|c|}
\hline
i \backslash s & 0 & 1 & 2 & 3 & 4 & 5 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 3 & 3 & 3 & 3 \\
2 & 0 & 0 & 3 & 4 & 4 & 7 \\
3 & 0 & 0 & 3 & 4 & 5 & 7 \\
4 & 0 & 0 & 3 & 4 & 5 & 7 \\
\hline
\end{array}
\end{align*}

\[ V(k, X) = \max \left\{ \begin{array}{ll}
V(k-1, X - s_k) + v_k & X \geq s_k \\
V(k-1, X) & \end{array} \right. \]

- fill out the base case row and column;
- the order of cells to be filled;
- \( V(i, X) = V(i-1, X) \) if \( X < s_i \);
- time complexity
Chapter 15. Dynamic Programming

Hidden Markov models and Viterbi’s algorithm
Chapter 15. Dynamic Programming

Hidden Markov models and Viterbi’s algorithm
Chapter 15. Dynamic Programming
Chapter 15. Dynamic Programming
Chapter 15. Dynamic Programming

(a)

\[
\begin{array}{c}
0.99 \\
0.01 \\
0.9 \\
\end{array}
\]

\[
\begin{array}{c}
1 \\
2 \\
\end{array}
\]

\[
\begin{array}{c}
A \ 0.4 \\
C \ 0.1 \\
G \ 0.1 \\
T \ 0.4 \\
\end{array}
\]

\[
\begin{array}{c}
A \ 0.05 \\
C \ 0.4 \\
G \ 0.5 \\
T \ 0.05 \\
\end{array}
\]

(b)

**state sequence (hidden):**

... 1 1 1 1 1 2 2 2 2 2 1 1 ...

**transitions:** ? 0.99 0.99 0.99 0.99 0.01 0.9 0.9 0.9 0.1 0.99

(c)

**symbol sequence (observable):**

... A T C A A G G C G A T ...

**emissions:** 0.4 0.4 0.1 0.4 0.4 0.5 0.5 0.4 0.4 0.4
Chapter 15. Dynamic Programming

Viterbi Algorithm

• Input: an HMM $M$, and data $D = d_1d_2 \ldots d_n$
Viterbi Algorithm

- Input: an HMM $\mathcal{M}$, and data $D = d_1d_2\ldots d_n$
- Output: hidden state sequences $H = s_1s_2\ldots s_n$ that generate $D$; such that

\[ \text{Prob}(H, D | \mathcal{M}) \] achieves the maximum
Viterbi Algorithm

- **Input:** an HMM $\mathcal{M}$, and data $D = d_1 d_2 \ldots d_n$
- **Output:** hidden state sequences $H = s_1 s_2 \ldots s_n$ that generate $D$; such that

$$\text{Prob}(H, D | \mathcal{M})$$

achieves the maximum
Chapter 15. Dynamic Programming

Viterbi Algorithm

- Input: an HMM $M$, and data $D = d_1 d_2 \ldots d_n$
- Output: hidden state sequences $H = s_1 s_2 \ldots s_n$ that generate $D$; such that

$$
\text{Prob}(H, D|M) \text{ achieves the maximum}
$$

... 1 3 2 4 6 6 6 4 1 6 5 6 6 2 4 2 1 2 3 5...
... F F F F L L L L L L L L F F F F F F F F...
A hidden Markov model (HMM) consists of \((S, T, e, t)\)
A hidden Markov model (HMM) consists of $(S, T, e, t)$

- a set of states $S = \{F, L\}$;
Chapter 15. Dynamic Programming

A hidden Markov model (HMM) consists of $(S, T, e, t)$

- a set of states $S = \{F, L\}$;
- a transition relation $T = \{(F, F), (F, L), (L, F), (L, L)\} \subseteq S \times S$, 

\begin{itemize}
  \item $e(F, d) = \frac{1}{6}$ for $d = 1, 2, \ldots, 6$
  \item $e(L, d) = \frac{1}{12}$ and $e(L, d) = \frac{1}{10}$ for $d = 1, 2, \ldots, 5$
  \item $t(F, F) = 0.95$, $t(F, L) = 0.05$, $t(L, F) = 0.1$, $t(L, L) = 0.90$
\end{itemize}
Chapter 15. Dynamic Programming

A hidden Markov model (HMM) consists of $(S, T, e, t)$

- a set of states $S = \{F, L\}$;
- a transition relation $T = \{(F, F), (F, L), (L, F), (L, L)\} \subseteq S \times S$,
- emission probability distribution $e$
A hidden Markov model (HMM) consists of \((S, T, e, t)\)

- a set of states \(S = \{F, L\}\);
- a transition relation \(T = \{(F, F), (F, L), (L, F), (L, L)\} \subseteq S \times S\),
- emission probability distribution \(e\)
  \[e(F, d) = \frac{1}{6} \text{ for } d = 1, 2, \ldots, 6;\]
Chapter 15. Dynamic Programming

A hidden Markov model (HMM) consists of \((S, T, e, t)\)

- a set of states \(S = \{F, L\}\);
- a transition relation \(T = \{(F, F), (F, L), (L, F), (L, L)\} \subseteq S \times S\),
- emission probability distribution \(e\)
  \[e(F, d) = \frac{1}{6} \text{ for } d = 1, 2, \ldots, 6;\]
  \[e(L, 6) = \frac{1}{2} \text{ and } e(L, d) = \frac{1}{10} \text{ for } d = 1, 2, \ldots, 5;\]
Chapter 15. Dynamic Programming

A hidden Markov model (HMM) consists of \((S, T, e, t)\)

- a set of states \(S = \{F, L\}\);
- a transition relation \(T = \{(F, F), (F, L), (L, F), (L, L)\} \subseteq S \times S\),
- emission probability distribution \(e\)
  \[
  e(F, d) = \frac{1}{6} \text{ for } d = 1, 2, \ldots, 6;
  \]
  \[
  e(L, 6) = \frac{1}{2} \text{ and } e(L, d) = \frac{1}{10} \text{ for } d = 1, 2, \ldots, 5;
  \]
- transition probability distribution \(t\)
  \[
  t(F, F) = 0.95, t(F, L) = 0.05, t(L, F) = 0.1, t(L, L) = 0.90
  \]
A hidden Markov model (HMM) consists of \((S, T, e, t)\)

- a set of states \(S = \{F, L\}\);
- a transition relation \(T = \{(F, F), (F, L), (L, F), (L, L)\} \subseteq S \times S\),
- emission probability distribution \(e\)
  \(e(F, d) = \frac{1}{6}\) for \(d = 1, 2, \ldots, 6\);
  \(e(L, 6) = \frac{1}{2}\) and \(e(L, d) = \frac{1}{10}\) for \(d = 1, 2, \ldots, 5\);
- transition probability distribution \(t\)
  \(t(F, F) = 0.95, t(F, L) = 0.05, t(L, F) = 0.1, t(L, L) = 0.90\);
Chapter 15. Dynamic Programming

The goal is to identify hidden states $s_1 s_2 \ldots s_n$ that generates $d_1 d_2 \ldots d_n$, to maximize the associated probability.
Chapter 15. Dynamic Programming

The goal is to identify hidden states $s_1 s_2 \ldots s_n$ that generates $d_1 d_2 \ldots d_n$, to maximize the associated probability

\[ P(s_1 s_2 \ldots s_n, d_1 d_2 \ldots d_n) \]
The goal is to identify hidden states $s_1s_2 \ldots s_n$ that generates $d_1d_2 \ldots d_n$, to maximize the associated probability

$$P(s_1s_2 \ldots s_n, d_1d_2 \ldots d_n)$$

e.g., $d_1 = 3$, $d_2 = 6$, there are 4 possible ways to generate them:
Chapter 15. Dynamic Programming

The goal is to identify hidden states $s_1 s_2 \ldots s_n$ that generates $d_1 d_2 \ldots d_n$, to maximize the associated probability

$$P(s_1 s_2 \ldots s_n, d_1 d_2 \ldots d_n)$$

e.g., $d_1 = 3$, $d_2 = 6$, there are 4 possible ways to generate them:

- $FF$, $FL$, $LL$, $LF$, 

Chapter 15. Dynamic Programming

The goal is to identify hidden states $s_1 s_2 \ldots s_n$ that generates $d_1 d_2 \ldots d_n$, to maximize the associated probability $P(s_1 s_2 \ldots s_n, d_1 d_2 \ldots d_n)$.

e.g., $d_1 = 3$, $d_2 = 6$, there are 4 possible ways to generate them:

- $FF$, $FL$, $LL$, $LF$, but with different probabilities
Chapter 15. Dynamic Programming

- $P(FF, 36) = 0.95 \times 1/6 \times 1/6 = 0.02639$

- $P(FL, 36) = e(F, 3) \times t(F, L) \times e(L, 6) = 0.9 \times 0.05 \times 1/2 = 0.00417$

- $P(LL, 36) = e(L, 3) \times t(L, L) \times e(L, 6) = 1/10 \times 0.9 \times 1/2 = 0.045$

- $P(LF, 36) = e(L, 3) \times t(L, F) \times e(F, 6) = 1/10 \times 0.1 \times 1/6 = 0.00167$

but for 10 digits, how many cases to consider?
Chapter 15. Dynamic Programming

- $P(FF, 3 6) = e(F, 3) \times t(F, F) \times e(F, 6)$
Chapter 15. Dynamic Programming

- \( P(FF, 36) = e(F, 3) \times t(F, F) \times e(F, 6) = \frac{1}{6} \times 0.95 \times \frac{1}{6} \)
Chapter 15. Dynamic Programming

- \( P(FF, 36) = e(F, 3) \times t(F, F) \times e(F, 6) = \frac{1}{6} \times 0.95 \times \frac{1}{6} = 0.02639 \)
Chapter 15. Dynamic Programming

- $P(FF, 36) = e(F, 3) \times t(F, F) \times e(F, 6) = \frac{1}{6} \times 0.95 \times \frac{1}{6} = 0.02639$
- $P(FL, 36)$
Chapter 15. Dynamic Programming

- \( P(FF, 36) = e(F, 3) \times t(F, F) \times e(F, 6) = \frac{1}{6} \times 0.95 \times \frac{1}{6} = 0.02639 \)
- \( P(FL, 36) = e(F, 3) \times t(F, L) \times e(L, 6) \)
Chapter 15. Dynamic Programming

- \( P(FF, 36) = e(F, 3) \times t(F, F) \times e(F, 6) = \frac{1}{6} \times 0.95 \times \frac{1}{6} = 0.02639 \)
- \( P(FL, 36) = e(F, 3) \times t(F, L) \times e(L, 6) = \frac{1}{6} \times 0.05 \times \frac{1}{2} = 0.00167 \)
Chapter 15. Dynamic Programming

- \( P(FF, 36) = e(F, 3) \times t(F, F) \times e(F, 6) = \frac{1}{6} \times 0.95 \times \frac{1}{6} = 0.02639 \)
- \( P(FL, 36) = e(F, 3) \times t(F, L) \times e(L, 6) = \frac{1}{6} \times 0.05 \times \frac{1}{2} = 0.00417 \)
Chapter 15. Dynamic Programming

- \( P(FF, 36) = e(F, 3) \times t(F, F) \times e(F, 6) = \frac{1}{6} \times 0.95 \times \frac{1}{6} = 0.02639 \)
- \( P(FL, 36) = e(F, 3) \times t(F, L) \times e(L, 6) = \frac{1}{6} \times 0.05 \times \frac{1}{2} = 0.00417 \)
- \( P(LL, 36) \) but for 10 digits, how many cases to consider?
Chapter 15. Dynamic Programming

- $P(FF, 36) = e(F, 3) \times t(F, F) \times e(F, 6) = \frac{1}{6} \times 0.95 \times \frac{1}{6} = 0.02639$
- $P(FL, 36) = e(F, 3) \times t(F, L) \times e(L, 6) = \frac{1}{6} \times 0.05 \times \frac{1}{2} = 0.00417$
- $P(LL, 36) = e(L, 3) \times t(L, L) \times e(L, 6)$

but for 10 digits, how many cases to consider?
Chapter 15. Dynamic Programming

- \( P(FF, 36) = e(F, 3) \times t(F, F) \times e(F, 6) = \frac{1}{6} \times 0.95 \times \frac{1}{6} = 0.02639 \)
- \( P(FL, 36) = e(F, 3) \times t(F, L) \times e(L, 6) = \frac{1}{6} \times 0.05 \times \frac{1}{2} = 0.00417 \)
- \( P(LL, 36) = e(L, 3) \times t(L, L) \times e(L, 6) = \frac{1}{10} \times 0.90 \times \frac{1}{2} = 0.045 \)

but for 10 digits, how many cases to consider?
Chapter 15. Dynamic Programming

- \( P(FF, 36) = e(F, 3) \times t(F, F) \times e(F, 6) = \frac{1}{6} \times 0.95 \times \frac{1}{6} = 0.02639 \)
- \( P(FL, 36) = e(F, 3) \times t(F, L) \times e(L, 6) = \frac{1}{6} \times 0.05 \times \frac{1}{2} = 0.00417 \)
- \( P(LL, 36) = e(L, 3) \times t(L, L) \times e(L, 6) = \frac{1}{10} \times 0.90 \times \frac{1}{2} = 0.045 \)
Chapter 15. Dynamic Programming

- \( P(FF, 36) = e(F, 3) \times t(F, F) \times e(F, 6) = \frac{1}{6} \times 0.95 \times \frac{1}{6} = 0.02639 \)
- \( P(FL, 36) = e(F, 3) \times t(F, L) \times e(L, 6) = \frac{1}{6} \times 0.05 \times \frac{1}{2} = 0.00417 \)
- \( P(LL, 36) = e(L, 3) \times t(L, L) \times e(L, 6) = \frac{1}{10} \times 0.90 \times \frac{1}{2} = 0.045 \)
- \( P(LF, 36) \)
Chapter 15. Dynamic Programming

- \( P(FF,36) = e(F,3) \times t(F,F) \times e(F,6) = \frac{1}{6} \times 0.95 \times \frac{1}{6} = 0.02639 \)
- \( P(FL,36) = e(F,3) \times t(F,L) \times e(L,6) = \frac{1}{6} \times 0.05 \times \frac{1}{2} = 0.00417 \)
- \( P(LL,36) = e(L,3) \times t(L,L) \times e(L,6) = \frac{1}{10} \times 0.90 \times \frac{1}{2} = 0.045 \)
- \( P(LF,36) = e(L,3) \times t(L,F) \times e(F,6) \)

but for 10 digits, how many cases to consider?
Chapter 15. Dynamic Programming

• \( P(FF, 36) = e(F, 3) \times t(F, F) \times e(F, 6) = \frac{1}{6} \times 0.95 \times \frac{1}{6} = 0.02639 \)
• \( P(FL, 36) = e(F, 3) \times t(F, L) \times e(L, 6) = \frac{1}{6} \times 0.05 \times \frac{1}{2} = 0.00417 \)
• \( P(LL, 36) = e(L, 3) \times t(L, L) \times e(L, 6) = \frac{1}{10} \times 0.90 \times \frac{1}{2} = 0.045 \)
• \( P(LF, 36) = e(L, 3) \times t(L, F) \times e(F, 6) = \frac{1}{10} \times 0.10 \times \frac{1}{6} \)
Chapter 15. Dynamic Programming

- $P(FF, 3 6) = e(F, 3) \times t(F, F) \times e(F, 6) = \frac{1}{6} \times 0.95 \times \frac{1}{6} = 0.02639$
- $P(FL, 3 6) = e(F, 3) \times t(F, L) \times e(L, 6) = \frac{1}{6} \times 0.05 \times \frac{1}{2} = 0.00417$
- $P(LL, 3 6) = e(L, 3) \times t(L, L) \times e(L, 6) = \frac{1}{10} \times 0.90 \times \frac{1}{2} = 0.045$
- $P(LF, 3 6) = e(L, 3) \times t(L, F) \times e(F, 6) = \frac{1}{10} \times 0.10 \times \frac{1}{6} = 0.00167$

but for 10 digits, how many cases to consider?
Chapter 15. Dynamic Programming

- $P(FF, 3 \ 6) = e(F, 3) \times t(F, F) \times e(F, 6) = \frac{1}{6} \times 0.95 \times \frac{1}{6} = 0.02639$
- $P(FL, 3 \ 6) = e(F, 3) \times t(F, L) \times e(L, 6) = \frac{1}{6} \times 0.05 \times \frac{1}{2} = 0.00417$
- $P(LL, 3 \ 6) = e(L, 3) \times t(L, L) \times e(L, 6) = \frac{1}{10} \times 0.90 \times \frac{1}{2} = 0.045$
- $P(LF, 3 \ 6) = e(L, 3) \times t(L, F) \times e(F, 6) = \frac{1}{10} \times 0.10 \times \frac{1}{6} = 0.00167$

but for 10 digits, how many cases to consider?
Chapter 15. Dynamic Programming

Take a dynamic programming approach:

Take a dynamic programming approach:
Take a dynamic programming approach:

For prefix $d_1 \ldots d_k$, define:

- $P(F,k)$ to be the maximum probability for states $\ldots F$ to generate $d_1 \ldots d_k$.
- $P(L,k)$ to be the maximum probability for states $\ldots L$ to generate $d_1 \ldots d_k$.
Chapter 15. Dynamic Programming

Take a dynamic programming approach:

For prefix $d_1 \ldots d_k$, define:

- $P(F, k)$ to be the maximum probability for states $\ldots F$ to generate $d_1 \ldots d_k$
Chapter 15. Dynamic Programming

Take a dynamic programming approach:

For prefix $d_1 \ldots d_k$, define:

- $P(F, k)$ to be the maximum probability for states $\ldots F$ to generate $d_1 \ldots d_k$

- $P(L, k)$ to be the maximum probability for states $\ldots L$ to generate $d_1 \ldots d_k$
Chapter 15. Dynamic Programming

Then recursively,

![Diagram showing transition probabilities between fair and loaded dice.](image-url)
Chapter 15. Dynamic Programming

Then recursively,

\[ P(F, k) = \max \left\{ P(F, k - 1) \times t(F, F) \times e(F, d_k), P(L, k - 1) \times t(L, F) \times e(F, d_k) \right\} \]

where \( F \) and \( L \) represent the states of the dice being fair or loaded, respectively, and \( d_k \) is the number of times the dice has been rolled.
Chapter 15. Dynamic Programming

Then recursively,

\[
P(F, k) = \max \left\{ P(F, k - 1) \times t(F, F) \times e(F, d_k), \ P(L, k - 1) \times t(F, L) \times e(F, d_k) \right\}
\]

\[
P(L, k) = \max \left\{ P(F, k - 1) \times t(L, F) \times e(F, d_k), \ P(L, k - 1) \times t(L, L) \times e(L, d_k) \right\}
\]
Then recursively,

\[ P(F, k) = \max \left\{ P(F, k-1) \times t(F, F) \times e(F, d_k), P(L, k-1) \times t(L, F) \times e(F, d_k) \right\} \]

\[ P(L, k) = \max \left\{ P(F, k-1) \times t(F, L) \times e(L, d_k), P(L, k-1) \times t(L, L) \times e(L, d_k) \right\} \]

\[ P(F, 1) = \frac{1}{6} \]
Then recursively,

\[ P(F, k) = \max \left\{ \begin{array}{c} P(F, k - 1) \times t(F, F) \times e(F, d_k) \\ P(L, k - 1) \times t(L, F) \times e(F, d_k) \end{array} \right\} \]

\[ P(L, k) = \max \left\{ \begin{array}{c} P(F, k - 1) \times t(F, L) \times e(L, d_k) \\ P(L, k - 1) \times t(L, L) \times e(L, d_k) \end{array} \right\} \]

\[ P(F, 1) = \frac{1}{6} \quad P(L, 1) = \begin{cases} \frac{1}{10} & 1 \leq d_1 \leq 5 \\ \frac{1}{2} & d_1 = 6 \end{cases} \]
The outlined method is Viterbi Algorithm.
Chapter 15. Dynamic Programming

The outlined method is Viterbi Algorithm.

- **Start** and **Finish** states can be added into the model, but they do not emit digits (called **silent states**);
The outlined method is Viterbi Algorithm.

- **Start** and **Finish** states can be added into the model, but they do not emit digits (called **silent states**);
- The state in phase $k - 1$ determines the state in phase $k$;
Chapter 15. Dynamic Programming

The outlined method is Viterbi Algorithm.

- **Start** and **Finish** states can be added into the model, but they do not emit digits (called **silent states**);
- The state in phase \( k - 1 \) determines the state in phase \( k \);
- Given a state in current phase \( k \), the previous state in phase \( k - 1 \) can only be one of those having transition to the current state.
Chapter 15. Dynamic Programming

A hidden Markov model (HMM) consists of \((S, T, e, t)\), where

- \(S\) is a set of states: \(s_0, s_1, \ldots, s_m\); where \(s_0\) is the begin state;
- transitions \(T \subseteq S \times S\);
- each state \(s_i\) is associated with an (emission) probability distribution \(e(i, a)\) for \(a \in \Sigma\), such that \(\sum_{a \in \Sigma} e(i, a) = 1\) for every \(i, 1 \leq i < m\);
- every \((s_i, s_j)\) is associated with a probability distribution \(t(i, j)\), such that \(\sum_{1 \leq j \leq m} t(i, j) = 1\) for every \(i, 0 \leq i < m\).
A hidden Markov model (HMM) consists of \((S, T, e, t)\), where

- \(S\) is a set of states: \(s_0, s_1, \ldots, s_m\); where \(s_0\) is the begin state;
- \(T \subseteq S \times S\);
- each state \(s_i\) is associated with a (emission) probability distribution \(e(i, a)\) for \(a \in \Sigma\), such that \(\sum_{a \in \Sigma} e(i, a) = 1\) for every \(i, 1 \leq i < m\);
- every \((s_i, s_j)\) \(\in T\) is associated with a probability distribution \(t(i, j)\), such that \(\sum_{1 \leq j \leq m} t(i, j) = 1\) for every \(i, 0 \leq i < m\).
Chapter 15. Dynamic Programming

A hidden Markov model (HMM) consists of \((S, T, e, t)\), where

- \(S\) is a set of states: \(s_0, s_1, \ldots, s_m\); where \(s_0\) is the begin state;
- transitions \(T \subseteq S \times S\);
Chapter 15. Dynamic Programming

A hidden Markov model (HMM) consists of \((S, T, e, t)\), where

- \(S\) is a set of states: \(s_0, s_1, \ldots, s_m\); where \(s_0\) is the begin state;
- transitions \(T \subseteq S \times S\);
- each state \(s_i\) is associated with a (emission) probability distribution \(e(i, a)\)
  for \(a \in \Sigma\), such that

\[
\sum_{a \in \Sigma} e(i, a) = 1 \quad \text{for every} \quad 0 \leq i < m
\]
Chapter 15. Dynamic Programming

A hidden Markov model (HMM) consists of \((S, T, e, t)\), where

- \(S\) is a set of states: \(s_0, s_1, \ldots, s_m\); where \(s_0\) is the begin state;
- transitions \(T \subseteq S \times S\);
- each state \(s_i\) is associated with a (emission) probability distribution \(e(i, a)\)
  for \(a \in \Sigma\), such that

\[
\sum_{a \in \Sigma} e(i, a) = 1 \quad \text{for every } i, \ 1 \leq i < m
\]
Chapter 15. Dynamic Programming

A hidden Markov model (HMM) consists of \((S, T, e, t)\), where

- \(S\) is a set of states: \(s_0, s_1, \ldots, s_m\); where \(s_0\) is the begin state;
- transitions \(T \subseteq S \times S\);
- each state \(s_i\) is associated with a (emission) probability distribution \(e(i, a)\) for \(a \in \Sigma\), such that
  \[
  \sum_{a \in \Sigma} e(i, a) = 1 \quad \text{for every } i, \, 1 \leq i < m
  \]
- every \((s_i, s_j) \in T\) is associated with a probability distribution \(t(i, j)\), such that
  \[
  \sum_{1 \leq j \leq m} t(i, j) = 1 \quad \text{for every } i, \, 0 \leq i < m
  \]
Chapter 15. Dynamic Programming

A hidden Markov model (HMM) consists of \((S, T, e, t)\), where

- \(S\) is a set of states: \(s_0, s_1, \ldots, s_m\); where \(s_0\) is the begin state;
- transitions \(T \subseteq S \times S\);
- each state \(s_i\) is associated with a (emission) probability distribution \(e(i, a)\) for \(a \in \Sigma\), such that
  \[
  \sum_{a \in \Sigma} e(i, a) = 1 \quad \text{for every } i, \quad 1 \leq i < m
  \]
- every \((s_i, s_j) \in T\) is associated with a probability distribution \(t(i, j)\), such that
  \[
  \sum_{1 \leq j \leq m} t(i, j) = 1 \quad \text{for every } i, \quad 0 \leq i < m
  \]

\[
\begin{align*}
\Sigma = \{a, b\} \\
\end{align*}
\]
HMM decoding problem:

**Input:** HMM $M$, sequence of symbols $x \in \Sigma^*$

**Output:** $y^* \in \Sigma^*$, a sequence of states such that $y^* = \text{arg max}_{y \in \Sigma^*} \{ \text{Prob}(y, x | M) \}$ where $y^*$ begins from $s_0$. 

Diagram:

- States: $s_0, s_1, s_4, s_i, s_k, s_j, s_m$
- Edges:
  - $e(i, a) = 0.2$
  - $e(i, b) = 0.8$
  - $t(i, k) = 0.6$
  - $t(i, j) = 0.4$

$\Sigma = \{a, b\}$
Chapter 15. Dynamic Programming

HMM decoding problem:
Chapter 15. Dynamic Programming

HMM decoding problem:

**INPUT:** HMM $\mathcal{M}$, sequence of symbols $x \in \Sigma^*$

**OUTPUT:** $y^*$, a sequence of states such that

$$y^* = \arg \max_y \{Prob(y, x|\mathcal{M})\}$$
Chapter 15. Dynamic Programming

HMM decoding problem:

**Input:** HMM $\mathcal{M}$, sequence of symbols $x \in \Sigma^*$
**Output:** $y^*$, a sequence of states such that

$$y^* = \arg \max_y \{ \text{Prob}(y, x | \mathcal{M}) \}$$

where $y$ begins from $s_0$. 

**Diagram:**

- States: $s_0$, $s_1$, $s_2$, $s_3$, $s_4$, $\ldots$, $s_m$
- Transition probabilities:
  - $t(i, k) = 0.6$
  - $t(i, j) = 0.4$
- Emission probabilities:
  - $e(i, a) = 0.2$
  - $e(i, b) = 0.8$
- Alphabet: $\Sigma = \{a, b\}$
Chapter 15. Dynamic Programming

The idea of Viterbi algorithm:
Chapter 15. Dynamic Programming

The idea of Viterbi algorithm:

For any prefix $x_1 \ldots x_k$, $k \geq 0$, 
Chapter 15. Dynamic Programming

The idea of Viterbi algorithm:
For any prefix $x_1 \ldots x_k$, $k \geq 0$, and state $s_j \in S$;
The idea of Viterbi algorithm:

For any prefix $x_1 \ldots x_k$, $k \geq 0$, and state $s_j \in S$;

$$\max_{y=s_0 \ldots s_j} \{ Prob(y, x_1 \ldots x_k | M) \}$$

the maximum probability of a path $s_0 \ldots s_j$ to generate $x_1 \ldots x_k$;
The idea of Viterbi algorithm:

For any prefix $x_1 \ldots x_k$, $k \geq 0$, and state $s_j \in S$;

$$\max_{y=s_0 \ldots s_j} \{\text{Prob}(y, x_1 \ldots x_k | M)\}$$

the maximum probability of a path $s_0 \ldots s_j$ to generate $x_1 \ldots x_k$;
Chapter 15. Dynamic Programming

Define $f(j,k) = \max_{y = s_0 \ldots s_j} \{ \text{Prob}(y, x_1 \ldots x_k) \}$

Recursively, $f(j,k) = \max_i \{ f(i,k-1) \times t(i,j) \times e(j,x_k) \}$

Base case: $f(0,0) = 1$, $f(i,0) = 0$ for all $i \geq 1$. 

Graph: 
- $s_0$ generates $x_1x_2\ldots x_{k-1}$.
- $s_j$ generates $x_k$.
- $y$ connects $s_0$ to $s_j$. 

Diagram:
- A red circle labeled $s_0$.
- A blue circle labeled $s_j$.
- A dashed line with a label $y$ connecting $s_0$ to $s_j$.
- A label $\text{generates } x_1x_2\ldots x_{k-1}$ connected to $s_0$.
- A label $\text{generates } x_k$ connected to $s_j$. 

Diagonal line connections between $s_0$ and $s_j$.
Chapter 15. Dynamic Programming

Define \( f(j, k) = \max_{y=s_0\ldots s_j} \{ \text{Prob}(y, x_1 \ldots x_k) \} \)

Recursively, \( f(j, k) = \max_{i} \{ f(i, k-1) \times t(i, j) \times e(j, x_k) \} \)

Base case: \( f(0, 0) = 1 \), \( f(i, 0) = 0 \) for all \( i \geq 1 \).
Chapter 15. Dynamic Programming

Define $f(j, k) = \max_{y=s_0 \ldots s_j} \{ \text{Prob}(y, x_1 \ldots x_k) \}$

Recursively, $f(j, k) = \max_i \{ f(i, k-1) \times t(i, j) \times e(j, x_k) \}$

Base case: $f(0, 0) = 1$, $f(i, 0) = 0$ for all $i \geq 1$. 
Chapter 15. Dynamic Programming

Define \( f(j, k) = \max_{y=s_0 \ldots s_j} \{\text{Prob}(y, x_1 \ldots x_k)\} \)

Recursively, \( f(j, k) = \)
Chapter 15. Dynamic Programming

Define \( f(j, k) = \max_{y=s_0...s_j} \{ \text{Prob}(y, x_1...x_k) \} \)

Recursively, \( f(j, k) = \max_i \{ f(i, k - 1) \times t(i, j) \times e(j, x_k) \} \)

Base case: \( f(0, 0) = 1 \), \( f(i, 0) = 0 \) for all \( i \geq 1 \).
Chapter 15. Dynamic Programming

Define \( f(j, k) = \max_{y = s_0 \ldots s_j} \{ \text{Prob}(y, x_1 \ldots x_k) \} \)

Recursively, \( f(j, k) = \max_i \{ f(i, k - 1) \times t(i, j) \times e(j, x_k) \} \)

Base case: \( f(0, 0) = 1 \), \( f(i, 0) = 0 \) for all \( i \geq 1 \).
Chapter 15. Dynamic Programming

Define $f(j, k) = \max_{y = s_0 \ldots s_j} \{Prob(y, x_1 \ldots x_k)\}$

Recursively, $f(j, k) = \max_i \{f(i, k - 1) \times t(i, j)\}$
Define \( f(j, k) = \max_{y=s_0\ldots s_j} \{Prob(y, x_1 \ldots x_k)\} \)

Recursively, \( f(j, k) = \max_i \left\{ f(i, k - 1) \times t(i, j) \times e(j, x_k) \right\} \)
Chapter 15. Dynamic Programming

Define \( f(j, k) = \max_{y=s_0...s_j} \{ Prob(y, x_1...x_k) \} \)

Recursively, \( f(j, k) = \max_i \{ f(i, k - 1) \times t(i, j) \times e(j, x_k) \} \)
Chapter 15. Dynamic Programming

Define $f(j, k) = \max_{y=s_0 \ldots s_j} \{Prob(y, x_1 \ldots x_k)\}$

Recursively, $f(j, k) = \max_i \{f(i, k - 1) \times t(i, j) \times e(j, x_k)\}$

Base case: $f(0, 0) =$
Chapter 15. Dynamic Programming

Define \( f(j, k) = \max_{y=s_0 \ldots s_j} \{ \text{Prob}(y, x_1 \ldots x_k) \} \)

Recursively, \( f(j, k) = \max_i \{ f(i, k - 1) \times t(i, j) \times e(j, x_k) \} \)

Base case: \( f(0, 0) = 1 \),
Define $f(j, k) = \max_{y=s_0 \ldots s_j} \{Prob(y, x_1 \ldots x_k)\}$

Recursively, $f(j, k) = \max_i \{f(i, k - 1) \times t(i, j) \times e(j, x_k)\}$

Base case: $f(0, 0) = 1$, $f(i, 0)$
Chapter 15. Dynamic Programming

Define $f(j, k) = \max_{y=s_0 \ldots s_j} \{ Prob(y, x_1 \ldots x_k) \}$

Recursively,

$$f(j, k) = \max_i \{ f(i, k-1) \times t(i, j) \times e(j, x_k) \}$$

Base case: $f(0, 0) = 1$, $f(i, 0) = 0$ for all $i \geq 1$. 
Algorithm $\text{VITERBI}(x, n, t, e, m)$

0. $T(0, 0) = 1; P(0, 0) = 0;$
Algorithm \textsc{Viterbi}(x, n, t, e, m)

0. \hspace{1cm} T(0, 0) = 1; P(0, 0) = 0;
1. \hspace{1cm} \textbf{for} \hspace{0.5cm} j = 1 \textbf{ to } m
2. \hspace{1cm} T(j, 0) = 0;
Chapter 15. Dynamic Programming

Algorithm \textsc{Viterbi}(x, n, t, e, m)

0. \(T(0, 0) = 1; \ P(0, 0) = 0;\)
1. \textbf{for} \(j = 1\ \textbf{to} \ m\)
2. \(T(j, 0) = 0;\)
3. \textbf{for} \(k = 1\ \textbf{to} \ n\)
Algorithm VITERBI($x, n, t, e, m$)

0. $T(0, 0) = 1; \ P(0, 0) = 0$
1. for $j = 1$ to $m$
2. $T(j, 0) = 0$
3. for $k = 1$ to $n$  for every prefix upto $k$th symbol
4. premax = 0
5. for $i = 0$ to $m$ try every 'predecessor state' $s_i$
6. if premax $< T(i, k - 1) \times t(i,j) \times e(j,x_k)$
7. premax $= T(i, k - 1) \times t(i,j) \times e(j,x_k)$
8. $P(j,k) = i$ memorize the predecessor
9. $T(j,k) = \text{premax}$
10. laststate = 1; $\max = T(1, n)$
11. for $j = 2$ to $m$ identify maximum probability
12. if $\max < T(j, n)$
13. laststate $= j$
14. $\max = T(j, n)$
15. return ($T$, $P$, $\max$, laststate)
Algorithm \textsc{Viterbi}\((x, n, t, e, m)\)

0. \(T(0, 0) = 1; P(0, 0) = 0;\)
1. \textbf{for} \(j = 1\) \textbf{to} \(m\)
2. \(T(j, 0) = 0;\)
3. \textbf{for} \(k = 1\) \textbf{to} \(n\) \hspace{1cm} \text{for every prefix upto } k\text{th symbol}
4. \textbf{for} \(j = 1\) \textbf{to} \(m\)
Chapter 15. Dynamic Programming

Algorithm \textsc{Viterbi}(x, n, t, e, m)

0. \quad T(0, 0) = 1; \quad P(0, 0) = 0;
1. \quad \textbf{for} \ j = 1 \ \textbf{to} \ m \\
2. \quad T(j, 0) = 0;
3. \quad \textbf{for} \ k = 1 \ \textbf{to} \ n \quad \text{for every prefix upto } k\text{th symbol} \\
4. \quad \textbf{for} \ j = 1 \ \textbf{to} \ m \quad \text{for every ‘last state’ } s_j
Chapter 15. Dynamic Programming

Algorithm VITERBI($x, n, t, e, m$)

0. $T(0, 0) = 1; P(0, 0) = 0;
1. for $j = 1$ to $m$
2. $T(j, 0) = 0$;
3. for $k = 1$ to $n$ for every prefix upto $k$th symbol
4. for $j = 1$ to $m$ for every ‘last state’ $s_j$
5. $premax = 0$

6. $premax = 0$
7. for $i = 0$ to $m$ try every ‘predecessor state’ $s_i$
8. if $premax < T(i, k - 1) \times t(i, j) \times e(j, x_k)$
9. $premax = T(i, k - 1) \times t(i, j) \times e(j, x_k)$
10. $P(j, k) = i$ memorize the predecessor
11. $T(j, k) = premax$
12. laststate = 1
13. for $j = 2$ to $m$ identify maximum probability
14. if $max < T(j, n)$
15. laststate = $j$
16. $max = T(j, n)$

17. return ($T, P, max, laststate$)

Time complexity: $O(nm^2)$
Algorithm VITERBI($x, n, t, e, m$)

0. $T(0, 0) = 1; P(0, 0) = 0$;
1. for $j = 1$ to $m$
2. $T(j, 0) = 0$;
3. for $k = 1$ to $n$ for every prefix upto $k$th symbol
4. for $j = 1$ to $m$ for every ‘last state’ $s_j$
5. $premax = 0$ initialize the prefix probability
6. for $i = 0$ to $m$ try every ‘predecessor state’ $s_i$
7. if $premax < T(i, k - 1) 	imes t(i, j) 	imes e(j, x_k)$
8. $premax = T(i, k - 1) 	imes t(i, j) 	imes e(j, x_k)$ update probability
9. $P(j, k) = i$ memorize the predecessor
10. $T(j, k) = premax$
11. $laststate = 1$;
12. max = $T(1, n)$
13. for $j = 2$ to $m$ identify maximum probability
14. if max < $T(j, n)$
15. laststate = $j$
16. max = $T(j, n)$
17. return ($T$, $P$, max, laststate)
Chapter 15. Dynamic Programming

