CSCI 4470/6470 Algorithms, Fall 2015

Lecture Note III

Liming Cai, Department of Computer Science, UGA

October 11, 2015

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

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

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

To enumerate all possible solutions to the problem instance

How?

- systematic examining all solutions

Chapter 14.9.Exhaustive Search

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

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

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

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

Example: Boolean Formula Satisfiability problem (SAT)

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.

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

$x_i \in \{\mathtt{T}, \mathtt{F}\}$, $i = 1, 2, \ldots, n$,

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

$x_i \in \{\mathtt{T}, \mathtt{F}\}$, $i = 1, 2, \ldots, n$, such that f is evaluated to $\mathtt{T}$.

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

$x_i \in \{\mathtt{T}, \mathtt{F}\}$, $i = 1, 2, \ldots, n$, such that f is evaluated to $\mathtt{T}$.

e.g,

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

$x_i \in \{\mathtt{T}, \mathtt{F}\}$, $i = 1, 2, \ldots, n$, such that f is evaluated to $\mathtt{T}$.

e.g,
$f(x_1, x_2, x_3) = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3)$ 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

$x_i \in \{\mathtt{T}, \mathtt{F}\}$, $i = 1, 2, \ldots, n$, such that f is evaluated to $\mathtt{T}$.

e.g,

$f(x_1, x_2, x_3) = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3)$ is satisfiable

$g(x_1, x_2) = (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2) \wedge (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$ is not!

Chapter 14.9.Exhaustive Search

Use exhaustive search to solve the SAT problem.

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.

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?

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 $x_1, \ldots, x_n$.

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 a iterative algorithm?

Chapter 14.9.Exhaustive Search

Solve SAT problem with a recursive algorithm:

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?

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?

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

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?

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

$f(x_1, \ldots, x_{n-1}, \mathtt{T})$

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?

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

$$f(x_1, \ldots, x_{n-1}, \mathtt{T}) \Longrightarrow g(x_1, \ldots, x_{n-1})$$

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

- what is the recursive case?

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

$f(x_1, \ldots, x_{n-1}, \mathtt{T}) \Longrightarrow g(x_1, \ldots, x_{n-1})$
$f(x_1, \ldots, x_{n-1}, \mathtt{F})$

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?

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

$$f(x_1, \ldots, x_{n-1}, \mathtt{T}) \Longrightarrow g(x_1, \ldots, x_{n-1})$$
$$f(x_1, \ldots, x_{n-1}, \mathtt{F}) \Longrightarrow h(x_1, \ldots, x_{n-1})$$

Chapter 14.9.Exhaustive Search

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

Chapter 14.9.Exhaustive Search

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

1. **if** $n = 0$, **return** (f);

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** $g(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, \mathtt{T})$

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** $g(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, \mathtt{T})$
3. $\quad h(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, \mathtt{F})$

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** $g(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, \mathtt{T})$
3. $h(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, \mathtt{F})$
4. **return** (SAT SOLVER$(g(x_1, \ldots, x_{n-1})) \vee$
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** $g(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, \mathtt{T})$
3. $h(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, \mathtt{F})$
4. **return** (SAT SOLVER$(g(x_1, \ldots, x_{n-1})) \vee$ 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** $g(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, \mathtt{T})$
3. $\quad h(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, \mathtt{F})$
4. $\quad$ **return** (SAT SOLVER$(g(x_1, \ldots, x_{n-1})) \vee$
   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 SAT SOLVER$(f(x_1, \ldots, x_{n-1}, x_n))$

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

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** $g(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, \mathtt{T})$
3. $\quad h(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, \mathtt{F})$
4. $\quad$ **return** $(\text{SAT SOLVER}(g(x_1, \ldots, x_{n-1})) \vee$
$\quad\quad\quad\quad \text{SAT SOLVER}(h(x_1, \ldots, x_{n-1})))$

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?

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** $g(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, \mathtt{T})$
3. $h(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, \mathtt{F})$
4. **return** (SAT SOLVER$(g(x_1, \ldots, x_{n-1})) \vee$ 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** $g(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, \mathtt{T})$
3. $h(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, \mathtt{F})$
4. **return** (SAT SOLVER$(g(x_1, \ldots, x_{n-1})) \vee$ 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 SAT SOLVER$(f(x_1, \ldots, x_{n-1}, x_n))$

1. **if** $n = 0$, **return** (f);
2. **else** $g(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, \mathtt{T})$
3. $h(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, \mathtt{F})$
4. **return** (SAT SOLVER$(g(x_1, \ldots, x_{n-1})) \vee$ 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? $T(n) = 2T(n-1) + cn$, $T(0) = c$,

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** $g(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, \mathtt{T})$
3. $h(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, \mathtt{F})$
4. **return** (SAT SOLVER$(g(x_1, \ldots, x_{n-1}))$ $\vee$ 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? $T(n) = 2T(n-1) + cn$, $T(0) = c$, $\Longrightarrow T(n) = \Theta(2^n)$

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?

Chapter 14.9.Exhaustive Search

Solve SAT problem with iterative algorithms

- How? what to iterate on?

  assignments

- what is the initial value?

  $x_1 = \mathtt{F}, x_2 = \mathtt{F}, \ldots, x_n = \mathtt{F},$

Chapter 14.9.Exhaustive Search

Solve SAT problem with iterative algorithms

- How? what to iterate on?

  assignments

- what is the initial value?

  $x_1 = \mathtt{F}, x_2 = \mathtt{F}, \ldots, x_n = \mathtt{F}$, or simply $(\mathtt{F}, \mathtt{F}, \ldots, \mathtt{F})$

Chapter 14.9.Exhaustive Search

Solve SAT problem with iterative algorithms

- How? what to iterate on?

  assignments

- what is the initial value?

  $x_1 = \mathtt{F}, x_2 = \mathtt{F}, \ldots, x_n = \mathtt{F}$, or simply $(\mathtt{F}, \mathtt{F}, \ldots, \mathtt{F})$

- what to increment

Chapter 14.9.Exhaustive Search

Solve SAT problem with iterative algorithms

- How? what to iterate on?

  assignments

- what is the initial value?

  $x_1 = \mathtt{F}, x_2 = \mathtt{F}, \ldots, x_n = \mathtt{F}$, or simply $(\mathtt{F}, \mathtt{F}, \ldots, \mathtt{F})$

- what to increment

  $(\ldots, \mathtt{F}, \mathtt{T}, \ldots, \mathtt{T})$

Chapter 14.9.Exhaustive Search

Solve SAT problem with iterative algorithms

- How? what to iterate on?

  assignments

- what is the initial value?

  $x_1 = \mathtt{F}, x_2 = \mathtt{F}, \ldots, x_n = \mathtt{F}$, or simply $(\mathtt{F}, \mathtt{F}, \ldots, \mathtt{F})$

- what to increment

  $(\ldots, \mathtt{F}, \mathtt{T}, \ldots, \mathtt{T}) \longrightarrow (\ldots, \mathtt{T}, \mathtt{F}, \ldots, \mathtt{F})$

Chapter 14.9.Exhaustive Search

Solve SAT problem with iterative algorithms

- How? what to iterate on?

  assignments

- what is the initial value?

  $x_1 = \mathtt{F}, x_2 = \mathtt{F}, \ldots, x_n = \mathtt{F}$, or simply $(\mathtt{F}, \mathtt{F}, \ldots, \mathtt{F})$

- what to increment

  $(\ldots, \mathtt{F}, \mathtt{T}, \ldots, \mathtt{T}) \longrightarrow (\ldots, \mathtt{T}, \mathtt{F}, \ldots, \mathtt{F})$
  always flip the last bit.

Chapter 14.9.Exhaustive Search

Chapter 14.9.Exhaustive Search

Algorithm 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** $\langle x_1, \ldots, x_n \rangle = \langle \mathtt{F}, \ldots, \mathtt{F} \rangle$ **to** $\langle \mathtt{T}, \ldots, \mathtt{T} \rangle$

Chapter 14.9.Exhaustive Search

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

1. **for** $\langle x_1, \ldots, x_n \rangle = \langle \mathtt{F}, \ldots, \mathtt{F} \rangle$ **to** $\langle \mathtt{T}, \ldots, \mathtt{T} \rangle$
2. $\quad V = Evaluate(f, x_1, \ldots, x_n)$

Chapter 14.9.Exhaustive Search

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

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

Chapter 14.9.Exhaustive Search

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

```
for ⟨x_1, ..., x_n⟩ = ⟨F, ..., F⟩ to ⟨T, ..., T⟩
    V = Evaluate(f, x_1, ..., x_n)
    if V = T, return (T)
return (F)
```

Chapter 14.9.Exhaustive Search

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

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

- **for** loop can be implemented by encoding vectors $\langle \mathtt{F}, \ldots, \mathtt{F} \rangle$, $\ldots$, $\langle \mathtt{T}, \ldots, \mathtt{T} \rangle$ with

Chapter 14.9.Exhaustive Search

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

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

- **for** loop can be implemented by encoding vectors $\langle \mathtt{F}, \ldots, \mathtt{F} \rangle$, $\ldots$, $\langle \mathtt{T}, \ldots, \mathtt{T} \rangle$ with binary numbers then

Chapter 14.9.Exhaustive Search

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

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

- **for** loop can be implemented by encoding vectors $\langle \mathtt{F}, \ldots, \mathtt{F} \rangle$, $\ldots$, $\langle \mathtt{T}, \ldots, \mathtt{T} \rangle$ 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** $\langle x_1, \ldots, x_n \rangle = \langle \mathtt{F}, \ldots, \mathtt{F} \rangle$ **to** $\langle \mathtt{T}, \ldots, \mathtt{T} \rangle$
2. $V = Evaluate(f, x_1, \ldots, x_n)$
3. **if** $V = \mathtt{T}$, **return** $(\mathtt{T})$
4. **return** $(\mathtt{F})$

- **for** loop can be implemented by encoding vectors $\langle \mathtt{F}, \ldots, \mathtt{F} \rangle$, $\ldots$, $\langle \mathtt{T}, \ldots, \mathtt{T} \rangle$ with binary numbers then further with integers
- a decoding process is needed to converting integers back to vectors

Chapter 14.9.Exhaustive Search

Chapter 14.9.Exhaustive Search

Iterative exhaustive search seems to be more convenient

Chapter 14.9.Exhaustive Search

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.)

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?

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 $(12 \ldots n)$

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 $(12 \ldots n)$
- how to encode these permutations as integers ?

Chapter 14.9.Exhaustive Search

Exhaustive search could be non-trivial

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

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

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.

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

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

- non-trivial: use a search tree, achieving a better time upper bound.

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

- 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

Chapter 14.9.Exhaustive Search

The algorithm follows a logical search tree

- given a graph G, it picks an arbitrary vertex v from G;

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:

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;

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.

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

Algorithm MaxIndSet (G)

Chapter 14.9.Exhaustive Search

Algorithm MaxIndSet (G)

1. **if** $G = \emptyset$ **return** $(\emptyset)$

Chapter 14.9.Exhaustive Search

Algorithm MaxIndSet (G)

1. **if** $G = \emptyset$ **return** ($\emptyset$)
2. **else** pick an arbitrary vertex v in G

Chapter 14.9.Exhaustive Search

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

Chapter 14.9.Exhaustive Search

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. $I_1 = \{v\} \cup$ MaxIndSet (G_1)

Chapter 14.9.Exhaustive Search

Algorithm MaxIndSet (G)

```
if  G = ∅ return (∅)
else pick an arbitrary vertex v in G
        let G_1 be G with v and all its neighbors removed
        I_1 = {v} ∪ MaxIndSet (G_1)
        let G_2 be G with v removed
```

Chapter 14.9.Exhaustive Search

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

```
if  G = ∅ return (∅)
else pick an arbitrary vertex v in G
        let G_1 be G with v and all its neighbors removed
        I_1 = {v} ∪ MaxIndSet (G_1)
        let G_2 be G with v removed
        I_2 = MaxIndSet (G_2)
```

Chapter 14.9.Exhaustive Search

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

```
if  G = ∅ return (∅)
else pick an arbitrary vertex v in G
        let G_1 be G with v and all its neighbors removed
        I_1 = {v} ∪ MaxIndSet (G_1)
        let G_2 be G with v removed
        I_2 = MaxIndSet (G_2)
        if |I_1| ≥ |I_2| return (I_1)
```

Chapter 14.9.Exhaustive Search

Algorithm MaxIndSet (G)

```
if  G = ∅ return (∅)
else pick an arbitrary vertex v in G
        let G_1 be G with v and all its neighbors removed
        I_1 = {v} ∪ MaxIndSet (G_1)
        let G_2 be G with v removed
        I_2 = MaxIndSet (G_2)
        if |I_1| ≥ |I_2| return (I_1)
        else return (I_2)
```

Chapter 14.9.Exhaustive Search

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

```
if  G = ∅ return (∅)
else pick an arbitrary vertex v in G
        let G_1 be G with v and all its neighbors removed
        I_1 = {v} ∪ MaxIndSet (G_1)
        let G_2 be G with v removed
        I_2 = MaxIndSet (G_2)
        if |I_1| ≥ |I_2| return (I_1)
        else return (I_2)
```

- the algorithm is a search tree

Chapter 14.9.Exhaustive Search

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

```
if  G = ∅ return (∅)
else pick an arbitrary vertex v in G
        let G_1 be G with v and all its neighbors removed
        I_1 = {v} ∪ MaxIndSet (G_1)
        let G_2 be G with v removed
        I_2 = MaxIndSet (G_2)
        if |I_1| ≥ |I_2| return (I_1)
        else return (I_2)
```

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

Chapter 14.9.Exhaustive Search

Algorithm MaxIndSet (G)

```
if  G = ∅ return (∅)
else pick an arbitrary vertex v in G
        let G_1 be G with v and all its neighbors removed
        I_1 = {v} ∪ MaxIndSet (G_1)
        let G_2 be G with v removed
        I_2 = MaxIndSet (G_2)
        if |I_1| ≥ |I_2| return (I_1)
        else return (I_2)
```

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

Chapter 14.9.Exhaustive Search

Algorithm MaxIndSet (G)

```
if  G = ∅ return (∅)
else pick an arbitrary vertex v in G
        let G_1 be G with v and all its neighbors removed
        I_1 = {v} ∪ MaxIndSet (G_1)
        let G_2 be G with v removed
        I_2 = MaxIndSet (G_2)
        if |I_1| ≥ |I_2| return (I_1)
        else 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

Algorithm MaxIndSet (G)

```
if  G = ∅ return (∅)
else pick an arbitrary vertex v in G
        let G_1 be G with v and all its neighbors removed
        I_1 = {v} ∪ MaxIndSet (G_1)
        let G_2 be G with v removed
        I_2 = MaxIndSet (G_2)
        if |I_1| ≥ |I_2| return (I_1)
        else 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,
how to guarantee a large m?

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

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

- $m \geq 0$, $T(n) \leq T(n-1) + T(n-1) + cn^2$, $\Longrightarrow$ $T(n) = O(2^n)$
- how to guarantee $m \geq 1$?

Chapter 14.9.Exhaustive Search

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

- $m \geq 0$, $T(n) \leq T(n-1) + T(n-1) + cn^2$, $\Longrightarrow$ $T(n) = O(2^n)$
- how to guarantee $m \geq 1$? if we can,
  $T(n) \leq T(n-2) + T(n-1) + cn^2$,

Chapter 14.9.Exhaustive Search

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

- $m \geq 0$, $T(n) \leq T(n-1) + T(n-1) + cn^2$, $\Longrightarrow$ $T(n) = O(2^n)$
- how to guarantee $m \geq 1$? if we can,
  $T(n) \leq T(n-2) + T(n-1) + cn^2$, $\Longrightarrow$ $T(n) = O(1.6181^n)$

Chapter 14.9.Exhaustive Search

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

- $m \geq 0$, $T(n) \leq T(n-1) + T(n-1) + cn^2$, $\Longrightarrow$ $T(n) = O(2^n)$
- how to guarantee $m \geq 1$? if we can,
  $T(n) \leq T(n-2) + T(n-1) + cn^2$, $\Longrightarrow$ $T(n) = O(1.6181^n)$
- how to guarantee $m \geq 2$?

Chapter 14.9.Exhaustive Search

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

- $m \geq 0$, $T(n) \leq T(n-1) + T(n-1) + cn^2$, $\Longrightarrow$ $T(n) = O(2^n)$
- how to guarantee $m \geq 1$? if we can,
  $T(n) \leq T(n-2) + T(n-1) + cn^2$, $\Longrightarrow$ $T(n) = O(1.6181^n)$
- how to guarantee $m \geq 2$? if we can,
  $T(n) \leq T(n-3) + T(n-1) + cn^2$,

Chapter 14.9.Exhaustive Search

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

- $m \geq 0$, $T(n) \leq T(n-1) + T(n-1) + cn^2$, $\Longrightarrow$ $T(n) = O(2^n)$
- how to guarantee $m \geq 1$? if we can,
  $T(n) \leq T(n-2) + T(n-1) + cn^2$, $\Longrightarrow$ $T(n) = O(1.6181^n)$
- how to guarantee $m \geq 2$? if we can,
  $T(n) \leq T(n-3) + T(n-1) + cn^2$, $\Longrightarrow$ $T(n) = O(1.5^n)$

Chapter 14.9.Exhaustive Search

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

- $m \geq 0$, $T(n) \leq T(n-1) + T(n-1) + cn^2$, $\Longrightarrow$ $T(n) = O(2^n)$
- how to guarantee $m \geq 1$? if we can,
  $T(n) \leq T(n-2) + T(n-1) + cn^2$, $\Longrightarrow$ $T(n) = O(1.6181^n)$
- how to guarantee $m \geq 2$? if we can,
  $T(n) \leq T(n-3) + T(n-1) + cn^2$, $\Longrightarrow$ $T(n) = O(1.5^n)$

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

Chapter 14.9.Exhaustive Search

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

- $m \geq 0$, $T(n) \leq T(n-1) + T(n-1) + cn^2$, $\Longrightarrow$ $T(n) = O(2^n)$
- how to guarantee $m \geq 1$? if we can,
  $T(n) \leq T(n-2) + T(n-1) + cn^2$, $\Longrightarrow$ $T(n) = O(1.6181^n)$
- how to guarantee $m \geq 2$? if we can,
  $T(n) \leq T(n-3) + T(n-1) + cn^2$, $\Longrightarrow$ $T(n) = O(1.5^n)$

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

  - can we guarantee $m \geq 3$?

Chapter 14.9.Exhaustive Search

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

- $m \geq 0$, $T(n) \leq T(n-1) + T(n-1) + cn^2$, $\Longrightarrow$ $T(n) = O(2^n)$
- how to guarantee $m \geq 1$? if we can,
  $T(n) \leq T(n-2) + T(n-1) + cn^2$, $\Longrightarrow$ $T(n) = O(1.6181^n)$
- how to guarantee $m \geq 2$? if we can,
  $T(n) \leq T(n-3) + T(n-1) + cn^2$, $\Longrightarrow$ $T(n) = O(1.5^n)$

  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

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

Chapter 14.9.Exhaustive Search

- Algorithms for SAT and $\textsc{MaxIndSet}$ run in exponential time $O(2^n)$ or $O(\gamma^n)$ for $1 < \gamma < 2$
- search tree (solution search space) is large, inherently large

Chapter 14.9.Exhaustive Search

- Algorithms for SAT and $\textsc{MaxIndSet}$ 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 $\textsc{MaxIndSet}$ 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

- we will deal with optimization problems

Chapter 15. Dynamic Programming

- we will deal with optimization problems

  the output solution satisfies certain desired optimality

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

Chapter 15. Dynamic Programming

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 ubproblems

Chapter 15. Dynamic Programming

Will discuss the following examples:

Chapter 15. Dynamic Programming

Will discuss the following examples:

- assembly-line scheduling

Chapter 15. Dynamic Programming

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 nth element of Fibonacci sequence

Chapter 15. Dynamic Programming

Revisit the problem of computing the nth 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 nth element of Fibonacci sequence

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

- it divides problems into two subproblems

Chapter 15. Dynamic Programming

Revisit the problem of computing the nth 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 nth 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

Chapter 15. Dynamic Programming

Revisit the problem of computing the nth 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

Chapter 15. Dynamic Programming

Revisit the problem of computing the nth 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