Algorithm VITERBI $(x, n, t, e, m)$

0. \( T(0, 0) = 1; P(0, 0) = 0; \)
1. \( \text{for } j = 1 \text{ to } m \)
2. \( T(j, 0) = 0; \)
3. \( \text{for } k = 1 \text{ to } n \) for every prefix upto \( k \)th symbol
4. \( \text{for } j = 1 \text{ to } m \) for every ‘last state’ \( s_j \)
5. \( \text{premax} = 0 \) initialize the prefix probability
6. \( \text{for } i = 0 \text{ to } m \) try every ‘predecessor state’ \( s_i \)
7. \( \text{if } \text{premax} < T(i, k - 1) \times t(i, j) \times e(j, x_k) \)
8. \( \text{premax} = T(i, k - 1) \times t(i, j) \times e(j, x_k) \) update probability
9. \( P(j, k) = i \) memorize the predecessor
10. \( T(j, k) = \text{premax} \)
11. \( \text{laststate} = 1 \)
12. \( \text{max} = T(1, n) \)
13. \( \text{for } j = 2 \text{ to } m \) identify maximum probability
14. \( \text{if } \text{max} < T(j, n) \)
15. \( \text{laststate} = j \)
16. \( \text{max} = T(j, n) \)
17. \( \text{return } (T, P, \text{max}, \text{laststate}) \)

Time complexity: \( O(nm^2) \)
Algorithm VITERBI($x, n, t, e, m$)

0. $T(0, 0) = 1; P(0, 0) = 0$
1. for $j = 1$ to $m$
2. $T(j, 0) = 0$
3. for $k = 1$ to $n$ for every prefix upto $k$th symbol
4. for $j = 1$ to $m$ for every ‘last state’ $s_j$
5. $\text{premax} = 0$ initialize the prefix probability
6. for $i = 0$ to $m$ try every ‘predecessor state’ $s_i$
7. if $\text{premax} < T(i, k - 1) \times t(i, j) \times e(j, x_k)$
Chapter 15. Dynamic Programming

Algorithm \textsc{Viterbi}(x, n, t, e, m)

0. \hspace{1cm} T(0, 0) = 1; \hspace{0.5cm} P(0, 0) = 0;
1. \hspace{1cm} \textbf{for} \hspace{0.5cm} j = 1 \hspace{0.5cm} \textbf{to} \hspace{0.5cm} m \\
2. \hspace{1cm} \hspace{1cm} T(j, 0) = 0; \\
3. \hspace{1cm} \textbf{for} \hspace{0.5cm} k = 1 \hspace{0.5cm} \textbf{to} \hspace{0.5cm} n \hspace{1cm} \text{for every prefix upto kth symbol} \\
4. \hspace{1cm} \hspace{1cm} \textbf{for} \hspace{0.5cm} j = 1 \hspace{0.5cm} \textbf{to} \hspace{0.5cm} m \hspace{1cm} \text{for every ‘last state’ } s_j \\
5. \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \text{premax} = 0 \hspace{1cm} \text{initialize the prefix probability} \\
6. \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \textbf{for} \hspace{0.5cm} i = 0 \hspace{0.5cm} \textbf{to} \hspace{0.5cm} m \hspace{1cm} \text{try every ‘predecessor state’ } s_i \\
7. \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \textbf{if} \hspace{0.5cm} \text{premax} < T(i, k - 1) \times t(i, j) \times e(j, x_k) \\
8. \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \textbf{premax} = T(i, k - 1) \times t(i, j) \times e(j, x_k) \\

\begin{center}
\textbf{Time complexity:} \hspace{1cm} \mathcal{O}(nm^2)
\end{center}
Chapter 15. Dynamic Programming

Algorithm VITERBI(x, n, t, e, m)

0. \( T(0, 0) = 1; \ P(0, 0) = 0; \)
1. for \( j = 1 \) to \( m \)
2. \( T(j, 0) = 0; \)
3. for \( k = 1 \) to \( n \) for every prefix upto \( k \)th symbol
4. for \( j = 1 \) to \( m \) for every ‘last state’ \( s_j \)
5. \( \text{premax} = 0 \) initialize the prefix probability
6. for \( i = 0 \) to \( m \) try every ‘predecessor state’ \( s_i \)
7. if \( \text{premax} < T(i, k - 1) \times t(i, j) \times e(j, x_k) \)
8. \( \text{premax} = T(i, k - 1) \times t(i, j) \times e(j, x_k) \) update probability
9. \( P(j, k) = i \) memorize the predecessor
10. \( T(j, k) = \text{premax} \)
11. \( \text{laststate} = 1; \text{max} = T(1, n) \)
12. for \( j = 2 \) to \( m \) identify maximum probability
13. if \( \text{max} < T(j, n) \)
14. \( \text{laststate} = j; \text{max} = T(j, n) \)
15. return \((T, P, \text{max}, \text{laststate})\)

Time complexity: \( O(nm^2) \)
Algorithm VITERBI($x, n, t, e, m$)

0. $T(0,0) = 1; P(0,0) = 0$
1. for $j = 1$ to $m$
2. $T(j,0) = 0$
3. for $k = 1$ to $n$ for every prefix upto $k$th symbol
4. for $j = 1$ to $m$ for every ‘last state’ $s_j$
5. premax = 0 initialize the prefix probability
6. for $i = 0$ to $m$ try every ‘predecessor state’ $s_i$
7. if premax < $T(i,k - 1) \times t(i,j) \times e(j,x_k)$
8. premax = $T(i,k - 1) \times t(i,j) \times e(j,x_k)$ update probability
9. $P(j,k) = i$
10. $T(j,k) = \text{premax}$
11. laststate = 1 max = $T(1,n)$
12. for $j = 2$ to $m$ identify maximum probability
13. if max < $T(j,n)$
14. laststate = $j$
15. max = $T(j,n)$
16. return ($T, P, \text{max}, \text{laststate}$)

Time complexity: $O(nm^2)$
Algorithm \textsc{Viterbi}(x, n, t, e, m)

0. \( T(0, 0) = 1; \ P(0, 0) = 0; \)
1. \textbf{for} \( j = 1 \) \textbf{to} \( m \)
2. \( T(j, 0) = 0; \)
3. \textbf{for} \( k = 1 \) \textbf{to} \( n \) \begin{itemize}
   \item for every prefix upto \( k \)th symbol
5. \( \text{premax} = 0 \)
6. \textbf{for} \( i = 0 \) \textbf{to} \( m \) \begin{itemize}
   \item try every 'predecessor state' \( s_i \)
   \item if \( \text{premax} < T(i, k - 1) \times t(i, j) \times e(j, x_k) \)
8. \( \text{premax} = T(i, k - 1) \times t(i, j) \times e(j, x_k) \) \textbf{update probability}
9. \( P(j, k) = i \) \textbf{memorize the predecessor}
\end{itemize}
\end{itemize}
10. \( T(j, k) = \text{premax} \)
11. \( \text{laststate} = 1; \ \text{max} = T(1, n) \)
12. \textbf{for} \( j = 2 \) \textbf{to} \( m \) \begin{itemize}
   \item identify maximum probability
   \item if \( \text{max} < T(j, n) \)
14. \( \text{laststate} = j \)
15. \( \text{max} = T(j, n) \)
16. \textbf{return} \((T, P, \text{max}, \text{laststate})\)

Time complexity: \( O(mn^2) \)
Algorithm VITERBI \((x, n, t, e, m)\)

0. \(T(0, 0) = 1; P(0, 0) = 0\);
1. \(\text{for } j = 1 \text{ to } m\)
2. \(T(j, 0) = 0;\)
3. \(\text{for } k = 1 \text{ to } n \text{ for every prefix upto } k\text{th symbol}\)
4. \(\text{for } j = 1 \text{ to } m \text{ for every } \text{last state } s_j\)
5. \(\text{premax} = 0 \text{ initialize the prefix probability}\)
6. \(\text{for } i = 0 \text{ to } m \text{ try every } \text{predecessor state } s_i\)
7. \(\text{if premax} < T(i, k - 1) \times t(i, j) \times e(j, x_k)\)
8. \(\text{premax} = T(i, k - 1) \times t(i, j) \times e(j, x_k) \text{ update probability}\)
9. \(P(j, k) = i \text{ memorize the predecessor}\)
10. \(T(j, k) = \text{premax}\)
Algorithm VITERBI\(x, n, t, e, m\)

0. \(T(0, 0) = 1; P(0, 0) = 0;\)
1. \(for\ j = 1\ to\ m\)
2. \(T(j, 0) = 0;\)
3. \(\text{for} \ k = 1\ \text{to} \ n\) \(\text{for every prefix upto kth symbol}\)
4. \(\text{for} \ j = 1\ \text{to} \ m\) \(\text{for every ‘last state’} \ s_j\)
5. \(\text{premax} = 0\) \(\text{initialize the prefix probability}\)
6. \(\text{for} \ i = 0\ \text{to} \ m\) \(\text{try every ‘predecessor state’} \ s_i\)
7. \(\text{if} \ \text{premax} < T(i, k - 1) \times t(i, j) \times e(j, x_k)\)
8. \(\text{premax} = T(i, k - 1) \times t(i, j) \times e(j, x_k)\) \(\text{update probability}\)
9. \(P(j, k) = i\) \(\text{memorize the predecessor}\)
10. \(T(j, k) = \text{premax}\)
11. \(\text{laststate} = 1; \ \text{max} = T(1, n)\)

Time complexity: \(O(nm^2)\)
Algorithm VITERBI$(x, n, t, e, m)$

0. $T(0, 0) = 1; P(0, 0) = 0$
1. for $j = 1$ to $m$
2. $T(j, 0) = 0$
3. for $k = 1$ to $n$ for every prefix upto $k$th symbol
4. for $j = 1$ to $m$ for every ‘last state’ $s_j$
5. $premax = 0$ initialize the prefix probability
6. for $i = 0$ to $m$ try every ‘predecessor state’ $s_i$
7. if $premax < T(i, k - 1) \times t(i, j) \times e(j, x_k)$
8. $premax = T(i, k - 1) \times t(i, j) \times e(j, x_k)$ update probability
9. $P(j, k) = i$ memorize the predecessor
10. $T(j, k) = premax$
11. laststate = 1; max = $T(1, n)$
12. for $j = 2$ to $m$ identify maximum probability
Chapter 15. Dynamic Programming

Algorithm VITERBI\((x, n, t, e, m)\)

0. \(T(0, 0) = 1; P(0, 0) = 0;\)
1. for \(j = 1\) to \(m\)
2. \(T(j, 0) = 0;\)
3. for \(k = 1\) to \(n\) for every prefix upto \(k\)th symbol
4. for \(j = 1\) to \(m\) for every ‘last state’ \(s_j\)
5. \(\text{premax} = 0\) initialize the prefix probability
6. for \(i = 0\) to \(m\) try every ‘predecessor state’ \(s_i\)
7. \(\text{if premax} < T(i, k - 1) \times t(i, j) \times e(j, x_k)\)
8. \(\text{premax} = T(i, k - 1) \times t(i, j) \times e(j, x_k)\) update probability
9. \(P(j, k) = i\) memorize the predecessor
10. \(T(j, k) = \text{premax}\)
11. \(\text{laststate} = 1; \text{max} = T(1, n)\)
12. for \(j = 2\) to \(m\) identify maximum probability
13. if \(\text{max} < T(j, n)\)
Chapter 15. Dynamic Programming

Algorithm VITERBI($x, n, t, e, m$)

0. $T(0, 0) = 1; P(0, 0) = 0$;
1. for $j = 1$ to $m$
2. 
3. for $k = 1$ to $n$ for every prefix upto $k$th symbol
4. 
5. premax = 0 initialize the prefix probability
6. for $i = 0$ to $m$ try every 'predecessor state' $s_i$
7. if premax < $T(i, k - 1) \times t(i, j) \times e(j, x_k)$
8. premax = $T(i, k - 1) \times t(i, j) \times e(j, x_k)$ update probability
9. 
10. $T(j, k) = \text{premax}$
11. $laststate = 1; \text{max} = T(1, n)$
12. for $j = 2$ to $m$ identify maximum probability
13. if max < $T(j, n)$
14. $laststate = j$;
Algorithm VITERBI(x, n, t, e, m)

0. $T(0, 0) = 1; P(0, 0) = 0$
1. for $j = 1$ to $m$
2. $T(j, 0) = 0$
3. for $k = 1$ to $n$ for every prefix up to $k$th symbol
4. for $j = 1$ to $m$ for every ‘last state’ $s_j$
5. $\text{premax} = 0$ initialize the prefix probability
6. for $i = 0$ to $m$ try every ‘predecessor state’ $s_i$
7. if $\text{premax} < T(i, k - 1) \times t(i, j) \times e(j, x_k)$
8. $\text{premax} = T(i, k - 1) \times t(i, j) \times e(j, x_k)$ update probability
9. $P(j, k) = i$ memorize the predecessor
10. $T(j, k) = \text{premax}$
11. $\text{laststate} = 1; \text{max} = T(1, n)$
12. for $j = 2$ to $m$ identify maximum probability
13. if $\text{max} < T(j, n)$
14. $\text{laststate} = j$
15. $\text{max} = T(j, n)$

Time complexity: $O(nm^2)$
Algorithm VITERBI\((x, n, t, e, m)\)

0. \(T(0, 0) = 1; \ P(0, 0) = 0;\)
1. \(\text{for } j = 1 \text{ to } m\)
2. \(T(j, 0) = 0;\)
3. \(\text{for } k = 1 \text{ to } n\) \(\text{for every prefix upto } k\text{th symbol}\)
4. \(\text{for } j = 1 \text{ to } m\) \(\text{for every 'last state' } s_j\)
5. \(\text{premax} = 0\) \(\text{initialize the prefix probability}\)
6. \(\text{for } i = 0 \text{ to } m\) \(\text{try every 'predecessor state' } s_i\)
7. \(\text{if } \text{premax} < T(i, k - 1) \times t(i, j) \times e(j, x_k)\)
8. \(\text{premax} = T(i, k - 1) \times t(i, j) \times e(j, x_k)\) \(\text{update probability}\)
9. \(P(j, k) = i\) \(\text{memorize the predecessor}\)
10. \(T(j, k) = \text{premax}\)
11. \(\text{laststate} = 1; \ \text{max} = T(1, n)\)
12. \(\text{for } j = 2 \text{ to } m\) \(\text{identify maximum probability}\)
13. \(\text{if } \text{max} < T(j, n)\)
14. \(\text{laststate} = j;\)
15. \(\text{max} = T(j, n)\)
16. \(\text{return } (T, P, \text{max}, \text{laststate})\)

Time complexity: \(O(nm^2)\)
Chapter 15. Dynamic Programming

Algorithm VITERBI($x, n, t, e, m$)

0. \( T(0, 0) = 1; \) \( P(0, 0) = 0; \)
1. for \( j = 1 \) to \( m \)
2. \( T(j, 0) = 0; \)
3. for \( k = 1 \) to \( n \) for every prefix upto \( k \)th symbol
4. for \( j = 1 \) to \( m \) for every ‘last state’ \( s_j \)
5. \( \text{premax} = 0 \) initialize the prefix probability
6. for \( i = 0 \) to \( m \) try every ‘predecessor state’ \( s_i \)
7. if \( \text{premax} < T(i, k - 1) \times t(i, j) \times e(j, x_k) \)
8. \( \text{premax} = T(i, k - 1) \times t(i, j) \times e(j, x_k) \) update probability
9. \( P(j, k) = i \) memorize the predecessor
10. \( T(j, k) = \text{premax} \)
11. \( \text{laststate} = 1; \) \( \text{max} = T(1, n) \)
12. for \( j = 2 \) to \( m \) identify maximum probability
13. if \( \text{max} < T(j, n) \)
14. \( \text{laststate} = j; \)
15. \( \text{max} = T(j, n) \)
16. return \( (T, P, \text{max}, \text{laststate}) \)

Time complexity:
Chapter 15. Dynamic Programming

Algorithm VITERBI($x, n, t, e, m$)

0. $T(0, 0) = 1; P(0, 0) = 0$;
1. for $j = 1$ to $m$
2. $T(j, 0) = 0$;
3. for $k = 1$ to $n$ for every prefix upto $k$th symbol
4. for $j = 1$ to $m$ for every 'last state' $s_j$
5. premax = 0 initialize the prefix probability
6. for $i = 0$ to $m$ try every 'predecessor state' $s_i$
7. if premax < $T(i, k - 1) \times t(i, j) \times e(j, x_k)$
8. premax = $T(i, k - 1) \times t(i, j) \times e(j, x_k)$ update probability
9. $P(j, k) = i$ memorize the predecessor
10. $T(j, k) = \text{premax}$
11. laststate = 1; max = $T(1, n)$
12. for $j = 2$ to $m$ identify maximum probability
13. if max < $T(j, n)$
14. laststate = $j$
15. max = $T(j, n)$
16. return $(T, P, \text{max}, \text{laststate})$

Time complexity: $O(nm^2)$
Use the computed table $P$ and $\text{laststate}$ to traceback the hidden states.
Use the computed table $P$ and $laststate$ to traceback the hidden states

Algorithm TraceHiddenStates($P, j, k, x$)

1. if $j > 0$
Use the computed table $P$ and $\text{laststate}$ to traceback the hidden states

Algorithm $\text{TraceHiddenStates}(P, j, k, x)$

1. if $j > 0$
2. $\text{TraceHiddenStates}(P, P[j, k], k - 1, x)$
Use the computed table $P$ and laststate to traceback the hidden states

Algorithm $\text{TraceHiddenStates}(P, j, k, x)$

1. if $j > 0$
2. $\text{TraceHiddenStates}(P, P[j, k], k - 1, x)$
3. print ('State' $j$ 'emits symbol' $x_k$)
Use the computed table $P$ and $laststate$ to traceback the hidden states.

Algorithm $\text{TraceHiddenStates}(P, j, k, x)$

1. if $j > 0$
2. $\text{TraceHiddenStates}(P, P[j, k], k - 1, x)$
3. print ('State' $j$ 'emits symbol' $x_k$)

First, call $\text{TraceHiddenStates}(P, laststate, n, x)$
Chapter 15. Dynamic Programming

There are other DP algorithms associated with HMMs.
Chapter 15. Dynamic Programming

There are other DP algorithms associated with HMMs.

- **Viterbi** algorithm belongs to the type of forward algorithms because it computes for prefixes of the given symbol sequence.
Chapter 15. Dynamic Programming

There are other DP algorithms associated with HMMs.

- **Viterbi** algorithm belongs to the type of **forward** algorithms because it computes for prefixes of the given symbol sequence.

  A DP algorithm can be designed so it computes for suffixes of the given symbols sequence, a **backward** algorithm.

\[
\begin{align*}
\text{Algorithm:} & \\
\text{How to find objective function?} & \\
\text{Recalled for forward, we defined} & \\
\quad f(j,k) &= \max_{y=s_0...s_j} \{ \text{Prob}(y, x_1...x_k) \} \\
\text{\(f(j,k)\) is the maximum probability to generate prefix \(x_1...x_k\) beginning from state \(s_0\) and ending with state \(s_j\);} & \\
\text{For backward, we may define} & \\
\quad b(l,k) &= \max_{y=s_l...s_\omega} \{ \text{Prob}(y, x_k...x_n) \} \\
\text{\(b(l,k)\) is the maximum probability to generate suffix \(x_k...x_n\) beginning from state \(s_l\) and ending with state \(s_\omega\);} & 
\end{align*}
\]
Chapter 15. Dynamic Programming

There are other DP algorithms associated with HMMs.

- **Viterbi** algorithm belongs to the type of forward algorithms because it computes for prefixes of the given symbol sequence.

  A DP algorithm can be designed so it computes for suffixes of the given symbols sequence, a backward algorithm.

**Algorithm IBRETIV**
Chapter 15. Dynamic Programming

There are other DP algorithms associated with HMMs.

- **Viterbi** algorithm belongs to the type of forward algorithms because it computes for prefixes of the given symbol sequence.

  A DP algorithm can be designed so it computes for suffixes of the given symbols sequence, a backward algorithm.

**Algorithm I$\mathbf{BRETIV}$**

How to find objective function?
Recalled for forward, we defined

\[
f(j, k) = \max_{y=s_0 \ldots s_j} \{\text{Prob}(y, x_1 \ldots x_k)\}
\]

\(f(j, k)\) is the maximum probability to generate prefix \(x_1 \ldots x_k\) beginning from state \(s_0\) and ending with state \(s_j\);
Chapter 15. Dynamic Programming

There are other DP algorithms associated with HMMs.

- **Viterbi** algorithm belongs to the type of forward algorithms because it computes for prefixes of the given symbol sequence.

  A DP algorithm can be designed so it computes for suffixes of the given symbols sequence, a backward algorithm.

**Algorithm I**

**RETIV**

How to find objective function?
Recalled for forward, we defined

\[
f(j, k) = \max_{y = s_0 \ldots s_j} \{\text{Prob}(y, x_1 \ldots x_k)\}
\]

\(f(j, k)\) is the maximum probability to generate prefix \(x_1 \ldots x_1\) beginning from state \(s_0\) and ending with state \(s_j\);

For backward, we may define

\[
b(l, k) = \max_{y = s_l \ldots s_\omega} \{\text{Prob}(y, x_k \ldots x_n)\}
\]

\(b(l, k)\) is the maximum probability to generate suffix \(x_k \ldots x_n\) beginning from state \(s_l\) and ending with state \(s_\omega\);
We need to amend the HMM model with a silent state $s_\omega$;

$$b(l, k) = \max_{y=s_l \ldots s_\omega} \{ \text{Prob}(y, x_k \ldots x_n) \}$$
Chapter 15. Dynamic Programming

- We need to amend the HMM model with a silent state $s_\omega$;

  $$b(l, k) = \max_{y=s_1 \ldots s_\omega} \{ \text{Prob}(y, x_k \ldots x_n) \}$$

- Recurrence (draw a diagram)

  $$b(l, k) =$$
Chapter 15. Dynamic Programming

- We need to amend the HMM model with a silent state $s_\omega$;

$$b(l, k) = \max_{y=s_1 \ldots s_\omega} \{\text{Prob}(y, x_k \ldots x_n)\}$$

- Recurrence (draw a diagram)

$$b(l, k) = \max_i$$
Chapter 15. Dynamic Programming

- We need to amend the HMM model with a silent state $s_\omega$

\[ b(l, k) = \max_{y = s_1 \ldots s_\omega} \{ \text{Prob}(y, x_k \ldots x_n) \} \]

- Recurrence (draw a diagram)

\[ b(l, k) = \max_i \{ b(i, k + 1) \]
Chapter 15. Dynamic Programming

- We need to amend the HMM model with a silent state $s_\omega$;

\[ b(l, k) = \max_{y=s_l \ldots s_\omega} \{ Prob(y, x_k \ldots x_n) \} \]

- Recurrence (draw a diagram)

\[ b(l, k) = \max_i \{ b(i, k + 1) \times t(l, i) \} \]
Chapter 15. Dynamic Programming

• We need to amend the HMM model with a silent state $s_\omega$;

$$b(l, k) = \max_{y=s_l \ldots s_\omega} \{ \text{Prob}(y, x_k \ldots x_n) \}$$

• Recurrence (draw a diagram)

$$b(l, k) = \max_i \{ b(i, k + 1) \times t(l, i) \times e(l, x_k) \}$$

the larger the $k$ value, the shortest the suffix is
Chapter 15. Dynamic Programming

• We need to amend the HMM model with a silent state $s_\omega$;

\[
b(l, k) = \max_{y=s_1...s_\omega} \{Prob(y, x_k ... x_n)\}
\]

• Recurrence (draw a diagram)

\[
b(l, k) = \max_i \{b(i, k + 1) \times t(l, i) \times e(l, x_k)\}
\]

the larger the $k$ value, the shortest the suffix is

• Base cases:
Chapter 15. Dynamic Programming

• We need to amend the HMM model with a silent state \( s_\omega \);

\[
b(l, k) = \max_{y=s_l \ldots s_\omega} \{ Prob(y, x_k \ldots x_n) \}
\]

• Recurrence (draw a diagram)

\[
b(l, k) = \max_i \{ b(i, k + 1) \times t(l, i) \times e(l, x_k) \}
\]

the larger the \( k \) value, the shortest the suffix is

• Base cases: \( b(s_\omega, \)
Chapter 15. Dynamic Programming

• We need to amend the HMM model with a silent state $s_\omega$;

$$b(l, k) = \max_{y=s_l \ldots s_\omega} \{ \text{Prob}(y, x_k \ldots x_n) \}$$

• Recurrence (draw a diagram)

$$b(l, k) = \max_i \{ b(i, k+1) \times t(l, i) \times e(l, x_k) \}$$

the larger the $k$ value, the shortest the suffix is

• Base cases: $b(s_\omega, n+1) =$
Chapter 15. Dynamic Programming

- We need to amend the HMM model with a silent state $s_\omega$;

$$b(l, k) = \max_{y=s_1 \ldots s_\omega} \{Prob(y, x_k \ldots x_n)\}$$

- Recurrence (draw a diagram)

$$b(l, k) = \max_i \{b(i, k + 1) \times t(l, i) \times e(l, x_k)\}$$

  the larger the $k$ value, the shortest the suffix is

- Base cases: $b(s_\omega, n + 1) = 1$, 
Chapter 15. Dynamic Programming

- We need to amend the HMM model with a silent state $s_\omega$;

$$b(l, k) = \max_{y=s_l \ldots s_\omega} \{Prob(y, x_k \ldots x_n)\}$$

- Recurrence (draw a diagram)

$$b(l, k) = \max_i \{b(i, k + 1) \times t(l, i) \times e(l, x_k)\}$$

the larger the $k$ value, the shortest the suffix is

- Base cases: $b(s_\omega, n + 1) = 1$, $b(j, n + 1) = 0$ for all $j \neq \omega$.

$k = n + 1$ is the case when the suffix is empty.
Chapter 15. Dynamic Programming

- We need to amend the HMM model with a silent state $s_\omega$;

$$b(l, k) = \max_{y=s_1...s_\omega} \{\text{Prob}(y, x_k ... x_n)\}$$

- Recurrence (draw a diagram)

$$b(l, k) = \max_i \{b(i, k + 1) \times t(l, i) \times e(l, x_k)\}$$

the larger the $k$ value, the shortest the suffix is

- Base cases: $b(s_\omega, n + 1) = 1$, $b(j, n + 1) = 0$ for all $j \neq \omega$.

$k = n + 1$ is the case when the suffix is empty.

What is the relationship between $\max_j \{f(j, n)\}$ and $\max_l \{b(l, 1)\}$?
Chapter 15. Dynamic Programming

- We need to amend the HMM model with a silent state $s_\omega$;

$$b(l, k) = \max_{y=s_l \ldots s_\omega} \{ Prob(y, x_k \ldots x_n) \}$$

- Recurrence (draw a diagram)

$$b(l, k) = \max_i \{ b(i, k + 1) \times t(l, i) \times e(l, x_k) \}$$

the larger the $k$ value, the shortest the suffix is

- Base cases: $b(s_\omega, n + 1) = 1$, $b(j, n + 1) = 0$ for all $j \neq \omega$.

$k = n + 1$ is the case when the suffix is empty.

What is the relationship between $\max_j \{ f(j, n) \}$ and $\max_l \{ b(l, 1) \}$? (draw diagram to show)
Chapter 15. Dynamic Programming

Computing the total probability (instead of the maximum prob)
Chapter 15. Dynamic Programming

Computing the total probability (instead of the maximum prob)

- often we are interested in knowing the chance for a given sequence of symbols to be produced by a model.
Chapter 15. Dynamic Programming

Computing the total probability (instead of the maximum prob)

- often we are interested in knowing the chance for a given sequence of symbols to be produced by a model.
  
  e.g., used in discrimination against background
Chapter 15. Dynamic Programming

Computing the total probability (instead of the maximum prob)

• often we are interested in knowing the chance for a given sequence of symbols to be produced by a model.

  e.g., used in discrimination against background

• given a HMM $\mathcal{M}$, and sequence $x \in \Sigma^*$, what is the probability

  $$\text{Prob}(x|\mathcal{M})$$
Computing the total probability (instead of the maximum prob)

- often we are interested in knowing the chance for a given sequence of symbols to be produced by a model.
  e.g., used in discrimination against background
- given a HMM $\mathcal{M}$, and sequence $x \in \Sigma^*$, what is the probability

$$Prob(x|\mathcal{M}) = \sum_y Prob(y, x|\mathcal{M})$$
Chapter 15. Dynamic Programming

Computing the total probability (instead of the maximum prob)

- often we are interested in knowing the chance for a given sequence of symbols to be produced by a model.
  e.g., used in discrimination against background
- given a HMM $M$, and sequence $x \in \Sigma^*$, what is the probability

$$Prob(x|M) = \sum_y Prob(y, x|M)$$

- consider prefix, $x_1 \ldots x_k$ and state $s_j$, define:
Computing the total probability (instead of the maximum prob)

• often we are interested in knowing the chance for a given sequence of symbols to be produced by a model.

  e.g., used in discrimination against background

• given a HMM $M$, and sequence $x \in \Sigma^*$, what is the probability

  $$\text{Prob}(x|M) = \sum_y \text{Prob}(y,x|M)$$

• consider prefix, $x_1 \ldots x_k$ and state $s_j$, define:

  $p(j,k)$ to be the total probability for $M$ to generate prefix $x_1 \ldots x_k$ ending at state $s_j$. 

Chapter 15. Dynamic Programming

Define $p(j, k)$ to be the total probability for $M$ to generate prefix $x_1 \ldots x_k$ ending at state $s_j$. 

\[
p(j, k) = \sum_i p(i, k-1) \times t(i, j) \times e(j, x_k)
\]

with base cases:

$p(0, 0) = 1$, and $p(j, 0) = 0$ for all $j \geq 1$. 

Define $p(j, k)$ to be the total probability for $\mathcal{M}$ to generate prefix $x_1 \ldots x_k$ ending at state $s_j$.

$$p(j, k) = \sum_i p(i, k - 1) \times t(i, j) \times e(j, x_k)$$

with base cases: $p(0, 0) = 1$, and $p(j, 0) = 0$ for all $j \geq 1$. 
Chapter 15. Dynamic Programming

Implementation of the total probability algorithm

• almost the same as for Viterbi algorithm.
• applications, for example
  - screen a stream of signals (symbols) for occasional but desired pattern
  - use HMM to model the desired pattern
  - scanning window on the long stream; computes the total probability
  - report segments with 'significant' probability values

but what does it mean by 'significant probability values'?
Implementation of the total probability algorithm

• almost the same as for Viterbi algorithm.

• applications, for example
  - screen a stream of signals (symbols) for occasional but desired pattern
  - use HMM to model the desired pattern
  - scanning window on the long stream; computes the total probability
  - report segments with 'significant' probability values

but what does it mean by 'significant probability values'?
Chapter 15. Dynamic Programming

Implementation of the total probability algorithm

• almost the same as for Viterbi algorithm.

• applications, for example

but what does it mean by 'significant probability values'?
Implementation of the total probability algorithm

- almost the same as for Viterbi algorithm.
- applications, for example
  - screen a stream of signals (symbols) for occasional but desired pattern
Implementation of the total probability algorithm

- almost the same as for Viterbi algorithm.

- applications, for example
  - screen a stream of signals (symbols) for occasional but desired pattern
  - use HMM to model the desired pattern

- what does it mean by 'significant probability values'?
Implementation of the total probability algorithm

- almost the same as for Viterbi algorithm.
- applications, for example
  - screen a stream of signals (symbols) for occasional but desired pattern
  - use HMM to model the desired pattern
  - scanning window on the long stream; computes the total probability

But what does it mean by 'significant probability values'?
Chapter 15. Dynamic Programming

Implementation of the total probability algorithm

• almost the same as for Viterbi algorithm.

• applications, for example
  - screen a stream of signals (symbols) for occasional but desired pattern
  - use HMM to model the desired pattern
  - scanning window on the long stream; computes the total probability
  - report segments with ‘significant’ probability values
Chapter 15. Dynamic Programming

Implementation of the total probability algorithm

- almost the same as for Viterbi algorithm.
- applications, for example
  - screen a stream of signals (symbols) for occasional but desired pattern
  - use HMM to model the desired pattern
  - scanning window on the long stream; computes the total probability
  - report segments with ‘significant’ probability values

  but what does it mean by ’significant probability values’?
Chapter 15. Dynamic Programming

Necessary elements for problems solvable with DP

(1) it is an optimization problem with an objective function to optimize by solutions
   e.g., solution: a path of stations, objective function: time

(2) solution has optimal substructure solutions are recursively definable
   e.g., a fastest path consists of other fastest subpaths

(3) overlapping subproblems e.g., two or more paths share a subpath.
Chapter 15. Dynamic Programming

Necessary elements for problems solvable with DP

1. It is an optimization problem
Chapter 15. Dynamic Programming

Necessary elements for problems solvable with DP

(1) it is an optimization problem

   with an objective function to optimize by solutions

   e.g., solution: a path of stations,
   objective function: time
Chapter 15. Dynamic Programming

Necessary elements for problems solvable with DP

(1) it is an optimization problem
   with an objective function to optimize by solutions
   e.g., solution: a path of stations,
   objective function: time

(2) solution has optimal substructure
Chapter 15. Dynamic Programming

Necessary elements for problems solvable with DP

(1) It is an optimization problem
   with an objective function to optimize by solutions
   e.g., solution: a path of stations,
   objective function: time

(2) Solution has optimal substructure
   solutions are recursively definable
   e.g., a fastest path consists of other fastest subpaths
Chapter 15. Dynamic Programming

Necessary elements for problems solvable with DP

(1) it is an optimization problem
   with an objective function to optimize by solutions
   e.g., solution: a path of stations,
     objective function: time

(2) solution has optimal substructure
   solutions are recursively definable
   e.g., a fastest path consists of other fastest subpaths

(3) overlapping subproblems
   e.g., two or more paths share a subpath.
Chapter 15. Dynamic Programming

Longest Common Subsequence (LCS)
Longest Common Subsequence (LCS)

\[ x = \text{ACCGGTCGAGTGCG} \]
\[ y = \text{GTCGTTCGGATGCC} \]
Chapter 15. Dynamic Programming

Longest Common Subsequence (LCS)

\[ x = \text{ACCGGTCCGAGTGCG} \]
\[ y = \text{GTCGTTCGGATGCCC} \]

To see how much the two sequences are related, we may examine how much the two are in common.
Chapter 15. Dynamic Programming

Longest Common Subsequence (LCS)

\[
x = \text{ACCGGTTCGAGTGCG}
\]
\[
y = \text{GTCGTTCGGATGCCC}
\]

To see how much the two sequences are related, we may examine how much the two are in common.

e.g., a common subsequence for the two sequences
Chapter 15. Dynamic Programming

Longest Common Subsequence (LCS)

\[ x = \text{ACCGGT}CGAGTGCG \]
\[ y = \text{GTCGTT}CGGATGCCC \]

To see how much the two sequences are related, we may examine how much the two are in common.

e.g., a common subsequence for the two sequences

\[ \text{ACCGG}T\text{TCGAGTGCG} \]
\[ \text{CGTCG}ATGC \quad \leftarrow \text{common sequence} \]
\[ \text{GTCGTT}CGGATGCCC \]
Chapter 15. Dynamic Programming

Longest Common Subsequence (LCS)

\[ x = \text{ACCGGTCGAGTGCG} \]
\[ y = \text{GTCGTTCGGATGCCC} \]

To see how much the two sequences are related, we may examine how much the two are in common.

e.g., a common subsequence for the two sequences

\[ \text{ACCGGTCGAGTGCG} \]
\[ \text{CGTCGATGC} \quad \leftarrow \text{common sequence} \]
\[ \text{GTCGTTCGGATGCCC} \]

significance of common sequence, viewed as:
Chapter 15. Dynamic Programming

Longest Common Subsequence (LCS)

\[ x = \text{ACCGGTTCGAGTGCG} \]
\[ y = \text{GTCGTTTCGGATGCCC} \]

To see how much the two sequences are related, we may examine how much the two are in common.

e.g., a common subsequence for the two sequences

\[
\begin{array}{c}
\text{ACCGGTTCGAGTGCG} \\
\text{CGTCGATGC} \quad \leftarrow \text{common sequence} \\
\text{GTCGTTTCGGATGCCC}
\end{array}
\]

significance of common sequence, viewed as:

\[
\begin{array}{c}
\text{ACCGGTTC} \quad \text{GAGTGCG} \\
\| \| \| \| \| \| \| \| \| \| \| \| \| \\
\text{CG} \quad \text{TC} \quad \text{GA} \quad \text{TGC} \quad \leftarrow \text{common sequence} \\
\| \| \| \| \| \| \| \| \| \| \| \| \| \\
\text{GTCGTTTCGGA} \quad \text{TGCCCC}
\end{array}
\]
Let $X = x_1 x_2 \cdots x_m$ be a sequence, $x_i \in \Sigma$. 
Let $X = x_1x_2\cdots x_m$ be a sequence, $x_i \in \Sigma$.

$Z = z_1z_2\cdots z_k$ is a subsequence of $X$
Chapter 15. Dynamic Programming

Let $X = x_1 x_2 \cdots x_m$ be a sequence, $x_i \in \Sigma$.

$Z = z_1 z_2 \cdots z_k$ is a subsequence of $X$

if there are indexes $i_1 < i_2 < \cdots , < i_k$ of $X$ such that
Chapter 15. Dynamic Programming

Let $X = x_1 x_2 \cdots x_m$ be a sequence, $x_i \in \Sigma$.

$Z = z_1 z_2 \cdots z_k$ is a subsequence of $X$ if there are indexes $i_1 < i_2 < \cdots < i_k$ of $X$ such that $z_j = x_{i_j}$ for $j = 1, \cdots, k$. 

Chapter 15. Dynamic Programming

Let \( X = x_1 x_2 \cdots x_m \) be a sequence, \( x_i \in \Sigma \).

\( Z = z_1 z_2 \cdots z_k \) is a \textit{subsequence} of \( X \)
if there are indexes \( i_1 < i_2 < \cdots < i_k \) of \( X \) such that
\[
z_j = x_{i_j} \quad \text{for} \quad j = 1, \cdots, k.
\]

\[
\begin{array}{cccccccccc}
34 & 6789 & 11 & 12 & 13 \\
X = & ACCGGTCGAGTGCG \\
| \hspace{1cm} | \hspace{1cm} | \hspace{1cm} | \hspace{1cm} | \\
Z = & CG & TCGA & TGC \\
12 & 3456 & 789
\end{array}
\]
Chapter 15. Dynamic Programming

Let $X = x_1 x_2 \cdots x_m$ be a sequence, $x_i \in \Sigma$.

$Z = z_1 z_2 \cdots z_k$ is a subsequence of $X$ if there are indexes $i_1 < i_2 < \cdots < i_k$ of $X$ such that $z_j = x_{i_j}$ for $j = 1, \cdots, k$.

$X = \text{ACCGGTCGAGTGCG}$

\[ 34 \ 6789 \ 11^{12}_{13} \]

$Z = \text{CG TCGA TGC}$

\[ 12 \ 3456 \ 789 \]

the corresponding indexes in $X$ are:

$3 < 4 < 6 < 7 < 8 < 9 < 11 < 12 < 13$
Chapter 15. Dynamic Programming

$Z$ is a *common subsequence* of $X$ and $Y$
if $Z$ is a subsequence of $X$ and $Z$ is a subsequence of $Y$.
Chapter 15. Dynamic Programming

A common subsequence $Z$ of $X$ and $Y$ is a subsequence of $X$ and $Z$ is a subsequence of $Y$.

\begin{align*}
\text{ACCGGTC} & \quad \text{GAGTGC} \\
\text{CG} & \quad \text{TC} \quad \text{GA} \quad \text{TGC} \quad \leftarrow \text{common sequence} \\
\text{GTCGTTCGGA} & \quad \text{TGCC}
\end{align*}
Chapter 15. Dynamic Programming

Z is a common subsequence of X and Y if Z is a subsequence of X and Z is a subsequence of Y.

\[
\begin{align*}
\text{ACCGGTC} & \quad \text{GAGTGC}G \\
\ | & \quad | & \quad | & \quad | & \quad | & \quad | & \quad |
\end{align*}
\]

\[
\begin{align*}
\text{CG TC GA TGC} & \quad \text{← common sequence} \\
\ | & \quad | & \quad | & \quad | & \quad | & \quad |
\end{align*}
\]

\[
\begin{align*}
\text{GTCGTTCGGA} & \quad \text{TGCCC} \\
\end{align*}
\]

Longest Common Subsequence (LCS) problem:

**Input:** \( X = x_1 x_2 \cdots x_m, \ Y = y_1 y_2 \cdots y_n. \)

**Output:** \( Z, \) a common subsequence of \( X \) and \( Y \) such that \( \text{length}(Z) \) is the maximum.
Step 1: Analyze the LCS problem to see if it has the desired features:
Step 1: Analyze the LCS problem to see if it has the desired features:

- optimal substructure
- overlapping subproblems
Step 1: Analyze the LCS problem to see if it has the desired features:

- optimal substructure
- overlapping subproblems
Chapter 15. Dynamic Programming

**Step 1:** Analyze the LCS problem to see if it has the desired features:

- optimal substructure
- overlapping subproblems

We examine prefixes $x_1x_2\ldots x_i, i \leq m$
and $y_1y_2\ldots y_j, j \leq n$
Chapter 15. Dynamic Programming

**Step 1:** Analyze the LCS problem to see if it has the desired features:

- optimal substructure
- overlapping subproblems

We examine prefixes $x_1x_2\ldots x_i$, $i \leq m$ and $y_1y_2\ldots y_j$, $j \leq n$

- how is the **LCS** of $x_1x_2\ldots x_i$ and $y_1y_2\ldots y_j$ related to the **LCS** of shorter prefixes of $X$ and $Y$?
Chapter 15. Dynamic Programming

If $x_i = y_j$, include $x_i$ in the LCS, reducing to LCS for $x_1 \ldots x_{i-1}x_i$ and $y_1 \ldots y_{j-1}y_j$.

If $x_i \neq y_j$, should we abandon either?

Two options:

1. Keep $x_i$ but abandon $y_j$, resulting in prefixes: $x_1 \ldots x_{i-1}x_i$ and $y_1 \ldots y_{j-1}y_j$.

2. Keep $y_j$ but abandon $x_i$, resulting in prefixes: $x_1 \ldots x_{i-1}x_i$ and $y_1 \ldots y_{j-1}y_j$. 
Chapter 15. Dynamic Programming

\[
x_1 \ldots x_{i-1} x_i \\
y_1 \ldots y_{j-1} y_j
\]

if \( x_i = y_j \), include \( x_i \) in the LCS, reducing to LCS for \( x_1 \ldots x_{i-1} \) and \( y_1 \ldots y_{j-1} \).
Chapter 15. Dynamic Programming

\[ x_1 \ldots x_{i-1} x_i \]
\[ y_1 \ldots y_{j-1} y_j \]

if \( x_i = y_j \), include \( x_i \) in the LCS, reducing to LCS for \( x_1 \ldots x_{i-1} \) and \( y_1 \ldots y_{j-1} \)

if \( x_i \neq y_j \), should we abandon either?
Chapter 15. Dynamic Programming

\[ x_1 \ldots x_{i-1}x_i \]
\[ y_1 \ldots y_{j-1}y_j \]

if \( x_i = y_j \), include \( x_i \) in the LCS, reducing to LCS for \( x_1 \ldots x_{i-1} \) and \( y_1 \ldots y_{j-1} \)

\[ x_1 \ldots x_{i-1}x_i \]
\[ y_1 \ldots y_{j-1}y_j \]

if \( x_i \neq y_j \), should we abandon either? two options

(1) keep \( x_i \) but abandon \( y_j \)
    resulting in prefixes: \( x_1x_2 \ldots x_i \) and \( y_1y_2 \ldots y_{j-1} \), or
Chapter 15. Dynamic Programming

\[ x_1 \ldots x_{i-1} x_i \]
\[ y_1 \ldots y_{j-1} y_j \]

if \( x_i = y_j \), include \( x_i \) in the LCS, reducing to LCS for \( x_1 \ldots x_{i-1} \) and \( y_1 \ldots y_{j-1} \)

if \( x_i \neq y_j \), should we abandon either? two options

(1) keep \( x_i \) but abandon \( y_j \)
resulting in prefixes: \( x_1 x_2 \ldots x_i \) and \( y_1 y_2 \ldots y_{j-1} \), or

(2) keep \( y_j \) but abandon \( x_i \)
resulting in prefixes: \( x_1 x_2 \ldots x_{i-1} \) and \( y_1 y_2 \ldots y_j \)
Chapter 15. Dynamic Programming

Let $LCS(i,j)$ be the longest common sequence for $x_1x_2\ldots x_i$ and $y_1y_2\ldots y_j$, then
Chapter 15. Dynamic Programming

Let $LCS(i, j)$ be the longest common sequence for $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_j$, then

- if $x_i = y_j$, then $LCS(i, j) = \text{concat}(LCS(i - 1, j - 1), x_i)$
- if $x_i \neq y_j$, then $LCS(i, j) = LCS(i, j - 1)$ OR $LCS(i, j) = LCS(i - 1, j)$ (depending on which is of a larger length.)

We conclude: LCS problem has the optimal substructure property; LCS problem has overlapping subproblems.
Let $LCS(i, j)$ be the longest common sequence for $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_j$, then

- if $x_i = y_j$, then $LCS(i, j) = \text{concat}(LCS(i - 1, j - 1), x_i)$
- if $x_i \neq y_j$, then

$$LCS(i, j) = LCS(i, j - 1) \text{ OR } LCS(i, j) = LCS(i - 1, j)$$

(depending on which is of a larger length.)
Let $LCS(i, j)$ be the longest common sequence for $x_1x_2\ldots x_i$ and $y_1y_2\ldots y_j$, then

- if $x_i = y_j$, then $LCS(i, j) = \text{concat}(LCS(i - 1, j - 1), x_i)$
- if $x_i \neq y_j$, then

$$LCS(i, j) = LCS(i, j - 1) \text{ OR } LCS(i, j) = LCS(i - 1, j)$$

(depending on which is of a larger length.)

We conclude:

LCS problem has the optimal substructure property;
LCS problem has overlapping subproblems.
Chapter 15. Dynamic Programming

**Step 2**: define objective function and formulate recurrence

Define $l(i,j)$ to be the length of the LCS for prefixes $x_1...x_i$ and $y_1...y_j$.

then

$$l(i,j) = \begin{cases} 
  l(i-1,j-1) + 1 & \text{if } x_i = y_j; \\
  \max\{l(i,j-1), l(i-1,j), l(i-1,j-1)\} & \text{if } x_i \neq y_j
\end{cases}$$

**base cases**: 
- $l(i,0) = 0$, for $0 \leq i \leq m$, either prefix is empty → LCS is empty
- $l(0,j) = 0$, for $0 \leq j \leq n$
Chapter 15. Dynamic Programming

**Step 2**: define objective function and formulate recurrence

Define $l(i, j)$ to be the length of the LCS for prefixes $x_1 \ldots x_i$ and $y_1 \ldots y_j$. 

- **base cases**: $l(i, 0) = 0$, for $0 \leq i \leq m$, either prefix is empty $\rightarrow$ LCS is empty.
- $l(0, j) = 0$, for $0 \leq j \leq n$. 

Step 2: define objective function and formulate recurrence

Define $l(i, j)$ to be the length of the LCS for prefixes $x_1 \ldots x_i$ and $y_1 \ldots y_j$. then
Step 2: define objective function and formulate recurrence

Define $l(i, j)$ to be the length of the LCS for prefixes $x_1 \ldots x_i$ and $y_1 \ldots y_j$. Then

$$l(i, j) = \begin{cases} 
  l(i - 1, j - 1) + 1 & \text{if } x_i = y_j; \\
  \max \left\{ l(i, j - 1); l(i - 1, j) \right\} & \text{if } x_i \neq y_j
\end{cases}$$

base cases:

$l(i, 0) = 0$, for $0 \leq i \leq m$, either prefix is empty → LCS is empty

$l(0, j) = 0$, for $0 \leq j \leq n$
Step 2: define objective function and formulate recurrence

Define $l(i, j)$ to be the length of the LCS for prefixes $x_1 \ldots x_i$ and $y_1 \ldots y_j$. then

$$l(i, j) = \begin{cases} 
  l(i - 1, j - 1) + 1 & \text{if } x_i = y_j; \\
  \max \left\{ l(i, j - 1); l(i - 1, j) \right\} & \text{if } x_i \neq y_j
\end{cases}$$

the value of $l(i, j)$ depends on values of $l(i - 1, j - 1)$, $l(i, j - 1)$, and $l(i - 1, j)$. 

base cases:

- $l(i, 0) = 0$, for $0 \leq i \leq m$, either prefix is empty → LCS is empty
- $l(0, j) = 0$, for $0 \leq j \leq n$
Step 2: define objective function and formulate recurrence

Define $l(i, j)$ to be the length of the LCS for prefixes $x_1 \ldots x_i$ and $y_1 \ldots y_j$. Then

$$l(i, j) = \begin{cases} 
  l(i - 1, j - 1) + 1 & \text{if } x_i = y_j; \\
  \max\{l(i, j - 1); l(i - 1, j)\} & \text{if } x_i \neq y_j 
\end{cases}$$

the value of $l(i, j)$ depends on values of $l(i - 1, j - 1)$, $l(i, j - 1)$, and $l(i - 1, j)$.

base cases: $l(i, 0) = 0$, for $0 \leq i \leq m$, either prefix is empty $\rightarrow$ LCS is empty
Step 2: define objective function and formulate recurrence

Define \( l(i, j) \) to be the length of the LCS for prefixes \( x_1 \ldots x_i \) and \( y_1 \ldots y_j \). Then

\[
l(i, j) = \begin{cases} 
  l(i - 1, j - 1) + 1 & \text{if } x_i = y_j; \\
  \max \{ l(i, j - 1); l(i - 1, j) \} & \text{if } x_i \neq y_j
\end{cases}
\]

the value of \( l(i, j) \) depends on values of \( l(i - 1, j - 1), l(i, j - 1), \) and \( l(i - 1, j) \).

base cases: \( l(i, 0) = 0 \), for \( 0 \leq i \leq m \), either prefix is empty \( \rightarrow \) LCS is empty

\( l(0, j) = 0 \), for \( 0 \leq j \leq n \)
Step 3: bottom-up table building for function \( l(i, j) \).
Chapter 15. Dynamic Programming

**Step 3**: bottom-up table building for function $l(i, j)$.

<table>
<thead>
<tr>
<th></th>
<th>G</th>
<th>C</th>
<th>C</th>
<th>C</th>
<th>T</th>
<th>A</th>
<th>G</th>
<th>C</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>G</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Chapter 15. Dynamic Programming**

**Step 3**: bottom-up table building for function $l(i, j)$.

<table>
<thead>
<tr>
<th></th>
<th>G</th>
<th>C</th>
<th>C</th>
<th>C</th>
<th>T</th>
<th>A</th>
<th>G</th>
<th>C</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>G</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>G</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Chapter 15. Dynamic Programming**

**Step 3:** bottom-up table building for function $l(i, j)$.

<table>
<thead>
<tr>
<th></th>
<th>G</th>
<th>C</th>
<th>C</th>
<th>C</th>
<th>T</th>
<th>A</th>
<th>G</th>
<th>C</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>G</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>G</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LCS result between prefixes GCCCTAGC and GCG:

GCCCTAGC

G   C   G
Algorithm LCS($x, y$)
1. $m = \text{length}(x), n = \text{length}(y)$;
Algorithm LCS($x, y$)
1. $m = \text{length}(x)$, $n = \text{length}(y)$;
2. for $i = 0$ to $m$
Algorithm LCS \((x, y)\)
1. \(m = \text{length}(x), n = \text{length}(y)\);
2. \(\textbf{for } i = 0 \textbf{ to } m\)
3. \(T[i, 0] = 0\)
Algorithm LCS($x, y$)
1. $m = \text{length}(x), n = \text{length}(y)$;
2. for $i = 0$ to $m$
3.  $T[i, 0] = 0$
4. for $j = 0$ to $n$
5. if $x_i = y_j$
6.  $T[i, j] = T[i-1, j-1] + 1$; $P[i, j] = '↖'$
7. else if $T[i, j-1] > T[i-1, j]$
8.  $T[i, j] = T[i, j-1]$; $P[i, j] = '←'$
9. else
10.  $T[i, j] = T[i-1, j]$; $P[i, j] = '↑'$
11. return ($T, P$)

Time complexity: $O(mn)$
Algorithm LCS($x, y$)
1. $m = \text{length}(x)$, $n = \text{length}(y)$;
2. for $i = 0$ to $m$
3. \hspace{1em} $T[i, 0] = 0$
4. for $j = 0$ to $n$
5. \hspace{1em} $T[0, j] = 0$

Time complexity: $O(mn)$
Algorithm LCS($x, y$)
1. $m = \text{length}(x)$, $n = \text{length}(y)$;
2. for $i = 0$ to $m$
   3. $T[i, 0] = 0$
4. for $j = 0$ to $n$
   5. $T[0, j] = 0$
6. for $i = 1$ to $m$
   7. for $j = 1$ to $n$
      8. if $x_i = y_j$
         9. $T[i, j] = T[i−1, j−1] + 1$
            $P[i, j] = \uparrow$
      10. else if $T[i, j−1] > T[i−1, j]$
            11. $T[i, j] = T[i, j−1]$
            $P[i, j] = \leftarrow$
      12. else
            13. $T[i, j] = T[i−1, j]$
            $P[i, j] = \uparrow$
14. return $(T, P)$

Time complexity: $O(mn)$
Algorithm LCS$(x, y)$

1. \( m = \text{length}(x) \), \( n = \text{length}(y) \);
2. \( \text{for } i = 0 \text{ to } m \)
3. \( T[i, 0] = 0 \)
4. \( \text{for } j = 0 \text{ to } n \)
5. \( T[0, j] = 0 \)
6. \( \text{for } i = 1 \text{ to } m \)
7. \( \text{for } j = 1 \text{ to } n \)
Algorithm LCS$(x, y)$
1. $m = \text{length}(x)$, $n = \text{length}(y)$;
2. for $i = 0$ to $m$
3.     $T[i, 0] = 0$
4. for $j = 0$ to $n$
5.     $T[0, j] = 0$
6. for $i = 1$ to $m$
7.     for $j = 1$ to $n$
8.         if $x_i = y_j$
9.             $T[i, j] = T[i-1, j-1] + 1$
10.            $P[i, j] = \text{↖}$
11. else if $T[i-1, j] > T[i, j-1]$
12.              $T[i, j] = T[i-1, j]$;
13.              $P[i, j] = \text{←}$
14. else
15.              $T[i, j] = T[i, j-1]$
16.              $P[i, j] = \text{↑}$
17. return $(T, P)$

Time complexity: $O(mn)$
Algorithm LCS($x, y$)
1. $m = \text{length}(x)$, $n = \text{length}(y)$;
2. for $i = 0$ to $m$
3.   $T[i, 0] = 0$
4. for $j = 0$ to $n$
5.   $T[0, j] = 0$
6. for $i = 1$ to $m$
7.   for $j = 1$ to $n$
8.       if $x_i = y_j$
9.       $T[i, j] = T[i - 1, j - 1] + 1; P[i, j] = '↖'
10.  else if $T[i, j - 1] > T[i - 1, j]$
11.     $T[i, j] = T[i, j - 1]; P[i, j] = '←'
12.   else
13.     $T[i, j] = T[i - 1, j]; P[i, j] = '↑'
14.  return ($T, P$)

Time complexity: $O(mn)$
Chapter 15. Dynamic Programming

Algorithm LCS($x, y$)
1. $m = \text{length}(x)$, $n = \text{length}(y)$;
2. for $i = 0$ to $m$
3. \hspace{1em} $T[i, 0] = 0$
4. for $j = 0$ to $n$
5. \hspace{1em} $T[0, j] = 0$
6. for $i = 1$ to $m$
7. \hspace{2em} for $j = 1$ to $n$
8. \hspace{3em} if $x_i = y_j$
9. \hspace{4em} $T[i, j] = T[i - 1, j - 1] + 1$; $P[i, j] = \swarrow$
10. \hspace{3em} else if $T[i, j - 1] > T[i - 1, j]$
11. \hspace{4em} $T[i, j] = T[i, j - 1]$; $P[i, j] = \leftarrow$
12. return $(T, P)$

Time complexity: $O(mn)$
Algorithm LCS(x, y)
1. \( m = \text{length}(x), n = \text{length}(y); \)
2. \textbf{for} \( i = 0 \) to \( m \)
3. \hspace{1em} \( T[i, 0] = 0 \)
4. \textbf{for} \( j = 0 \) to \( n \)
5. \hspace{1em} \( T[0, j] = 0 \)
6. \textbf{for} \( i = 1 \) to \( m \)
7. \hspace{1em} \textbf{for} \( j = 1 \) to \( n \)
8. \hspace{2em} \textbf{if} \( x_i = y_j \)
9. \hspace{3em} \( T[i, j] = T[i - 1, j - 1] + 1; \ P[i, j] = \, \uparrow \downarrow \)\)
10. \hspace{2em} \textbf{else if} \( T[i, j - 1] > T[i - 1, j] \)
11. \hspace{3em} \( T[i, j] = T[i, j - 1]; \ P[i, j] = \, \uparrow \downarrow \)\)
12. \hspace{2em} \textbf{else} \( T[i, j] = T[i - 1, j] \)
13. \hspace{3em} \( P[i, j] = \, \uparrow \downarrow \)
Chapter 15. Dynamic Programming

Algorithm LCS($x, y$)
1. $m = length(x), n = length(y)$;
2. for $i = 0$ to $m$
3. \hspace{1cm} $T[i, 0] = 0$
4. for $j = 0$ to $n$
5. \hspace{1cm} $T[0, j] = 0$
6. for $i = 1$ to $m$
7. \hspace{1cm} for $j = 1$ to $n$
8. \hspace{2cm} if $x_i = y_j$
9. \hspace{3cm} $T[i, j] = T[i - 1, j - 1] + 1$; $P[i, j] = '↖'$
10. \hspace{2cm} else if $T[i, j - 1] > T[i - 1, j]$
11. \hspace{3cm} $T[i, j] = T[i, j - 1]$; $P[i, j] = '←'$
12. \hspace{2cm} else $T[i, j] = T[i - 1, j]$; $P[i, j] = '↑'$

Time complexity: $O(mn)$
Algorithm LCS($x, y$)
1. $m = \text{length}(x), \ n = \text{length}(y)$;
2. for $i = 0$ to $m$
3. \hspace{1em} $T[i, 0] = 0$
4. for $j = 0$ to $n$
5. \hspace{1em} $T[0, j] = 0$
6. for $i = 1$ to $m$
7. \hspace{1em} for $j = 1$ to $n$
8. \hspace{2em} if $x_i = y_j$
9. \hspace{3em} $T[i, j] = T[i - 1, j - 1] + 1$; $P[i, j] = '↖'$
10. \hspace{2em} else if $T[i, j - 1] > T[i - 1, j]$
11. \hspace{3em} $T[i, j] = T[i, j - 1]$; $P[i, j] = '←'$
12. \hspace{2em} else $T[i, j] = T[i - 1, j]$; $P[i, j] = '↑'$
13. return $(T, P)$

Time complexity: $O(mn)$
Algorithm LCS($x, y$)
1. \( m = \text{length}(x), \ n = \text{length}(y); \)
2. \textbf{for} \( i = 0 \) \textbf{to} \( m \)
3. \hspace{1em} \( T[i, 0] = 0 \)
4. \textbf{for} \( j = 0 \) \textbf{to} \( n \)
5. \hspace{1em} \( T[0, j] = 0 \)
6. \textbf{for} \( i = 1 \) \textbf{to} \( m \)
7. \hspace{1em} \textbf{for} \( j = 1 \) \textbf{to} \( n \)
8. \hspace{2em} \textbf{if} \( x_i = y_j \)
9. \hspace{3em} \( T[i, j] = T[i - 1, j - 1] + 1; \ P[i, j] = \leftarrow \)
10. \hspace{2em} \textbf{else if} \( T[i, j - 1] > T[i - 1, j] \)
11. \hspace{3em} \( T[i, j] = T[i, j - 1]; \ P[i, j] = \downarrow \)
12. \hspace{2em} \textbf{else} \( T[i, j] = T[i - 1, j]; \ P[i, j] = \uparrow \)
13. \hspace{1em} \textbf{return} \( (T, P) \)

Time complexity: \( O(mn) \)
Chapter 15. Dynamic Programming

Traceback the longest common subsequence
either recursive or iterative algorithm
Chapter 15. Dynamic Programming

Traceback the longest common subsequence
either recursive or iterative algorithm

Algorithm PRINTLCS($P, x, i, j$)
Traceback the longest common subsequence
either recursive or iterative algorithm

Algorithm \texttt{PRINTLCS}(P, x, i, j)
1. \textbf{if} \ (i > 0) \land (j > 0)
Traceback the longest common subsequence either recursive or iterative algorithm

Algorithm PRINTLCS\((P, x, i, j)\)
1. \(\text{if } (i > 0) \land (j > 0)\)
2. \(\text{if } P[i, j] = \backslash\)
3. \(\text{PrintLCS}(P, x, i - 1, j - 1)\)
4. \(\text{Print}(x_i)\)
5. \(\text{else if } P[i, j] = \backslash\)
6. \(\text{PrintLCS}(P, x, i, j - 1)\)
7. \(\text{else } \text{PrintLCS}(P, x, i - 1, j)\)
8. \(\text{return } ()\)
Chapter 15. Dynamic Programming

Traceback the longest common subsequence

either recursive or iterative algorithm

Algorithm \text{PRINTLCS}(P, x, i, j)
1. \textbf{if} \ (i > 0) \land \ (j > 0)
2. \quad \textbf{if} \ P[i, j] = '\text\text{'\textbackslash'
3. \quad \text{PRINTLCS}(P, x, i - 1, j - 1)
Chapter 15. Dynamic Programming

Traceback the longest common subsequence

either recursive or iterative algorithm

Algorithm `PRINTLCS(P, x, i, j)`
1. `if (i > 0) ∧ (j > 0)`
2. `if P[i, j] = '↖'`
3. `PRINTLCS(P, x, i - 1, j - 1)`
4. `Print (x_i)`
Chapter 15. Dynamic Programming

Traceback the longest common subsequence

either recursive or iterative algorithm

Algorithm \text{PRINTLCS}(P, x, i, j)
1. \textbf{if} \ (i > 0) \land (j > 0)
2. \quad \textbf{if} \ P[i, j] ='\langle '\
3. \quad \text{PRINTLCS}(P, x, i - 1, j - 1)
4. \quad \textbf{Print} \ (x_i)
5. \quad \textbf{else if} \ P[i, j] ='\leftarrow '\
Traceback the longest common subsequence

either recursive or iterative algorithm

Algorithm PRINTLCS($P, x, i, j$)
1. if ($i > 0$) ∧ ($j > 0$)
2. if $P[i, j] = \text{↖} \backslash$
3. PRINTLCS($P, x, i - 1, j - 1$)
4. Print ($x_i$)
5. else if $P[i, j] = \text{←} ^{\prime}$
6. PRINTLCS($P, x, i, j - 1$)
Chapter 15. Dynamic Programming

Traceback the longest common subsequence

either recursive or iterative algorithm

Algorithm PRINTLCS($P, x, i, j$)
1. if ($i > 0$) $\land$ ($j > 0$)
2. if $P[i, j] = \text{'\textbackslash'}$
   3. PRINTLCS($P, x, i - 1, j - 1$)
4. Print ($x_i$)
5. else if $P[i, j] = \text{'\textleft'}$
   6. PRINTLCS($P, x, i, j - 1$)
7. else PRINTLCS($P, x, i - 1, j$)
Traceback the longest common subsequence

either recursive or iterative algorithm

Algorithm \texttt{PRINTLCS}(P, x, i, j)

1. \textbf{if} \ (i > 0) \land (j > 0)
2. \hspace{1em} \textbf{if} \ P[i, j] = \text{'\textless\textbackslash'}
3. \hspace{2em} \text{PRINTLCS}(P, x, i - 1, j - 1)
4. \hspace{1em} \textbf{Print} \ (x_i)
5. \hspace{1em} \textbf{else if} \ P[i, j] = \text{'\textless\textleft'}
6. \hspace{2em} \text{PRINTLCS}(P, x, i, j - 1)
7. \hspace{1em} \textbf{else} \ \text{PRINTLCS}(P, x, i - 1, j)
8. \hspace{1em} \textbf{else return} \ ()
Chapter 15. Dynamic Programming

**Pairwise Alignment**
Chapter 15. Dynamic Programming

Pairwise Alignment

\begin{align*}
\text{ACCGGTCGAGTGCG} & \quad \text{CGTCGATGC} \quad \text{← common sequence} \\
\text{GTCGTTCGGATGCCC} & \\
\end{align*}

biologically more meaningful view:

\begin{align*}
\text{ACCGGTCGAGTGCG} & \quad \text{CGTCGATGC} \quad \text{← common sequence} \\
\text{GTCGTTCGGATGCCC} & \\
\end{align*}

blue letters are conserved; red letters are mutated; purple letters are inserted or deleted;
Chapter 15. Dynamic Programming

Pairwise Alignment

ACCGGT CGAGT GCG  
   CGTCGATGC ← common sequence  
GTCGT TCGGATGCCC

biologically more meaningful view:

ACCGGTC GAGT GCG
   || || || || 
   CG TC GA TGC ← common sequence  
   || || || || 
GTCGT TCGGA TGCCC
Chapter 15. Dynamic Programming

Pairwise Alignment

ACCGGT TCGAGTGCG
CGTCGATGC ← common sequence
GTCGTTCGGATGCC

biologically more meaningful view:

ACCGGTC GAGT GCG
|| || || |||
CG TC GA TGC ← common sequence
|| || || |||
GTCGTTCGGGA TGCC

blue letters are conserved;
red letters are mutated;
purple letters are inserted or deleted;
Chapter 15. Dynamic Programming

Alignment between two sequences:

ACCGGTC
GAGTGCG

|| || || |||
CG TC GA TGC ←− common sequence

|| || || |||
GTCGTTCGGA
TGCCC

• two sequences padded with 's called gaps;
• two sequences are of the same length, aligned;
• sum of column scores is used to identify the best alignment

For example, a score scheme
+5 for conservation columns, (reward)
-2 for substitution columns, (penalty)
-6 for delete/insert columns, (penalty)

the above alignment has the total score

\[ (-2) + (-2) + 5 + 5 + (-2) + 5 + 5 + (-6) + 5 + 5 + (-6) + 5 + 5 + 5 + (-2) + (-6) \]

LCS is a special case

0 + 0 + 1 + 1 + 0 + 1 + 1 + 0 + 1 + 1 + 0 + 1 + 1 + 1 + 0 + 0
Chapter 15. Dynamic Programming

Alignment between two sequences:

```
ACCGGTC_GAGTGCG_
  || || || ||||
  CG TC GA TGC ← common sequence
  || || || ||||
GTCGTTCGGA_TGCC
```

For example, a score scheme

- +5 for conservation columns, (reward)
- -2 for substitution columns, (penalty)
- -6 for delete/insert columns, (penalty)

The above alignment has the total score

\[ (-2) + (-2) + 5 + 5 + (-2) + 5 + 5 + (-6) + 5 + 5 + (-6) + 5 + 5 + 5 + (-2) + (-6) \]
Chapter 15. Dynamic Programming

Alignment between two sequences:

```
ACCGGTC_GAGTGCG_
   || || || ||
CG TC GA TGC ← common sequence
   || || || ||
GTCGTTCGGA_TGCC
```

- two sequences padded with ‘_’s called **gaps**;
Chapter 15. Dynamic Programming

Alignment between two sequences:

\[
\begin{align*}
\text{ACCGGTC}_\text{GAGTGC}_G \text{GCG} \\
\quad | | | | | | | |
\text{CG TC GA TGC} \quad \leftarrow \text{common sequence} \\
\quad | | | | | | | |
\text{GTCGTTCGGA}_\text{TGCC} \text{C}
\end{align*}
\]

- two sequences padded with ’_’s called gaps;
- two sequences are of the same length, aligned;
Alignment between two sequences:

ACCGGTC\_GAGTGCG\_  
|| || || |||
CG TC GA TGC ← common sequence  
|| || || |||
GTCGGTTCGGA\_TGCCC

- two sequences padded with ‘\_’s called gaps;
- two sequences are of the same length, aligned;
- sum of column scores is used to identify the best alignment
Chapter 15. Dynamic Programming

Alignment between two sequences:

```
ACCGGTC_GAGTGC
  || || || |||  
  CG TC GA TGC ← common sequence
  || || || |||  
GTCGTTGCAGA_TGCC
```

- two sequences padded with ‘_’s called gaps;
- two sequences are of the same length, aligned;
- sum of column scores is used to identify the best alignment

For example, a score scheme

+5 for conservation columns, (reward)
-2 for substitution columns, (penalty)
-6 for delete/insert columns, (penalty)

the above alignment has the total score

\[-2 + (-2) + 5 + 5 + (-2) + 5 + 5 + (-6) + 5 + 5 + (-6) + 5 + 5 + 5 + (-2) + (-6)\]
Alignment between two sequences:

ACCGGTC_GAGTGCG_
  || || || |||
CG TC GA TGC ← common sequence
  || || || |||
GTCGTTCGGA_TGCC

• two sequences padded with ‘_’s called gaps;
• two sequences are of the same length, aligned;
• sum of column scores is used to identify the best alignment

For example, a score scheme
  +5 for conservation columns, (reward)
  -2 for substitution columns, (penalty)
  -6 for delete/insert columns, (penalty)
Alignment between two sequences:

\[
\text{ACCGGTC}_{-} \text{GAGTGCG}_{-} \\
\mid \mid | \mid \mid \mid \mid \mid \\
\text{CG TC GA TGC} \quad \longrightarrow \text{common sequence} \\
\mid \mid | \mid \mid \mid \mid \mid \\
\text{GTCGTTCGGA}_{-} \text{TGCC} \\
\text{GTCGTTCGGA}_{-} \text{TGCC}
\]

- two sequences padded with ‘\_’s called \textbf{gaps};
- two sequences are of the same length, aligned;
- sum of column scores is used to identify the best alignment

For example, a score scheme

- \(+5\) for conservation columns, (reward)
- \(-2\) for substitution columns, (penalty)
- \(-6\) for delete/insert columns, (penalty)

the above alignment has the total score

\((-2) + (-2) + 5 + 5 + (-2) + 5 + 5 + (-6) + 5 + 5 + (-6) + 5 + 5 + 5 + (-2) + (-6)\)
Alignment between two sequences:

\[
\begin{align*}
\text{ACCGGTC} & \_ \_ \_ \_ \_ \_ \_ \_ \\
\text{GAGTGCG} & \_ \_ \_ \_ \_ \_ \_ \_ \\
\text{CG} & \text{ TC} \text{ GA} \text{ TGC} & \leftarrow \text{common sequence} \\
\text{GTCGTTCGGA} & \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \\
\end{align*}
\]

- two sequences padded with ‘_’’s called gaps;
- two sequences are of the same length, aligned;
- sum of column scores is used to identify the best alignment

For example, a score scheme
- +5 for conservation columns, (reward)
- -2 for substitution columns, (penalty)
- -6 for delete/insert columns, (penalty)

the above alignment has the total score
\((-2) + (-2) + 5 + 5 + (-2) + 5 + 5 + (-6) + 5 + 5 + (-2) + 5 + 5 + (-6)\)

\[\text{LCS is a special case}\]
Chapter 15. Dynamic Programming

Alignment between two sequences:

```
ACCGGTC_GAGTGCG_
|| || || |||
CG TC GA TGC ← common sequence
|| || || |||
GTCGTTCGGA_TGCC
```

- two sequences padded with ‘_’s called gaps;
- two sequences are of the same length, aligned;
- sum of column scores is used to identify the best alignment

For example, a score scheme
- +5 for conservation columns, (reward)
- -2 for substitution columns, (penalty)
- -6 for delete/insert columns, (penalty)

the above alignment has the total score

\((-2) + (-2) + 5 + 5 + (-2) + 5 + 5 + (-6) + 5 + 5 + (-6) + 5 + 5 + 5 + (-2) + (-6)\)

LCS is a special case

```
0 + 0 +1+1+ 0 +1+1+ 0 +1+1+ 0 +1+1+1+ 0 + 0
```
Chapter 15. Dynamic Programming

Significance of sequence alignments:

Sequence Homology Reveals Functions

Homology reveals evolution of structure/function

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Alignment</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>FOS_RAT</td>
<td>MMFSGFTADYEASSSRSACSPAGDSLSYYHSPAEDSFSMGSPVNTQDFCADLSVSSANF</td>
<td>60</td>
</tr>
<tr>
<td>FOS_MOUSE</td>
<td>MMYCGFTADYEASSSRSACSPAGDSLSYYHSPAEDSFSMGSPVNTQDFCADLSVSSANF</td>
<td>60</td>
</tr>
<tr>
<td>FOS_CHICK</td>
<td>MMYCGFTADYEASSSRSACSPAGDSLSYYHSPAEDSFSMGSPVNTQDFCADLSVSSANF</td>
<td>60</td>
</tr>
<tr>
<td>FOSB_MOUSE</td>
<td>MFQAPFGDYDS-SGSRSS-SPSAESQ-YLSSVDFSFGPPTAAASQE-CAGLGEQMPGSF</td>
<td>54</td>
</tr>
<tr>
<td>FOSB_HUMAN</td>
<td>MFQAPFGDYDS-SGSRSS-SPSAESQ-YLSSVDFSFGPPTAAASQE-CAGLGEQMPGSF</td>
<td>54</td>
</tr>
</tbody>
</table>

Consensus

Homology reveals regulatory structure (E. Coli promoters)

<table>
<thead>
<tr>
<th>Gene</th>
<th>Consensus</th>
</tr>
</thead>
<tbody>
<tr>
<td>tyr</td>
<td>TCTCAACGTAACCTTTACAGGCCTTCCCGATAAAGGG</td>
</tr>
<tr>
<td>rnm D1</td>
<td>GTAACCGTGTTTATTTACAGGCCTTCCCGATAAAGGG</td>
</tr>
<tr>
<td>rnm X1</td>
<td>ATTCGAACCGTCTTTACAGGCCTTCCCGATAAAGGG</td>
</tr>
<tr>
<td>rnm (DXE)2</td>
<td>CCGAGAGGCAGTTATTTACAGGCCTTCCCGATAAAGGG</td>
</tr>
<tr>
<td>rnm E1</td>
<td>TCTCAACGTAACCTTTACAGGCCTTCCCGATAAAGGG</td>
</tr>
<tr>
<td>rnm A1</td>
<td>TTTCGAGAGGCAGTTATTTACAGGCCTTCCCGATAAAGGG</td>
</tr>
<tr>
<td>rnm A2</td>
<td>CGAGAAATCTTGCAGTTATTTACAGGCCTTCCCGATAAAGGG</td>
</tr>
<tr>
<td>A FR</td>
<td>TAAAGACGAGGTGTTTATTTACAGGCCTTCCCGATAAAGGG</td>
</tr>
<tr>
<td>A LR</td>
<td>GCTCATGTGCTTTATTTACAGGCCTTCCCGATAAAGGG</td>
</tr>
<tr>
<td>T7 A3</td>
<td>GTGAACACCGTCTTTATTTACAGGCCTTCCCGATAAAGGG</td>
</tr>
<tr>
<td>T7 A1</td>
<td>TGCAGGCCTTTATTTACAGGCCTTCCCGATAAAGGG</td>
</tr>
<tr>
<td>T7 A2</td>
<td>AGCAAGACCGTCTTTATTTACAGGCCTTCCCGATAAAGGG</td>
</tr>
<tr>
<td>fd VIII</td>
<td>GATCAATATCCTTTATTTACAGGCCTTCCCGATAAAGGG</td>
</tr>
</tbody>
</table>

-35
-10
+1
Chapter 15. Dynamic Programming

Pairwise alignment problem:

**Input:** sequences $x = x_1x_2 \ldots x_m$, $y = y_1y_2 \ldots y_n$, and scoring scheme $\text{score}$

**Output:** $x'$ and $y'$, which are $x$ and $y$ padded with '-'s, respectively, such that total score $\sum_{i=1}^{l} \text{score}(x'_i, y'_i)$ is the maximum, where $l = |x'| = |y'|$.