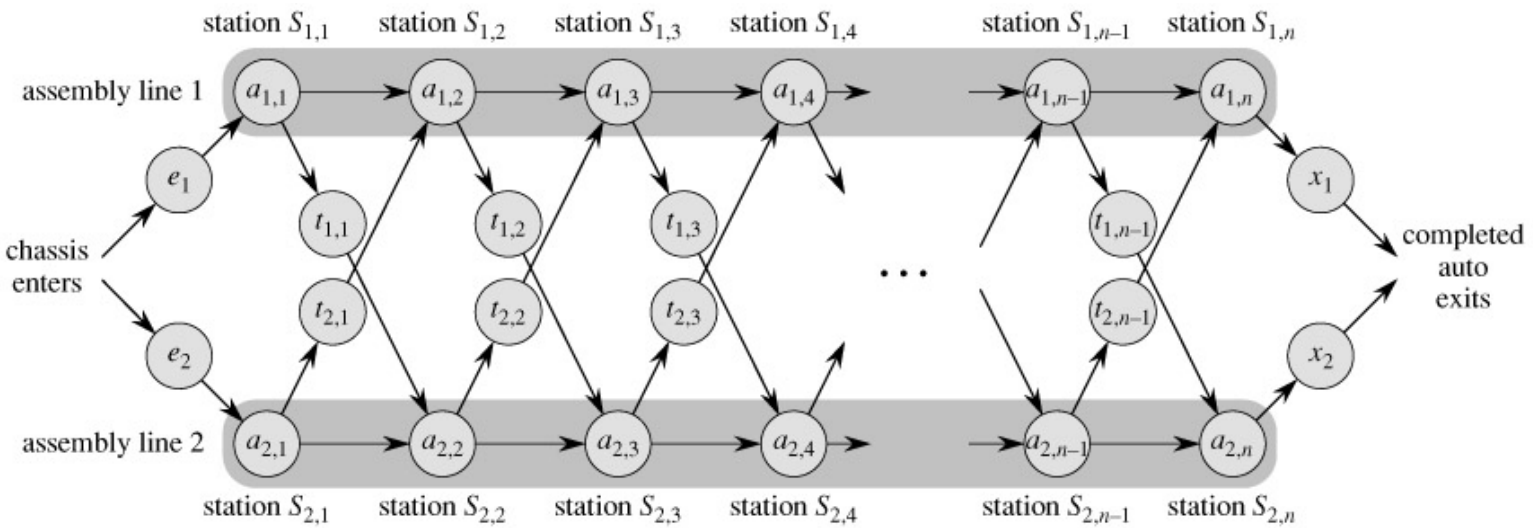
  compute table $T[1..n]$ from left to right,
  by looking up values that have already been computed

Chapter 15. Dynamic Programming

Assembly-line scheduling

Chapter 15. Dynamic Programming

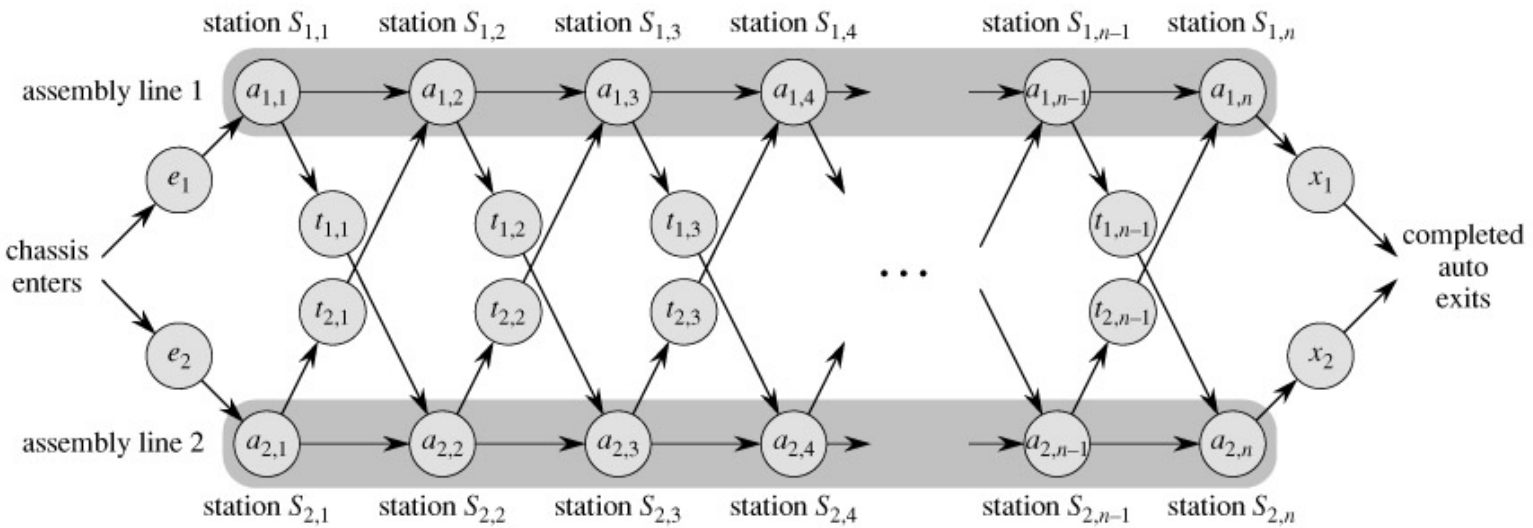
Assembly-line scheduling



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

Chapter 15. Dynamic Programming

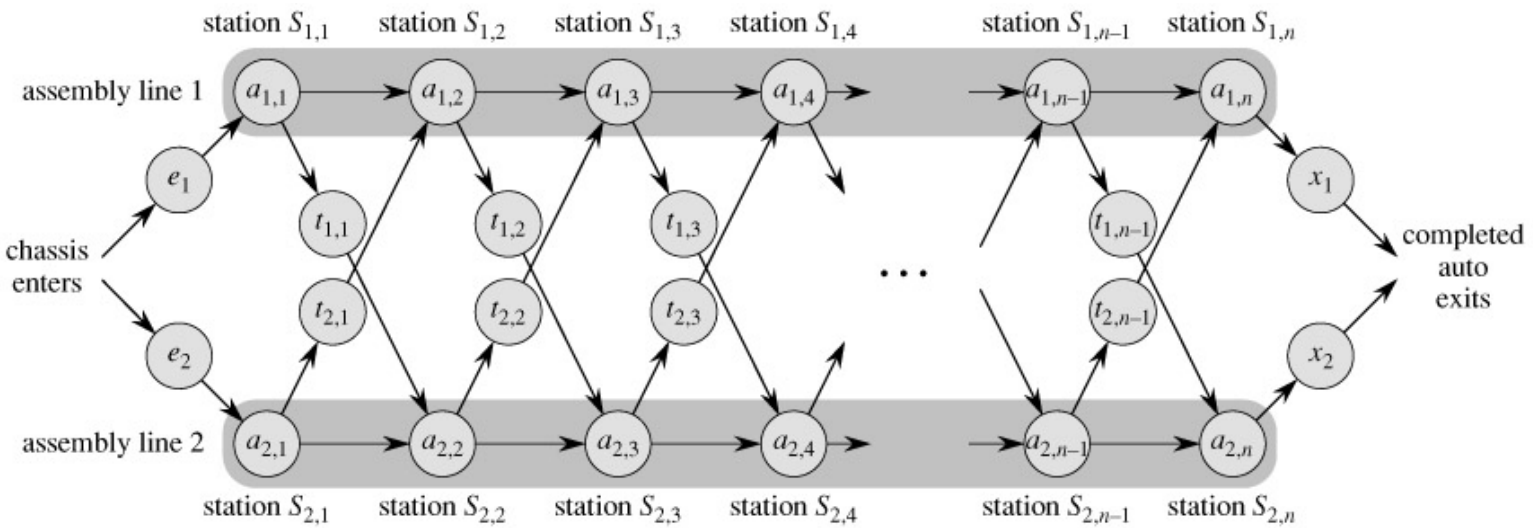
Assembly-line scheduling



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

Chapter 15. Dynamic Programming

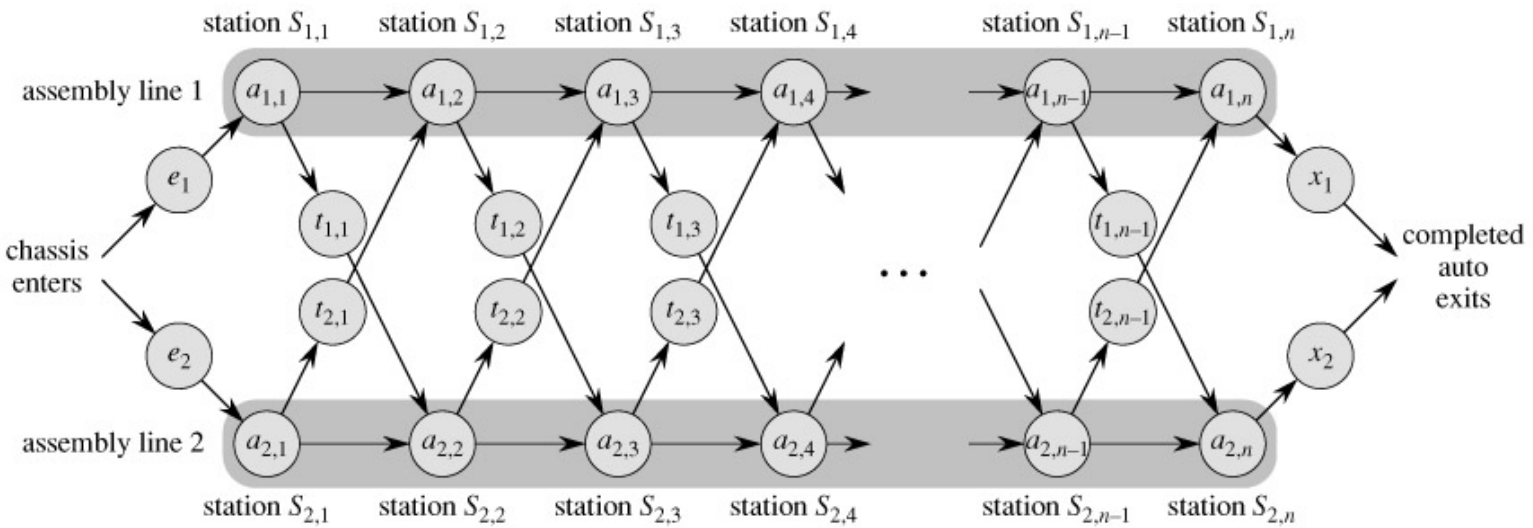
Assembly-line scheduling



- $S_{i,j}$: the jth 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 $S_{i,j}$

Chapter 15. Dynamic Programming

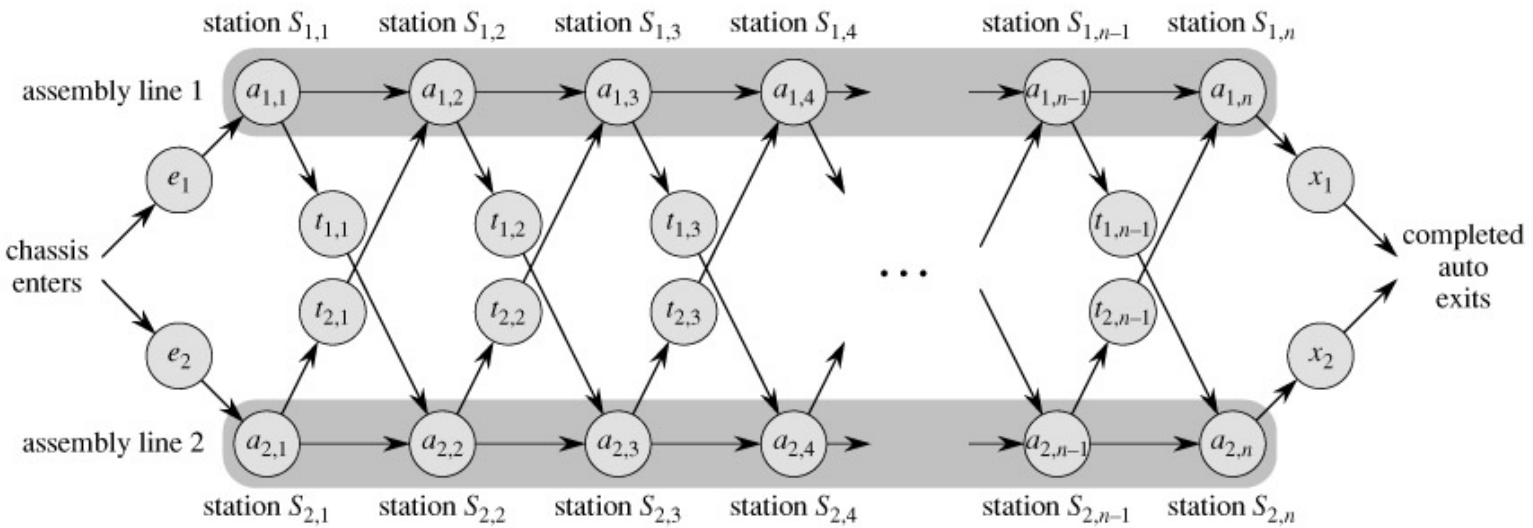
Assembly-line scheduling



- $S_{i,j}$: the jth 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

Chapter 15. Dynamic Programming

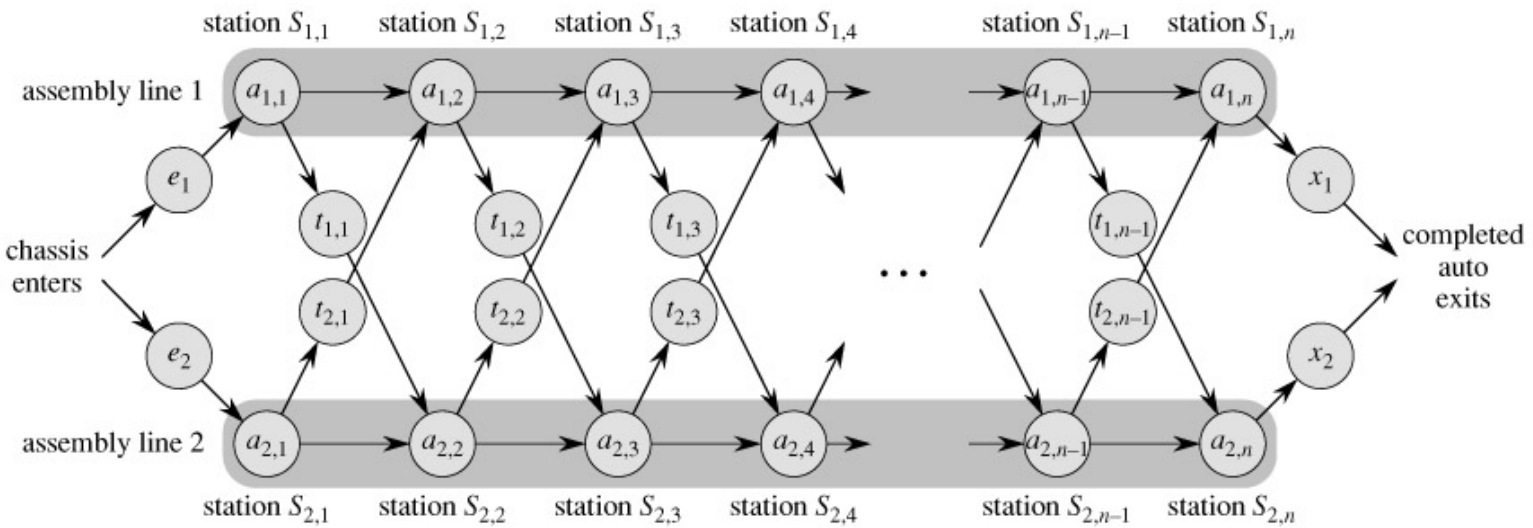
Assembly-line scheduling



- $S_{i,j}$: the jth 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
- x_1 and x_2 are exiting time costs

Chapter 15. Dynamic Programming

Assembly-line scheduling

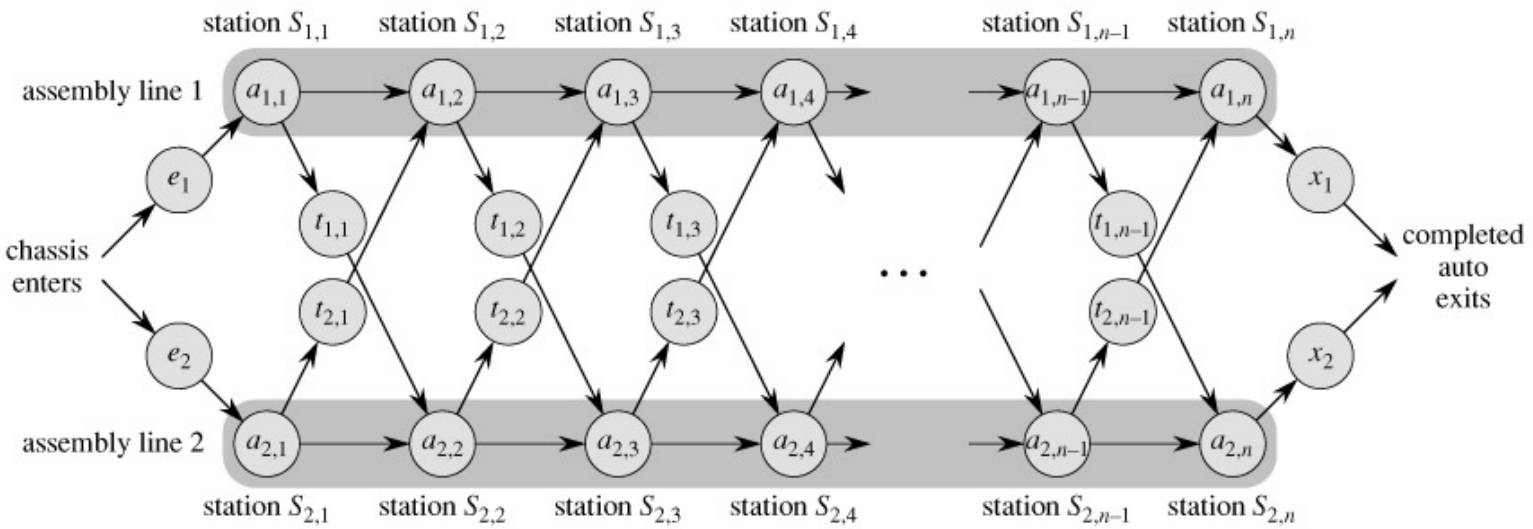


- $S_{i,j}$: the jth station on line i, $i = 1, 2$, $j = 1, 2, \cdots, n$.
- $a_{i,j}$ is the assembly time at station $S_{i,j}$
- $t_{i,j}$ is time for transferring lines from station $S_{i,j}$
- e_1 and e_2 are entering time costs
- x_1 and x_2 are exiting time costs

To determine the fastest way through a factory

Chapter 15. Dynamic Programming

Assembly-line scheduling



- $S_{i,j}$: the jth 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
- 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 general $j, 1 \leq j \leq n$, the fastest way through $S_{1,j}$ 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 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 $S_{1,j-1} \rightarrow S_{1,j}$,

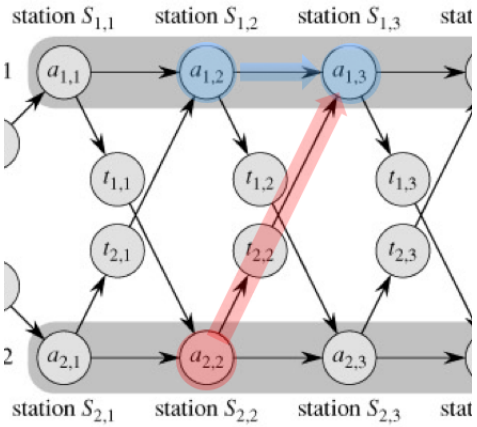
Chapter 15. Dynamic Programming

Analyzing the problem:

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

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 $S_{2,j-1} \rightarrow S_{1,j}$.



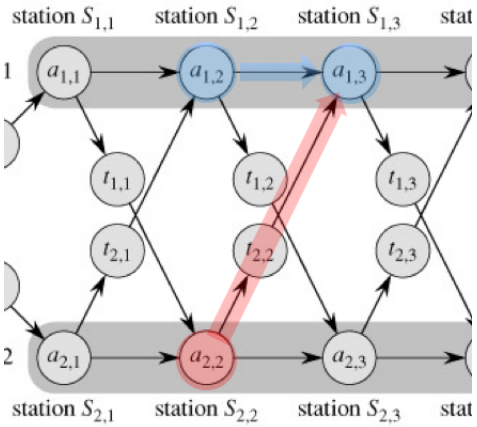
Chapter 15. Dynamic Programming

Analyzing the problem:

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

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

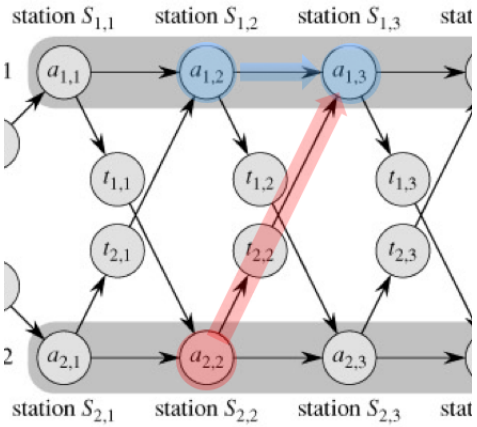
Chapter 15. Dynamic Programming

Analyzing the problem:

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

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

Take a dynamic programming approach:

Chapter 15. Dynamic Programming

Take a dynamic programming approach:

step 1: the above analysis

Chapter 15. Dynamic Programming

Take a dynamic programming approach:

step 1: the above analysis

step 2: formulate a recursive solution to the problem

Chapter 15. Dynamic Programming

Take 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 $j = 1, \cdots, n$.