How to solve it?
Pairwise alignment problem:

**Input**: sequences \( x = x_1x_2 \ldots x_m \), \( y = y_1y_2 \ldots y_n \), and scoring scheme \( \text{score} \)

**Output**: \( x' \) and \( y' \), which are \( x \) and \( y \) padded with '-'s, respectively such that total score \( \sum_{i=1}^{l} \text{score}(x'_i, y'_i) \) is the maximum, where \( l = |x'| = |y'| \).

How to solve it?

Similar to LCS problem, we consider prefixes \( x_1 \ldots x_i \) of \( x \), \( y_1 \ldots y_j \) of \( y \).
Chapter 15. Dynamic Programming

Pairwise alignment problem:

**Input:** sequences \( x = x_1x_2 \ldots x_m \), \( y = y_1y_2 \ldots y_n \), and scoring scheme \( \text{score} \)

**Output:** \( x' \) and \( y' \), which are \( x \) and \( y \) padded with \( '-' \)'s, respectively such that total score \( \sum_{i=1}^{l} \text{score}(x'_i, y'_i) \) is the maximum, where \( l = |x'| = |y'| \).

How to solve it?

Similar to LCS problem, we consider prefixes \( x_1 \ldots x_i \) of \( x \), \( y_1 \ldots y_j \) of \( y \).

(1) \[ x_1 \ldots x_{i-1} x_i \]
(2) \[ x_1 \ldots x_{i-1} x_i \]
(3) \[ x_1 \ldots x_{i-1} x_i \]
Chapter 15. Dynamic Programming

Pairwise alignment problem:

**Input**: sequences \( x = x_1 x_2 \ldots x_m \), \( y = y_1 y_2 \ldots y_n \), and scoring scheme \( \text{score} \)

**Output**: \( x' \) and \( y' \), which are \( x \) and \( y \) padded with ‘_’s, respectively such that total score \( \sum_{i=1}^{l} \text{score}(x'_i, y'_i) \) is the maximum, where \( l = |x'| = |y'| \).

How to solve it?

Similar to LCS problem, we consider prefixes \( x_1 \ldots x_i \) of \( x \), \( y_1 \ldots y_j \) of \( y \).

\[
\begin{align*}
\text{(1)} & \quad x_1 \ldots x_{i-1} \, x_i \\
& \quad y_1 \ldots y_{j-1} \, y_j
\end{align*}
\]

\[
\begin{align*}
\text{(2)} & \quad x_1 \ldots x_{i-1} \, x_i \\
& \quad y_1 \ldots y_{j-1} \, y_j
\end{align*}
\]

\[
\begin{align*}
\text{(3)} & \quad x_1 \ldots x_{i-1} \, x_i \\
& \quad y_1 \ldots y_{j-1} \, y_j
\end{align*}
\]

case (1) contributes to a match/mismatch score
Chapter 15. Dynamic Programming

Pairwise alignment problem:

**INPUT:** sequences \( x = x_1 x_2 \ldots x_m \), \( y = y_1 y_2 \ldots y_n \), and scoring scheme \( \text{score} \)

**OUTPUT:** \( x' \) and \( y' \), which are \( x \) and \( y \) padded with '-'s, respectively such that total score \( \sum_{i=1}^{l} \text{score}(x'_i, y'_i) \) is the maximum, where \( l = |x'| = |y'| \).

How to solve it?

Similar to LCS problem, we consider prefixes \( x_1 \ldots x_i \) of \( x \), \( y_1 \ldots y_j \) of \( y \).

1. \( x_1 \ldots x_{i-1} x_i \)
2. \( x_1 \ldots x_{i-1} x_i \)
3. \( x_1 \ldots x_{i-1} x_i \)

(1) \( y_1 \ldots y_{j-1} y_j \)
(2) \( y_1 \ldots y_{j-1} y_j \)
(3) \( y_1 \ldots y_{j-1} y_j \)

case (1) contributes to a match/mismatch score

cases (2) and (3) contribute to an insertion/deletion score.
Chapter 15. Dynamic Programming

Scoring function for pairwise alignment

Example (1)
+5 for conservation columns, (reward)
-2 for substitution columns, (penalty)
-6 for delete/insert columns, (penalty)

Example (2)
Why sum of column scores?
product of independent column probabilities
Chapter 15. Dynamic Programming

Scoring function for pairwise alignment

Example (1)
- +5 for conservation columns, (reward)
- -2 for substitution columns, (penalty)
- -6 for delete/insert columns, (penalty)
Chapter 15. Dynamic Programming

Scoring function for pairwise alignment

Example (1)
- +5 for conservation columns, (reward)
- -2 for substitution columns, (penalty)
- -6 for delete/insert columns, (penalty)

Example (2)

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>C</th>
<th>G</th>
<th>T</th>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>5</td>
<td>-1</td>
<td>-2</td>
<td>-1</td>
<td>-4</td>
</tr>
<tr>
<td>C</td>
<td>-1</td>
<td>5</td>
<td>-3</td>
<td>-2</td>
<td>-4</td>
</tr>
<tr>
<td>G</td>
<td>-2</td>
<td>-3</td>
<td>5</td>
<td>-2</td>
<td>-4</td>
</tr>
<tr>
<td>T</td>
<td>-1</td>
<td>-2</td>
<td>-2</td>
<td>5</td>
<td>-4</td>
</tr>
<tr>
<td>-</td>
<td>-4</td>
<td>-4</td>
<td>-4</td>
<td>-4</td>
<td>*</td>
</tr>
</tbody>
</table>
Chapter 15. Dynamic Programming

Scoring function for pairwise alignment

Example (1)

+5 for conservation columns, (reward)
-2 for substitution columns, (penalty)
-6 for delete/insert columns, (penalty)

Example (2)

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>C</th>
<th>G</th>
<th>T</th>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>5</td>
<td>-1</td>
<td>-2</td>
<td>-1</td>
<td>-4</td>
</tr>
<tr>
<td>C</td>
<td>-1</td>
<td>5</td>
<td>-3</td>
<td>-2</td>
<td>-4</td>
</tr>
<tr>
<td>G</td>
<td>-2</td>
<td>-3</td>
<td>5</td>
<td>-2</td>
<td>-4</td>
</tr>
<tr>
<td>T</td>
<td>-1</td>
<td>-2</td>
<td>-2</td>
<td>5</td>
<td>-4</td>
</tr>
<tr>
<td>-</td>
<td>-4</td>
<td>-4</td>
<td>-4</td>
<td>-4</td>
<td>*</td>
</tr>
</tbody>
</table>

Why sum of column scores?
Chapter 15. Dynamic Programming

Scoring function for pairwise alignment

Example (1)

+5 for conservation columns, (reward)
-2 for substitution columns, (penalty)
-6 for delete/insert columns, (penalty)

Example (2)

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>C</th>
<th>G</th>
<th>T</th>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>5</td>
<td>-1</td>
<td>-2</td>
<td>-1</td>
<td>-4</td>
</tr>
<tr>
<td>C</td>
<td>-1</td>
<td>5</td>
<td>-3</td>
<td>-2</td>
<td>-4</td>
</tr>
<tr>
<td>G</td>
<td>-2</td>
<td>-3</td>
<td>5</td>
<td>-2</td>
<td>-4</td>
</tr>
<tr>
<td>T</td>
<td>-1</td>
<td>-2</td>
<td>-2</td>
<td>5</td>
<td>-4</td>
</tr>
<tr>
<td>-</td>
<td>-4</td>
<td>-4</td>
<td>-4</td>
<td>-4</td>
<td>*</td>
</tr>
</tbody>
</table>

Why sum of column scores?

product of independent column probabilities
Chapter 15. Dynamic Programming

As for the LCS problem, we can define

\[ A(i, j) \] is the maximum score alignment between \( x_1 \ldots x_i \)
and \( y_1 \ldots y_j \), then

\[
A(i, j) = \begin{cases} 
\text{concat}(A(i - 1, j - 1), x_i \ y_j) & \text{if this has the highest score}; \\
\end{cases}
\]

\text{score} \text{exercise}: Table filling and traceback
Chapter 15. Dynamic Programming

As for the LCS problem, we can define

\[ A(i, j) \] is the maximum score alignment between \( x_1 \ldots x_i \) and \( y_1 \ldots y_j \), then

\[
A(i, j) = \begin{cases} 
\text{concat}(A(i - 1, j - 1), x_i) & \text{if this has the highest score;} \\
\text{contat}(A(i, j - 1), -y_j) & \text{if this has the highest score;}
\end{cases}
\]

exercise: Table filling and traceback
Chapter 15. Dynamic Programming

As for the LCS problem, we can define

\[ A(i, j) \] is the maximum score alignment between \( x_1 \ldots x_i \)

and \( y_1 \ldots y_j \), then

\[
A(i, j) = \begin{cases} 
\text{concat}(A(i - 1, j - 1), x_i) & \text{if this has the highest score}; \\
\text{contat}(A(i, j - 1), y_j) & \text{if this has the highest score}; \\
\text{contat}(A(i - 1, j), x_i) & \text{if this has the highest score};
\end{cases}
\]
Chapter 15. Dynamic Programming

As for the LCS problem, we can define

\[ A(i, j) \] is the maximum score alignment between \( x_1 \ldots x_i \) and \( y_1 \ldots y_j \), then

\[
A(i, j) = \begin{cases} 
\text{concat}(A(i - 1, j - 1), x_i) & \text{if this has the highest score;} \\
\text{contat}(A(i, j - 1), y_j) & \text{if this has the highest score;} \\
\text{contat}(A(i - 1, j), x_i) & \text{if this has the highest score;}
\end{cases}
\]

Instead, we define objective function \( S(i, j) \) to be the maximum alignment score between \( x_1 \ldots x_i \) and \( y_1 \ldots y_j \)

\[
S(i, j) = \max \left\{ S(i - 1, j - 1) + \text{score}(x_i, y_j) \mid i \geq 1, j \geq 1 \right\}
\]
Chapter 15. Dynamic Programming

As for the LCS problem, we can define

\[ A(i, j) \] is the maximum score alignment between \( x_1 \ldots x_i \)
and \( y_1 \ldots y_j \), then

\[
A(i, j) = \begin{cases} 
\text{concat}(A(i - 1, j - 1), x_i) & \text{if this has the highest score;} \\
\text{concat}(A(i, j - 1), y_j) & \text{if this has the highest score;} \\
\text{concat}(A(i - 1, j), x_i) & \text{if this has the highest score;}
\end{cases}
\]

Instead, we define objective function \( S(i, j) \) to be the maximum alignment score
between \( x_1 \ldots x_i \)
and \( y_1 \ldots y_j \)

\[
S(i, j) = \max \left\{ 
S(i - 1, j - 1) + \text{score}(x_i, y_j) & i \geq 1, j \geq 1; \\
S(i, j - 1) + \text{score}(-, y_j) & j \geq 1;
\right\}
\]
Chapter 15. Dynamic Programming

As for the LCS problem, we can define

\[ A(i, j) \] is the maximum score alignment between \( x_1 \ldots x_i \) and \( y_1 \ldots y_j \), then

\[
A(i, j) = \begin{cases} 
\text{concat}(A(i - 1, j - 1), \ x_i \ ) & \text{if this has the highest score;} \\
\text{contat}(A(i, j - 1), \ y_j ) & \text{if this has the highest score;} \\
\text{contat}(A(i - 1, j), \ x_i ) & \text{if this has the highest score;}
\end{cases}
\]

Instead, we define objective function \( S(i, j) \) to be the maximum alignment score between \( x_1 \ldots x_i \) and \( y_1 \ldots y_j \)

\[
S(i, j) = \max \begin{cases} 
S(i - 1, j - 1) + \text{score}(x_i, y_j) & i \geq 1, j \geq 1; \\
S(i, j - 1) + \text{score}(-, y_j ) & j \geq 1; \\
S(i - 1, j) + \text{score}(x_i, -) & i \geq 1;
\end{cases}
\]

scoring function \( \text{score} \)
As for the LCS problem, we can define $A(i, j)$ is the maximum score alignment between $x_1 \ldots x_i$ and $y_1 \ldots y_j$, then

$$A(i, j) = \begin{cases} 
\text{concat}(A(i-1, j-1), x_i) & \text{if this has the highest score}; \\
\text{concat}(A(i, j-1), y_j) & \text{if this has the highest score}; \\
\text{concat}(A(i-1, j), x_i) & \text{if this has the highest score}; 
\end{cases}$$

Instead, we define objective function $S(i, j)$ to be the maximum alignment score between $x_1 \ldots x_i$ and $y_1 \ldots y_j$

$$S(i, j) = \max \begin{cases} 
S(i-1, j-1) + \text{score}(x_i, y_j) & i \geq 1, j \geq 1; \\
S(i, j-1) + \text{score}(-, y_j) & j \geq 1; \\
S(i-1, j) + \text{score}(x_i, -) & i \geq 1; 
\end{cases}$$

scoring function $\text{score}$

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>C</th>
<th>G</th>
<th>T</th>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>5</td>
<td>-1</td>
<td>-2</td>
<td>-1</td>
<td>-4</td>
</tr>
<tr>
<td>C</td>
<td>-1</td>
<td>5</td>
<td>-3</td>
<td>-2</td>
<td>-4</td>
</tr>
<tr>
<td>G</td>
<td>-2</td>
<td>-3</td>
<td>5</td>
<td>-2</td>
<td>-4</td>
</tr>
<tr>
<td>T</td>
<td>-1</td>
<td>-2</td>
<td>-2</td>
<td>5</td>
<td>-4</td>
</tr>
<tr>
<td>-</td>
<td>-4</td>
<td>-4</td>
<td>-4</td>
<td>-4</td>
<td>*</td>
</tr>
</tbody>
</table>

exercise: Table filling and traceback
Chapter 15. Dynamic Programming

Affine gap penalty: a challenge for DP

Consider: new gap penalty score:

\[ \text{penalty} = -\text{op} - \text{ext} (l - 1) \]

for a gap covering \( l \) positions, where \( \text{op} > \text{ext} \).

How to modify recurrences:

\[
S(i, j) = \max \begin{cases}
S(i - 1, j - 1) + \text{score}(x_i, y_j) \\
S(i - 1, j) + \text{score}(x_i, -)
\end{cases}
\]
Chapter 15. Dynamic Programming

Affine gap penalty: a challenge for DP

Consider: new gap penalty score:

\[ \text{penalty} = -\text{op} - \text{ext} (l-1) \]

for a gap covering \( l \) positions, where \( \text{op} > \text{ext} \).

How to modify recurrences:

\[
S(i,j) = \max \begin{cases} 
S(i-1,j-1) + \text{score}(x_i, y_j) \\
S(i,j-1) + \text{score}(x_i, -) \\
S(i-1,j) + \text{score}(-, y_j) 
\end{cases}
\]
Chapter 15. Dynamic Programming

Affine gap penalty: a challenge for DP

Consider: new gap penalty score:

- $op$: penalty to open a gap;
- $ext$: penalty to extend an already opened gap

$$
\text{penalty} = -op - ext(l - 1)
$$

for a gap covering $l$ positions, where $op > ext$.
Chapter 15. Dynamic Programming

Affine gap penalty: a challenge for DP

Consider: new gap penalty score:

\(\text{op}\): penalty to open a gap;
\(\text{ext}\): penalty to extend an already opened gap

penalty = \(-\text{op} - \text{ext}(l - 1)\) for a gap covering \(l\) positions, where \(\text{op} > \text{ext}\).
Chapter 15. Dynamic Programming

Affine gap penalty: a challenge for DP

Consider: new gap penalty score:

- $op$: penalty to open a gap;
- $ext$: penalty to extend an already opened gap

penalty = $-op - ext(l - 1)$ for a gap covering $l$ positions, where $op > ext$.

how to modify recurrences:

$$S(i, j) = \max \left\{ S(i - 1, j - 1) + \text{score}(x_i, y_j) \right\}$$
Chapter 15. Dynamic Programming

Affine gap penalty: a challenge for DP

Consider: new gap penalty score:

- $op$: penalty to open a gap;
- $ext$: penalty to extend an already opened gap

$$\text{penalty} = -op - ext(l - 1)$$
for a gap covering $l$ positions, where $op > ext$.

how to modify recurrences:

$$S(i, j) = \max \begin{cases} 
S(i - 1, j - 1) + \text{score}(x_i, y_j) \\
S(i, j - 1) + \text{score}(-, y_j)
\end{cases}$$
Affine gap penalty: a challenge for DP

Consider: new gap penalty score:

- \( op \): penalty to open a gap;
- \( ext \): penalty to extend an already opened gap

\[
\text{penalty} = -op - ext(l - 1) \quad \text{for a gap covering} \quad l \quad \text{positions, where} \quad op > ext.
\]

how to modify recurrences:

\[
S(i, j) = \max \left\{ \begin{array}{l}
S(i - 1, j - 1) + \text{score}(x_i, y_j) \\
S(i, j - 1) + \text{score}(-, y_j) \\
S(i - 1, j) + \text{score}(x_i, -) \\
\end{array} \right. 
\]
Chapter 15. Dynamic Programming

Issue: we do not know how long a gap is.

\[ S(i, j) = \max \left\{ \begin{array}{l} S(i - 1, j - 1) + \text{score}(x_i, y_j) \\ S(i, j - 1) + \text{score}(\text{-}, y_j) \\ S(i - 1, j) + \text{score}(x_i, \text{-}) \end{array} \right\} \]

Solution 1: we can try all different length \( l \)

\[ S(i, j) = \max \left\{ \begin{array}{l} S(i - 1, j - 1) + \text{score}(x_i, y_j) \end{array} \right\} \]

But the complexity increased to: \( O(nm \times \max\{n, m\}) \)

Why does the complexity increase? a lot of recomputations, but where?
Chapter 15. Dynamic Programming

Issue: we do not know how long a gap is.

\[
S(i, j) = \max \begin{cases} 
S(i - 1, j - 1) + \text{score}(x_i, y_j) \\
S(i, j - 1) + \text{score}(-, y_j) \\
S(i - 1, j) + \text{score}(x_i, -) 
\end{cases}
\]

**Solution 1:** we can try all different length \(l\)

\[
S(i, j) = \max \begin{cases} 
S(i - 1, j - 1) + \text{score}(x_i, y_j) \\
\max_{1 \leq l \leq j} S(i, j - l) - \text{op} - (l - 1) \times \text{ext}
\end{cases}
\]

But the complexity increased to: \(O(nm \times \max\{n, m\})\)

Why does the complexity increase? a lot of recomputations, but where?
Chapter 15. Dynamic Programming

Issue: we do not know how long a gap is.

\[ S(i, j) = \max \begin{cases} 
S(i - 1, j - 1) + \text{score}(x_i, y_j) \\
S(i, j - 1) + \text{score}(-, y_j) \\
S(i - 1, j) + \text{score}(x_i, -) 
\end{cases} \]

Solution 1: we can try all different length \( l \)

\[ S(i, j) = \max \begin{cases} 
S(i - 1, j - 1) + \text{score}(x_i, y_j) \\
\max_{1 \leq l \leq j} S(i, j - l) - \text{op} - (l - 1) \times \text{ext} \\
\max_{1 \leq l \leq i} S(i - l, j) - \text{op} - (l - 1) \times \text{ext} 
\end{cases} \]

But the complexity increased to: \( O(nm \times \max\{n, m\}) \)

Why does the complexity increase?

a lot of recomputations, but where?
Chapter 15. Dynamic Programming

Issue: we do not know how long a gap is.

\[ S(i, j) = \max \begin{cases} 
  S(i - 1, j - 1) + \text{score}(x_i, y_j) \\
  S(i, j - 1) + \text{score}(-, y_j) \\
  S(i - 1, j) + \text{score}(x_i, -) 
\end{cases} \]

Solution 1: we can try all different length \( l \)

\[ S(i, j) = \max \begin{cases} 
  S(i - 1, j - 1) + \text{score}(x_i, y_j) \\
  \max_{1 \leq l \leq j} S(i, j - l) - \text{op} - (l - 1) \times \text{ext} \\
  \max_{1 \leq l \leq i} S(i - l, j) - \text{op} - (l - 1) \times \text{ext} 
\end{cases} \]

But the complexity increased to: \( O(nm \times \max\{n, m\}) \)
Chapter 15. Dynamic Programming

Issue: we do not know how long a gap is.

\[ S(i, j) = \max \begin{cases} 
S(i - 1, j - 1) + \text{score}(x_i, y_j) \\
S(i, j - 1) + \text{score}(-, y_j) \\
S(i - 1, j) + \text{score}(x_i, -) 
\end{cases} \]

Solution 1: we can try all different length \( l \)

\[ S(i, j) = \max \begin{cases} 
S(i - 1, j - 1) + \text{score}(x_i, y_j) \\
\max_{1 \leq l \leq j} S(i, j - l) - \text{op} - (l - 1) \times \text{ext} \\
\max_{1 \leq l \leq i} S(i - l, j) - \text{op} - (l - 1) \times \text{ext} 
\end{cases} \]

But the complexity increased to: \( O(nm \times \max\{n, m\}) \)

Why does the complexity increase?
Chapter 15. Dynamic Programming

Issue: we do not know how long a gap is.

\[ S(i, j) = \max \begin{cases} 
S(i - 1, j - 1) + \text{score}(x_i, y_j) \\
S(i, j - 1) + \text{score}(\_, y_j) \\
S(i - 1, j) + \text{score}(x_i, \_) 
\end{cases} \]

**Solution 1:** we can try all different length \( l \)

\[ S(i, j) = \max \begin{cases} 
S(i - 1, j - 1) + \text{score}(x_i, y_j) \\
\max_{1 \leq l \leq j} S(i, j - l) - \text{op} - (l - 1) \times \text{ext} \\
\max_{1 \leq l \leq i} S(i - l, j) - \text{op} - (l - 1) \times \text{ext} 
\end{cases} \]

But the complexity increased to: \( O(nm \times \max\{n, m\}) \)

Why does the complexity increase?

a lot of recomputations,
Chapter 15. Dynamic Programming

Issue: we do not know how long a gap is.

\[
S(i, j) = \max \begin{cases} 
S(i - 1, j - 1) + \text{score}(x_i, y_j) \\
S(i, j - 1) + \text{score}(-, y_j) \\
S(i - 1, j) + \text{score}(x_i, -) 
\end{cases}
\]

Solution 1: we can try all different length \( l \)

\[
S(i, j) = \max \begin{cases} 
S(i - 1, j - 1) + \text{score}(x_i, y_j) \\
\max_{1 \leq l \leq j} S(i, j - l) - \text{op} - (l - 1) \times \text{ext} \\
\max_{1 \leq l \leq i} S(i - l, j) - \text{op} - (l - 1) \times \text{ext} 
\end{cases}
\]

But the complexity increased to: \( O(nm \times \max\{n, m\}) \)

Why does the complexity increase?

a lot of recomputations, but where?
So we want to do recursion "one step at a time":

Chapter 15. Dynamic Programming
Chapter 15. Dynamic Programming

So we want to do recursion "one step at a time":

**but at the same time, differentiate different "states"**

where the two tails are

**Substitute**

```
\[
S(i, j) \\
S(i-1, j-1)
\]

\[
x_1 \ldots x_{i-1} x_i \\
y_1 \ldots y_{j-1} y_j
\]

```

```
\[
S(i, j) \\
l(i-1, j-1)
\]

\[
x_1 \ldots x_{i-1} x_i \\
y_1 \ldots y_{j-1} y_j
\]

```

```
\[
S(i, j) \\
d(i-1, j-1)
\]

\[
x_1 \ldots x_{i-1} x_i \\
y_1 \ldots y_{j-1} y_j
\]
```
Chapter 15. Dynamic Programming

So we want to do recursion "one step at a time":

but at the same time, differentiate different "states"
where the two tails are

Substitute

Insertion

\[
\begin{align*}
S(i, j) &
\begin{array}{c}
\textit{x}_1 \cdots \textit{x}_{i-1} \textit{x}_i \\
\textit{y}_1 \cdots \textit{y}_{j-1} \textit{y}_j
\end{array} \\
S(i-1, j-1)
\end{align*}
\]

\[
\begin{align*}
S(i, j) &
\begin{array}{c}
\textit{x}_1 \cdots \textit{x}_{i-1} \textit{x}_i \\
\textit{y}_1 \cdots \textit{y}_{j-1} \textit{y}_j
\end{array} \\
I(i-1, j-1)
\end{align*}
\]

\[
\begin{align*}
S(i, j) &
\begin{array}{c}
\textit{x}_1 \cdots \textit{x}_{i-1} \textit{x}_i \\
\textit{y}_1 \cdots \textit{y}_{j-1} \textit{y}_j
\end{array} \\
D(i-1, j-1)
\end{align*}
\]

\[
\begin{align*}
I(i, j) &
\begin{array}{c}
\textit{x}_1 \cdots \textit{x}_i \\
\textit{y}_1 \cdots \textit{y}_{j-1} \textit{y}_j
\end{array} \\
I(i, j-1)
\end{align*}
\]

\[
\begin{align*}
I(i, j) &
\begin{array}{c}
\textit{x}_1 \cdots \textit{x}_i \\
\textit{y}_1 \cdots \textit{y}_{j-1} \textit{y}_j
\end{array} \\
S(i-1, j)
\end{align*}
\]
Chapter 15. Dynamic Programming

So we want to do recursion "one step at a time":

**but at the same time, differentiate different "states"**
where the two tails are

**Substitute**

**Insertion**

**Deletion**
Define $S(i,j)$ to be the maximum score of alignment between $x_1 \ldots x_i$ and $y_1 \ldots y_j$ with $x_i$ being aligned to $y_j$. 

\[
S(i,j) = \max \begin{cases} 
S(i-1,j-1) + \text{score}(x_i, y_j) \\
I(i-1,j-1) + \text{score}(x_i, y_j) \\
D(i-1,j-1) + \text{score}(x_i, y_j) 
\end{cases}
\]

Define $I(i,j)$ to be the maximum score of alignment between $x_1 \ldots x_i$ and $y_1 \ldots y_j$ with $y_j$ being insertion.
Define $S(i, j)$ to be the maximum score of alignment between $x_1 \ldots x_i$ and $y_1 \ldots y_j$ with $x_i$ being aligned to $y_j$.

$$S(i, j) = \max \left\{ S(i-1, j-1) + \text{score}(x_i, y_j), I(i-1, j-1) + \text{score}(x_i, y_j), D(i-1, j-1) + \text{score}(x_i, y_j) \right\}$$
Chapter 15. Dynamic Programming

Define $S(i, j)$ to be the maximum score of alignment between $x_1 \ldots x_i$ and $y_1 \ldots y_j$ with $x_i$ being aligned to $y_j$.

$$S(i, j) = \max \begin{cases} 
S(i-1, j-1) + \text{score}(x_i, y_j) \\
I(i-1, j-1) + \text{score}(x_i, y_j) \\
D(i-1, j-1) + \text{score}(x_i, y_j) 
\end{cases}$$

Define $I(i, j)$ to be the maximum score of alignment between $x_1 \ldots x_i$ and $y_1 \ldots y_j$ with $y_j$ being insertion.
Define $S(i, j)$ to be the maximum score of alignment between $x_1 \ldots x_i$ and $y_1 \ldots y_j$ with $x_i$ being aligned to $y_j$. 

$$S(i, j) = \max \left\{ S(i-1, j-1) + \text{score}(x_i, y_j), \ I(i-1, j-1) + \text{score}(x_i, y_j), \ D(i-1, j-1) + \text{score}(x_i, y_j) \right\}$$

Define $I(i, j)$ to be the maximum score of alignment between $x_1 \ldots x_i$ and $y_1 \ldots y_j$ with $y_j$ being insertion.

$$I(i, j) = \max \left\{ I(i, j-1) - \text{ext}, \ S(i, j-1) - \text{op} \right\}$$
Chapter 15. Dynamic Programming

Define $D(i,j)$ to be the maximum score of alignment between $x_1...x_i$ and $y_1...y_j$ with $x_i$ being deletion.

$D(i,j) = \max \{ D(i-1,j) - extS(i-1,j) - op \}$

• Note that there is no deletion right after insertion, neither insertion right after deletion. why?

• What is the desired value?

$\max \{ S(m,n), I(m,n), D(m,n) \}$

• How to fill the table(s) and to traceback the optimal alignment?
Define $D(i, j)$ to be the maximum score of alignment between $x_1 \ldots x_i$ and $y_1 \ldots y_j$ with $x_i$ being deletion.
Chapter 15. Dynamic Programming

Define $D(i, j)$ to be the maximum score of alignment between $x_1 \ldots x_i$ and $y_1 \ldots y_j$ with $x_i$ being deletion.

$$D(i, j) = \max \begin{cases} 
D(i - 1, j) - ext \\
S(i - 1, j) - op 
\end{cases}$$
Chapter 15. Dynamic Programming

Define $D(i, j)$ to be the maximum score of alignment between $x_1 \ldots x_i$ and $y_1 \ldots y_j$ with $x_i$ being deletion.

$D(i, j) = \max \left\{ \begin{array}{l} D(i - 1, j) - \text{ext} \\ S(i - 1, j) - \text{op} \end{array} \right\}$

- Note that there is no deletion right after insertion, neither insertion right after deletion. \textbf{why?}
Define $D(i, j)$ to be the maximum score of alignment between $x_1 \ldots x_i$ and $y_1 \ldots y_j$ with $x_i$ being deletion.

\[
D(i, j) = \max \left\{ D(i - 1, j) - \text{ext}, S(i - 1, j) - \text{op} \right\}
\]

- Note that there is no deletion right after insertion, neither insertion right after deletion. Why?
Chapter 15. Dynamic Programming

Define \( D(i, j) \) to be the maximum score of alignment between \( x_1 \ldots x_i \) and \( y_1 \ldots y_j \) with \( x_i \) being deletion.

\[
D(i, j) = \max \{ D(i - 1, j) - ext, S(i - 1, j) - op \}
\]

- Note that there is no deletion right after insertion, neither insertion right after deletion. why?

- What is the desired value?
Chapter 15. Dynamic Programming

Define $D(i, j)$ to be the maximum score of alignment between $x_1 \ldots x_i$ and $y_1 \ldots y_j$ with $x_i$ being deletion.

$$D(i, j) = \max \left\{ D(i - 1, j) - \text{ext}, S(i - 1, j) - \text{op} \right\}$$

- Note that there is no deletion right after insertion, neither insertion right after deletion. why?

- What is the desired value? max \left\{ S(m, n), I(m, n), D(m, n) \right\}
Chapter 15. Dynamic Programming

Define $D(i, j)$ to be the maximum score of alignment between $x_1 \ldots x_i$ and $y_1 \ldots y_j$ with $x_i$ being deletion.

$$D(i, j) = \max \left\{ \begin{array}{l} D(i-1, j) - \text{ext} \\ S(i-1, j) - \text{op} \end{array} \right. $$

- Note that there is no deletion right after insertion, neither insertion right after deletion. why?

- What is the desired value? \[ \max \left\{ \begin{array}{l} S(m, n) \\ I(m, n) \\ D(m, n) \end{array} \right. \]

- How to fill the table(s) and to traceback the optimal alignment?
Chapter 15. Dynamic Programming

DP applied to summation problems
Chapter 15. Dynamic Programming

DP applied to summation problems

Unmatched Socks Problem

There are $n$ pairs of socks, each a different color;
• each day, remove arbitrarily two individual socks;
compute the probability that, on none of the $k$ days, a pair of color-matched socks is removed.

When $n = 5$, $k = 1$

$$P_1 = \frac{\text{number of color-not-matched pairs}}{\text{number of total pairs}} = \frac{10 \times 8}{10^2} = \frac{40}{45} = \frac{8}{9}$$

When $n = 5$, $k = 2$

$$P_2 = P_1 \times \left( P_{u} + P_{m} + P_{u,m} \right) = \frac{8}{9} \times \left( \frac{1}{8^2} + \frac{6 \times 4}{2 \times 8^2} + 2 \times 6 \times 8^2 \right) = \frac{8}{9} \times \frac{25}{28}$$

$$P_3 = P_2 \times \cdots$$
Chapter 15. Dynamic Programming

DP applied to summation problems

Unmatched Socks Problem

There are \( n \) pairs of socks, each a different color;

- each day, remove arbitrarily two individual socks;

compute the probability that, on none of the \( k \) days, a pair of color-matched socks is removed.
Chapter 15. Dynamic Programming

DP applied to summation problems

Unmatched Socks Problem

There are $n$ pairs of socks, each a different color;
• each day, remove arbitrarily two individual socks;
compute the probability that, on none of the $k$ days, a pair of color-matched socks is removed.

When $n = 5$, $k = 1$
Chapter 15. Dynamic Programming

DP applied to summation problems

Unmatched Socks Problem

There are \( n \) pairs of socks, each a different color;
- each day, remove arbitrarily two individual socks;
compute the probability that, on none of the \( k \) days, a pair of color-matched socks
is removed.

When \( n = 5, \ k = 1 \)

\[
P_1 = \frac{\text{number of color-not-matched pairs}}{\text{number of total pairs}} = \frac{10 \times 8}{2(10^2)} = \frac{40}{45} = \frac{8}{9}
\]
Chapter 15. Dynamic Programming

DP applied to summation problems

Unmatched Socks Problem

There are \( n \) pairs of socks, each a different color;
• each day, remove arbitrarily two individual socks;
compute the probability that, on none of the \( k \) days, a pair of color-matched socks is removed.

When \( n = 5 \), \( k = 1 \)

\[
P_1 = \frac{\text{number of color-not-matched pairs}}{\text{number of total pairs}}
\]
Chapter 15. Dynamic Programming

**DP applied to summation problems**

**Unmatched Socks Problem**

There are $n$ pairs of socks, each a different color;
- each day, remove arbitrarily two individual socks;
compute the probability that, on *none of the $k$ days*, a pair of color-matched socks is removed.

When $n = 5$, $k = 1$

$$P_1 = \frac{\text{number of color-not-matched pairs}}{\text{number of total pairs}} = \frac{(10 \times 8)/2}{(10 \times 8)/2}$$
Chapter 15. Dynamic Programming

DP applied to summation problems

Unmatched Socks Problem

There are \( n \) pairs of socks, each a different color;
- each day, remove arbitrarily two individual socks;
compute the probability that, on none of the \( k \) days, a pair of color-matched socks is removed.

When \( n = 5, \ k = 1 \)

\[
P_1 = \frac{\text{number of color-not-matched pairs}}{\text{number of total pairs}} = \frac{(10 \times 8)/2}{\binom{10}{2}} = \]

Chapter 15. Dynamic Programming

DP applied to summation problems

Unmatched Socks Problem

There are \( n \) pairs of socks, each a different color;
• each day, remove arbitrarily two individual socks;
compute the probability that, on none of the \( k \) days, a pair of color-matched socks
is removed.

When \( n = 5, \ k = 1 \)

\[
P_1 = \frac{\text{number of color-not-matched pairs}}{\text{number of total pairs}} = \frac{(10 \times 8)/2}{\binom{10}{2}} = \frac{40}{45} =
\]
Chapter 15. Dynamic Programming

DP applied to summation problems

Unmatched Socks Problem

There are $n$ pairs of socks, each a different color;
• each day, remove arbitrarily two individual socks;
compute the probability that, on none of the $k$ days, a pair of color-matched socks is removed.

When $n = 5$, $k = 1$

$$P_1 = \frac{\text{number of color-not-matched pairs}}{\text{number of total pairs}} = \frac{(10 \times 8)/2}{\binom{10}{2}} = \frac{40}{45} = \frac{8}{9}$$
Chapter 15. Dynamic Programming

**DP applied to summation problems**

**Unmatched Socks Problem**

There are \( n \) pairs of socks, each a different color;
- each day, remove arbitrarily two individual socks;
compute the probability that, on *none of the \( k \) days*, a pair of color-matched socks is removed.

When \( n = 5 \), \( k = 1 \)

\[
P_1 = \frac{\text{number of color-not-matched pairs}}{\text{number of total pairs}} = \frac{(10 \times 8)/2}{\binom{10}{2}} = \frac{40}{45} = \frac{8}{9}
\]

When \( n = 5 \), \( k = 2 \)
Chapter 15. Dynamic Programming

**DP applied to summation problems**

**Unmatched Socks Problem**

There are $n$ pairs of socks, each a different color;  
- each day, remove arbitrarily two individual socks;  
compute the probability that, on **none of the $k$ days**, a pair of color-matched socks  
is removed.

When $n = 5$, $k = 1$

$$P_1 = \frac{\text{number of color-not-matched pairs}}{\text{number of total pairs}} = \frac{(10 \times 8)/2}{\binom{10}{2}} = \frac{40}{45} = \frac{8}{9}$$

When $n = 5$, $k = 2$

$$P_2 = P_1$$
Chapter 15. Dynamic Programming

DP applied to summation problems

Unmatched Socks Problem

There are \( n \) pairs of socks, each a different color;
• each day, remove arbitrarily two individual socks;
compute the probability that, on none of the \( k \) days, a pair of color-matched socks is removed.

When \( n = 5 \), \( k = 1 \)

\[
P_1 = \frac{\text{number of color-not-matched pairs}}{\text{number of total pairs}} = \frac{(10 \times 8)/2}{\binom{10}{2}} = \frac{40}{45} = \frac{8}{9}
\]

When \( n = 5 \), \( k = 2 \)

\[
P_2 = P_1 \times (P_u + P_m + P_{u,m}) = \]

Chapter 15. Dynamic Programming

DP applied to summation problems

Unmatched Socks Problem

There are $n$ pairs of socks, each a different color;
• each day, remove arbitrarily two individual socks;
compute the probability that, on none of the $k$ days, a pair of color-matched socks
is removed.

When $n = 5$, $k = 1$

\[ P_1 = \frac{\text{number of color-not-matched pairs}}{\text{number of total pairs}} = \frac{(10 \times 8)/2}{\binom{10}{2}} = \frac{40}{45} = \frac{8}{9} \]

When $n = 5$, $k = 2$

\[ P_2 = P_1 \times (P_u + P_m + P_{u,m}) = \frac{8}{9} \times \left( \frac{1}{\binom{8}{2}} + \frac{(6 \times 4)/2}{\binom{8}{2}} + \frac{2 \times 6}{\binom{8}{2}} \right) = \]
Chapter 15. Dynamic Programming

**DP applied to summation problems**

**Unmatched Socks Problem**

There are \( n \) pairs of socks, each a different color;
- each day, remove arbitrarily two individual socks;
compute the probability that, on none of the \( k \) days, a pair of color-matched socks is removed.

When \( n = 5 \), \( k = 1 \)

\[
P_1 = \frac{\text{number of color-not-matched pairs}}{\text{number of total pairs}} = \frac{(10 \times 8)/2}{\binom{10}{2}} = \frac{40}{45} = \frac{8}{9}
\]

When \( n = 5 \), \( k = 2 \)

\[
P_2 = P_1 \times (P_u + P_m + P_{u,m}) = \frac{8}{9} \times \left( \frac{1}{\binom{8}{2}} + \frac{(6 \times 4)/2}{\binom{8}{2}} + \frac{2 \times 6}{\binom{8}{2}} \right) = \frac{8}{9} \times \frac{25}{28}
\]
Chapter 15. Dynamic Programming

DP applied to summation problems

Unmatched Socks Problem

There are \( n \) pairs of socks, each a different color;
• each day, remove arbitrarily two individual socks;
compute the probability that, on none of the \( k \) days, a pair of color-matched socks
is removed.

When \( n = 5, \ k = 1 \)

\[
P_1 = \frac{\text{number of color-not-matched pairs}}{\text{number of total pairs}} = \frac{(10 \times 8)/2}{\binom{10}{2}} = \frac{40}{45} = \frac{8}{9}
\]

When \( n = 5, \ k = 2 \)

\[
P_2 = P_1 \times (P_u + P_m + P_{u,m}) = \frac{8}{9} \times \left( \frac{1}{\binom{8}{2}} + \frac{(6 \times 4)/2}{\binom{8}{2}} + \frac{2 \times 6}{\binom{8}{2}} \right) = \frac{8}{9} \times \frac{25}{28}
\]

\[
P_3 = P_2 \times ??
\]
Chapter 15. Dynamic Programming

Unfortunately, the probability of individual days cannot be independently calculated,
Unfortunately, the probability of individual days cannot be independently calculated, but conditional on the choices made on the previous day.
Chapter 15. Dynamic Programming

Unfortunately, the probability of individual days cannot be independently calculated, but conditional on the choices made on the previous day. So we want to catalog choices to correct computation.
Chapter 15. Dynamic Programming

Unfortunately, the probability of individual days cannot be independently calculated, but conditional on the choices made on the previous day.

So we want to catalog choices to correct computation.

Let $m_k$ be the number of pairs of color-matched socks after selection on the $k$th day.
Chapter 15. Dynamic Programming

Unfortunately, the probability of individual days cannot be independently calculated, but conditional on the choices made on the previous day.

So we want to catalog choices to correct computation.

Let $m_k$ be the number of pairs of color-matched socks after selection on the $k$th day. Let $u_k$ be the number of unmatched socks after selection on the $k$th day.
Unfortunately, the probability of individual days cannot be independently calculated, but conditional on the choices made on the previous day.

So we want to catalog choices to correct computation.

Let $m_k$ be the number of pairs of color-matched socks after selection on the $k$th day.

Let $u_k$ be the number of unmatched socks after selection on the $k$th day.

Note that

$$2m_k + u_k = 2n - 2k, \text{ for every } k \leq n$$
Chapter 15. Dynamic Programming

Unfortunately, the probability of individual days cannot be independently calculated, but conditional on the choices made on the previous day.

So we want to catalog choices to correct computation.

Let $m_k$ be the number of pairs of color-matched socks after selection on the $k$th day.

Let $u_k$ be the number of unmatched socks after selection on the $k$th day.

Note that

$$2m_k + u_k = 2n - 2k, \text{ for every } k \leq n$$

We define $P_k(m_k, u_k)$ to be the total probability that not a pair of matched socks is selected on any day of the first $k$ days, resulting in $m_k$ and $u_k$. Then
Unfortunately, the probability of individual days cannot be independently calculated, but conditional on the choices made on the previous day. So we want to catalog choices to correct computation.

Let \( m_k \) be the number of pairs of color-matched socks after selection on the \( k \)th day. Let \( u_k \) be the number of unmatched socks after selection on the \( k \)th day.

Note that

\[
2m_k + u_k = 2n - 2k, \quad \text{for every } k \leq n
\]

We define \( P_k(m_k, u_k) \) to be the total probability that not a pair of matched socks is selected on any day of the first \( k \) days, resulting in \( m_k \) and \( u_k \). Then

\[
P_k(m_k, u_k) = \sum \left\{ \begin{array}{c}
P_{k-1}(m_k + 2, u_k - 2) \times \frac{(m_k+2)(2m_k+2)}{(2m_k+2+u_k)} \\
P_{k-1}(m_k + 1, u_k) \times \frac{2(m_k+1)u_k}{(2m_k+2+u_k)} \\
P_{k-1}(m_k, u_k + 2) \times \frac{u_k+2}{(2m_k+2+u_k)} \\
\end{array} \right. \]
Chapter 15. Dynamic Programming

We have seen a number of examples with "forward DP" solutions.

Matrix Chain Multiplication problem
Input: matrices $A_1, \cdots, A_n$, where $A_i$ has dimension $p_{i-1} \times p_i$;
Output: a parenthesization by which the product $A_1 \times A_2 \times \cdots \times A_n$ uses the minimum number of scalar multiplications.
Chapter 15. Dynamic Programming

We have seen a number of examples with "forward DP" solutions.

(Most of these problems are solvable with "backward DP" as well).

Matrix Chain Multiplication problem

Input: matrices $A_1, \ldots, A_n$, where $A_i$ has dimension $(p_{i-1} \times p_i)$;

Output: a parenthesization by which the product $A_1 \times A_2 \times \cdots \times A_n$ uses the minimum number of scalar multiplications.
Chapter 15. Dynamic Programming

We have seen a number of examples with "forward DP" solutions. (Most of these problems are solvable with "backward DP" as well). We now consider an example with an "inside-out DP" solution.

Matrix Chain Multiplication problem
Input: matrices $A_1, \cdots, A_n$, where $A_i$ has dimension $p_{i-1} \times p_i$;
Output: a parenthesization by which the product $A_1 \times A_2 \times \cdots \times A_n$ uses the minimum number of scalar multiplications.
We have seen a number of examples with "forward DP" solutions. (Most of these problems are solvable with "backward DP" as well).

We now consider an example with an "inside-out DP" solution.

**Matrix Chain Multiplication problem**
Chapter 15. Dynamic Programming

We have seen a number of examples with ”forward DP” solutions.
(Most of these problems are solvable with ”backward DP” as well).
We now consider an example with an ”inside-out DP” solution.

Matrix Chain Multiplication problem

**Input:** matrices $A_1, \cdots, A_n$, where $A_i$ has dimension $p_{i-1} \times p_i$;
**Output:** a parenthesization by which the product $A_1 \times A_2 \times \cdots \times A_n$
uses the minimum number of scalar multiplications.
Chapter 15. Dynamic Programming

We have seen a number of examples with "forward DP" solutions. 
(Most of these problems are solvable with "backward DP" as well).

We now consider an example with an "inside-out DP" solution.

Matrix Chain Multiplication problem

**Input:** matrices $A_1, \ldots, A_n$, where $A_i$ has dimension $p_{i-1} \times p_i$;

**Output:** a parenthesization by which the product $A_1 \times A_2 \times \cdots \times A_n$
uses the minimum number of scalar multiplications.
Chapter 15. Dynamic Programming

\[ 5 \times 5 \times 5 \times 4 \times 8 = 5 \times 2 \]
Chapter 15. Dynamic Programming

scalable multiplications

= \( 5 \times 4 \times 8 \)
Chapter 15. Dynamic Programming

Many possible parenthesizations

\[
\begin{align*}
\text{5x5} \times \text{5x4} \times \text{4x8} \times \text{8x2} = \text{5x2}
\end{align*}
\]
Many possible parenthesizations

1. \((A (B (C D)))\)
   \[5(5)2 + 5(4)2 + 4(8)2 = 154\]
2. \((A ((B C) D))\)
   \[5(5)2 + 5(4)8 + 5(8)2 = 290\]
3. \(((A B) (C D))\)
   \[5(5)4 + 4(8)2 + 5(4)2 = 204\]
4. \(((A (B C)) D)\)
   \[5(4)8 + 5(5)8 + 5(8)2 = 440\]
5. \(((A B) C) D\)
   \[5(5)4 + 5(4)8 + 5(8)2 = 340\]
Chapter 15. Dynamic Programming

Many possible parenthesizations

Note: The number of all possible parenthesizations is

\[ P(n) = \sum_{k=1}^{n-1} P(k) P(n-k), \text{ Catalan number} \]
Chapter 15. Dynamic Programming

**Step 1:** identify *optimal substructure*:
Optimal solution in terms of optimal solutions to subproblems:

1. The best parenthesization of $A_i A_{i+1} \cdots A_j$ must be $(A_i \cdots A_k) (A_k+1 \cdots A_j)$ for some $k, i \leq k < j$.
2. The optimal cost of $A_i A_{i+1} \cdots A_j$ must be the smallest among optimal costs for $(A_i \cdots A_k) (A_k+1 \cdots A_j)$ for $k = i, i+1, \ldots, j-1$.
3. For each $k$, the optimal cost for the above sequence is the optimal cost for $(A_i \cdots A_k) +$ the optimal cost for $(A_k+1 \cdots A_j) +$ the number of scalar multiplications to multiply the two terms.
Chapter 15. Dynamic Programming

**Step 1**: identify *optimal substructure*:

Optimal solution in terms of optimal solutions to subproblems:

1. The best parenthesization of $A_i A_{i+1} \cdots A_j$ must be

   $$(A_i \cdots A_k) (A_{k+1} \cdots A_j)$$

   for some $k, i \leq k < j$. 
Chapter 15. Dynamic Programming

**Step 1:** identify *optimal substructure*:

Optimal solution in terms of optimal solutions to subproblems:

1. The best parenthesization of \( A_i A_{i+1} \cdots A_j \) must be

\[
(A_i \cdots A_k)(A_{k+1} \cdots A_j)
\]

for some \( k, i \leq k < j \).

2. The optimal cost of \( A_i A_{i+1} \cdots A_j \) must be the smallest among optimal costs for

\[
(A_i \cdots A_k)(A_{k+1} \cdots A_j)
\]

\( k = i, i + 1, \cdots, j - 1 \).
Chapter 15. Dynamic Programming

**Step 1:** identify *optimal substructure*:

Optimal solution in terms of optimal solutions to subproblems:

1. The best parenthesization of $A_i A_{i+1} \cdots A_j$ must be

   $$(A_i \cdots A_k)(A_{k+1} \cdots A_j)$$

   for some $k, i \leq k < j$.

2. The optimal cost of $A_i A_{i+1} \cdots A_j$ must be the smallest among optimal costs for

   $$(A_i \cdots A_k)(A_{k+1} \cdots A_j)$$

   $k = i, i + 1, \ldots, j - 1$.

3. For each $k$, the optimal cost for the above =

   - the optimal cost for $(A_i \cdots A_k) +$
   - the optimal cost for $(A_{k+1} \cdots A_j) +$
   - the number of scalar multiplications to multiply the two terms.
Step 2: objective function and recurrence

Define $f(i, j)$ to be the minimum number of scalar multiplications needed for $A_i A_{i+1} \cdots A_j$. Then

$$f(i, j) = \min_{i \leq k < j} \{ f(i, k) + f(k+1, j) + p_{i-1} p_k p_{j-1} \}$$

where base case is $f(i, j) = 0$ when $i = j$, for single matrix $A_i$. 
Step 2: objective function and recurrence

Define $f(i, j)$ to be the minimum number of scalar multiplications needed for $A_i A_{i+1} \cdots A_j$. 
**Step 2**: objective function and recurrence

Define $f(i, j)$ to be the minimum number of scalar multiplications needed for $A_i A_{i+1} \cdots A_j$. Then

$$f(i, j) = \min_{i \leq k < j} \{ f(i, k) + f(k+1, j) + p_{i-1}p_kp_j \}$$

where base case is

$$f(i, j) = 0 \text{ when } i = j, \text{ for single matrix } A_i.$$
Step 3. Fill out the DP table for function $f$. 
Step 3. Fill out the DP table for function $f$.

- Table size $n \times n$;
Step 3. Fill out the DP table for function \( f \).

- Table size \( n \times n \);
- base cases are on the main diagonal;
Step 3. Fill out the DP table for function $f$.

- Table size $n \times n$;
- base cases are on the main diagonal;
- smaller instances are of smaller $j-1$ values;
Step 3. Fill out the DP table for function $f$.

- Table size $n \times n$;
- base cases are on the main diagonal;
- smaller instances are of smaller $j-1$ values;
- diagonal cells have the same $j-1$ values;
Chapter 15. Dynamic Programming

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>20</td>
<td>35</td>
<td>53</td>
<td>95</td>
<td>99</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>60</td>
<td>132</td>
<td>270</td>
<td>95</td>
<td>95</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>90</td>
<td>231</td>
<td>75</td>
<td>75</td>
<td>75</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>126</td>
<td>60</td>
<td>60</td>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>42</td>
<td>42</td>
<td>42</td>
<td>42</td>
<td>42</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Side lengths: 1, 4, 5, 3, 6, 7, 1

\[ A_1^{(1 \times 4)} \times A_2^{(4 \times 5)} \times A_3^{(5 \times 3)} \times A_4^{(3 \times 6)} \times A_5^{(6 \times 7)} \times A_6^{(7 \times 1)} \]
Algorithm $\text{MatrixChainOrder}(p)$
1. $n = \text{length}[p] - 1$;
Chapter 15. Dynamic Programming

Algorithm MatrixChainOrder(p)
1. \( n = \text{length}[p] - 1; \)
2. \textbf{for } i = 1 \textbf{ to } n
Algorithm \textsc{MatrixChainOrder}(p)
1. \hspace{1em} n = \text{length}[p] - 1;
2. \hspace{1em} \textbf{for} i = 1 \textbf{ to } n
3. \hspace{2em} M[i, i] = 0;
Chapter 15. Dynamic Programming

Algorithm \textsc{MatrixChainOrder}(p)
1. \( n = \text{length}[p] - 1; \)
2. \textbf{for} \( i = 1 \) \textbf{to} \( n \)
3. \( M[i, i] = 0; \)
4. \textbf{for} \( l = 2 \) \textbf{to} \( n \)
Algorithm MatrixChainOrder($p$)
1. \( n = \text{length}[p] - 1; \)
2. \( \text{for } i = 1 \text{ to } n \)
3. \( M[i, i] = 0; \)
4. \( \text{for } l = 2 \text{ to } n \)
5. \( \text{for } i = 1 \text{ to } n - l + 1 \)
Chapter 15. Dynamic Programming

Algorithm \textsc{MatrixChainOrder}(p)
1. \hspace{1em} n = \text{length}[p] - 1;
2. \hspace{1em} \textbf{for} \ i = 1 \ \textbf{to} \ n
3. \hspace{2em} M[i, i] = 0;
4. \hspace{1em} \textbf{for} \ l = 2 \ \textbf{to} \ n
5. \hspace{2em} \textbf{for} \ i = 1 \ \textbf{to} \ n - l + 1
6. \hspace{3em} j = i + l - 1;
Algorithm \textsc{MatrixChainOrder}(p)

1. \hspace{1em} n = \text{length}[p] - 1;
2. \hspace{1em} \textbf{for} \ i = 1 \ \textbf{to} \ n \hspace{1em}
3. \hspace{1em} \hspace{1em} M[i, i] = 0;
4. \hspace{1em} \textbf{for} \ l = 2 \ \textbf{to} \ n \hspace{1em}
5. \hspace{1em} \hspace{1em} \textbf{for} \ i = 1 \ \textbf{to} \ n - l + 1 \hspace{1em}
6. \hspace{1em} \hspace{1em} \hspace{1em} j = i + l - 1; \hspace{1em}
7. \hspace{1em} \hspace{1em} \hspace{1em} M[i, j] = \infty;
Chapter 15. Dynamic Programming

Algorithm \textsc{MatrixChainOrder}(p)

1. \hspace{1em} \textit{n} = \text{length}[p] - 1;
2. \hspace{1em} \textbf{for} \ i = 1 \ \textbf{to} \ n
3. \hspace{3em} M[i,i] = 0;
4. \hspace{1em} \textbf{for} \ l = 2 \ \textbf{to} \ n
5. \hspace{3em} \textbf{for} \ i = 1 \ \textbf{to} \ n - l + 1
6. \hspace{5em} j = i + l - 1;
7. \hspace{3em} M[i,j] = \infty;
8. \hspace{3em} \textbf{for} \ k = i \ \textbf{to} \ j - 1
Algorithm \textsc{MatrixChainOrder}(p)

1. \( n = \text{length}[p] - 1; \)
2. \hspace{1em} \textbf{for} \( i = 1 \) \textbf{to} \( n \)
3. \hspace{2em} \( M[i, i] = 0; \)
4. \hspace{1em} \textbf{for} \( l = 2 \) \textbf{to} \( n \)
5. \hspace{2em} \textbf{for} \( i = 1 \) \textbf{to} \( n - l + 1 \)
6. \hspace{3em} \( j = i + l - 1; \)
7. \hspace{2em} \( M[i, j] = \infty; \)
8. \hspace{2em} \textbf{for} \( k = i \) \textbf{to} \( j - 1 \)
9. \hspace{3em} \( q = M[i, k] + M[k + 1, j] + p[i - 1]p[k]p[j] \)
Algorithm \textsc{MatrixChainOrder}(p)

1. \hspace{1em} n = \text{length}[p] - 1;
2. \hspace{1em} \textbf{for} \ i = 1 \ \textbf{to} \ n \\
3. \hspace{1em} M[i, i] = 0;
4. \hspace{1em} \textbf{for} \ l = 2 \ \textbf{to} \ n \\
5. \hspace{2em} \textbf{for} \ i = 1 \ \textbf{to} \ n - l + 1 \\
6. \hspace{3em} j = i + l - 1; \\
7. \hspace{3em} M[i, j] = \infty; \\
8. \hspace{3em} \textbf{for} \ k = i \ \textbf{to} \ j - 1 \\
10. \hspace{4em} \textbf{if} \ q < M[i, j]
Algorithm MatrixChainOrder(p)
1. \( n = \text{length}[p] - 1; \)
2. \( \text{for } i = 1 \text{ to } n \)
3. \( M[i, i] = 0; \)
4. \( \text{for } l = 2 \text{ to } n \)
5. \( \text{for } i = 1 \text{ to } n - l + 1 \)
6. \( j = i + l - 1; \)
7. \( M[i, j] = \infty; \)
8. \( \text{for } k = i \text{ to } j - 1 \)
10. \( \text{if } q < M[i, j] \)
11. \( M[i, j] = q \)
Algorithm \textsc{MatrixChainOrder}(p)

1. \( n = \text{length}[p] - 1; \)
2. \textbf{for} \( i = 1 \) \textbf{to} \( n \)
3. \( M[i, i] = 0; \)
4. \textbf{for} \( l = 2 \) \textbf{to} \( n \)
5. \textbf{for} \( i = 1 \) \textbf{to} \( n - l + 1 \)
6. \( j = i + l - 1; \)
7. \( M[i, j] = \infty; \)
8. \textbf{for} \( k = i \) \textbf{to} \( j - 1 \)
10. \textbf{if} \( q < M[i, j] \)
11. \( M[i, j] = q \)
12. \( S[i, j] = k \)

Time complexity: \( O(n^2 \times n) \)
Algorithm \textsc{MatrixChainOrder}(p) 
1. \hspace{0.5cm} n = length[p] - 1;
2. \hspace{0.5cm} \textbf{for } i = 1 \textbf{ to } n
3. \hspace{0.5cm} M[i, i] = 0;
4. \hspace{0.5cm} \textbf{for } l = 2 \textbf{ to } n
5. \hspace{1.5cm} \textbf{for } i = 1 \textbf{ to } n - l + 1
6. \hspace{2cm} j = i + l - 1;
7. \hspace{1.5cm} M[i, j] = \infty;
8. \hspace{2cm} \textbf{for } k = i \textbf{ to } j - 1
9. \hspace{3cm} q = M[i, k] + M[k + 1, j] + p[i - 1]p[k]p[j]
10. \hspace{3cm} \textbf{if } q < M[i, j]
11. \hspace{4cm} M[i, j] = q
12. \hspace{4cm} S[i, j] = k
13. \hspace{0.5cm} \textbf{return} (M, S)
Algorithm MatrixChainOrder\((p)\)

1. \(n = \text{length}[p] - 1\);
2. for \(i = 1\) to \(n\)
3. \(M[i, i] = 0\);
4. for \(l = 2\) to \(n\)
5. for \(i = 1\) to \(n - l + 1\)
6. \(j = i + l - 1\);
7. \(M[i, j] = \infty\);
8. for \(k = i\) to \(j - 1\)
9. \(q = M[i, k] + M[k + 1, j] + p[i - 1]p[k]p[j]\)
10. if \(q < M[i, j]\)
11. \(M[i, j] = q\)
12. \(S[i, j] = k\)
13. return \((M, S)\)

Time complexity: \(O(n^2 \times n)\)
Step 4. obtain the optimization parenthesization
Step 4. obtain the optimization parenthesization

Algorithm \textsc{MatrixChainParenthesization}(i, j, S)
1. if $i < j$
2. \hspace{1cm} $k = s[i, j]$
3. \hspace{1cm} \textbf{print} $(i, j, " : ", k)$
4. \hspace{1cm} \textsc{MatrixChainParenthesization}(i, k, S)
5. \hspace{1cm} \textsc{MatrixChainParenthesization}(k + 1, j, S)
6. \hspace{1cm} return
Step 4. obtain the optimization parenthesization

Algorithm $\text{MatrixChainParenthesization}(i, j, S)$
1. if $i < j$
2. $k = s[i, j]$
3. print $(i, j, " : ", k)$
4. $\text{MatrixChainParenthesization}(i, k, S)$
5. $\text{MatrixChainParenthesization}(k + 1, j, S)$
6. return

Usage: call $\text{MatrixChainParenthesization}(1, n, S)$
Step 4. obtain the optimization parenthesization

Algorithm \textsc{MatrixChainParenthesization}(i, j, S)
1. if $i < j$
2. \hspace{1em} $k = s[i, j]$
3. \hspace{1em} print $(i, j, " : ", k)$
4. \hspace{1em} \textsc{MatrixChainParenthesization}(i, k, S)
5. \hspace{1em} \textsc{MatrixChainParenthesization}(k + 1, j, S)
6. return

Usage: call \textsc{MatrixChainParenthesization}(1, n, S)

Time complexity for \textsc{MatrixChainParenthesization}:
**Step 4.** obtain the optimization parenthesization

Algorithm $\text{MatrixChainParenthesization}(i, j, S)$

1. if $i < j$
2. $k = s[i, j]$
3. print $(i, j, " : ", k)$
4. $\text{MatrixChainParenthesization}(i, k, S)$
5. $\text{MatrixChainParenthesization}(k + 1, j, S)$
6. return

Usage: call $\text{MatrixChainParenthesization}(1, n, S)$

Time complexity for $\text{MatrixChainParenthesization}$: $O(n)$. 
Chapter 16. Greedy Algorithms

Chapter 16 Greedy Algorithms
Chapter 16. Greedy Algorithms

Chapter 16 Greedy Algorithms

- Dynamic programming is to consider all possible choices and select the best.
Chapter 16. Greedy Algorithms

- Dynamic programming is to \textit{consider all possible choices and select the best.}
  
  a DP approach always leads to the optimal solution
Chapter 16. Greedy Algorithms

Chapter 16 Greedy Algorithms

- Dynamic programming is to *consider all possible choices and select the best.*
  - A DP approach always leads to the optimal solution
- A greedy algorithm may *ignore some choices and select the one that is locally the best.*
Chapter 16. Greedy Algorithms

Chapter 16 Greedy Algorithms

- Dynamic programming is to *consider all possible choices and select the best.*
  
a DP approach always leads to the optimal solution

- A greedy algorithm may *ignore some choices and select the one that is locally the best.*
  
a greedy strategy may or may NOT lead to the optimal solution
Chapter 16. Greedy Algorithms

Activity-selection problem
Activity-selection problem

**Input:** \( n \) activities \( S = \{a_1, \ldots, a_n\} \), each with a start time \( s_i \) and finish time \( f_i \), \( 1 \leq i \leq n \)

**Output:** subset \( A \subseteq S \) of mutually compatible activities such that \( |A| \) is the maximum.
Chapter 16. Greedy Algorithms

**Activity-selection problem**

**Input:** \( n \) activities \( S = \{a_1, \ldots, a_n\} \), each with a start time \( s_i \) and finish time \( f_i \), \( 1 \leq i \leq n \)

**Output:** subset \( A \subseteq S \) of mutually compatible activities such that \( |A| \) is the maximum.

\[
A = \{a_1, a_3, a_6, a_8\}, \text{ or }
\]
Chapter 16. Greedy Algorithms

Activity-selection problem

**Input:** \( n \) activities \( S = \{a_1, \ldots, a_n\} \), each with a start time \( s_i \) and finish time \( f_i \), \( 1 \leq i \leq n \)

**Output:** subset \( A \subseteq S \) of mutually compatible activities such that \( |A| \) is the maximum.

\[
A = \{a_1, a_3, a_6, a_8\}, \text{ or } A = \{a_2, a_5, a_7, a_8\}, \text{ or }
\]
Chapter 16. Greedy Algorithms

Activity-selection problem

**Input:** \( n \) activities \( S = \{a_1, \ldots, a_n\} \), each with a start time \( s_i \) and finish time \( f_i \), \( 1 \leq i \leq n \)

**Output:** subset \( A \subseteq S \) of mutually compatible activities such that \(|A|\) is the maximum.

\[ A = \{a_1, a_3, a_6, a_8\}, \text{ or } A = \{a_2, a_5, a_7, a_8\}, \text{ or } A = \{a_2, a_5, a_7, a_9\}, \]
Chapter 16. Greedy Algorithms

Activity-selection problem

**Input:** \( n \) activities \( S = \{a_1, \ldots, a_n\} \), each with a start time \( s_i \) and finish time \( f_i \), \( 1 \leq i \leq n \)

**Output:** subset \( A \subseteq S \) of mutually compatible activities such that \( |A| \) is the maximum.

\[
A = \{a_1, a_3, a_6, a_8\}, \text{ or } A = \{a_2, a_5, a_7, a_8\}, \text{ or } A = \{a_2, a_5, a_7, a_9\}, \ldots
\]
Chapter 16. Greedy Algorithms

A dynamic programming solution
A dynamic programming solution

**Step 1** analysis of problem
Chapter 16. Greedy Algorithms

A dynamic programming solution

Step 1 analysis of problem
Chapter 16. Greedy Algorithms

A dynamic programming solution

**Step 1** analysis of problem

How to solve it recursively?
Chapter 16. Greedy Algorithms

For example, selecting $a_3$ would result in two subsets of activities to further consider: 
\{a_1\} and \{a_6, a_7, a_8, a_9\};
selecting $a_5$ would result in two subsets of activities to further consider: 
\{a_1, a_2\} and \{a_7\}.
For example, selecting $a_3$ would result in two subsets of activities to further consider: \( \{a_1\} \) and \( \{a_6, a_7, a_8, a_9\} \);
For example, selecting $a_3$ would result in two subsets of activities to further consider: $\{a_1\}$ and $\{a_6, a_7, a_8, a_9\}$;

selecting $a_5$ would result in two subsets of activities to further consider: $\{a_1, a_2\}$ and $\{a_7, a_8, a_9\}$;
selecting $a_5$ would result in two subsets of activities to further consider:

\{a_1, a_2\} and \{a_7, a_8, a_9\};

but if selecting again $a_8$ would result in two smaller subsets:

\{a_7\} and \{\}\; these subproblems are not “prefixes” or “suffixes”, they could be “middle segments”; we approach such problems with inside-out instead of forward or backward methods.
selecting $a_5$ would result in two subsets of activities to further consider: 
\{a_1, a_2\} and \{a_7, a_8, a_9\};
selecting $a_5$ would result in two subsets of activities to further consider: 
\{a_1, a_2\} and \{a_7, a_8, a_9\};

but if selecting again $a_8$ would result in two smaller subsets: 
\{a_7\} and \{\};
selecting $a_5$ would result in two subsets of activities to further consider: 
$\{a_1, a_2\}$ and $\{a_7, a_8, a_9\}$;

but if selecting again $a_8$ would result in two smaller subsets: 
$\{a_7\}$ and $\{\}$;

these subproblems are not “prefixes” or “suffixes”, they could be “middle segments”;}
selecting $a_5$ would result in two subsets of activities to further consider: 
$\{a_1, a_2\}$ and $\{a_7, a_8, a_9\}$; 

but if selecting again $a_8$ would result in two smaller subsets: 
$\{a_7\}$ and $\{\}$; 

these subproblems are not “prefixes” or “suffixes”, they could be “middle segments”; 

we approach such problems with inside-out instead of forward or backward methods.
Chapter 16. Greedy Algorithms

So we consider solve activity selection on subsets of tasks:

Define subset $S_{i,j} = \{ a_k \in S : f_i \leq s_k < f_k \leq s_j \}$

e.g., $S_{2,9} = \{ a_5, a_6, a_7 \}$

$S_{3,8} = \{ a_6, a_7 \}$.

Introducing dummy activities: $a_0$ with $s_0 = f_0$ = the earliest start time of all, and

$a_{n+1}$ with $s_{n+1} = f_{n+1}$ = latest finish time of all.
So we consider solve activity selection on subsets of tasks:

Define subset $S_{i,j} = \{a_k \in S: f_i \leq s_k < f_k \leq s_j\}$

- $S_{2,9} = \{a_5, a_6, a_7\}$
- $S_{3,8} = \{a_6, a_7\}$

Introducing dummy activities:
- $a_0$ with $s_0 = f_0$ = the earliest start time of all,
- $a_{n+1}$ with $s_{n+1} = f_{n+1} = $ latest finish time of all.
Chapter 16. Greedy Algorithms

So we consider solve activity selection on subsets of tasks:

Define subset $S_{i,j} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$

So we consider solve activity selection on **subsets of tasks**:

Define subset $S_{i,j} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$
Chapter 16. Greedy Algorithms

So we consider solve activity selection on subsets of tasks:

Define subset $S_{i,j} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$

e.g.,

$S_{2,9} = \{a_k \in S : f_2 \leq s_k < f_k \leq s_9\} =$
Chapter 16. Greedy Algorithms

So we consider solve activity selection on subsets of tasks:

Define subset $S_{i,j} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$

e.g.,

$S_{2,9} = \{a_k \in S : f_2 \leq s_k < f_k \leq s_9\} = \{a_5, a_6, a_7\}$
So we consider solve activity selection on subsets of tasks:

Define subset $S_{i,j} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$

e.g.,

$S_{2,9} = \{a_k \in S : f_2 \leq s_k < f_k \leq s_9\} = \{a_5, a_6, a_7\} \quad S_{3,8} = \{a_6, a_7\}$. 
Chapter 16. Greedy Algorithms

So we consider solve activity selection on subsets of tasks:

Define subset $S_{i,j} = \{ a_k \in S : f_i \leq s_k < f_k \leq s_j \}$

e.g.,

$S_{2,9} = \{ a_k \in S : f_2 \leq s_k < f_k \leq s_9 \} = \{ a_5, a_6, a_7 \}$

$S_{3,8} = \{ a_6, a_7 \}$.

introducing dummy activities:

$a_0$ with $s_0 = f_0 =$the earliest start time of all, and

$a_{n+1}$ with $s_{n+1} = f_{n+1} =$ latest finish time of all.
Define subset $S_{i,j} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$
Define subset $S_{i,j} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$

Define solution $A_{i,j} \subseteq S_{i,j}$ is the maximum size set of compatible activities.
Define subset $S_{i,j} = \{ a_k \in S : f_i \leq s_k < f_k \leq s_j \}$

Define solution $A_{i,j} \subseteq S_{i,j}$ is the maximum size set of compatible activities.

Then

$$|A_{i,j}| = \max_{a_k \in S_{i,j}} \{|A_{i,k} \cup \{a_k\} \cup A_{k,j}|\}$$
Chapter 16. Greedy Algorithms

Define subset $S_{i,j} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$

Define solution $A_{i,j} \subseteq S_{i,j}$ is the maximum size set of compatible activities.

Then

$$|A_{i,j}| = \max_{a_k \in S_{i,j}} \{|A_{i,k} \cup \{a_k\} \cup A_{k,j}|\}$$

e.g.,

$$|A_{0,10}| = \max \left\{ |A_{0,6} \cup \{a_6\} \cup A_{6,10}|, \text{ where } S_{0,6} = \{a_1, \ldots, a_4\}, S_{6,10} = \{a_8, a_9\} \right\}$$
Chapter 16. Greedy Algorithms

Instead of solving this with DP, we wonder if all but one of the subproblems could be made empty.

\[
A_{i,j} = \max_{a_k \in S_{i,j}} \{|A_{i,k} \cup \{a_k\} \cup A_{k,j}|\}
\]

i.e., a greedy choice of \(a_k \in S_{i,j}\) makes \(A_{i,l} \cup \{a_l\} \cup A_{l,i}\) not needed, for all \(l \neq k\).

But the chosen \(a_k\) has to guarantee to maintain \(|A_{i,j}|\) is the maximum.

called greedy-choice property (need to be proved)
Instead of solving this with DP, we wonder if all but one of the subproblems could be made empty.

\[ |A_{i,j}| = \max_{a_k \in S_{i,j}} \{|A_{i,k} \cup \{a_k\} \cup A_{k,j}|\} \]
Chapter 16. Greedy Algorithms

Instead of solving this with DP, we wonder if all but one of the subproblems could be made empty.

\[ |A_{i,j}| = \max_{a_k \in S_{i,j}} \{|A_{i,k} \cup \{a_k\} \cup A_{k,j}| \} \]

i.e., a greedy choice of \(a_k \in S_{i,j}\) makes \(A_{i,l} \cup \{a_l\} \cup A_{l,i}\) not needed, for all \(l \neq k\)
Instead of solving this with DP, we wonder if all but one of the subproblems could be made empty.

\[ |A_{i,j}| = \max_{a_k \in S_{i,j}} \{|A_{i,k} \cup \{a_k\} \cup A_{k,j}| \} \]

i.e., a greedy choice of \( a_k \in S_{i,j} \) makes \( A_{i,l} \cup \{a_l\} \cup A_{l,i} \) not needed, for all \( l \neq k \)

But the chosen \( a_k \) has to guarantee to maintain \( |A_{i,j}| \) is the maximum.
Instead of solving this with DP, we wonder if all but one of the subproblems could be made empty.

\[ |A_{i,j}| = \max_{a_k \in S_{i,j}} \{|A_{i,k} \cup \{a_k\} \cup A_{k,j}|\} \]

i.e., a greedy choice of \(a_k \in S_{i,j}\) makes \(A_{i,l} \cup \{a_l\} \cup A_{l,i}\) not needed, for all \(l \neq k\)

But the chosen \(a_k\) has to guarantee to maintain \(|A_{i,j}|\) is the maximum.

called greedy-choice property
Instead of solving this with DP, we wonder if all but one of the subproblems could be made empty.

\[ |A_{i,j}| = \max_{a_k \in S_{i,j}} \{|A_{i,k} \cup \{a_k\} \cup A_{k,j}|\} \]

i.e., a greedy choice of \(a_k \in S_{i,j}\) makes \(A_{i,l} \cup \{a_l\} \cup A_{l,i}\) not needed, for all \(l \neq k\)

But the chosen \(a_k\) has to guarantee to maintain \(|A_{i,j}|\) is the maximum.

called greedy-choice property (need to be proved)
Chapter 16. Greedy Algorithms
Chapter 16. Greedy Algorithms

Activity Selection problem has the greedy-choice property.

Theorem 16.1: Let $S_{i,j}$ be a subproblem and $a_k \in S_{i,j}$ be the activity of the earliest finish time, then $a_k$ is in some optimal solution for $S_{i,j}$.