Chapter 15. Dynamic Programming

Take 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 $\min\{f_1(n) + x_1, f_2(n) + x_2\}$

Chapter 15. Dynamic Programming

Take 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:

Chapter 15. Dynamic Programming

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

Chapter 15. Dynamic Programming

Take 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

Take 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 $f_2(1) = e_2 + a_{2,1}$

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

$f_1(1)$	$f_1(2)$	$f_1(3)$	$\ldots$	$f_1(n)$
$f_2(1)$	$f_2(2)$	$f_2(3)$	$\ldots$	$f_2(n)$

- 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

$f_1(1)$	$f_1(2)$	$f_1(3)$	$\ldots$	$f_1(n)$
$f_2(1)$	$f_2(2)$	$f_2(3)$	$\ldots$	$f_2(n)$

- 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

$f_1(1)$	$f_1(2)$	$f_1(3)$	$\ldots$	$f_1(n)$
$f_2(1)$	$f_2(2)$	$f_2(3)$	$\ldots$	$f_2(n)$

- 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

$f_1(1)$	$f_1(2)$	$f_1(3)$	$\ldots$	$f_1(n)$
$f_2(1)$	$f_2(2)$	$f_2(3)$	$\ldots$	$f_2(n)$

- 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

$f_1(1)$	$f_1(2)$	$f_1(3)$	$\ldots$	$f_1(n)$
$f_2(1)$	$f_2(2)$	$f_2(3)$	$\ldots$	$f_2(n)$

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

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

$f_1(1)$	$f_1(2)$	$f_1(3)$	$\ldots$	$f_1(n)$
$f_2(1)$	$f_2(2)$	$f_2(3)$	$\ldots$	$f_2(n)$

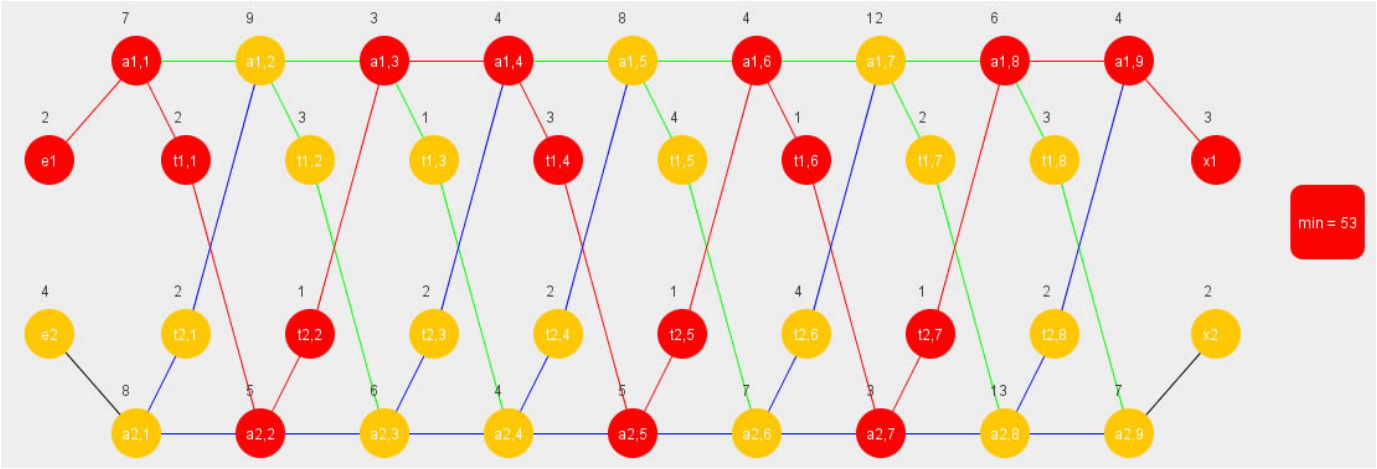
- using the recurrences to compute; and
- looking up already-computed values, where

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

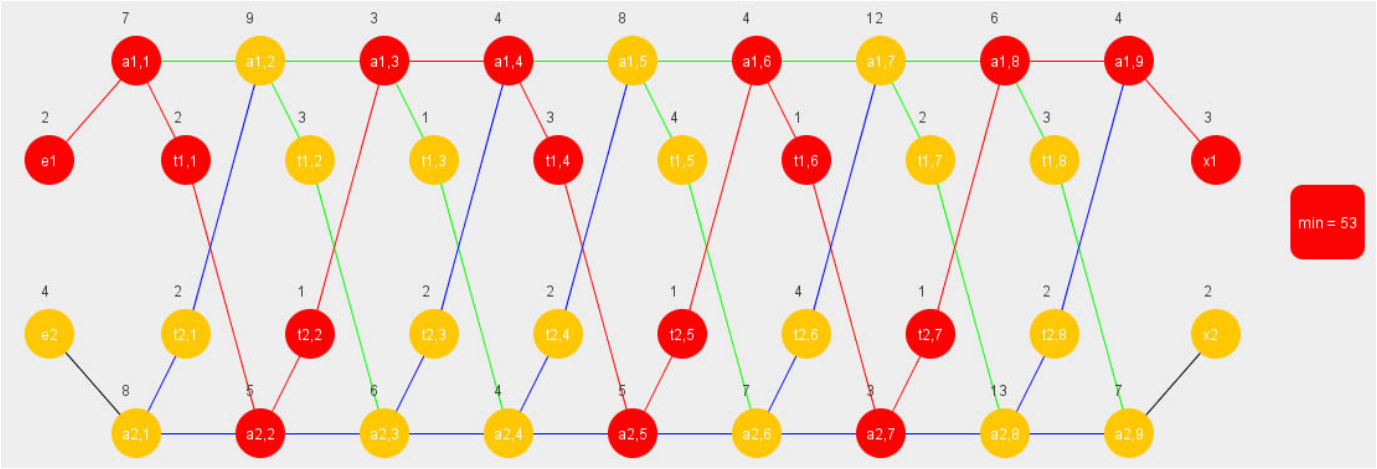
Chapter 15. Dynamic Programming

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



Chapter 15. Dynamic Programming

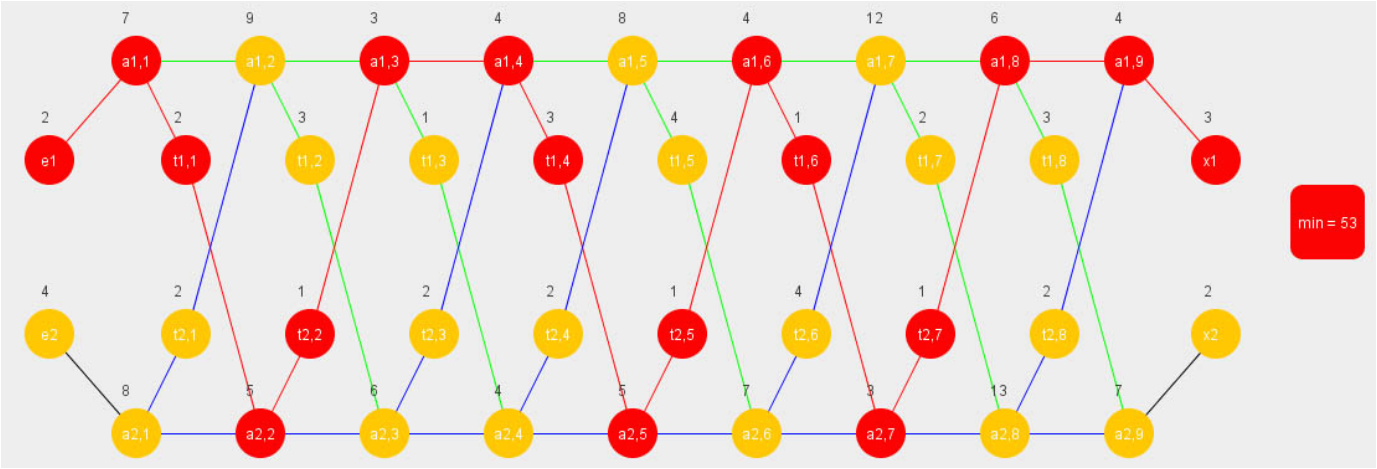
step 4: get the fastest path, not just the faster time.



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.

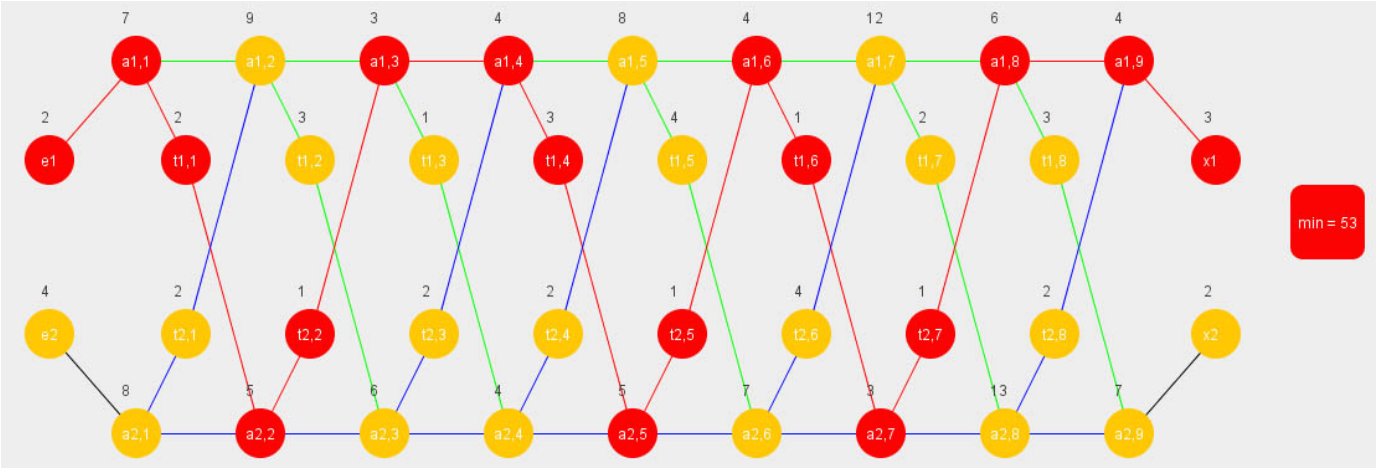


Do we know the fastest path from the fastest time f^*?