The theorem is the same as that selecting the activity with the earliest finish time

• makes all other subproblems disappearing, and
• maintains the optimality of solution

Proof: (Using the "swapping method", or "exchange method")
Let $A_{i,j}$ be an optimal solution for $S_{i,j}$.

(1) If $a_k \in A_{i,j}$, then the theorem is true.

(2) If $a_k \not\in A_{i,j}$, assume $a_p$ is the one with earliest finish time in $A_{i,j}$.

We define $B_{i,j} = A_{i,j} - \{a_p\} \cup \{a_k\}$. Then

• because $f_p \geq f_k$, $a_k$ is compatible with other activities in $B_{i,j}$.
  So $B_{i,j}$ is a solution for $S_{i,j}$.

• $|B_{i,j}| = |A_{i,j}|$, so $B_{i,j}$ is also an optimal solution for $S_{i,j}$.

Since $a_k \in B_{i,j}$, we prove the theorem.
Activity Selection problem has the greedy-choice property.

**Theorem 16.1:** Let $S_{i,j}$ be a subproblem and $a_k \in S_{i,j}$ be the activity of the earliest finish time, then $a_k$ is in some optimal solution for $S_{i,j}$.
Activity Selection problem has the greedy-choice property.

**Theorem 16.1**: Let \( S_{i,j} \) be a subproblem and \( a_k \in S_{i,j} \) be the activity of the earliest finish time, then \( a_k \) is in some optimal solution for \( S_{i,j} \).

The theorem is the same as that selecting the activity with the earliest finish time.
Activity Selection problem has the greedy-choice property.

**Theorem 16.1**: Let $S_{i,j}$ be a subproblem and $a_k \in S_{i,j}$ be the activity of the earliest finish time, then $a_k$ is in some optimal solution for $S_{i,j}$.

The theorem is the same as that selecting the activity with the earliest finish time

- makes all other subproblems disappearing, and
Chapter 16. Greedy Algorithms

Activity Selection problem has the greedy-choice property.

**Theorem 16.1:** Let $S_{i,j}$ be a subproblem and $a_k \in S_{i,j}$ be the activity of the earliest finish time, then $a_k$ is in some optimal solution for $S_{i,j}$.

The theorem is the same as that selecting the activity with the earliest finish time
- makes all other subproblems disappearing, and
- maintains the optimality of solution
Activity Selection problem has the greedy-choice property.

**Theorem 16.1:** Let $S_{i,j}$ be a subproblem and $a_k \in S_{i,j}$ be the activity of the earliest finish time, then $a_k$ is in some optimal solution for $S_{i,j}$.

The theorem is the same as that selecting the activity with the earliest finish time

- makes all other subproblems disappearing, and
- maintains the optimality of solution

**Proof:** (Using the “swapping method”, or “exchange method”)

Activity Selection problem has the greedy-choice property.

**Theorem 16.1**: Let $S_{i,j}$ be a subproblem and $a_k \in S_{i,j}$ be the activity of the earliest finish time, then $a_k$ is in some optimal solution for $S_{i,j}$.

The theorem is the same as that **selecting the activity with the earliest finish time**

- makes all other subproblems disappearing, and
- maintains the optimality of solution

**Proof**: (Using the “swapping method”, or “exchange method”)

Let $A_{i,j}$ be an optimal solution for $S_{i,j}$. 
Activity Selection problem has the greedy-choice property.

**Theorem 16.1**: Let $S_{i,j}$ be a subproblem and $a_k \in S_{i,j}$ be the activity of the earliest finish time, then $a_k$ is in some optimal solution for $S_{i,j}$.

The theorem is the same as that selecting the activity with the earliest finish time

- makes all other subproblems disappearing, and
- maintains the optimality of solution

**Proof**: (Using the “swapping method”, or “exchange method”)

Let $A_{i,j}$ be an optimal solution for $S_{i,j}$.

(1) If $a_k \in A_{i,j}$, then the theorem is true.
Activity Selection problem has the greedy-choice property.

**Theorem 16.1:** Let $S_{i,j}$ be a subproblem and $a_k \in S_{i,j}$ be the activity of the earliest finish time, then $a_k$ is in **some** optimal solution for $S_{i,j}$.

The theorem is the same as that selecting the activity with the earliest finish time

- makes all other subproblems disappearing, and
- maintains the optimality of solution

**Proof:** (Using the “swapping method”, or “exchange method”)

Let $A_{i,j}$ be an optimal solution for $S_{i,j}$.

(1) If $a_k \in A_{i,j}$, then the theorem is true.

(2) If $a_k \not\in A_{i,j}$, assume $a_p$ is the one with earliest finish time in $A_{i,j}$.
Chapter 16. Greedy Algorithms

Activity Selection problem has the greedy-choice property.

**Theorem 16.1**: Let $S_{i,j}$ be a subproblem and $a_k \in S_{i,j}$ be the activity of the earliest finish time, then $a_k$ is in some optimal solution for $S_{i,j}$.

The theorem is the same as that selecting the activity with the earliest finish time

- makes all other subproblems disappearing, and
- maintains the optimality of solution

**Proof**: (Using the “swapping method”, or “exchange method”)

Let $A_{i,j}$ be an optimal solution for $S_{i,j}$.

(1) If $a_k \in A_{i,j}$, then the theorem is true.

(2) If $a_k \not\in A_{i,j}$, assume $a_p$ is the one with earliest finish time in $A_{i,j}$.

We define $B_{i,j} = A_{i,j} - \{a_p\} \cup \{a_k\}$. Then

- because $f_p \geq f_k$, $a_k$ is compatible with other activities in $B_{i,j}$.
  So $B_{i,j}$ is a solution for $S_{i,j}$. 

Chapter 16. Greedy Algorithms

Activity Selection problem has the greedy-choice property.

**Theorem 16.1**: Let $S_{i,j}$ be a subproblem and $a_k \in S_{i,j}$ be the activity of the earliest finish time, then $a_k$ is in some optimal solution for $S_{i,j}$.

The theorem is the same as that selecting the activity with the earliest finish time

- makes all other subproblems disappearing, and
- maintains the optimality of solution

**Proof**: (Using the “swapping method”, or “exchange method”)

Let $A_{i,j}$ be an optimal solution for $S_{i,j}$.

(1) If $a_k \in A_{i,j}$, then the theorem is true.

(2) If $a_k \not\in A_{i,j}$, assume $a_p$ is the one with earliest finish time in $A_{i,j}$.

We define $B_{i,j} = A_{i,j} - \{a_p\} \cup \{a_k\}$. Then

- because $f_p \geq f_k$, $a_k$ is compatible with other activities in $B_{i,j}$.
  So $B_{i,j}$ is a solution for $S_{i,j}$.
- $|B_{i,j}| = |A_{i,j}|$, so $B_{i,j}$ is also an optimal solution for $S_{i,j}$. 
Activity Selection problem has the greedy-choice property.

**Theorem 16.1**: Let $S_{i,j}$ be a subproblem and $a_k \in S_{i,j}$ be the activity of the earliest finish time, then $a_k$ is in some optimal solution for $S_{i,j}$.

The theorem is the same as that selecting the activity with the earliest finish time

- makes all other subproblems disappearing, and
- maintains the optimality of solution

**Proof**: (Using the “swapping method”, or “exchange method”)

Let $A_{i,j}$ be an optimal solution for $S_{i,j}$.

(1) If $a_k \in A_{i,j}$, then the theorem is true.

(2) If $a_k \notin A_{i,j}$, assume $a_p$ is the one with earliest finish time in $A_{i,j}$.

We define $B_{i,j} = A_{i,j} - \{a_p\} \cup \{a_k\}$. Then
- because $f_p \geq f_k$, $a_k$ is compatible with other activities in $B_{i,j}$.
  So $B_{i,j}$ is a solution for $S_{i,j}$.
- $|B_{i,j}| = |A_{i,j}|$, so $B_{i,j}$ is also an optimal solution for $S_{i,j}$.

Since $a_k \in B_{i,j}$, we prove the theorem.
Chapter 16. Greedy Algorithms

Using the Theorem 16.1 and the recurrence

$$|A_{i,j}| = \max_{a_k \in S_{i,j}} \{|A_{i,k} \cup \{a_k\} \cup A_{k,j}|\}$$

we can derive greedy algorithms for the Activity Selection problem.
Assume that the activities are sorted according to their finish times.
\[ f_1 \leq f_2 \leq \cdots \leq f_n. \]
Assume that the activities are sorted according to their finish times.

\[ f_1 \leq f_2 \leq \cdots \leq f_n. \]

Adding hypothetical activity \( a_0 \) (with \( s_0 = f_0 = s_1 \))
Assume that the activities are sorted according to their finish times.
\[ f_1 \leq f_2 \leq \cdots \leq f_n. \]

Adding hypothetical activity \( a_0 \) (with \( s_0 = f_0 = s_1 \)) and activity \( a_{n+1} \) (with \( s_{n+1} = f_{n+1} = f_n \)).

Algorithm Activity-Selection \((s,f,k,n)\)

1. \( m = k + 1 \)
2. while \( m \leq n \) and \( s_m < f_k \)
3. \( m = m + 1 \)
4. if \( m \leq n \)
5. return \( \{a_m\} \cup \text{Activity-Selection} (s,f,m,n) \)
6. else return \( (\emptyset) \)

First called with \( \text{Activity-Selection} (s,f_0,n) \)

Time complexity: \( O(n) \).
Assume that the activities are sorted according to their finish times. 
\( f_1 \leq f_2 \leq \cdots \leq f_n \).

Adding hypothetical activity \( a_0 \) (with \( s_0 = f_0 = s_1 \)) and activity \( a_{n+1} \) (with \( s_{n+1} = f_{n+1} = f_n \)).

**Algorithm** 

\( \text{Activity-Selection}(s, f, k, n) \)
Assume that the activities are sorted according to their finish times. 
\[ f_1 \leq f_2 \leq \cdots \leq f_n. \]

Adding hypothetical activity \( a_0 \) (with \( s_0 = f_0 = s_1 \)) and activity \( a_{n+1} \) (with \( s_{n+1} = f_{n+1} = f_n \)).

Algorithm **Activity-Selection** \( (s, f, k, n) \)

1. \( m = k + 1 \)
Chapter 16. Greedy Algorithms

Assume that the activities are sorted according to their finish times.
\[ f_1 \leq f_2 \leq \cdots \leq f_n. \]

Adding hypothetical activity \( a_0 \) (with \( s_0 = f_0 = s_1 \)) and activity \( a_{n+1} \) (with \( s_{n+1} = f_{n+1} = f_n \)).

Algorithm \textsc{Activity-Selection}(s, f, k, n)
1. \quad \text{\textit{m} = k + 1}
2. \quad \textbf{while} \text{ \textit{m} \leq n} \text{ and } s_\text{m} < f_k
   \quad \text{\textit{m} = \textit{m} + 1}
3. \quad \text{if} \text{ \textit{m} \leq n}
4. \quad \text{\text{\textit{return}}} \{a_\text{m}\} \cup \text{\textsc{Activity-Selection}(s, f, \text{m}, n)}
5. \quad \text{\textit{else} return } (\emptyset)
Chapter 16. Greedy Algorithms

Assume that the activities are sorted according to their finish times.
\[ f_1 \leq f_2 \leq \cdots \leq f_n. \]

Adding hypothetical activity \( a_0 \) (with \( s_0 = f_0 = s_1 \)) and activity \( a_{n+1} \) (with \( s_{n+1} = f_{n+1} = f_n \)).

Algorithm \textsc{Activity-Selection}(s, f, k, n)

1. \( m = k + 1 \)
2. \textbf{while} \( m \leq n \) and \( s_m < f_k \)
3. \( m = m + 1 \)
Assume that the activities are sorted according to their finish times. 
\[ f_1 \leq f_2 \leq \cdots \leq f_n. \]

Adding hypothetical activity \( a_0 \) (with \( s_0 = f_0 = s_1 \)) and activity \( a_{n+1} \) (with \( s_{n+1} = f_{n+1} = f_n \)).

Algorithm \textbf{Activity-Selection}\((s, f, k, n)\)

1. \( m = k + 1 \)
2. \textbf{while} \( m \leq n \) and \( s_m < f_k \)
3. \( m = m + 1 \)
4. \textbf{if} \( m \leq n \)
Assume that the activities are sorted according to their finish times. 
\( f_1 \leq f_2 \leq \cdots \leq f_n \).

Adding hypothetical activity \( a_0 \) (with \( s_0 = f_0 = s_1 \)) and activity \( a_{n+1} \) (with \( s_{n+1} = f_{n+1} = f_n \)).

Algorithm \textsc{Activity-Selection}(s, f, k, n)
1. \( m = k + 1 \)
2. \textbf{while} \( m \leq n \) and \( s_m < f_k \)
3. \( m = m + 1 \)
4. \textbf{if} \( m \leq n \)
5. \textbf{return} \( \{a_m\} \cup \text{Activity-Selection}(s, f, m, n) \)
Chapter 16. Greedy Algorithms

Assume that the activities are sorted according to their finish times.
\[ f_1 \leq f_2 \leq \cdots \leq f_n. \]

Adding hypothetical activity \( a_0 \) (with \( s_0 = f_0 = s_1 \)) and activity \( a_{n+1} \) (with \( s_{n+1} = f_{n+1} = f_n \)).

Algorithm \texttt{Activity-Selection}(\( s, f, k, n \))

1. \( m = k + 1 \)
2. \textbf{while} \( m \leq n \) and \( s_m < f_k \)
3. \( m = m + 1 \)
4. \textbf{if} \( m \leq n \)
5. \textbf{return} \( \{a_m\} \cup \texttt{Activity-Selection}(s, f, m, n) \)
6. \textbf{else return} \( (\emptyset) \)
Chapter 16. Greedy Algorithms

Assume that the activities are sorted according to their finish times.
\[ f_1 \leq f_2 \leq \cdots \leq f_n. \]

Adding hypothetical activity \( a_0 \) (with \( s_0 = f_0 = s_1 \)) and activity \( a_{n+1} \) (with \( s_{n+1} = f_{n+1} = f_n \)).

Algorithm \textsc{Activity-Selection}(s, f, k, n)
1. \( m = k + 1 \)
2. \textbf{while} \( m \leq n \) and \( s_m < f_k \)
3. \( m = m + 1 \)
4. \textbf{if} \( m \leq n \)
5. \textbf{return} \( \{a_m\} \cup \textsc{Activity-Selection}(s, f, m, n) \)
6. \textbf{else return} (\( \emptyset \))

First called with \textsc{Activity-Selection}(s, f_0, n)
Chapter 16. Greedy Algorithms

Assume that the activities are sorted according to their finish times.
\[ f_1 \leq f_2 \leq \cdots \leq f_n. \]
Adding hypothetical activity \( a_0 \) (with \( s_0 = f_0 = s_1 \)) and activity \( a_{n+1} \) (with \( s_{n+1} = f_{n+1} = f_n \)).

Algorithm \( \text{Activity-Selection}(s, f, k, n) \)
1. \( m = k + 1 \)
2. \textbf{while} \( m \leq n \) and \( s_m < f_k \)
3. \hspace{1em} \( m = m + 1 \)
4. \textbf{if} \( m \leq n \)
5. \hspace{1em} \textbf{return} \( \{a_m\} \cup \text{Activity-Selection}(s, f, m, n) \)
6. \hspace{1em} \textbf{else return} \( (\emptyset) \)

First called with \( \text{Activity-Selection}(s, f_0, n) \)

Time complexity: \( O(n) \).
Chapter 16. Greedy Algorithms

Fractional Knapsack Problem:

- **Input**: \( n \) items of sizes \( s_1, ..., s_n \) and values \( v_1, ..., v_n \); and a knapsack size \( B \).
- **Output**: a subset \( A \subseteq \{1, 2, ..., n\} \), each with a fraction \( 0 < f_i \leq 1 \), which maximizes \( \sum_{i \in A} f_i v_i \) under the constraint \( \sum_{i \in A} f_i s_i \leq B \).

Define:
\[
K(S, X) \text{ be the maximum value of packing items from set } S \text{ into space } X.
\]

Then
\[
K(S, X) = \max_{0 < f_i \leq 1, i \in S} \left\{ f_i v_i + K(S \setminus \{i\}, X - f_i s_i) \right\}
\]

- We hope to select some item \( i \) that makes other subproblems disappear.
- We select item \( i \) such that it has the highest density \( d_i = \frac{v_i}{s_i} \).
- For convenience, assume the items are sorted according to the non-decreasing order of density.

**Theorem**: Fractional Knapsack problem has the greedy-choice property.
Chapter 16. Greedy Algorithms

Fractional Knapsack Problem:

**Input:** $n$ items of sizes $s_1, \ldots, s_n$ and values $v_1, \ldots, v_n$; and a knapsack size $B$

**Output:** a subset $A \subseteq \{1, 2, \ldots, n\}$, each with a fraction $0 < f_i \leq 1$, which maximizes
$$\sum_{i \in A} f_i v_i$$
under the constraint
$$\sum_{i \in A} f_i s_i \leq B$$

Define:
$$K(S, X)$$
be the maximum value of packing items from set $S$ into space $X$.

Then
$$K(S, X) = \max_{0 < f_i \leq 1, i \in S} \{ f_i v_i + K(S \setminus \{i\}, X - f_i s_i) \}$$

We hope to select some item $i$ that makes other subproblems disappear.

We select item $i$ such that it has the highest density $d_i = \frac{v_i}{s_i}$.

For convenience, assume the items are sorted according to the non-decreasing order of density.

**Theorem:** Fractional Knapsack problem has the greedy-choice property.
Fractional Knapsack Problem:

**INPUT:** $n$ items of sizes $s_1, \ldots, s_n$ and values $v_1, \ldots, v_n$; and a knapsack size $B$

**OUTPUT:** a subset $A \subseteq \{1, 2, \ldots, n\}$, each with a fraction $0 < f_i \leq 1$, where $f_i$ is the fraction of the $i$th item included in the subset.
Chapter 16. Greedy Algorithms

Fractional Knapsack Problem:

**INPUT:** $n$ items of sizes $s_1, \ldots, s_n$ and values $v_1, \ldots, v_n$; and a knapsack size $B$

**OUTPUT:** a subset $A \subseteq \{1, 2, \ldots, n\}$, each with a fraction $0 < f_i \leq 1$, which maximizes $\sum_{i \in A} f_i v_i$ under the constraint $\sum_{i \in A} f_i s_i \leq B$
Chapter 16. Greedy Algorithms

Fractional Knapsack Problem:

**INPUT:** $n$ items of sizes $s_1, \ldots, s_n$ and values $v_1, \ldots, v_n$; and a knapsack size $B$

**OUTPUT:** a subset $A \subseteq \{1, 2, \ldots, n\}$, each with a fraction $0 < f_i \leq 1$, which maximizes $\sum_{i \in A} f_i v_i$ under the constraint $\sum_{i \in A} f_i s_i \leq B$

Define: $K(S, X)$ be the maximum value of packing items from set $S$ into space $X$. 
Fractional Knapsack Problem:

**Input:** \( n \) items of sizes \( s_1, \ldots, s_n \) and values \( v_1, \ldots, v_n \); and a knapsack size \( B \)

**Output:** a subset \( A \subseteq \{1, 2, \ldots, n\} \), each with a fraction \( 0 < f_i \leq 1 \), which maximizes \( \sum_{i \in A} f_i v_i \) under the constraint \( \sum_{i \in A} f_i s_i \leq B \)

Define: \( K(S, X) \) be the maximum value of packing items from set \( S \) into space \( X \). Then

\[
K(S, X) = \max_{0 < f_i \leq 1, i \in S} \sum_{i \in A} f_i v_i
\]
Chapter 16. Greedy Algorithms

Fractional Knapsack Problem:

**Input:** \( n \) items of sizes \( s_1, \ldots, s_n \) and values \( v_1, \ldots, v_n; \)
and a knapsack size \( B \)

**Output:** a subset \( A \subseteq \{1, 2, \ldots, n\} \), each with a fraction \( 0 < f_i \leq 1 \),
which maximizes \( \sum_{i \in A} f_i v_i \) under the constraint \( \sum_{i \in A} f_i s_i \leq B \)

Define: \( K(S, X) \) be the maximum value of packing items from set \( S \) into space \( X \).
Then

\[
K(S, X) = \max_{0 < f_i \leq 1, i \in S} \{ f_i v_i + K(S - \{i\}, X - f_i s_i) \}
\]
Chapter 16. Greedy Algorithms

Fractional Knapsack Problem:

**INPUT:** \( n \) items of sizes \( s_1, \ldots, s_n \) and values \( v_1, \ldots, v_n \); and a knapsack size \( B \)

**OUTPUT:** a subset \( A \subseteq \{1, 2, \ldots, n\} \), each with a fraction \( 0 < f_i \leq 1 \), which maximizes \( \sum_{i \in A} f_i v_i \) under the constraint \( \sum_{i \in A} f_i s_i \leq B \)

Define: \( K(S, X) \) be the maximum value of packing items from set \( S \) into space \( X \). Then

\[
K(S, X) = \max_{0 < f_i \leq 1, i \in S} \left\{ f_i v_i + K(S - \{i\}, X - f_i s_i) \right\}
\]

- We hope to select some item \( i \) that makes other subproblems disappear.
Chapter 16. Greedy Algorithms

Fractional Knapsack Problem:

**INPUT:** \( n \) items of sizes \( s_1, \ldots, s_n \) and values \( v_1, \ldots, v_n \); and a knapsack size \( B \)

**OUTPUT:** a subset \( A \subseteq \{1, 2, \ldots, n\} \), each with a fraction \( 0 < f_i \leq 1 \), which maximizes \( \sum_{i \in A} f_i v_i \) under the constraint \( \sum_{i \in A} f_i s_i \leq B \)

Define: \( K(S, X) \) be the maximum value of packing items from set \( S \) into space \( X \). Then

\[
K(S, X) = \max_{0 < f_i \leq 1, i \in S} \{ f_i v_i + K(S - \{i\}, X - f_i s_i) \}
\]

- We hope to select some item \( i \) that makes other subproblems disappear.
- We select item \( i \) such that it has the **highest density** \( d_i = \frac{v_i}{s_i} \).
Chapter 16. Greedy Algorithms

Fractional Knapsack Problem:

**INPUT:** $n$ items of sizes $s_1, \ldots, s_n$ and values $v_1, \ldots, v_n$;
and a knapsack size $B$

**OUTPUT:** a subset $A \subseteq \{1, 2, \ldots, n\}$, each with a fraction $0 < f_i \leq 1$,
which maximizes $\sum_{i \in A} f_i v_i$ under the constraint $\sum_{i \in A} f_i s_i \leq B$

Define: $K(S, X)$ be the maximum value of packing items from set $S$ into space $X$. Then

$$K(S, X) = \max_{0 < f_i \leq 1, i \in S} \{ f_i v_i + K(S - \{i\}, X - f_i s_i) \}$$

- We hope to select some item $i$ that makes other subproblems disappear.
- We select item $i$ such that it has the **highest density** $d_i = \frac{v_i}{s_i}$.
- For convenience, assume the items are sorted according to the **non-decreasing order** of density.
Chapter 16. Greedy Algorithms

Fractional Knapsack Problem:

**Input:** $n$ items of sizes $s_1, \ldots, s_n$ and values $v_1, \ldots, v_n$; and a knapsack size $B$

**Output:** a subset $A \subseteq \{1, 2, \ldots, n\}$, each with a fraction $0 < f_i \leq 1$, which maximizes $\sum_{i \in A} f_i v_i$ under the constraint $\sum_{i \in A} f_i s_i \leq B$

Define: $K(S, X)$ be the maximum value of packing items from set $S$ into space $X$. Then

$$K(S, X) = \max_{0 < f_i \leq 1, i \in S} \{f_i v_i + K(S - \{i\}, X - f_i s_i)\}$$

- We hope to select some item $i$ that makes other subproblems disappear.
- We select item $i$ such that it has the **highest density** $d_i = \frac{v_i}{s_i}$.
- For convenience, assume the items are sorted according to the non-decreasing order of density.

**Theorem:** Fractional Knapsack problem has the greedy-choice property.
Chapter 16. Greedy Algorithms

**Theorem**: Fractional Knapsack problem has the greedy-choice property.
Chapter 16. Greedy Algorithms

**Theorem**: Fractional Knapsack problem has the greedy-choice property.

We need to prove the following:
Theorem: Fractional Knapsack problem has the greedy-choice property.

We need to prove the following:

Given a subset of items $S \subseteq \{1, 2, \ldots, n\}$ and space $X$. Assume item $i$ has the highest density in $S$. Then there is an optimal solution for $S$ which contains the fraction $f_i$ of the item $i$, for some maximum $f_i$, $0 \leq f_i \leq 1$ such that $f_is_i \leq X$. (This is left as an exercise)
Chapter 16. Greedy Algorithms

**Theorem:** Fractional Knapsack problem has the greedy-choice property.

We need to prove the following:

Given a subset of items $S \subseteq \{1, 2, \ldots, n\}$ and space $X$. Assume item $i$ has the highest density in $S$. Then there is an optimal solution for $S$ which contains the fraction $f_i$ of the item $i$, for some maximum $f_i$, $0 \leq f_i \leq 1$ such that $f_is_i \leq X$.

(This is left as an exercise)
Quiz # 3

Solve the following problem of **Coin Change** problem with **DP**:

**Input:** $X$ cents of money and coin denominations $\{d_1, d_2, d_3\}$ where $1 \leq d_1 < d_2 < d_3$;

**Output:** the minimum number of coins with total amount $= X$. 

1. What does your recursive solution look like?
2. What should the objective function be?
3. What is the recurrence for the objective function?
4. Does the problem have a greedy-choice property?
Solve the following problem of Coin Change problem with DP:

**Input**: \( X \) cents of money and coin denominations \( \{d_1, d_2, d_3\} \) where \( 1 \leq d_1 < d_2 < d_3 \);

**Output**: the minimum number of coins with total amount = \( X \).
Solve the following problem of Coin Change problem with DP:

**Input:** $X$ cents of money and coin denominations \( \{d_1, d_2, d_3\} \) where \( 1 \leq d_1 < d_2 < d_3 \);

**Output:** the minimum number of coins with total amount $= X$.

e.g., $X = 12$, $d_1 = 1$, $d_2 = 3$, $d_3 = 5$, then an optimal solution is: $5, 5, 1, 1$

$5, 3, 3, 1$ is also an optimal solution.
Solve the following problem of Coin Change problem with DP:

**Input:** $X$ cents of money and coin denominations \{d_1, d_2, d_3\} where $1 \leq d_1 < d_2 < d_3$;

**Output:** the minimum number of coins with total amount $= X$.

e.g., $X = 12$, $d_1 = 1$, $d_2 = 3$, $d_3 = 5$,
then an optimal solution is: 5, 5, 1, 1
5, 3, 3, 1 is also an optimal solution.

1. What does your recursive solution look like?
2. What should the objective function be?
3. What is the recurrence for the objective function?
4. Does the problem have a greedy-choice property?
Review for Midterm I

- Big-$O$, Big-$\Omega$, lower bound, upper bound
Review for Midterm I

- Big-$O$, Big-$\Omega$, lower bound, upper bound
- recurrences for recursive algorithms’ complexity, methods for solving recurrences
Review for Midterm I

- Big-$O$, Big-$\Omega$, lower bound, upper bound
- recurrences for recursive algorithms’ complexity, methods for solving recurrences
- decision tree based lower bound proof technique
Review for Midterm I

- Big-$O$, Big-$\Omega$, lower bound, upper bound
- recurrences for recursive algorithms’ complexity, methods for solving recurrences
- decision tree based lower bound proof technique
- average time complexity and randomized algorithms
Review for Midterm I

- Big-$O$, Big-$\Omega$, lower bound, upper bound
- recurrences for recursive algorithms’ complexity, methods for solving recurrences
- decision tree based lower bound proof technique
- average time complexity and randomized algorithms
- dynamic programming, 4 steps to solve problems with DP
Review for Midterm I

- Big-$O$, Big-$\Omega$, lower bound, upper bound
- recurrences for recursive algorithms’ complexity, methods for solving recurrences
- decision tree based lower bound proof technique
- average time complexity and randomized algorithms
- dynamic programming, 4 steps to solve problems with DP
- greedy algorithms, greedy-choice property proof
Chapter 16. Greedy Algorithms

Huffman Code
Chapter 16. Greedy Algorithms

Huffman Code

compressing data using binary bits
Chapter 16. Greedy Algorithms

Huffman Code

compressing data using binary bits
code: a compressing scheme,
Chapter 16. Greedy Algorithms

Huffman Code

compressing data using binary bits
code: a compressing scheme, fixed length vs variable length code
Huffman Code

compressing data using binary bits
code: a compressing scheme, fixed length vs variable length code

<table>
<thead>
<tr>
<th>character</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>frequency</td>
<td>.45</td>
<td>.13</td>
<td>.12</td>
<td>.16</td>
<td>.09</td>
<td>.05</td>
</tr>
<tr>
<td>fixed length code</td>
<td>000 001 010 011 100 101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>variable length code</td>
<td>0 101 100 111 1101 1100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Chapter 16. Greedy Algorithms

Huffman Code

compressing data using binary bits
code: a compressing scheme, fixed length vs variable length code

<table>
<thead>
<tr>
<th>character</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>frequency</td>
<td>.45</td>
<td>.13</td>
<td>.12</td>
<td>.16</td>
<td>.09</td>
<td>.05</td>
</tr>
<tr>
<td>fixed length code</td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
</tr>
<tr>
<td>variable length code</td>
<td>0</td>
<td>101</td>
<td>100</td>
<td>111</td>
<td>1101</td>
<td>1100</td>
</tr>
</tbody>
</table>

(a) and (b) show the Huffman coding trees for the given frequencies.
prefix code: a prefix of a codeword cannot be another codeword.
Chapter 16. Greedy Algorithms

**prefix code**: a prefix of a codeword cannot be another codeword.

The Optimal Prefix Code Problem:
prefix code: a prefix of a codeword cannot be another codeword.

The Optimal Prefix Code Problem:

**Input:** character set $C$ and frequencies $f$;
prefix code: a prefix of a codeword cannot be another codeword.

The Optimal Prefix Code Problem:

**Input:** character set $C$ and frequencies $f$;
**Output:** a code tree $T$ such that the cost
prefix code: a prefix of a codeword cannot be another codeword.

The Optimal Prefix Code Problem:

**Input:** character set $C$ and frequencies $f$;

**Output:** a code tree $T$ such that the cost

$$B(T) = \sum_{c \in C} f(c)d_T(c)$$

achieves the minimum

where $d_T(c)$ is the depth of character $c$ in tree $T$. 
prefix code: a prefix of a codeword cannot be another codeword.

The Optimal Prefix Code Problem:

**Input:** character set $C$ and frequencies $f$;
**Output:** a code tree $T$ such that the cost

$$B(T) = \sum_{c \in C} f(c)d_T(c)$$

achieves the minimum

where $d_T(c)$ is the depth of character $c$ in tree $T$.

Note that
- $d_T(c)$ is the length of codeword for $c$ under the code scheme $T$. 
prefix code: a prefix of a codeword cannot be another codeword.

The Optimal Prefix Code Problem:

**Input:** character set \( C \) and frequencies \( f \);

**Output:** a code tree \( T \) such that the cost

\[
B(T) = \sum_{c \in C} f(c)d_T(c)
\]

achieves the minimum where \( d_T(c) \) is the depth of character \( c \) in tree \( T \).

Note that

- \( d_T(c) \) is the length of codeword for \( c \) under the code scheme \( T \).
- \( B(T) \times n \) is the number of bits required to code a file (of \( n \) characters).
Algorithm \textsc{Huffman}(C, f)
Algorithm \texttt{HUFFMAN}(C, f)

1. $n = |C|$
Algorithm $\text{HUFFMAN}(C, f)$

1. $n = |C|$
2. $Q = C$

3. for $i = 1$ to $n - 1$
4.   \begin{align*}
5. & \text{newnode} (z) \\
6. & x = \text{Extract-Min} (Q) \quad \text{extract two least frequent characters} \\
7. & z.\text{leftchild} = x \\
8. & y = \text{Extract-Min} (Q) \\
9. & z.\text{rightchild} = y \\
10. & f(z) = f(x) + f(y)
11. & \text{Insert} (Q, z)
12. & \text{return} (\text{Extract-Min} (Q))
\end{align*}
Chapter 16. Greedy Algorithms

Algorithm \textsc{Huffman}(C, f)

1. \[ n = |C| \]
2. \[ Q = C \]
3. \textbf{for} \[ i = 1 \textbf{to} n - 1 \]
Algorithm $\text{HUFFMAN}(C, f)$

1. $n = |C|$
2. $Q = C$
3. for $i = 1$ to $n - 1$
4. newnode($z$)
Chapter 16. Greedy Algorithms

Algorithm $\text{HUFFMAN}(C, f)$

1. $n = |C|$
2. $Q = C$
3. for $i = 1$ to $n - 1$
4. \hspace{1em} $\text{newnode}(z)$
5. \hspace{1em} $x = \text{EXTRACT-MIN}(Q)$ \hspace{1em} extract two least frequent characters
Algorithm \texttt{HUFFMAN}(C, f)

1. \( n = |C| \)
2. \( Q = C \)
3. \textbf{for} \( i = 1 \) \textbf{to} \( n - 1 \)
4. \texttt{newnode}(z)
5. \( x = \texttt{EXTRACT-MIN}(Q) \) \hspace{1cm} \text{extract two least frequent characters}
6. \( z\.leftchild = x \)
Algorithm **HUFFMAN**(C, f)

1. \( n = |C| \)
2. \( Q = C \)
3. for \( i = 1 \) to \( n - 1 \)
4. \( \text{newnode}(z) \)
5. \( x = \text{EXTRACT-MIN}(Q) \)  
   extract two least frequent characters
6. \( z.leftchild = x \)
7. \( y = \text{EXTRACT-MIN}(Q) \)
Chapter 16. Greedy Algorithms

Algorithm HUFFMAN($C, f$)

1. $n = |C|$
2. $Q = C$
3. for $i = 1$ to $n - 1$
4. newnode($z$)
5. $x = \text{EXTRACT-MIN}(Q)$ extract two least frequent characters
6. $z.leftchild = x$
7. $y = \text{EXTRACT-MIN}(Q)$
8. $z.rightchild = y$
9. $f(z) = f(x) + f(y)$

return $\text{EXTRACT-MIN}(Q)$
Algorithm **HUFFMAN**\((C, f)\)

1. \(n = |C|\)
2. \(Q = C\)
3. **for** \(i = 1\) **to** \(n - 1\)
4. **newnode**\((z)\)
5. \(x = \text{EXTRACT-MIN}(Q)\) \hspace{1cm} \text{extract two least frequent characters}
6. \(z.leftchild = x\)
7. \(y = \text{EXTRACT-MIN}(Q)\)
8. \(z.rightchild = y\)
9. \(f(z) = f(x) + f(y)\)
Algorithm \textsc{Huffman}(C, f)

1. \( n = |C| \)
2. \( Q = C \)
3. \textbf{for} \( i = 1 \) \textbf{to} \( n - 1 \)
4. \textbf{newnode}(z)
5. \( x = \textsc{Extract-Min}(Q) \) \hspace{1cm} \text{extract two least frequent characters}
6. \( z.\text{leftchild} = x \)
7. \( y = \textsc{Extract-Min}(Q) \)
8. \( z.\text{rightchild} = y \)
9. \( f(z) = f(x) + f(y) \)
9. \textsc{Insert}(Q, z)
Chapter 16. Greedy Algorithms

Algorithm HUFFMAN($C, f$)

1. $n = |C|$
2. $Q = C$
3. \textbf{for} $i = 1 \text{ to } n - 1$
4. \hspace{1em} \textbf{newnode}(z)
5. $x = \text{EXTRACT-MIN}(Q)$ \hspace{2em} extract two least frequent characters
6. $z.leftchild = x$
7. $y = \text{EXTRACT-MIN}(Q)$
8. $z.rightchild = y$
9. $f(z) = f(x) + f(y)$
9. \hspace{1em} \text{INSERT}(Q, z)$
9. \hspace{1em} \textbf{return} (\text{EXTRACT-MIN}(Q))
Chapter 16. Greedy Algorithms

An example of Huffman algorithm

- Symbols: A, B, C, D, E, F, G
  freq.: 2, 3, 5, 8, 13, 15, 18

- Huffman codes:
  A: 10100  B: 10101  C: 1011
  D: 100    E: 00    F: 01
  G: 11

A Huffman code Tree
Chapter 16. Greedy Algorithms

Correctness of Huffman’s algorithm
Chapter 16. Greedy Algorithms

Correctness of Huffman’s algorithm

Lemma 16.2 (Greedy choice property)
Correctness of Huffman’s algorithm

**Lemma 16.2** (Greedy choice property)

Let $C$ be an alphabet and $f$ be the frequency function for characters in $C$. Let $x$ and $y$ be two characters in $C$ having the lowest frequencies. Then there exists an optimal prefix code for $C$ in which the codewords for $x$ and $y$ have the same length and differ in the last bit.

Proof: Assume $C$, $f$, $x$, and $y$ are as described in the lemma condition. Let $T$ be any optimal prefix code for $C$.

(a) If $T$ contains the situation about $x$ and $y$ stated in the lemma, done for the proof!

(b) Let $a, b \in C$ be two characters such that their codewords have the same length and differ in the last bit. Also assume that their codeword length $d_T(a) = d_T(b) \geq d_T(c)$, for every $c \in C$. We note that such $a$ and $b$ can be found from $T$ without difficulty.
Correctness of Huffman’s algorithm

Lemma 16.2 (Greedy choice property)

Let $C$ be an alphabet and $f$ be the frequency function for characters in $C$. Let $x$ and $y$ be two characters in $C$ having the lowest frequencies. Then there exists an optimal prefix code for $C$ in which the codewords for $x$ and $y$ have the same length and differ in the last bit.

Proof:
Assume $C$, $f$, $x$ and $y$ are as described in the lemma condition.
Chapter 16. Greedy Algorithms

Correctness of Huffman’s algorithm

Lemma 16.2 (Greedy choice property)

Let $C$ be an alphabet and $f$ be the frequency function for characters in $C$. Let $x$ and $y$ be two characters in $C$ having the lowest frequencies. Then there exists an optimal prefix code for $C$ in which the codewords for $x$ and $y$ have the same length and differ in the last bit.

Proof:
Assume $C$, $f$, $x$ and $y$ are as described in the lemma condition. Let $T$ be any optimal prefix code for $C$. 

Chapter 16. Greedy Algorithms

Correctness of Huffman’s algorithm

Lemma 16.2 (Greedy choice property)

Let $C$ be an alphabet and $f$ be the frequency function for characters in $C$. Let $x$ and $y$ be two characters in $C$ having the lowest frequencies. Then there exists an optimal prefix code for $C$ in which the codewords for $x$ and $y$ have the same length and differ in the last bit.

Proof:
Assume $C$, $f$, $x$ and $y$ are as described in the lemma condition. Let $T$ be any optimal prefix code for $C$.

(a) if $T$ contains the situation about $x$ and $y$ stated in the lemma. Done for the proof!
Chapter 16. Greedy Algorithms

Correctness of Huffman’s algorithm

Lemma 16.2 (Greedy choice property)

Let $C$ be an alphabet and $f$ be the frequency function for characters in $C$. Let $x$ and $y$ be two characters in $C$ having the lowest frequencies. Then there exists an optimal prefix code for $C$ in which the codewords for $x$ and $y$ have the same length and differ in the last bit.

Proof:
Assume $C$, $f$, $x$ and $y$ are as described in the lemma condition. Let $T$ be any optimal prefix code for $C$.

(a) if $T$ contains the situation about $x$ and $y$ stated in the lemma. Done for the proof!

(b) let $a, b \in C$ be two characters such that their codewords have the same length and differ in the last bit.
Chapter 16. Greedy Algorithms

Correctness of Huffman’s algorithm

**Lemma 16.2** (Greedy choice property)

Let $C$ be an alphabet and $f$ be the frequency function for characters in $C$. Let $x$ and $y$ be two characters in $C$ having the lowest frequencies. Then there exists an optimal prefix code for $C$ in which the codewords for $x$ and $y$ have the same length and differ in the last bit.

**Proof:**
Assume $C$, $f$, $x$ and $y$ are as described in the lemma condition. Let $T$ be any optimal prefix code for $C$.

(a) if $T$ contains the situation about $x$ and $y$ stated in the lemma. Done for the proof!

(b) let $a, b \in C$ be two characters such that their codewords have the same length and differ in the last bit. Also assume that their codeword length

$$d_T(a) = d_T(b) \geq d_T(c), \text{ for every } c \in C$$

We note that such $a$ and $b$ can be found from $T$ without difficulty.
We use the following swapping strategy to obtain another prefix code $T'$ for $C$. 

\[ B(T) - B(T') = \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c) = f(a) d_T(a) + f(b) d_T(b) + f(x) d_T(x) + f(y) d_T(y) - f(a) d_{T'}(a) - f(b) d_{T'}(b) - f(x) d_{T'}(x) - f(y) d_{T'}(y) = (f(a) - f(x)) d_T(a) + (f(x) - f(a)) d_T(x) + (f(b) - f(y)) d_T(b) + (f(y) - f(b)) d_T(y) \geq 0 \]

which implies $B(T) \geq B(T')$. So $T'$ is also an optimal code for $C$. Therefore, $T'$ satisfies the lemma.
We use the following swapping strategy to obtain another prefix code $T'$ for $C$. Without loss of generality, assume $\{a, b\} \cap \{x, y\} = \phi$. The created code $T'$ is the same as $T$ except the positions $a$ and $x$ are swapped in $T'$; so are the positions of $b$ and $y$ swapped in $T'$. Therefore, 

$$B(T) - B(T') = \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c) = f(a) d_T(a) + f(b) d_T(b) + f(x) d_T(x) + f(y) d_T(y) - f(a) d_{T'}(a) - f(b) d_{T'}(b) - f(x) d_{T'}(x) - f(y) d_{T'}(y) = (f(a) - f(x))(d_T(a) - d_T(x)) + (f(x) - f(a))(d_T(x) - d_T(y)) \geq 0$$

which implies $B(T) \geq B(T')$. So $T'$ is also an optimal code for $C$. So $T'$ satisfies the lemma.
We use the following swapping strategy to obtain another prefix code $T'$ for $C$.

Without loss of generality, assume $\{a, b\} \cap \{x, y\} = \emptyset$.

The created code $T'$ is the same as $T$ except the positions $a$ and $x$ are swapped in $T'$; so are the positions of $b$ and $y$ swapped in $T'$.
We use the following swapping strategy to obtain another prefix code $T'$ for $C$.

Without loss of generality, assume $\{a, b\} \cap \{x, y\} = \emptyset$.

The created code $T'$ is the same as $T$ except the positions $a$ and $x$ are swapped in $T'$; so are the positions of $b$ and $y$ swapped in $T'$.

$$B(T) - B(T') = \sum_{c \in C} f(c)d_T(c) - \sum_{c \in C} f(c)d_{T'}(c)$$
We use the following swapping strategy to obtain another prefix code $T'$ for $C$.

Without loss of generality, assume $\{a, b\} \cap \{x, y\} = \emptyset$.

The created code $T'$ is the same as $T$ except the positions $a$ and $x$ are swapped in $T'$; so are the positions of $b$ and $y$ swapped in $T'$.

\[
B(T) - B(T') = \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c)
= f(a) d_T(a) + f(b) d_T(b) + f(x) d_T(x) + f(y) d_T(y)
\]
Chapter 16. Greedy Algorithms

We use the following swapping strategy to obtain another prefix code $T'$ for $C$.

Without loss of generality, assume $\{a, b\} \cap \{x, y\} = \phi$.

The created code $T'$ is the same as $T$ except the positions $a$ and $x$ are swapped in $T'$; so are the positions of $b$ and $y$ swapped in $T'$.

$$B(T) - B(T') = \sum_{c \in C} f(c)d_T(c) - \sum_{c \in C} f(c)d_{T'}(c)$$

$$= f(a)d_T(a) + f(b)d_T(b) + f(x)d_T(x) + f(y)d_T(y)$$

$$- f(a)d_{T'}(a) - f(b)d_{T'}(b) - f(x)d_{T'}(x) - f(y)d_{T'}(y)$$

which implies $B(T) \geq B(T')$. So $T'$ is also an optimal code for $C$.

So $T'$ satisfies the lemma.
Chapter 16. Greedy Algorithms

We use the following swapping strategy to obtain another prefix code $T'$ for $C$.

Without loss of generality, assume $\{a, b\} \cap \{x, y\} = \phi$.

The created code $T'$ is the same as $T$ except the positions $a$ and $x$ are swapped in $T'$; so are the positions of $b$ and $y$ swapped in $T'$.

$$B(T) - B(T') = \sum_{c \in C} f(c)d_T(c) - \sum_{c \in C} f(c)d_{T'}(c)$$

$$= f(a)d_T(a) + f(b)d_T(b) + f(x)d_T(x) + f(y)d_T(y)$$

$$- f(a)d_{T'}(a) - f(b)d_{T'}(b) - f(x)d_{T'}(x) - f(y)d_{T'}(y)$$

$$= (f(a) - f(x))d_T(a) + (f(x) - f(a))d_T(x) + (f(b) - f(y))d_T(b)$$

$$+(f(y) - f(b))d_T(y)$$

which implies $B(T) \geq B(T')$. So $T'$ is also an optimal code for $C$.

So $T'$ satisfies the lemma.
We use the following swapping strategy to obtain another prefix code $T'$ for $C$.

Without loss of generality, assume $\{a, b\} \cap \{x, y\} = \emptyset$.

The created code $T'$ is the same as $T$ except the positions $a$ and $x$ are swapped in $T'$; so are the positions of $b$ and $y$ swapped in $T'$.

$$B(T) - B(T') = \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c)$$

$$= f(a) d_T(a) + f(b) d_T(b) + f(x) d_T(x) + f(y) d_T(y)$$

$$- f(a) d_{T'}(a) - f(b) d_{T'}(b) - f(x) d_{T'}(x) - f(y) d_{T'}(y)$$

$$= (f(a) - f(x)) d_T(a) + (f(x) - f(a)) d_T(x) + (f(b) - f(y)) d_T(b)$$

$$+ (f(y) - f(b)) d_T(y)$$

$$= (f(a) - f(x))(d_T(a) - d_T(x)) + (f(b) - f(y))(d_T(b) - d_T(y)) \geq 0$$

which implies $B(T) \geq B(T')$. So $T'$ is also an optimal code for $C$.

So $T'$ satisfies the lemma.
Chapter 16. Greedy Algorithms

We use the following swapping strategy to obtain another prefix code $T'$ for $C$.

Without loss of generality, assume $\{a, b\} \cap \{x, y\} = \emptyset$.

The created code $T'$ is the same as $T$ except the positions $a$ and $x$ are swapped in $T'$; so are the positions of $b$ and $y$ swapped in $T'$.

$$B(T) - B(T') = \sum_{c \in C} f(c)d_T(c) - \sum_{c \in C} f(c)d_{T'}(c)$$

$$= f(a)d_T(a) + f(b)d_T(b) + f(x)d_T(x) + f(y)d_T(y)$$

$$- f(a)d_{T'}(a) - f(b)d_{T'}(b) - f(x)d_{T'}(x) - f(y)d_{T'}(y)$$

$$= (f(a) - f(x))d_T(a) + (f(x) - f(a))d_T(x) + (f(b) - f(y))d_T(b)$$

$$+ (f(y) - f(b))d_T(y)$$

$$= (f(a) - f(x))(d_T(a) - d_T(x)) + (f(b) - f(y))(d_T(b) - d_T(y)) \geq 0$$

which implies $B(T) \geq B(T')$. So $T'$ is also an optimal code for $C$. 

We use the following swapping strategy to obtain another prefix code $T'$ for $C$.

Without loss of generality, assume $\{a, b\} \cap \{x, y\} = \emptyset$.

The created code $T'$ is the same as $T$ except the positions $a$ and $x$ are swapped in $T'$; so are the positions of $b$ and $y$ swapped in $T'$.

$$B(T) - B(T') = \sum_{c \in C} f(c)d_T(c) - \sum_{c \in C} f(c)d_{T'}(c)$$

$$= f(a)d_T(a) + f(b)d_T(b) + f(x)d_T(x) + f(y)d_T(y) - f(a)d_{T'}(a) - f(b)d_{T'}(b) - f(x)d_{T'}(x) - f(y)d_{T'}(y)$$

$$= (f(a) - f(x))d_T(a) + (f(x) - f(a))d_T(x) + (f(b) - f(y))d_T(b) + (f(y) - f(b))d_T(y)$$

$$= (f(a) - f(x))(d_T(a) - d_T(x)) + (f(b) - f(y))(d_T(b) - d_T(y)) \geq 0$$

which implies $B(T) \geq B(T')$. So $T'$ is also an optimal code for $C$.

So $T'$ satisfies the lemma.
Chapter 16. Greedy Algorithms

Lemma 16.3 (Optimal substructure)
Lemma 16.3 (Optimal substructure)
Let $C$ be an alphabet and $f$ be the frequency function for characters in $C$. Let $x$ and $y$ be two characters in $C$ having the lowest frequencies.

Statement:
Let $C'$ be an alphabet and $f$ be the frequency function for characters in $C$. Let $x$ and $y$ be two characters in $C$ having the lowest frequencies.

Let $C' = C - \{x, y\} \cup \{z\}$ and the frequency function for $C'$ is the same as $f$ except that $f(z) = f(x) + f(y)$.

Let $T'$ be any optimal prefix code tree for $C'$. Then the tree $T$ obtained from $T'$ by replacing the leaf node $z$ with an internal node having $x$ and $y$ as children, is an optimal prefix code tree for $C$.

Proof:
The code $T$ for $C$ constructed from $T'$, a code for $C'$, has the following objective function value

$$B(T) = \sum_{c \in C} f(c) d_T(c) = \sum_{c \in C - \{x, y\}} f(c) d_T(c) + (f(x) + f(y)) d_T(x) + f(y) d_T(y)$$

$$= \sum_{c \in C - \{x, y\}} f(c) d_{T'}(c) + (f(x) + f(y)) (d_{T'}(z) + 1)$$

$$= \sum_{c \in C - \{x, y\} \cup \{z\}} f(c) d_{T'}(c) + (f(x) + f(y)) (d_{T'}(z) + 1)$$

$$= \sum_{c \in C - \{x, y\} \cup \{z\}} f(c) d_{T'}(c) + (f(x) + f(y)) (d_{T'}(z) + 1)$$

$$= \sum_{c \in C - \{x, y\} \cup \{z\}} f(c) d_{T'}(c) + (f(x) + f(y)) (d_{T'}(z) + 1)$$

$$= \sum_{c \in C'} f(c) d_{T'}(c) + (f(x) + f(y)) (d_{T'}(z) + 1)$$

$$= B(T') + (f(x) + f(y)) (d_{T'}(z) + 1)$$

The objective function value of $T'$ is $B(T')$ and the objective function value of $T$ is $B(T) + (f(x) + f(y)) (d_{T'}(z) + 1)$.

Since $T'$ is an optimal prefix code tree for $C'$, $B(T')$ is the minimum possible objective function value.

Therefore, $B(T) = B(T') + (f(x) + f(y)) (d_{T'}(z) + 1)$.

Since $d_{T'}(z) + 1$ is a non-negative integer, $B(T)$ is the minimum possible objective function value.

Hence, $T$ is an optimal prefix code tree for $C$. 


Lemma 16.3 (Optimal substructure)

Let $C$ be an alphabet and $f$ be the frequency function for characters in $C$. Let $x$ and $y$ be two characters in $C$ having the lowest frequencies. Let

$$C' = C - \{x, y\} \cup \{z\}$$

and the frequency function for $C'$ is the same as $f$ except that $f(z) = f(x) + f(y)$. 
Chapter 16. Greedy Algorithms

Lemma 16.3 (Optimal substructure)

Let $C$ be an alphabet and $f$ be the frequency function for characters in $C$. Let $x$ and $y$ be two characters in $C$ having the lowest frequencies. Let

$$C' = C - \{x, y\} \cup \{z\}$$

and the frequency function for $C'$ is the same as $f$ except that $f(z) = f(x) + f(y)$.

Let $T'$ be any an optimal prefix code tree for $C'$. Then the tree $T$ obtained from $T'$ by replacing the leaf node $z$ with an internal node having $x$ and $y$ as children, is an optimal prefix code tree for $C$. 

Proof:
The code $T$ for $C$ constructed from $T'$, a code for $C'$, has the following objective function value

$$B(T) = \sum_{c \in C} f(c) \cdot d_T(c) = \sum_{c \in C - \{x, y\}} f(c) \cdot d_{T'}(c) + [f(x) + f(y)] \cdot d_{T'}(z) + 1 = \sum_{c \in C - \{x, y\} \cup \{z\}} f(c) \cdot d_{T'}(c) + [f(x) + f(y)] (d_{T'}(z) + 1) = \sum_{c \in C - \{x, y\} \cup \{z\}} f(c) \cdot d_{T'}(c) + [f(x) + f(y)] (d_{T'}(z) + 1) = \sum_{c \in C'} f(c) \cdot d_{T'}(c) + [f(x) + f(y)] = B(T') + f(x) + f(y)$$

for an optimal prefix code tree $T$ for $C$. 


Chapter 16. Greedy Algorithms

Lemma 16.3 (Optimal substructure)

Let $C$ be an alphabet and $f$ be the frequency function for characters in $C$. Let $x$ and $y$ be two characters in $C$ having the lowest frequencies. Let

$$C' = C - \{x, y\} \cup \{z\}$$

and the frequency function for $C'$ is the same as $f$ except that $f(z) = f(x) + f(y)$.

Let $T'$ be any an optimal prefix code tree for $C'$. Then the tree $T$ obtained from $T'$ by replacing the leaf node $z$ with an internal node having $x$ and $y$ as children, is an optimal prefix code tree for $C$.

Proof:
The code $T$ for $C$ constructed from $T'$, a code for $C'$, has the following objective function value
Lemma 16.3 (Optimal substructure)

Let \( C \) be an alphabet and \( f \) be the frequency function for characters in \( C \). Let \( x \) and \( y \) be two characters in \( C \) having the lowest frequencies. Let

\[
C' = C - \{x, y\} \cup \{z\}
\]

and the frequency function for \( C' \) is the same as \( f \) except that \( f(z) = f(x) + f(y) \).

Let \( T' \) be any an optimal prefix code tree for \( C' \). Then the tree \( T \) obtained from \( T' \) by replacing the leaf node \( z \) with an internal node having \( x \) and \( y \) as children, is an optimal prefix code tree for \( C \).

Proof:
The code \( T \) for \( C \) constructed from \( T' \), a code for \( C' \), has the following objective function value

\[
B(T) = \sum_{c \in C} f(c)d_T(c) = \sum_{c \in C - \{x, y\}} f(c)d_T(c) + f(x)d_T(x) + f(y)d_T(y)
\]
Lemma 16.3 (Optimal substructure)

Let $C$ be an alphabet and $f$ be the frequency function for characters in $C$. Let $x$ and $y$ be two characters in $C$ having the lowest frequencies. Let

$$C' = C - \{x, y\} \cup \{z\}$$

and the frequency function for $C'$ is the same as $f$ except that $f(z) = f(x) + f(y)$.

Let $T'$ be any an optimal prefix code tree for $C'$. Then the tree $T$ obtained from $T'$ by replacing the leaf node $z$ with an internal node having $x$ and $y$ as children, is an optimal prefix code tree for $C$.

Proof:
The code $T$ for $C$ constructed from $T'$, a code for $C'$, has the following objective function value

$$B(T) = \sum_{c \in C} f(c)d_T(c) = \sum_{c \in C - \{x, y\}} f(c)d_T(c) + f(x)d_T(x) + f(y)d_T(y)$$

$$= \sum_{c \in C - \{x, y\}} f(c)d_T(c) + [f(x) + f(y)]d_T(x)$$
Chapter 16. Greedy Algorithms

Lemma 16.3 (Optimal substructure)

Let $C$ be an alphabet and $f$ be the frequency function for characters in $C$. Let $x$ and $y$ be two characters in $C$ having the lowest frequencies. Let

$$C' = C - \{x, y\} \cup \{z\}$$

and the frequency function for $C'$ is the same as $f$ except that $f(z) = f(x) + f(y)$.

Let $T'$ be any an optimal prefix code tree for $C'$. Then the tree $T$ obtained from $T'$ by replacing the leaf node $z$ with an internal node having $x$ and $y$ as children, is an optimal prefix code tree for $C$.

Proof:
The code $T$ for $C$ constructed from $T'$, a code for $C'$, has the following objective function value

$$B(T) = \sum_{c \in C} f(c)d_T(c) = \sum_{c \in C - \{x, y\}} f(c)d_T(c) + f(x)d_T(x) + f(y)d_T(y)$$

$$= \sum_{c \in C - \{x, y\}} f(c)d_T(c) + [f(x) + f(y)]d_T(x)$$

$$= \sum_{c \in C - \{x, y\}} f(c)d_{T'}(c) + [f(x) + f(y)](d_{T'}(z) + 1)$$
Chapter 16. Greedy Algorithms

Lemma 16.3 (Optimal substructure)

Let $C$ be an alphabet and $f$ be the frequency function for characters in $C$. Let $x$ and $y$ be two characters in $C$ having the lowest frequencies. Let

$$C' = C - \{x, y\} \cup \{z\}$$

and the frequency function for $C'$ is the same as $f$ except that $f(z) = f(x) + f(y)$.

Let $T'$ be any an optimal prefix code tree for $C'$. Then the tree $T$ obtained from $T'$ by replacing the leaf node $z$ with an internal node having $x$ and $y$ as children, is an optimal prefix code tree for $C$.

Proof:
The code $T$ for $C$ constructed from $T'$, a code for $C'$, has the following objective function value

$$B(T) = \sum_{c \in C} f(c)d_T(c) = \sum_{c \in C - \{x, y\}} f(c)d_T(c) + f(x)d_T(x) + f(y)d_T(y)$$

$$= \sum_{c \in C - \{x, y\}} f(c)d_T(c) + [f(x) + f(y)]d_T(x)$$

$$= \sum_{c \in C - \{x, y\}} f(c)d_{T'}(c) + [f(x) + f(y)](d_{T'}(z) + 1)$$

$$= \sum_{c \in C - \{x, y\}} f(c)d_{T'}(c) + [f(x) + f(y)]d_{T'}(z) + [f(x) + f(y)]$$
Chapter 16. Greedy Algorithms

Lemma 16.3 (Optimal substructure)

Let $C$ be an alphabet and $f$ be the frequency function for characters in $C$. Let $x$ and $y$ be two characters in $C$ having the lowest frequencies. Let

$$C' = C - \{x, y\} \cup \{z\}$$

and the frequency function for $C'$ is the same as $f$ except that $f(z) = f(x) + f(y)$.

Let $T'$ be any an optimal prefix code tree for $C'$. Then the tree $T$ obtained from $T'$ by replacing the leaf node $z$ with an internal node having $x$ and $y$ as children, is an optimal prefix code tree for $C$.

Proof:
The code $T$ for $C$ constructed from $T'$, a code for $C'$, has the following objective function value

$$B(T) = \sum_{c \in C} f(c)d_T(c) = \sum_{c \in C - \{x, y\}} f(c)d_T(c) + f(x)d_T(x) + f(y)d_T(y)$$

$$= \sum_{c \in C - \{x, y\}} f(c)d_T(c) + [f(x) + f(y)]d_T(x)$$

$$= \sum_{c \in C - \{x, y\}} f(c)d_{T'}(c) + [f(x) + f(y)](d_{T'}(z) + 1)$$

$$= \sum_{c \in C - \{x, y\}} f(c)d_{T'}(c) + [f(x) + f(y)]d_{T'}(z) + [f(x) + f(y)]$$

$$= \sum_{c \in C - \{x, y\}} f(c)d_{T'}(c) + f(z)d_{T'}(z) + [f(x) + f(y)]$$
Chapter 16. Greedy Algorithms

Lemma 16.3 (Optimal substructure)

Let \( C \) be an alphabet and \( f \) be the frequency function for characters in \( C \). Let \( x \) and \( y \) be two characters in \( C \) having the lowest frequencies. Let

\[
C' = C - \{x, y\} \cup \{z\}
\]

and the frequency function for \( C' \) is the same as \( f \) except that \( f(z) = f(x) + f(y) \).

Let \( T' \) be any an optimal prefix code tree for \( C' \). Then the tree \( T \) obtained from \( T' \) by replacing the leaf node \( z \) with an internal node having \( x \) and \( y \) as children, is an optimal prefix code tree for \( C \).

Proof:
The code \( T \) for \( C \) constructed from \( T' \), a code for \( C' \), has the following objective function value

\[
B(T) = \sum_{c \in C} f(c)d_T(c) = \sum_{c \in C - \{x, y\}} f(c)d_T(c) + f(x)d_T(x) + f(y)d_T(y)
\]

\[
= \sum_{c \in C - \{x, y\}} f(c)d_T(c) + [f(x) + f(y)]d_T(x)
\]

\[
= \sum_{c \in C - \{x, y\}} f(c)d_{T'}(c) + [f(x) + f(y)](d_{T'}(z) + 1)
\]

\[
= \sum_{c \in C - \{x, y\}} f(c)d_{T'}(c) + [f(x) + f(y)]d_{T'}(z) + [f(x) + f(y)]
\]

\[
= \sum_{c \in C - \{x, y\} \cup \{z\}} f(c)d_{T'}(c) + [f(x) + f(y)]
\]
Lemma 16.3 (Optimal substructure)

Let $C$ be an alphabet and $f$ be the frequency function for characters in $C$. Let $x$ and $y$ be two characters in $C$ having the lowest frequencies. Let

$$C' = C - \{x, y\} \cup \{z\}$$

and the frequency function for $C'$ is the same as $f$ except that $f(z) = f(x) + f(y)$.

Let $T'$ be any an optimal prefix code tree for $C'$. Then the tree $T$ obtained from $T'$ by replacing the leaf node $z$ with an internal node having $x$ and $y$ as children, is an optimal prefix code tree for $C$.

Proof:
The code $T$ for $C$ constructed from $T'$, a code for $C'$, has the following objective function value

$$B(T) = \sum_{c \in C} f(c)d_T(c) = \sum_{c \in C - \{x, y\}} f(c)d_T(c) + f(x)d_T(x) + f(y)d_T(y)$$

$$= \sum_{c \in C - \{x, y\}} f(c)d_T(c) + [f(x) + f(y)]d_T(x)$$

$$= \sum_{c \in C - \{x, y\}} f(c)d_{T'}(c) + [f(x) + f(y)](d_{T'}(z) + 1)$$

$$= \sum_{c \in C - \{x, y\}} f(c)d_{T'}(c) + [f(x) + f(y)]d_{T'}(z) + [f(x) + f(y)]$$

$$= \sum_{c \in C - \{x, y\}} f(c)d_{T'}(c) + f(z)d_{T'}(z) + [f(x) + f(y)]$$

$$= \sum_{c \in C - \{x, y\} \cup \{z\}} f(c)d_{T'}(c) + [f(x) + f(y)]$$

$$= \sum_{c \in C'} f(c)d_{T'}(c) + [f(x) + f(y)] = B(T') + f(x) + f(y)$$
Lemma 16.3 (Optimal substructure)

Let $C$ be an alphabet and $f$ be the frequency function for characters in $C$. Let $x$ and $y$ be two characters in $C$ having the lowest frequencies. Let

$$C' = C - \{x, y\} \cup \{z\}$$

and the frequency function for $C'$ is the same as $f$ except that $f(z) = f(x) + f(y)$.

Let $T'$ be any an optimal prefix code tree for $C'$. Then the tree $T$ obtained from $T'$ by replacing the leaf node $z$ with an internal node having $x$ and $y$ as children, is an optimal prefix code tree for $C$.

Proof:
The code $T$ for $C$ constructed from $T'$, a code for $C'$, has the following objective function value

$$B(T) = \sum_{c \in C} f(c)d_T(c) = \sum_{c \in C - \{x, y\}} f(c)d_T(c) + f(x)d_T(x) + f(y)d_T(y)$$

$$= \sum_{c \in C - \{x, y\}} f(c)d_T(c) + [f(x) + f(y)]d_T(x)$$

$$= \sum_{c \in C - \{x, y\}} f(c)d_{T'(c)} + [f(x) + f(y)](d_{T'(z)} + 1)$$

$$= \sum_{c \in C - \{x, y\}} f(c)d_{T'(c)} + [f(x) + f(y)]d_{T'(z)} + [f(x) + f(y)]$$

$$= \sum_{c \in C - \{x, y\}} f(c)d_{T'(c)} + f(z)d_{T'(z)} + [f(x) + f(y)]$$

$$= \sum_{c \in C - \{x, y\} \cup \{z\}} f(c)d_{T'(c)} + [f(x) + f(y)]$$

$$= \sum_{c \in C'} f(c)d_{T'(c)} + [f(x) + f(y)] = B(T') + f(x) + f(y)$$
Now if $T$ is not optimal for $C$, assume optimal prefix code $S$ for $C$ such that $B(S) < B(T)$.
Chapter 16. Greedy Algorithms

Now if $T$ is not optimal for $C$, assume optimal prefix code $S$ for $C$ such that $B(S) < B(T)$.

By Lemma 16.2, we assume $S$ to be such that codewords for $x$ and $y$ are of the same depth and differ in the last bit.
Chapter 16. Greedy Algorithms

Now if $T$ is not optimal for $C$, assume optimal prefix code $S$ for $C$ such that $B(S) < B(T)$.

By Lemma 16.2, we assume $S$ to be such that codewords for $x$ and $y$ are of the same depth and differ in the last bit.

We construct another code $S'$ for $C'$ by removing the branches for $x$ and $y$ and assign $z$ to its parent node. We have $B(S') = B(S) - f(x) - f(y)$. But

$$B(S') < B(T) - f(x) - f(y) = B(T') + f(x) + f(y) - f(x) - f(y) = B(T')$$
Now if $T$ is not optimal for $C$, assume optimal prefix code $S$ for $C$ such that $B(S) < B(T)$.

By Lemma 16.2, we assume $S$ to be such that codewords for $x$ and $y$ are of the same depth and differ in the last bit.

We construct another code $S'$ for $C'$ by removing the branches for $x$ and $y$ and assign $z$ to its parent node. We have $B(S') = B(S) - f(x) - f(y)$. But

$$B(S') < B(T) - f(x) - f(y) = B(T') + f(x) + f(y) - f(x) - f(y) = B(T')$$

This implies $T'$ is not optimal code for $C'$. Contradict.
Now if $T$ is not optimal for $C$, assume optimal prefix code $S$ for $C$ such that $B(S) < B(T)$.

By Lemma 16.2, we assume $S$ to be such that codewords for $x$ and $y$ are of the same depth and differ in the last bit.

We construct another code $S'$ for $C'$ by removing the branches for $x$ and $y$ and assign $z$ to its parent node. We have $B(S') = B(S) - f(x) - f(y)$. But

$$B(S') < B(T) - f(x) - f(y) = B(T') + f(x) + f(y) - f(x) - f(y) = B(T')$$

This implies $T'$ is not optimal code for $C'$. Contradict.

So $T$ should be optimal for $C$. 

**Theorem 16.4**

Huffman’s algorithm produces an optimal prefix code.
Now if $T$ is not optimal for $C$, assume optimal prefix code $S$ for $C$ such that $B(S) < B(T)$.

By Lemma 16.2, we assume $S$ to be such that codewords for $x$ and $y$ are of the same depth and differ in the last bit.

We construct another code $S'$ for $C'$ by removing the branches for $x$ and $y$ and assign $z$ to its parent node. We have $B(S') = B(S) - f(x) - f(y)$. But

$$B(S') < B(T) - f(x) - f(y) = B(T') + f(x) + f(y) - f(x) - f(y) = B(T')$$

This implies $T'$ is not optimal code for $C'$. Contradict. So $T$ should be optimal for $C$.

**Theorem 16.4** Huffman’s algorithm produces an optimal prefix code.
Chapter 16. Greedy Algorithms

Coin change problem

Input: money of value $n$, currency coin denominations $D = \{v_1, ..., v_m\}$

Output: the least number of coins with denominations in $D$ whose values sum up exactly to $n$.

A mathematical formulation of the problem

Input: positive integer $n$, and set of positive integers $D = \{v_1, ..., v_m\}$

Output: a set of positive integers $X = \{x_1, ..., x_m\}$ such that

$$\sum_{i=1}^{m} x_i v_i = n$$

to minimize

$$\sum_{i=1}^{m} x_i$$
Chapter 16. Greedy Algorithms

Coin change problem

**Input**: money of value $n$, currency coin denominations $D = \{v_1, \ldots, v_m\}$
Chapter 16. Greedy Algorithms

Coin change problem

**INPUT:** money of value $n$, currency coin denominations $D = \{v_1, \ldots, v_m\}$

**OUTPUT:** the least number of coins with denominations in $D$ whose values sum up exactly to $n$. 
Coin change problem

**INPUT:** money of value \( n \), currency coin denominations \( D = \{v_1, \ldots, v_m\} \)

**OUTPUT:** the least number of coins with denominations in \( D \) whose values sum up exactly to \( n \).

A mathematical formulation of the problem

**INPUT:** positive integer \( n \), and set of positive integers \( D = \{v_1, \ldots, v_m\} \)
Coin change problem

**Input:** money of value $n$, currency coin denominations $D = \{v_1, \ldots, v_m\}$

**Output:** the least number of coins with denominations in $D$ whose values sum up exactly to $n$.

A mathematical formulation of the problem

**Input:** positive integer $n$, and set of positive integers $D = \{v_1, \ldots, v_m\}$

**Output:** a set of positive integers $X = \{x_1, \ldots, x_m\}$ such that
Chapter 16. Greedy Algorithms

Coin change problem

Input: money of value $n$, currency coin denominations $D = \{v_1, \ldots, v_m\}$
Output: the least number of coins with denominations in $D$ whose values sum up exactly to $n$.

A mathematical formulation of the problem

Input: positive integer $n$, and set of positive integers $D = \{v_1, \ldots, v_m\}$
Output: a set of positive integers $X = \{x_1, \ldots, x_m\}$ such that

$$\sum_{i=1}^{m} x_i v_i = n \text{ to minimize } \left(\sum_{i=1}^{m} x_i\right)$$
Chapter 16. Greedy Algorithms

A dynamic programming solution

...
A dynamic programming solution

**analysis:**
- for any value $k$, one coin of the $m$ denominations has to be used.
A dynamic programming solution

**analysis:**
- for any value $k$, one coin of the $m$ denominations has to be used.
- then for the left value, one coin of the $m$ denominations has to be used
A dynamic programming solution

**analysis:**
- for any value $k$, one coin of the $m$ denominations has to be used.
- then for the left value, one coin of the $m$ denominations has to be used
- ...
A dynamic programming solution

**analysis:**
- for any value $k$, one coin of the $m$ denominations has to be used.
- then for the left value, one coin of the $m$ denominations has to be used
- ...

**define** objective function:
- $c(k)$ to be the least number of coins to change value $k$. Then
A dynamic programming solution

**analysis:**
- for any value $k$, one coin of the $m$ denominations has to be used.
- then for the left value, one coin of the $m$ denominations has to be used
- ...

**define** objective function:
- $c(k)$ to be the least number of coins to change value $k$. Then

\[
c(k) = \min \left\{ \begin{array}{c}
\end{array} \right\}
\]
Chapter 16. Greedy Algorithms

A dynamic programming solution

**analysis:**
- for any value $k$, one coin of the $m$ denominations has to be used.
- then for the left value, one coin of the $m$ denominations has to be used.
- ...

**define** objective function:
- $c(k)$ to be the least number of coins to change value $k$. Then

$$c(k) = \min \begin{cases} 
  c(k - v_1) + 1 & k \geq v_1 \\
  c(k - v_m) + 1 & k \geq v_m
\end{cases}$$
Chapter 16. Greedy Algorithms

A dynamic programming solution

**analysis:**
- for any value $k$, one coin of the $m$ denominations has to be used.
- then for the left value, one coin of the $m$ denominations has to be used
- ...

**define** objective function:
- $c(k)$ to be the least number of coins to change value $k$. Then

$$c(k) = \min \begin{cases} 
  c(k - v_1) + 1 & k \geq v_1 \\
  c(k - v_2) + 1 & k \geq v_2 
\end{cases}$$
A dynamic programming solution

**analysis:**
- for any value $k$, one coin of the $m$ denominations has to be used.
- then for the left value, one coin of the $m$ denominations has to be used
- ...

**define** objective function:
- $c(k)$ to be the least number of coins to change value $k$. Then

$$c(k) = \min \begin{cases} 
  c(k - v_1) + 1 & k \geq v_1 \\
  c(k - v_2) + 1 & k \geq v_2 \\
  \ldots & \end{cases}$$
A dynamic programming solution

analysis:
- for any value $k$, one coin of the $m$ denominations has to be used.
- then for the left value, one coin of the $m$ denominations has to be used
- ...

define objective function:
- $c(k)$ to be the least number of coins to change value $k$. Then

\[
c(k) = \min \left\{ \begin{array}{ll}
c(k - v_1) + 1 & k \geq v_1 \\
c(k - v_2) + 1 & k \geq v_2 \\
\vdots \\
c(k - v_m) + 1 & k \geq v_m \\
\end{array} \right.
\]
Chapter 16. Greedy Algorithms

A dynamic programming solution

analysis:
• for any value $k$, one coin of the $m$ denominations has to be used.
• then for the left value, one coin of the $m$ denominations has to be used
• ...

define objective function:
• $c(k)$ to be the least number of coins to change value $k$. Then

$$c(k) = \min \begin{cases} 
  c(k - v_1) + 1 & k \geq v_1 \\
  c(k - v_2) + 1 & k \geq v_2 \\
  \cdots \\
  c(k - v_m) + 1 & k \geq v_m 
\end{cases}$$
Chapter 16. Greedy Algorithms

A dynamic programming solution

analysis:

- for any value $k$, one coin of the $m$ denominations has to be used.
- then for the left value, one coin of the $m$ denominations has to be used
- ...

define objective function:

- $c(k)$ to be the least number of coins to change value $k$. Then

\[
  c(k) = \min \begin{cases} 
    c(k - v_1) + 1 & k \geq v_1 \\
    c(k - v_2) + 1 & k \geq v_2 \\
    \ldots \\
    c(k - v_m) + 1 & k \geq v_m 
  \end{cases}
\]

- base case: $c(v_1) = 1$,
Chapter 16. Greedy Algorithms

A dynamic programming solution

**analysis:**
- for any value $k$, one coin of the $m$ denominations has to be used.
- then for the left value, one coin of the $m$ denominations has to be used
- ...

**define** objective function:
- $c(k)$ to be the least number of coins to change value $k$. Then

$$c(k) = \min \left\{ \begin{array}{ll}
  c(k - v_1) + 1 & k \geq v_1 \\
  c(k - v_2) + 1 & k \geq v_2 \\
  \ldots & \\
  c(k - v_m) + 1 & k \geq v_m 
\end{array} \right.$$

- base case: $c(v_1) = 1, \ldots, c(v_m) = 1$. 
Chapter 16. Greedy Algorithms

A greedy algorithm

Theorem: For the US coin denominations $D_{us} = \{1, 5, 10, 25\}$, the Coin Change problem has the greedy-choice property.