$f_1(1)$	$f_1(2)$	$f_1(3)$	$\ldots$	$f_1(n-1)$	$f_1(n)$
$f_2(1)$	$f_2(2)$	$f_2(3)$	$\ldots$	$f_2(n-1)$	$f_2(n)$

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^*?

$f_1(1)$	$f_1(2)$	$f_1(3)$	$\dots$	$f_1(n-1)$	$f_1(n)$
$f_2(1)$	$f_2(2)$	$f_2(3)$	$\dots$	$f_2(n-1)$	$f_2(n)$

$$
\begin{aligned}
f^* &= \min\{f_1(n) + x_1, f_2(n) + x_2\} \\
f_1(j) &= \min\{f_1(j-1) + a_{1,j}, f_2(j-1) + t_{2,j-1} + a_{1,j}\} \\
f_2(j) &= \min\{f_2(j-1) + a_{2,j}, f_1(j-1) + t_{1,j-1} + a_{2,j}\}
\end{aligned}
$$

Chapter 15. Dynamic Programming

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

Chapter 15. Dynamic Programming

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

1. $M[1,1] = e_1 + a_{1,1}$

Chapter 15. Dynamic Programming

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

1. $M[1,1] = e_1 + a_{1,1}$
2. $M[2,1] = e_2 + a_{2,1}$

Chapter 15. Dynamic Programming

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

```
M[1,1] = e_1 + a_{1,1}
M[2,1] = e_2 + a_{2,1}
for j=2 to n
```

Chapter 15. Dynamic Programming

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

```
M[1,1] = e_1 + a_{1,1}
M[2,1] = e_2 + a_{2,1}
for j=2 to n
    if M[1, j-1] + a_{1,j} ≤ M[2, j-1] + t_{2,j-1} + a_{1,j}
```

Chapter 15. Dynamic Programming

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

```
M[1,1] = e_1 + a_{1,1}
M[2,1] = e_2 + a_{2,1}
for j=2 to n
    if M[1,j-1] + a_{1,j} ≤ M[2,j-1] + t_{2,j-1} + a_{1,j}
        M[1,j] = M[1,j-1] + a_{1,j}
```

Chapter 15. Dynamic Programming

```
Algorithm AssemblyLine(a, t, e, x, n)
    M[1,1] = e_1 + a_{1,1}
    M[2,1] = e_2 + a_{2,1}
    for j=2 to n
        if M[1,j-1] + a_{1,j} ≤ M[2,j-1] + t_{2,j-1} + a_{1,j}
            M[1,j] = M[1,j-1] + a_{1,j}
            P[1,j] = 1
```

Chapter 15. Dynamic Programming

```
Algorithm AssemblyLine(a, t, e, x, n)
    M[1,1] = e_1 + a_{1,1}
    M[2,1] = e_2 + a_{2,1}
    for j=2 to n
        if M[1,j-1] + a_{1,j} ≤ M[2,j-1] + t_{2,j-1} + a_{1,j}
            M[1,j] = M[1,j-1] + a_{1,j}
            P[1,j] = 1
        else M[1,j] = M[2,j-1] + t_{2,j-1} + a_{1,j}
```

Chapter 15. Dynamic Programming

```
Algorithm AssemblyLine(a, t, e, x, n)
    M[1,1] = e_1 + a_{1,1}
    M[2,1] = e_2 + a_{2,1}
    for j=2 to n
        if M[1,j-1] + a_{1,j} ≤ M[2,j-1] + t_{2,j-1} + a_{1,j}
            M[1,j] = M[1,j-1] + a_{1,j}
            P[1,j] = 1
        else M[1,j] = M[2,j-1] + t_{2,j-1} + a_{1,j}
                P[1,j] = 2
```

Chapter 15. Dynamic Programming

```
Algorithm AssemblyLine(a, t, e, x, n)
    M[1,1] = e_1 + a_{1,1}
    M[2,1] = e_2 + a_{2,1}
    for j=2 to n
        if M[1,j-1] + a_{1,j} ≤ M[2,j-1] + t_{2,j-1} + a_{1,j}
            M[1,j] = M[1,j-1] + a_{1,j}
            P[1,j] = 1
        else M[1,j] = M[2,j-1] + t_{2,j-1} + a_{1,j}
                P[1,j] = 2
        if M[2,j-1] + a_{2,j} ≤ M[1,j-1] + t_{1,j-1} + a_{2,j}
```

Chapter 15. Dynamic Programming

```
Algorithm AssemblyLine(a, t, e, x, n)
    M[1,1] = e_1 + a_{1,1}
    M[2,1] = e_2 + a_{2,1}
    for j=2 to n
        if M[1,j-1] + a_{1,j} ≤ M[2,j-1] + t_{2,j-1} + a_{1,j}
            M[1,j] = M[1,j-1] + a_{1,j}
            P[1,j] = 1
        else M[1,j] = M[2,j-1] + t_{2,j-1} + a_{1,j}
                P[1,j] = 2
        if M[2,j-1] + a_{2,j} ≤ M[1,j-1] + t_{1,j-1} + a_{2,j}
            M[2,j] = M[2,j-1] + a_{2,j}
```

Chapter 15. Dynamic Programming

```
Algorithm AssemblyLine(a, t, e, x, n)
    M[1,1] = e_1 + a_{1,1}
    M[2,1] = e_2 + a_{2,1}
    for j=2 to n
        if M[1,j-1] + a_{1,j} ≤ M[2,j-1] + t_{2,j-1} + a_{1,j}
            M[1,j] = M[1,j-1] + a_{1,j}
            P[1,j] = 1
        else M[1,j] = M[2,j-1] + t_{2,j-1} + a_{1,j}
                P[1,j] = 2
        if M[2,j-1] + a_{2,j} ≤ M[1,j-1] + t_{1,j-1} + a_{2,j}
            M[2,j] = M[2,j-1] + a_{2,j}
            P[2,j] = 2
```

Chapter 15. Dynamic Programming

```
Algorithm AssemblyLine(a, t, e, x, n)
    M[1,1] = e_1 + a_{1,1}
    M[2,1] = e_2 + a_{2,1}
    for j=2 to n
        if M[1,j-1] + a_{1,j} ≤ M[2,j-1] + t_{2,j-1} + a_{1,j}
            M[1,j] = M[1,j-1] + a_{1,j}
            P[1,j] = 1
        else M[1,j] = M[2,j-1] + t_{2,j-1} + a_{1,j}
                P[1,j] = 2
        if M[2,j-1] + a_{2,j} ≤ M[1,j-1] + t_{1,j-1} + a_{2,j}
            M[2,j] = M[2,j-1] + a_{2,j}
            P[2,j] = 2
        else M[2,j] = M[1,j-1] + t_{1,j-1} + a_{2,j}
```

Chapter 15. Dynamic Programming

```
Algorithm AssemblyLine(a, t, e, x, n)
    M[1,1] = e_1 + a_{1,1}
    M[2,1] = e_2 + a_{2,1}
    for j=2 to n
        if M[1,j-1] + a_{1,j} ≤ M[2,j-1] + t_{2,j-1} + a_{1,j}
            M[1,j] = M[1,j-1] + a_{1,j}
            P[1,j] = 1
        else M[1,j] = M[2,j-1] + t_{2,j-1} + a_{1,j}
                P[1,j] = 2
        if M[2,j-1] + a_{2,j} ≤ M[1,j-1] + t_{1,j-1} + a_{2,j}
            M[2,j] = M[2,j-1] + a_{2,j}
            P[2,j] = 2
         else M[2,j] = M[1,j-1] + t_{1,j-1} + a_{2,j}
                P[2,j] = 1
```

Chapter 15. Dynamic Programming

```
Algorithm AssemblyLine(a, t, e, x, n)
    M[1,1] = e_1 + a_{1,1}
    M[2,1] = e_2 + a_{2,1}
    for j=2 to n
        if M[1,j-1] + a_{1,j} ≤ M[2,j-1] + t_{2,j-1} + a_{1,j}
            M[1,j] = M[1,j-1] + a_{1,j}
            P[1,j] = 1
        else M[1,j] = M[2,j-1] + t_{2,j-1} + a_{1,j}
                P[1,j] = 2
        if M[2,j-1] + a_{2,j} ≤ M[1,j-1] + t_{1,j-1} + a_{2,j}
            M[2,j] = M[2,j-1] + a_{2,j}
            P[2,j] = 2
        else M[2,j] = M[1,j-1] + t_{1,j-1} + a_{2,j}
                P[2,j] = 1
    return (min{f_1(n) + x_1, f_2(n) + x_2})
```

Algorithm $\textsc{PrintPathMain}(M, P, f^*, x)$

1. **if** $(f^* - x_1) = M[1, n]$

Algorithm $\textsc{PrintPathMain}(M, P, f^*, x)$

```
if (f* − x_1) = M[1, n]
    i = 1
```

Algorithm $\textsc{PrintPathMain}(M, P, f^*, x)$

```
if (f* − x_1) = M[1, n]
    i = 1
else i = 2
```

Algorithm $\textsc{PrintPathMain}(M, P, f^*, x)$

```
if (f* − x_1) = M[1, n]
    i = 1
else i = 2
PrintPath (P, i, n)
```

Algorithm PRINTPATHMAIN(M, P, f^*, x)

1. **if** $(f^* - x_1) = M[1, n]$
2. $\quad i = 1$
3. **else** $i = 2$
4. PRINTPATH (P, i, n)

Algorithm PRINTPATH (P, i, j)

1. **if** $j \geq 1$

Algorithm PRINTPATHMAIN(M, P, f^*, x)

1. **if** $(f^* - x_1) = M[1, n]$
2. $\quad i = 1$
3. **else** $i = 2$
4. PRINTPATH (P, i, n)

Algorithm PRINTPATH (P, i, j)

1. **if** $j \geq 1$
2. $\quad$ PRINTPATH $(P, P[i, j], j - 1)$

Algorithm PRINTPATHMAIN(M, P, f^*, x)

```
if (f* − x_1) = M[1, n]
    i = 1
else i = 2
PRINTPATH (P, i, n)
```

Algorithm PRINTPATH (P, i, j)

```
if j ≥ 1
    PRINTPATH (P, P[i, j], j − 1)
    print (i, j)
```

Chapter 15. Dynamic Programming

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

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

(3) computing optimal cost (bottom-up approach)

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

(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.

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 $k = 1, 2, 3$.

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 ?

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!

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?

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

Chapter 15. Dynamic Programming

Example-2. Casino dice and decoding algorithm

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

Chapter 15. Dynamic Programming

Example-2. Casino dice and decoding algorithm

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

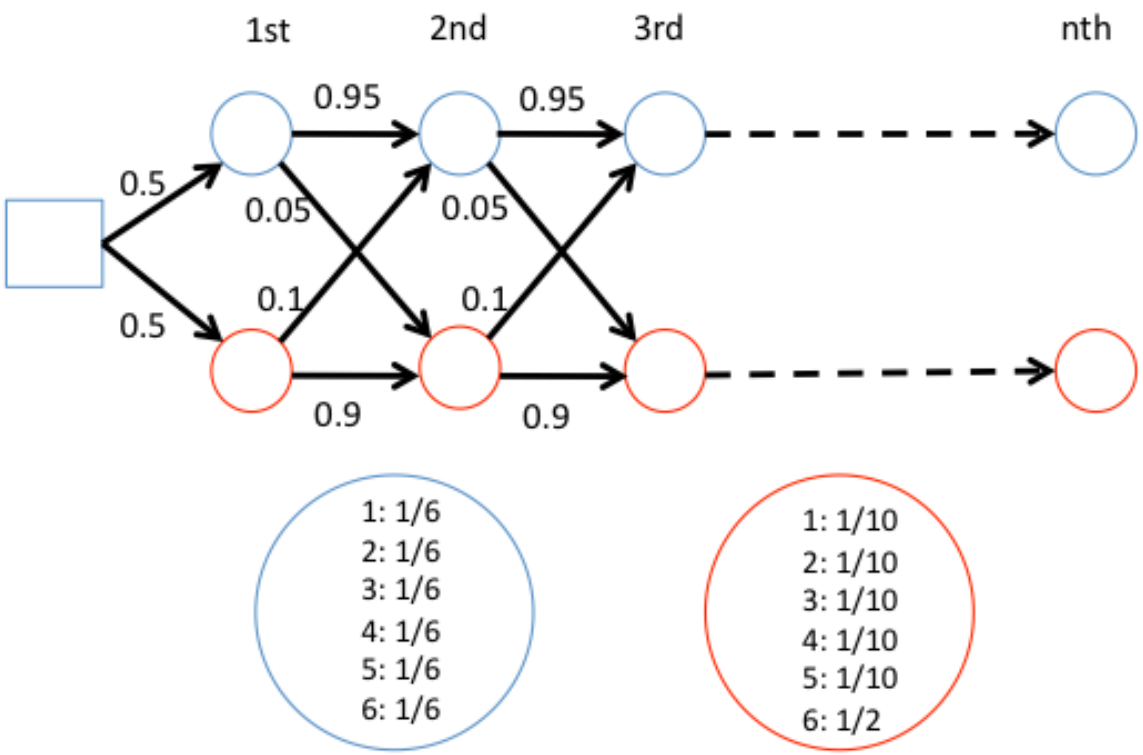
- not honest, switch between dice: fair (F) and loaded (L)

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

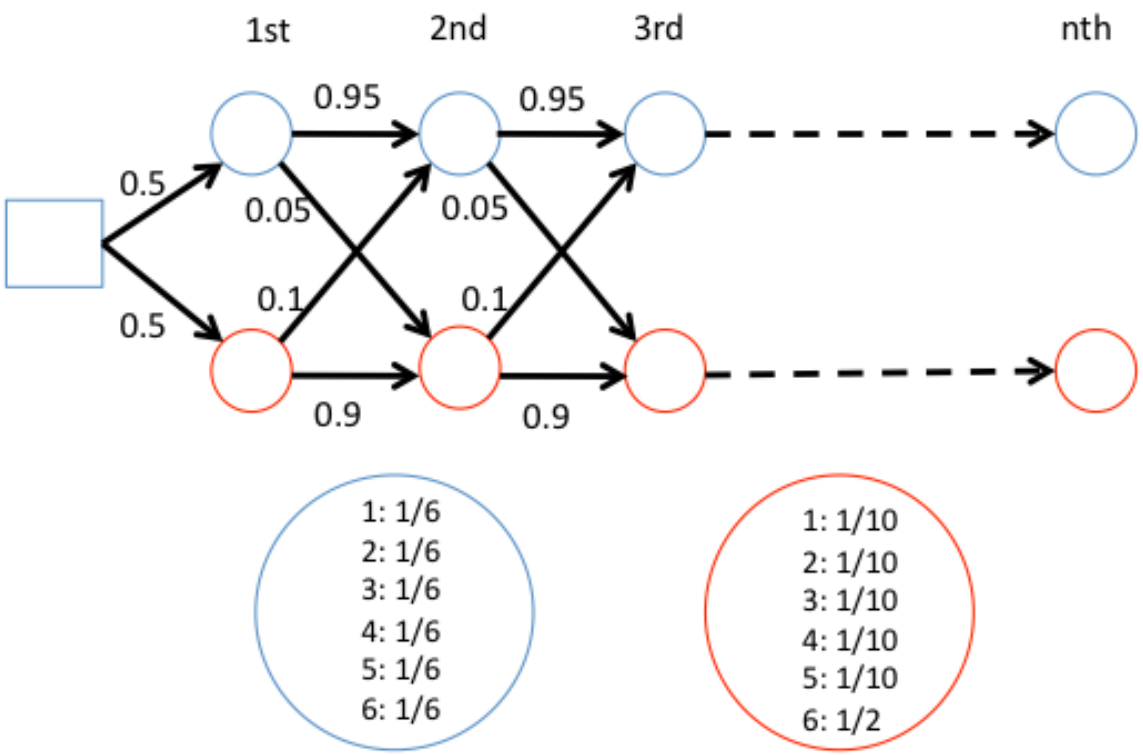


Chapter 15. Dynamic Programming

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



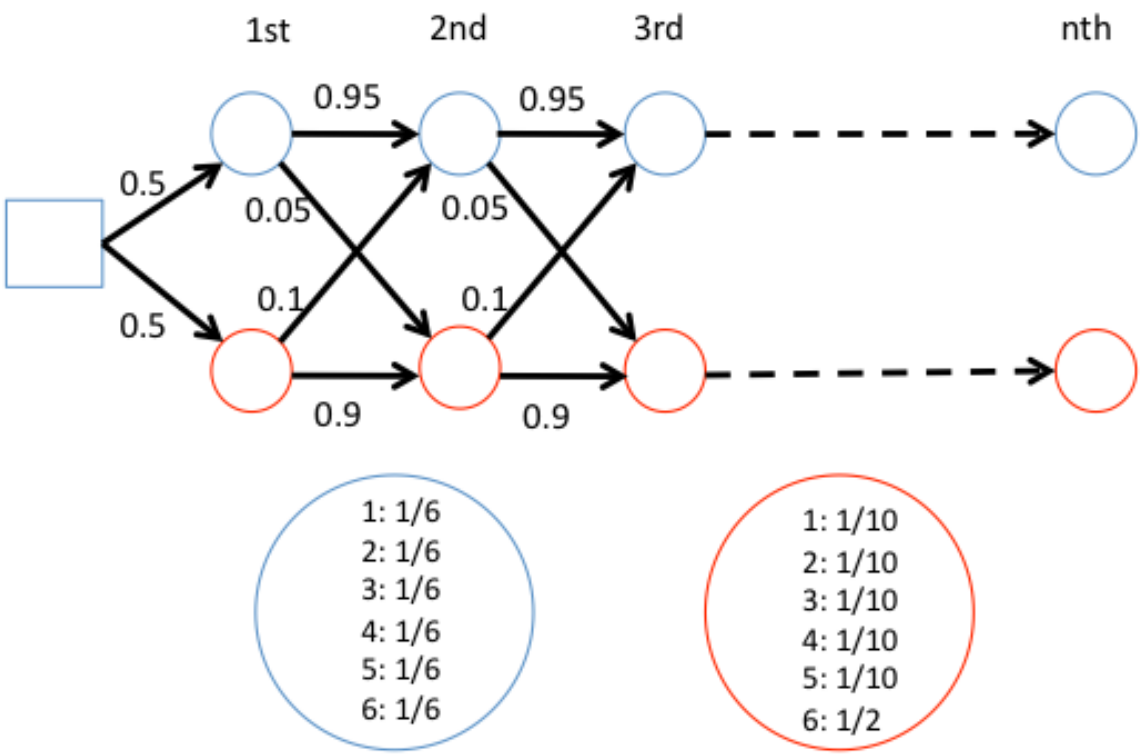
Given the sequence of numbers, e.g., $5, 3, 2, 5, 6, 6, 1, \ldots, 1$, what is the sequence of dices used

Chapter 15. Dynamic Programming

Example-2. Casino dice and decoding algorithm

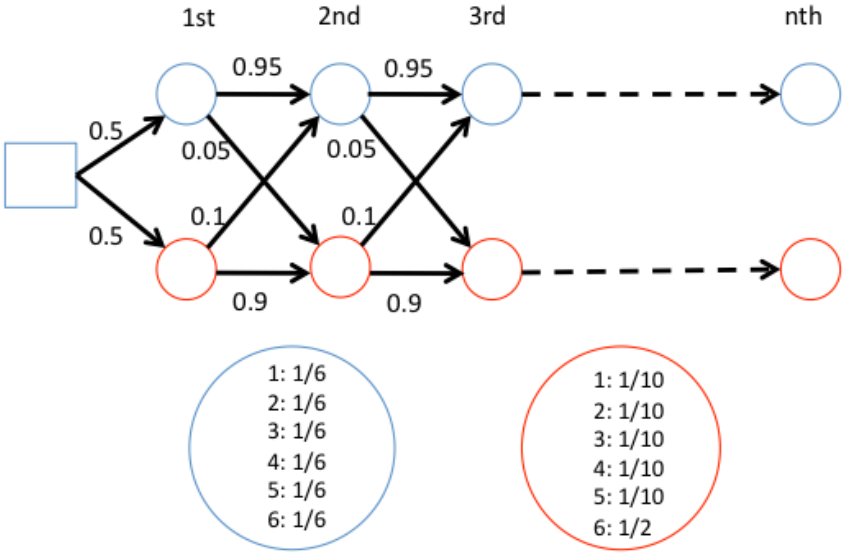
The house rolls die n times: e.g., $5, 3, 2, 5, 6, 6, 1, \ldots, 1$

- 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



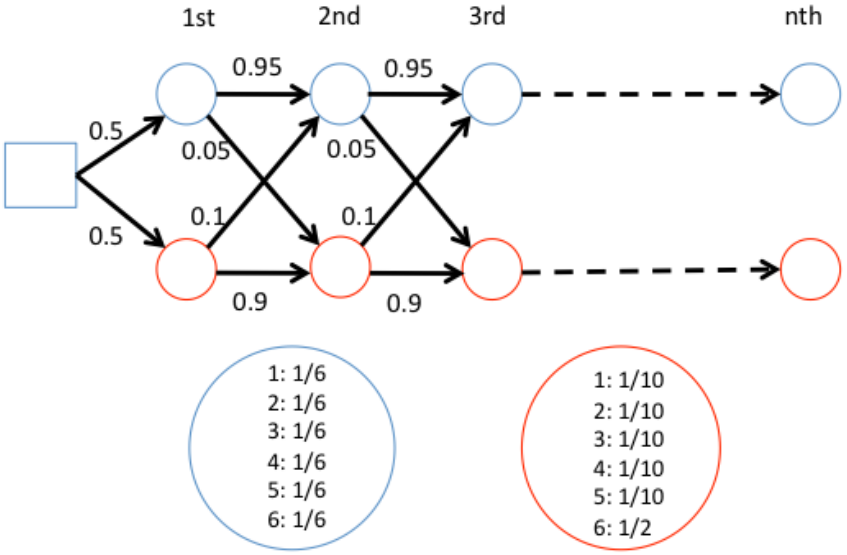
Given the sequence of numbers, e.g., $5, 3, 2, 5, 6, 6, 1, \ldots, 1$, what is the sequence of dices used **most likely**?