• always use the coin of the largest possible value.

but not all coin denominations have such property, e.g., $\{1, 3, 4, 10\}$, for $n = 6$. 
A greedy algorithm

\[ c(k) = \min \left\{ \begin{array}{ll}
  c(k - v_1) + 1 & k \geq v_1 \\
  c(k - v_2) + 1 & k \geq v_2 \\
  \ldots \\
  c(k - v_m) + 1 & k \geq v_m
\end{array} \right. \]

only need to consider one of the denominations
A greedy algorithm

\[ c(k) = \min \begin{cases} 
  c(k - v_1) + 1 & k \geq v_1 \\
  c(k - v_2) + 1 & k \geq v_2 \\
  \ldots \\
  c(k - v_m) + 1 & k \geq v_m 
\end{cases} \]

Theorem: For the US coin denominations \( D_{us} = \{1, 5, 10, 25\} \), the Coin Change problem has the greedy-choice property.
Chapter 16. Greedy Algorithms

A greedy algorithm

\[ c(k) = \min \begin{cases} 
  c(k - v_1) + 1 & k \geq v_1 \\
  c(k - v_2) + 1 & k \geq v_2 \\
  \cdots \ \\
  c(k - v_m) + 1 & k \geq v_m 
\end{cases} \]

only need to consider one of the denominations

Theorem: For the US coin denominations \( D_{us} = \{1, 5, 10, 25\} \), the Coin Change problem has the greedy-choice property.

- always use the coin of the largest possible value.
A greedy algorithm

\[ c(k) = \min \left\{ \begin{array}{ll} c(k - v_1) + 1 & k \geq v_1 \\ c(k - v_2) + 1 & k \geq v_2 \\ \vdots \\ c(k - v_m) + 1 & k \geq v_m \end{array} \right. \]

**Theorem:** For the US coin denominations \( D_{us} = \{1, 5, 10, 25\} \), the Coin Change problem has the greedy-choice property.

- always use the coin of the largest possible value.

but not all coin denominations have such property,
A greedy algorithm

\[
c(k) = \min \left\{ \begin{array}{ll}
c(k - v_1) + 1 & k \geq v_1 \\
c(k - v_2) + 1 & k \geq v_2 \\
& \cdots \\
& c(k - v_m) + 1 & k \geq v_m \\
\end{array} \right. 
\]

Theorem: For the US coin denominations \( D_{us} = \{1, 5, 10, 25\} \), the Coin Change problem has the greedy-choice property.

- always use the coin of the largest possible value.

but not all coin denominations have such property, e.g., \( \{1, 3, 4, 10\} \), for \( n = 6 \).
Proof: Assume $A$ is the optimal solution for $n$. 

• I. For $5 \leq n < 10$, assume $5 \not\in A$, but all coins in $A$ have to be one-cent coins. Let $B = A - \{1, 1, 1, 1, 1\} \cup \{5\}$. $|B| < |A|$, which is impossible. So $5 \in A$. 

• II. For $10 \leq n < 25$, assume $10 \not\in A$, but all coins in $A$ are either five-cent or one-cent coins. 
   1. all coins in $A$ are one-cent coins, let $B = A - \{1, 1, 1, 1, 1, 1, 1, 1, 1\} \cup \{10\}$, then $|B| < |A|$, which is impossible. So $10 \in A$; 
   2. there is one five-cent coin and some one-cent coins in $A$, let $B = A - \{1, 1, 1, 1, 1, 5\} \cup \{10\}$, $|B| < |A|$, which is impossible. So $10 \in A$; 
   3. there is at least two five-cent coins in $A$, so $B = A - \{5, 5\} \cup \{10\}$, $|B| < |A|$, which is impossible. So $10 \in A$;
Proof: Assume $A$ is the optimal solution for $n$. Consider various cases
Chapter 16. Greedy Algorithms

**Proof**: Assume $A$ is the optimal solution for $n$. Consider various cases

- I. For $5 \leq n < 10$, assume $5 \notin A$, but all coins in $A$ have to be one-cent coins.
Chapter 16. Greedy Algorithms

Proof: Assume $A$ is the optimal solution for $n$. Consider various cases

- I. For $5 \leq n < 10$, assume $5 \notin A$, but all coins in $A$ have to be one-cent coins. Let $B = A - \{1, 1, 1, 1\} \cup \{5\}$. $|B| < |A|$, which is impossible. So $5 \in A$. 

- II. For $10 \leq n < 25$, assume $10 \notin A$, but all coins in $A$ are either five-cent or one-cent coins.
  - (1) all coins in $A$ are one-cent coins, let $B = A - \{1, 1, 1, 1\} \cup \{10\}$, then $|B| < |A|$, which is impossible. So $10 \in A$;
  - (2) there is one five-cent coin and some one-cent coins in $A$, let $B = A - \{1, 1, 1, 1, 5\} \cup \{10\}$, $|B| < |A|$, which is impossible. So $10 \in A$;
  - (3) there is at least two five-cent coins in $A$, so $B = A - \{5, 5\} \cup \{10\}$, $|B| < |A|$, which is impossible. So $10 \in A$;
Chapter 16. Greedy Algorithms

Proof: Assume $A$ is the optimal solution for $n$. Consider various cases

- I. For $5 \leq n < 10$, assume $5 \not\in A$, but all coins in $A$ have to be one-cent coins.
  Let $B = A - \{1, 1, 1, 1, 1\} \cup \{5\}$. $|B| < |A|$, which is impossible. So $5 \in A$.

- II. For $10 \leq n < 25$, assume $10 \not\in A$, but all coins in $A$ are either five-cent or one-cent coins.
Proof: Assume $A$ is the optimal solution for $n$. Consider various cases

**I.** For $5 \leq n < 10$, assume $5 \notin A$, but all coins in $A$ have to be one-cent coins. Let $B = A - \{1,1,1,1,1\} \cup \{5\}$. $|B| < |A|$, which is impossible. So $5 \in A$.

**II.** For $10 \leq n < 25$, assume $10 \notin A$, but all coins in $A$ are either five-cent or one-cent coins.

(1) all coins in $A$ are one-cent coins, let $B = A - \{1,1,1,1,1,1,1,1\} \cup \{10\}$,
Proof: Assume $A$ is the optimal solution for $n$. Consider various cases

- I. For $5 \leq n < 10$, **assume** $5 \not\in A$, but all coins in $A$ have to be one-cent coins.
  Let $B = A - \{1, 1, 1, 1, 1\} \cup \{5\}$. $|B| < |A|$, which is impossible. So $5 \in A$.

- II. For $10 \leq n < 25$, **assume** $10 \not\in A$, but all coins in $A$ are either five-cent or one-cent coins.
  
  (1) all coins in $A$ are one-cent coins, let $B = A - \{1, 1, 1, 1, 1, 1, 1, 1\} \cup \{10\}$, then $|B| < |A|$, which is impossible. So $10 \in A$;
Proof: Assume $A$ is the optimal solution for $n$. Consider various cases

• I. For $5 \leq n < 10$, assume $5 \not\in A$, but all coins in $A$ have to be one-cent coins. Let $B = A - \{1, 1, 1, 1, 1\} \cup \{5\}$. $|B| < |A|$, which is impossible. So $5 \in A$.

• II. For $10 \leq n < 25$, assume $10 \not\in A$, but all coins in $A$ are either five-cent or one-cent coins.

  (1) all coins in $A$ are one-cent coins, let $B = A - \{1, 1, 1, 1, 1, 1, 1, 1\} \cup \{10\}$, then $|B| < |A|$, which is impossible. So $10 \in A$;

  (2) there is one five-cent coin and some one-cent coins in $A$, 


Proof: Assume $A$ is the optimal solution for $n$. Consider various cases

- I. For $5 \leq n < 10$, assume $5 \not\in A$, but all coins in $A$ have to be one-cent coins. Let $B = A - \{1, 1, 1, 1, 1\} \cup \{5\}$. $|B| < |A|$, which is impossible. So $5 \in A$.

- II. For $10 \leq n < 25$, assume $10 \not\in A$, but all coins in $A$ are either five-cent or one-cent coins.
  
  (1) all coins in $A$ are one-cent coins, let $B = A - \{1, 1, 1, 1, 1, 1, 1, 1\} \cup \{10\}$, then $|B| < |A|$, which is impossible. So $10 \in A$;

  (2) there is one five-cent coin and some one-cent coins in $A$, let $B = A - \{1, 1, 1, 1, 5\} \cup \{10\}$, $|B| < |A|$, which is impossible.
Chapter 16. Greedy Algorithms

Proof: Assume $A$ is the optimal solution for $n$. Consider various cases

- I. For $5 \leq n < 10$, assume $5 \not\in A$, but all coins in $A$ have to be one-cent coins.
  Let $B = A - \{1, 1, 1, 1, 1\} \cup \{5\}$. $|B| < |A|$, which is impossible. So $5 \in A$.

- II. For $10 \leq n < 25$, assume $10 \not\in A$, but all coins in $A$ are either five-cent or one-cent coins.
  
  (1) all coins in $A$ are one-cent coins, let $B = A - \{1, 1, 1, 1, 1, 1, 1, 1, 1\} \cup \{10\}$, then $|B| < |A|$, which is impossible. So $10 \in A$;

  (2) there is one five-cent coin and some one-cent coins in $A$,
  let $B = A - \{1, 1, 1, 1, 5\} \cup \{10\}$, $|B| < |A|$, which is impossible. So $10 \in A$;
Proof: Assume $A$ is the optimal solution for $n$. Consider various cases

- I. For $5 \leq n < 10$, assume $5 \notin A$, but all coins in $A$ have to be one-cent coins. Let $B = A - \{1, 1, 1, 1, 1\} \cup \{5\}$. $|B| < |A|$, which is impossible. So $5 \in A$.

- II. For $10 \leq n < 25$, assume $10 \notin A$, but all coins in $A$ are either five-cent or one-cent coins.
  
  (1) all coins in $A$ are one-cent coins, let $B = A - \{1, 1, 1, 1, 1, 1, 1, 1\} \cup \{10\}$, then $|B| < |A|$, which is impossible. So $10 \in A$;

  (2) there is one five-cent coin and some one-cent coins in $A$, let $B = A - \{1, 1, 1, 1, 1, 5\} \cup \{10\}$, $|B| < |A|$, which is impossible. So $10 \in A$;

  (3) there is at least two five-cent coins in $A$, so $B = A - \{5, 5\} \cup \{10\}$,
Proof: Assume $A$ is the optimal solution for $n$. Consider various cases

- I. For $5 \leq n < 10$, assume $5 \notin A$, but all coins in $A$ have to be one-cent coins.
  Let $B = A - \{1, 1, 1, 1, 1\} \cup \{5\}$. $|B| < |A|$, which is impossible. So $5 \in A$.

- II. For $10 \leq n < 25$, assume $10 \notin A$, but all coins in $A$ are either five-cent or one-cent coins.
  
  (1) all coins in $A$ are one-cent coins, let $B = A - \{1, 1, 1, 1, 1, 1, 1, 1\} \cup \{10\}$,
  then $|B| < |A|$, which is impossible. So $10 \in A$;

  (2) there is one five-cent coin and some one-cent coins in $A$,
  let $B = A - \{1, 1, 1, 1, 5\} \cup \{10\}$, $|B| < |A|$, which is impossible.
  So $10 \in A$;

  (3) there is at least two five-cent coins in $A$, so $B = A - \{5, 5\} \cup \{10\}$,
  $|B| < |A|$, which is impossible. So $10 \in A$;
Proof: Assume $A$ is the optimal solution for $n$. Consider various cases

- I. For $5 \leq n < 10$, assume $5 \notin A$, but all coins in $A$ have to be one-cent coins. Let $B = A - \{1, 1, 1, 1, 1\} \cup \{5\}$. $|B| < |A|$, which is impossible. So $5 \in A$.

- II. For $10 \leq n < 25$, assume $10 \notin A$, but all coins in $A$ are either five-cent or one-cent coins.
  
  (1) all coins in $A$ are one-cent coins, let $B = A - \{1, 1, 1, 1, 1, 1, 1, 1\} \cup \{10\}$, then $|B| < |A|$, which is impossible. So $10 \in A$;

  (2) there is one five-cent coin and some one-cent coins in $A$, let $B = A - \{1, 1, 1, 1, 1, 5\} \cup \{10\}$, $|B| < |A|$, which is impossible. So $10 \in A$;

  (3) there is at least two five-cent coins in $A$, so $B = A - \{5, 5\} \cup \{10\}$, $|B| < |A|$, which is impossible. So $10 \in A$;
• III. For $25 \leq n$, assume $25 \notin A$, but all coins in $A$ are either ten-cents, five-cents or one-cent.
• III. For $25 \leq n$, assume $25 \notin A$, but all coins in $A$ are either ten-cents, five-cents or one-cent.
Chapter 16. Greedy Algorithms

• III. For $25 \leq n$, assume $25 \notin A$, but all coins in $A$ are either ten-cents, five-cents or one-cent.

  (1) all coins in $A$ are one cent coins,
III. For $25 \leq n$, assume $25 \not\in A$, but all coins in $A$ are either ten-cents, five-cents or one-cent.

(1) all coins in $A$ are one cent coins, ...
Chapter 16. Greedy Algorithms

• III. For $25 \leq n$, assume $25 \not\in A$, but all coins in $A$ are either ten-cents, five-cents or one-cent.

  (1) all coins in $A$ are one cent coins, ... So $25 \in A$;

  (2) there are some five-cent and one-cent coins in $A$. $B = A - C \cup \{25\}$, where $C$ contains five-cent and one-cent coins making up to 25 cents, $|C| > 1$
Chapter 16. Greedy Algorithms

III. For $25 \leq n$, assume $25 \not\in A$, but all coins in $A$ are either ten-cents, five-cents or one-cent.

(1) all coins in $A$ are one cent coins, ... So $25 \in A$;

(2) there are some five-cent and one-cent coins in $A$. $B = A - C \cup \{25\}$, where $C$ contains five-cent and one-cent coins making up to 25 cents, $|C| > 1$ So $|B| < |A|$, which is impossible. So $25 \in A$;
III. For $25 \leq n$, assume $25 \not\in A$, but all coins in $A$ are either ten-cents, five-cents or one-cent.

(1) all coins in $A$ are one cent coins, ... So $25 \in A$;

(2) there are some five-cent and one-cent coins in $A$. $B = A - C \cup \{25\}$, where $C$ contains five-cent and one-cent coins making up to 25 cents, $|C| > 1$ So $|B| < |A|$, which is impossible. So $25 \in A$;

(3) there is at least one ten-cent coin in $A$. Let $m = n - 10$. 
Chapter 16. Greedy Algorithms

- III. For $25 \leq n$, assume $25 \not\in A$, but all coins in $A$ are either ten-cents, five-cents or one-cent.

  1. all coins in $A$ are one cent coins, ... So $25 \in A$;

  2. there are some five-cent and one-cent coins in $A$. $B = A - C \cup \{25\}$, where $C$ contains five-cent and one-cent coins making up to 25 cents, $|C| > 1$ So $|B| < |A|$, which is impossible. So $25 \in A$;

  3. there is at least one ten-cent coin in $A$. Let $m = n - 10$.
    - If $25 \leq m$, the solution for $m$ has to be either case III.(1) or III.(2),
Chapter 16. Greedy Algorithms

- III. For $25 \leq n$, assume $25 \not\in A$, but all coins in $A$ are either ten-cents, five-cents or one-cent.

(1) all coins in $A$ are one cent coins, . . . So $25 \in A$;

(2) there are some five-cent and one-cent coins in $A$. $B = A - C \cup \{25\}$, where $C$ contains five-cent and one-cent coins making up to 25 cents, $|C| > 1$

So $|B| < |A|$, which is impossible. So $25 \in A$;

(3) there is at least one ten-cent coin in $A$. Let $m = n - 10$.
- If $25 \leq m$, the solution for $m$ has to be either case III.(1) or III.(2), then $25$ is in the solution for $m$, thus $25 \in A$. 
Chapter 16. Greedy Algorithms

• III. For $25 \leq n$, assume $25 \not\in A$, but all coins in $A$ are either ten-cents, five-cents or one-cent.

(1) all coins in $A$ are one cent coins, . . . So $25 \in A$;

(2) there are some five-cent and one-cent coins in $A$. $B = A - C \cup \{25\}$, where $C$ contains five-cent and one-cent coins making up to 25 cents, $|C| > 1$ So $|B| < |A|$, which is impossible. So $25 \in A$;

(3) there is at least one ten-cent coin in $A$. Let $m = n - 10$.

• If $25 \leq m$, the solution for $m$ has to be either case III.(1) or III.(2), then $25$ is in the solution for $m$, thus $25 \in A$.

• If $10 \leq m < 25$, according to II, 10 is in the solution for $m$. So we conclude that there are at least two ten-cent coins are in $A$. 


Chapter 16. Greedy Algorithms

• III. For $25 \leq n$, assume $25 \notin A$, but all coins in $A$ are either ten-cents, five-cents or one-cent.

(1) all coins in $A$ are one cent coins, . . . So $25 \in A$;

(2) there are some five-cent and one-cent coins in $A$. $B = A - C \cup \{25\}$, where $C$ contains five-cent and one-cent coins making up to 25 cents, $|C| > 1$
So $|B| < |A|$, which is impossible. So $25 \in A$;

(3) there is at least one ten-cent coin in $A$. Let $m = n - 10$.
• If $25 \leq m$, the solution for $m$ has to be either case III.(1) or III.(2), then $25$ is in the solution for $m$, thus $25 \in A$.

• If $10 \leq m < 25$, according to II, 10 is in the solution for $m$.
So we conclude that there are at least two ten-cent coins are in $A$. Now we assume $k = n - 2 \times 10$. 

• If $25 \leq m$, the solution for $m$ has to be either case III.(1) or III.(2), then $25$ is in the solution for $m$, thus $25 \in A$.
Chapter 16. Greedy Algorithms

• III. For $25 \leq n$, assume $25 \not\in A$, but all coins in $A$ are either ten-cents, five-cents or one-cent.

(1) all coins in $A$ are one cent coins, ... So $25 \in A$;

(2) there are some five-cent and one-cent coins in $A$. $B = A - C \cup \{25\}$, where $C$ contains five-cent and one-cent coins making up to 25 cents, $|C| > 1$.

So $|B| < |A|$, which is impossible. So $25 \in A$;

(3) there is at least one ten-cent coin in $A$. Let $m = n - 10$.

• If $25 \leq m$, the solution for $m$ has to be either case III.(1) or III.(2), then $25$ is in the solution for $m$, thus $25 \in A$.

• If $10 \leq m < 25$, according to II, 10 is in the solution for $m$.

So we conclude that there are at least two ten-cent coins are in $A$.

Now we assume $k = n - 2 \times 10$.

If $10 \leq k < 25$, according to II, 10 is in the solution for $k$,
Chapter 16. Greedy Algorithms

• III. For $25 \leq n$, assume $25 \not\in A$, but all coins in $A$ are either ten-cents, five-cents or one-cent.

(1) all coins in $A$ are one cent coins, ... So $25 \in A$;

(2) there are some five-cent and one-cent coins in $A$. $B = A - C \cup \{25\}$, where $C$ contains five-cent and one-cent coins making up to 25 cents, $|C| > 1$ So $|B| < |A|$, which is impossible. So $25 \in A$;

(3) there is at least one ten-cent coin in $A$. Let $m = n - 10$.
  • If $25 \leq m$, the solution for $m$ has to be either case III.(1) or III.(2), then $25$ is in the solution for $m$, thus $25 \in A$.
  
  • If $10 \leq m < 25$, according to II, 10 is in the solution for $m$. So we conclude that there are at least two ten-cent coins are in $A$.

Now we assume $k = n - 2 \times 10$.

If $10 \leq k < 25$, according to II, 10 is in the solution for $k$, then $\{10, 10, 10\} \subseteq A$, 

...
Chapter 16. Greedy Algorithms

- III. For $25 \leq n$, assume $25 \notin A$, but all coins in $A$ are either ten-cents, five-cents or one-cent.

  (1) all coins in $A$ are one cent coins, . . . So $25 \in A$;

  (2) there are some five-cent and one-cent coins in $A$. $B = A - C \cup \{25\}$, where $C$ contains five-cent and one-cent coins making up to 25 cents, $|C| > 1$
  So $|B| < |A|$, which is impossible. So $25 \in A$;

  (3) there is at least one ten-cent coin in $A$. Let $m = n - 10$.
  - If $25 \leq m$, the solution for $m$ has to be either case III.(1) or III.(2), then $25$ is in the solution for $m$, thus $25 \in A$.

  - If $10 \leq m < 25$, according to II, 10 is in the solution for $m$. So we conclude that there are at least two ten-cent coins are in $A$.
    Now we assume $k = n - 2 \times 10$.

    If $10 \leq k < 25$, according to II, 10 is in the solution for $k$, then $\{10, 10, 10\} \subseteq A$, let $B = A - \{10, 10, 10\} \cup \{25, 5\}$,
Chapter 16. Greedy Algorithms

• III. For $25 \leq n$, assume $25 \not\in A$, but all coins in $A$ are either ten-cents, five-cents or one-cent.

(1) all coins in $A$ are one cent coins, . . . So $25 \in A$;

(2) there are some five-cent and one-cent coins in $A$. $B = A - C \cup \{25\}$, where $C$ contains five-cent and one-cent coins making up to 25 cents, $|C| > 1$

So $|B| < |A|$, which is impossible. So $25 \in A$;

(3) there is at least one ten-cent coin in $A$. Let $m = n - 10$.

• If $25 \leq m$, the solution for $m$ has to be either case III.(1) or III.(2), then $25$ is in the solution for $m$, thus $25 \in A$.

• If $10 \leq m < 25$, according to II, $10$ is in the solution for $m$. So we conclude that there are at least two ten-cent coins are in $A$. Now we assume $k = n - 2 \times 10$.

If $10 \leq k < 25$, according to II, $10$ is in the solution for $k$, then $\{10, 10, 10\} \subseteq A$, let $B = A - \{10, 10, 10\} \cup \{25, 5\}$, $|B| < |A|$, which is impossible. So $25 \in A$;
Chapter 16. Greedy Algorithms

• III. For $25 \leq n$, assume $25 \not\in A$, but all coins in $A$ are either ten-cents, five-cents or one-cent.

(1) all coins in $A$ are one cent coins, . . . So $25 \in A$;

(2) there are some five-cent and one-cent coins in $A$. $B = A - C \cup \{25\}$, where $C$ contains five-cent and one-cent coins making up to 25 cents, $|C| > 1$.
So $|B| < |A|$, which is impossible. So $25 \in A$;

(3) there is at least one ten-cent coin in $A$. Let $m = n - 10$.
  • If $25 \leq m$, the solution for $m$ has to be either case III.(1) or III.(2), then $25$ is in the solution for $m$, thus $25 \in A$.
  • If $10 \leq m < 25$, according to II, $10$ is in the solution for $m$.
So we conclude that there are at least two ten-cent coins are in $A$.
Now we assume $k = n - 2 \times 10$.

If $10 \leq k < 25$, according to II, $10$ is in the solution for $k$, then $\{10, 10, 10\} \subseteq A$, let $B = A - \{10, 10, 10\} \cup \{25, 5\}$, $|B| < |A|$, which is impossible. So $25 \in A$;

Otherwise, $5 \geq k < 10$, according to I, $5$ is in the solution for $k$. 
• III. For $25 \leq n$, assume $25 \notin A$, but all coins in $A$ are either ten-cents, five-cents or one-cent.

(1) all coins in $A$ are one cent coins, . . . So $25 \in A$;

(2) there are some five-cent and one-cent coins in $A$. $B = A - C \cup \{25\}$, where $C$ contains five-cent and one-cent coins making up to 25 cents, $|C| > 1$
So $|B| < |A|$, which is impossible. So $25 \in A$;

(3) there is at least one ten-cent coin in $A$. Let $m = n - 10$.
  • If $25 \leq m$, the solution for $m$ has to be either case III.(1) or III.(2),
    then 25 is in the solution for $m$, thus $25 \in A$.

• If $10 \leq m < 25$, according to II, 10 is in the solution for $m$.
So we conclude that there are at least two ten-cent coins are in $A$.
Now we assume $k = n - 2 \times 10$.

If $10 \leq k < 25$, according to II, 10 is in the solution for $k$,
then $\{10, 10, 10\} \subseteq A$, let $B = A - \{10, 10, 10\} \cup \{25, 5\}$,
$|B| < |A|$, which is impossible. So $25 \in A$;

Otherwise, $5 \geq k < 10$, according to I, 5 is in the solution for $k$.
then $\{10, 10, 5\} \subseteq A$. Let $B = A - \{10, 10, 5\} \cup \{25\}$
Chapter 16. Greedy Algorithms

• III. For \(25 \leq n\), assume \(25 \notin A\), but all coins in \(A\) are either ten-cents, five-cents or one-cent.

(1) all coins in \(A\) are one cent coins, \ldots So \(25 \in A\);

(2) there are some five-cent and one-cent coins in \(A\). \(B = A - C \cup \{25\}\), where \(C\) contains five-cent and one-cent coins making up to 25 cents, \(|C| > 1\). So \(|B| < |A|\), which is impossible. So \(25 \in A\);

(3) there is at least one ten-cent coin in \(A\). Let \(m = n - 10\).
   • If \(25 \leq m\), the solution for \(m\) has to be either case III.(1) or III.(2), then \(25\) is in the solution for \(m\), thus \(25 \in A\).
   
   • If \(10 \leq m < 25\), according to II, \(10\) is in the solution for \(m\). So we conclude that there are at least two ten-cent coins are in \(A\).
     Now we assume \(k = n - 2 \times 10\).

     If \(10 \leq k < 25\), according to II, \(10\) is in the solution for \(k\), then \(\{10, 10, 10\} \subseteq A\), let \(B = A - \{10, 10, 10\} \cup \{25, 5\}\), \(|B| < |A|\), which is impossible. So \(25 \in A\);

     Otherwise, \(5 \geq k < 10\), according to I, \(5\) is in the solution for \(k\). then \(\{10, 10, 5\} \subseteq A\). Let \(B = A - \{10, 10, 5\} \cup \{25\}\) \(|B| < |A|\), which is impossible. So \(25 \in A\).
Review for Parts I, II, and III

Summaries for Parts I, II, and III
Review for Parts I, II, and III

Summaries for Parts I, II, and III

Part I: Fundamentals of Analysis of algorithms
Summaries for Parts I, II, and III

Part I: Fundamentals of Analysis of algorithms

- time/space complexity functions
Review for Parts I, II, and III

Summaries for Parts I, II, and III

Part I: Fundamentals of Analysis of algorithms

- time/space complexity functions
  input size
Review for Parts I, II, and III

Summaries for Parts I, II, and III

Part I: Fundamentals of Analysis of algorithms

- time/space complexity functions
  - input size
  - upper bound

- asymptotic notations
  - big-O
  - big-Omega

- recurrences for complexities of recursive algorithms
- recurrence proof methods:
  - recursive tree (unfolding)
  - substitution (induction)
Review for Parts I, II, and III

Summaries for Parts I, II, and III

Part I: Fundamentals of Analysis of algorithms

- time/space complexity functions
  - input size
  - upper bound
  - lower bound

- asymptotic notations
  - big-O
  - big-Ω

- recurrences for complexities of recursive algorithms

- recurrence proof methods:
  - recursive tree (unfolding)
  - substitution (induction)
Summaries for Parts I, II, and III

Part I: Fundamentals of Analysis of algorithms

- time/space complexity functions
  - input size
  - upper bound
  - lower bound

- asymptotic notations
Summaries for Parts I, II, and III

Part I: Fundamentals of Analysis of algorithms

- time/space complexity functions
  - input size
  - upper bound
  - lower bound

- asymptotic notations
  - big-$O$
Summaries for Parts I, II, and III

Part I: Fundamentals of Analysis of algorithms

- time/space complexity functions
  input size
  upper bound
  lower bound

- asymptotic notations
  big-$O$
  big-$\Omega$
Summaries for Parts I, II, and III

Part I: Fundamentals of Analysis of algorithms

- time/space complexity functions
  - input size
  - upper bound
  - lower bound

- asymptotic notations
  - big-$O$
  - big-$\Omega$
Reviewer for Parts I, II, and III

Summaries for Parts I, II, and III

Part I: Fundamentals of Analysis of algorithms

- time/space complexity functions
  input size
  upper bound
  lower bound

- asymptotic notations
  big-\(O\)
  big-\(\Omega\)

- recurrences for complexities of recursive algorithms
Summaries for Parts I, II, and III

Part I: Fundamentals of Analysis of algorithms

• time/space complexity functions
  input size
  upper bound
  lower bound

• asymptotic notations
  big-$O$
  big-$\Omega$

• recurrences for complexities of recursive algorithms

• recurrence proof methods:
Summaries for Parts I, II, and III

Part I: Fundamentals of Analysis of algorithms

- time/space complexity functions
  - input size
  - upper bound
  - lower bound

- asymptotic notations
  - big-$O$
  - big-$\Omega$

- recurrences for complexities of recursive algorithms

- recurrence proof methods:
  - recursive tree (unfolding)
Summaries for Parts I, II, and III

Part I: Fundamentals of Analysis of algorithms

- time/space complexity functions
  - input size
  - upper bound
  - lower bound
- asymptotic notations
  - big-$O$
  - big-$\Omega$
- recurrences for complexities of recursive algorithms
- recurrence proof methods:
  - recursive tree (unfolding)
  - substitution (induction)
Review for Parts I, II, and III

Part II: Sorting and order statistics

• sorting algorithms: heap, insertion, merge, quick
  their worst case time complexities (upper and lower bound)

• randomize sorting algorithms
  quick sort, expected time complexity

• lower bound for sorting (comparison-based model)

• linear time sorting algorithms

• linear time to find $k$th smallest element
Review for Parts I, II, and III

Part II: Sorting and order statistics

- sorting algorithms: heap, insertion, merge, quick
Review for Parts I, II, and III

Part II: Sorting and order statistics

- sorting algorithms: heap, insertion, merge, quick
  their worst case time complexities (upper and lower bound)
Review for Parts I, II, and III

Part II: Sorting and order statistics

- sorting algorithms: heap, insertion, merge, quick
  their worst case time complexities (upper and lower bound)
- randomize sorting algorithms
Review for Parts I, II, and III

Part II: Sorting and order statistics

• sorting algorithms: heap, insertion, merge, quick
  their worst case time complexities (upper and lower bound)

• randomize sorting algorithms
  quick sort, expected time complexity
Review for Parts I, II, and III

Part II: Sorting and order statistics

- sorting algorithms: heap, insertion, merge, quick
  their worst case time complexities (upper and lower bound)
- randomize sorting algorithms
  quick sort, expected time complexity
- lower bound for sorting (comparison-based model)
Review for Parts I, II, and III

Part II: Sorting and order statistics

- sorting algorithms: heap, insertion, merge, quick
  their worst case time complexities (upper and lower bound)

- randomize sorting algorithms
  quick sort, expected time complexity

- lower bound for sorting (comparison-based model)

- linear time sorting algorithms
Part II: Sorting and order statistics

- sorting algorithms: heap, insertion, merge, quick
  their worst case time complexities (upper and lower bound)

- randomize sorting algorithms
  quick sort, expected time complexity

- lower bound for sorting (comparison-based model)

- linear time sorting algorithms

- linear time to find $k$th smallest element
Review for Parts I, II, and III

Part III. Exhaustive search, DP and greedy algorithms
Part III. Exhaustive search, DP and greedy algorithms

- exhaustive search by

  - exhaustive search by simple enumeration of all possible solutions
    search tree method resulting in $T(n) = T(n-1) + T(n-1-m)$, $m \geq 1$

- dynamic programming recursive solution (with optimal substructure)
  recurrence for numerical objective function
  iterative, bottom-up table-filling
  traceback of solution.

  - 'forward DP', 'backward DP', and 'inside-out DP'

- greedy algorithm
  DP solution
  proof of greedy-choice property (via swapping method)
  efficient algorithms
Part III. Exhaustive search, DP and greedy algorithms

- exhaustive search by
  simple enumeration of all possible solutions

\[ T(n) = T(n-1) + T(n-1-m), m \geq 1. \]

- dynamic programming
  recursive solution (with optimal substructure)
  recurrence for numerical objective function
  iterative, bottom-up table-filling
  traceback of solution
  'forward DP', 'backward DP', and 'inside-out DP'

- greedy algorithm
  DP solution
  proof of greedy-choice property (via swapping method)
  efficient algorithms
Part III. Exhaustive search, DP and greedy algorithms

- exhaustive search by
  simple enumeration of all possible solutions
  search tree method

- dynamic programming
  recursive solution (with optimal substructure)
  recurrence for numerical objective function
  iterative, bottom-up table-filling
  traceback of solution.
  'forward DP', 'backward DP', and 'inside-out DP'

- greedy algorithm
  DP solution
  proof of greedy-choice property (via swapping method)
  efficient algorithms
Part III. Exhaustive search, DP and greedy algorithms

- exhaustive search by
  simple enumeration of all possible solutions
  search tree method resulting in \( T(n) = T(n - 1) + T(n - 1 - m), m \geq 1 \).
Review for Parts I, II, and III

Part III. Exhaustive search, DP and greedy algorithms

- exhaustive search by
  - simple enumeration of all possible solutions
  - search tree method resulting in $T(n) = T(n - 1) + T(n - 1 - m)$, $m \geq 1$. 

- dynamic programming
  - recursive solution (with optimal substructure)
  - recurrence for numerical objective function
  - iterative, bottom-up table-filling
  - traceback of solution.

- greedy algorithm
  - DP solution
  - proof of greedy-choice property (via swapping method)
  - efficient algorithms
Part III. Exhaustive search, DP and greedy algorithms

- exhaustive search by
  simple enumeration of all possible solutions
  search tree method resulting in $T(n) = T(n - 1) + T(n - 1 - m), \ m \geq 1$.

- dynamic programming
Part III. Exhaustive search, DP and greedy algorithms

- exhaustive search by
  simple enumeration of all possible solutions
  search tree method resulting in $T(n) = T(n - 1) + T(n - 1 - m), \ m \geq 1$.

- dynamic programming
  recursive solution (with optimal substructure)
Part III. Exhaustive search, DP and greedy algorithms

- exhaustive search by
  simple enumeration of all possible solutions
  search tree method resulting in $T(n) = T(n - 1) + T(n - 1 - m)$, $m \geq 1$.

- dynamic programming
  recursive solution (with optimal substructure)
  recurrence for numerical objective function
Part III. Exhaustive search, DP and greedy algorithms

- exhaustive search by simple enumeration of all possible solutions
  search tree method resulting in $T(n) = T(n - 1) + T(n - 1 - m), \ m \geq 1$

- dynamic programming recursive solution (with optimal substructure)
  recurrence for numerical objective function
  iterative, bottom-up table-filling
Part III. Exhaustive search, DP and greedy algorithms

- **exhaustive search by**
  simple enumeration of all possible solutions
  search tree method resulting in \( T(n) = T(n - 1) + T(n - 1 - m), m \geq 1 \).

- **dynamic programming**
  recursive solution (with optimal substructure)
  recurrence for numerical objective function
  iterative, bottom-up table-filling
  traceback of solution.
Part III. Exhaustive search, DP and greedy algorithms

- exhaustive search by
  simple enumeration of all possible solutions
  search tree method resulting in $T(n) = T(n - 1) + T(n - 1 - m), m \geq 1$.

- dynamic programming
  recursive solution (with optimal substructure)
  recurrence for numerical objective function
  iterative, bottom-up table-filling
  traceback of solution.

  ‘forward DP’, ’backward DP’, and ’inside-out DP’
Part III. Exhaustive search, DP and greedy algorithms

- exhaustive search by simple enumeration of all possible solutions search tree method resulting in \( T(n) = T(n - 1) + T(n - 1 - m), \ m \geq 1 \).
- dynamic programming recursive solution (with optimal substructure) recurrence for numerical objective function iterative, bottom-up table-filling traceback of solution.

  ‘forward DP’, ’backward DP’, and ’inside-out DP’
- greedy algorithm
Review for Parts I, II, and III

Part III. Exhaustive search, DP and greedy algorithms

- exhaustive search by
  simple enumeration of all possible solutions
  search tree method resulting in $T(n) = T(n-1) + T(n-1-m)$, $m \geq 1$.

- dynamic programming
  recursive solution (with optimal substructure)
  recurrence for numerical objective function
  iterative, bottom-up table-filling
  traceback of solution.
  ‘forward DP’, ’backward DP’, and ’inside-out DP’

- greedy algorithm
  DP solution
Part III. Exhaustive search, DP and greedy algorithms

- exhaustive search by
  simple enumeration of all possible solutions
  search tree method resulting in $T(n) = T(n - 1) + T(n - 1 - m), m \geq 1$.

- dynamic programming
  recursive solution (with optimal substructure)
  recurrence for numerical objective function
  iterative, bottom-up table-filling
  traceback of solution.

  ‘forward DP’, ’backward DP’, and ’inside-out DP’

- greedy algorithm
  DP solution
  proof of greedy-choice property (via swapping method)
Part III. Exhaustive search, DP and greedy algorithms

- exhaustive search by
  simple enumeration of all possible solutions
  search tree method resulting in $T(n) = T(n-1) + T(n-1-m), m \geq 1$.

- dynamic programming
  recursive solution (with optimal substructure)
  recurrence for numerical objective function
  iterative, bottom-up table-filling
  traceback of solution.
  ‘forward DP’, ’backward DP’, and ’inside-out DP’

- greedy algorithm
  DP solution
  proof of greedy-choice property (via swapping method)
  efficient algorithms