Chapter 15. Dynamic Programming



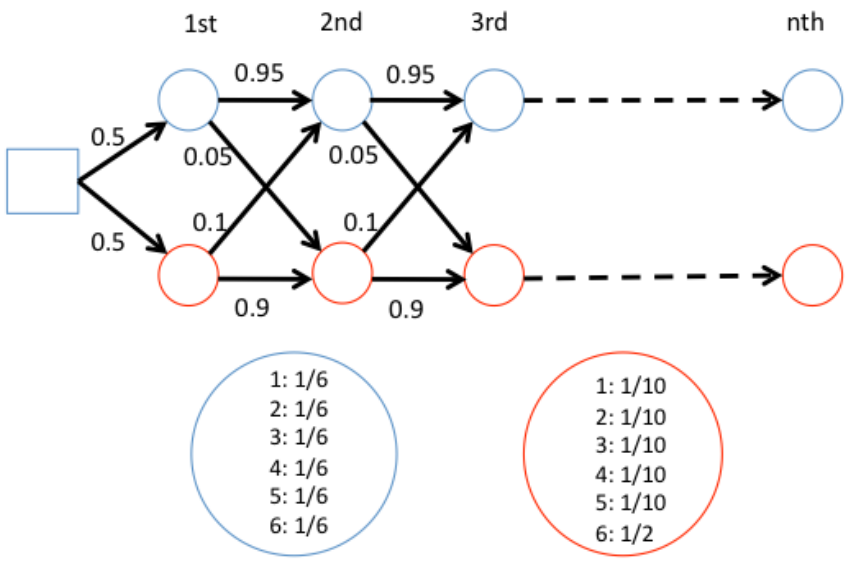
We compute the most probable path of dices to roll a given sequence of numbers $S = d_1 \dots d_n$, where $d_i \in \{1, 2, \dots, 6\}$

Chapter 15. Dynamic Programming



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

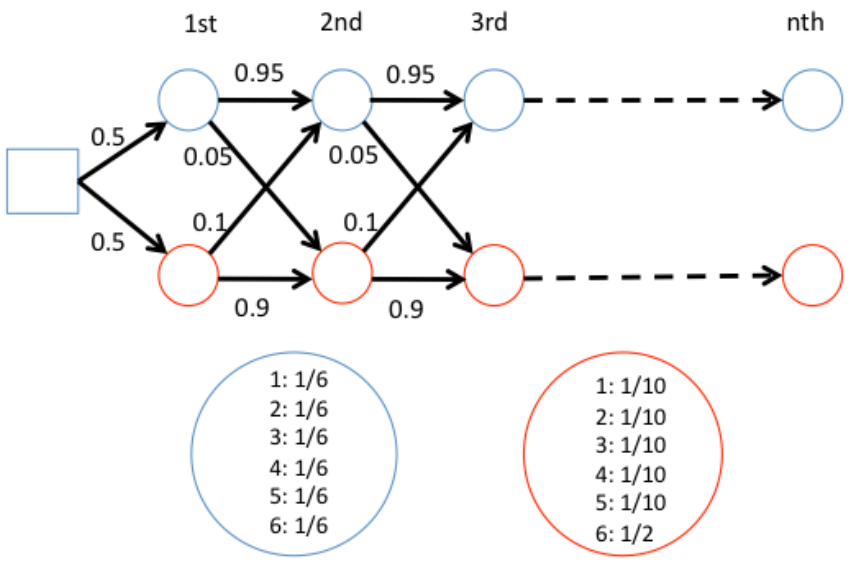
Chapter 15. Dynamic Programming



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

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

Chapter 15. Dynamic Programming

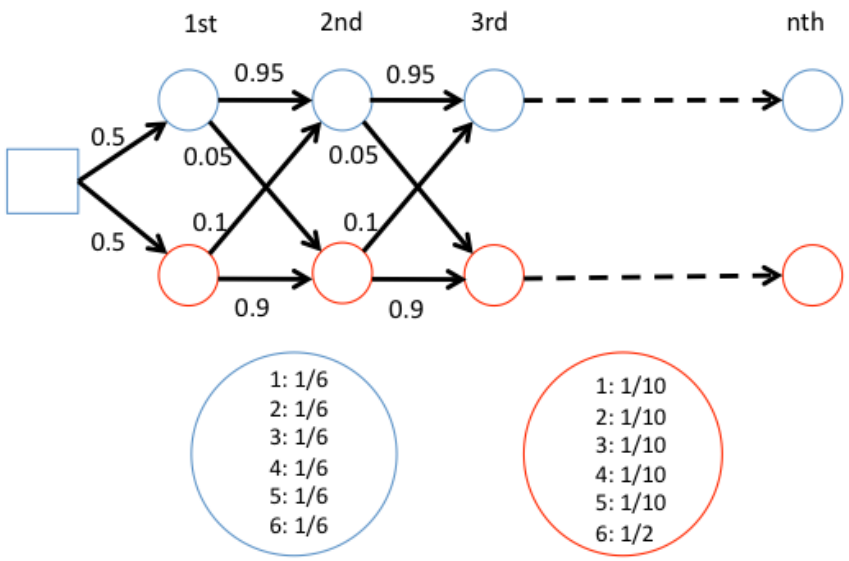


We compute the most probable path of dices to roll a given sequence of numbers $S = d_1 \dots d_n$, where $d_i \in \{1, 2, \dots, 6\}$

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

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

Chapter 15. Dynamic Programming



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

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

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

- $p(S, i, F)$ 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 $d_i = 6$

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 $i =$

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 $i = 1$

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 $i = 1$

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

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

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 $d_1 = 6$

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 $d_1 d_2 \ldots d_n$

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 $d_1 d_2 \ldots d_n$
- 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...

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...

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

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

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

... 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 L F F F F F F F...

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

... 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 L F F F F F F F...

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

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

... 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 L F F F F F F F...

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

The answer is definitely YES!

Chapter 15. Dynamic Programming

```
AGGACCATAAAACTCCAGTCAGTGAAC
AAACAAGTTAATAAACTAAAACTTTCA
TGGTTCTGGCATCGATGAAGAACGCAG
GTAATGTGAATTGCAGAATTCAGTGAA
GAACGCACATTGCGCCCCTTGGTATTC
TGTTCGAGCGTCATTTCAACCCTCAAG
TGGGCTCCGTCCTCCACGGACGCGCCT
GGTGGCGTCTTGCCTCAAGCGTAGTAG
TTGGAGCGCACGGCGTCGCCCGCCGGA
TATTTCTCAAGGTTGACCTCGGATCAT
AAGGTAAGAAAGTTTTTCCTTCCGCTG
CTGGGTGCTGGGTGCTGGGTGCTGGGT
TTGCCTTATCGCTTCGGTGAGGGGCAT
TTGGCCCGCGCTAAGCCTCGTTCGGGC
CGCATCTGGTTTTTTTGCGACCGGCGT
```

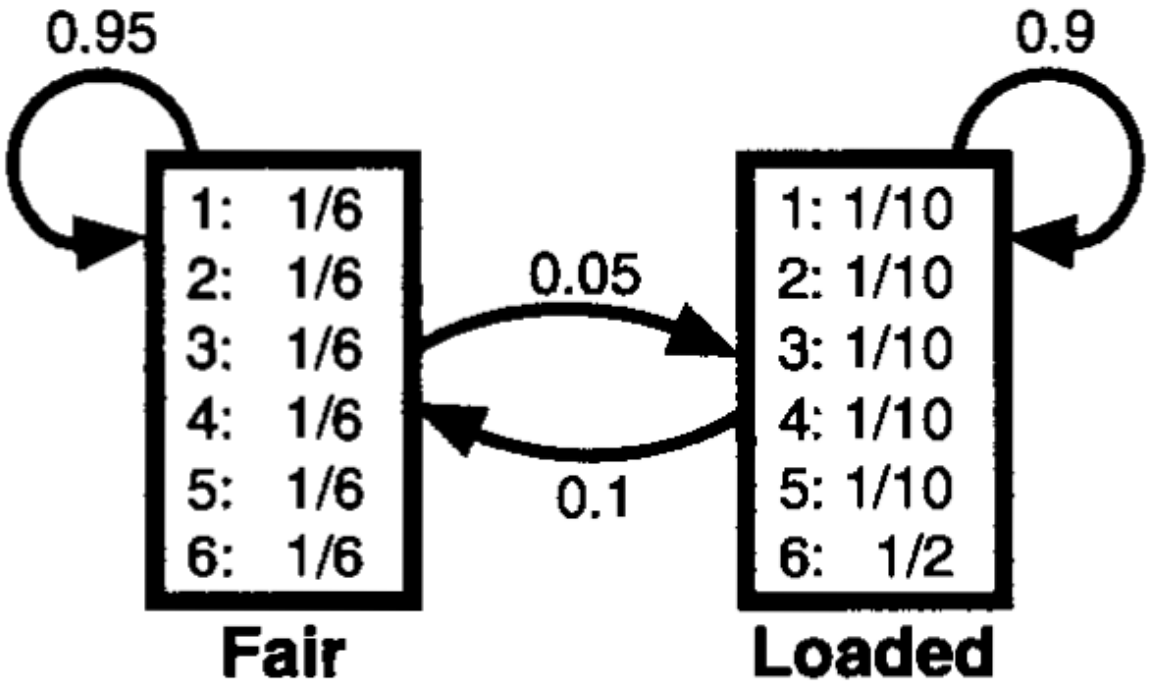
a segment of DNA sequence, is it meaningful?

Chapter 15. Dynamic Programming

if the highlighted segment is not random, it may be a gene.

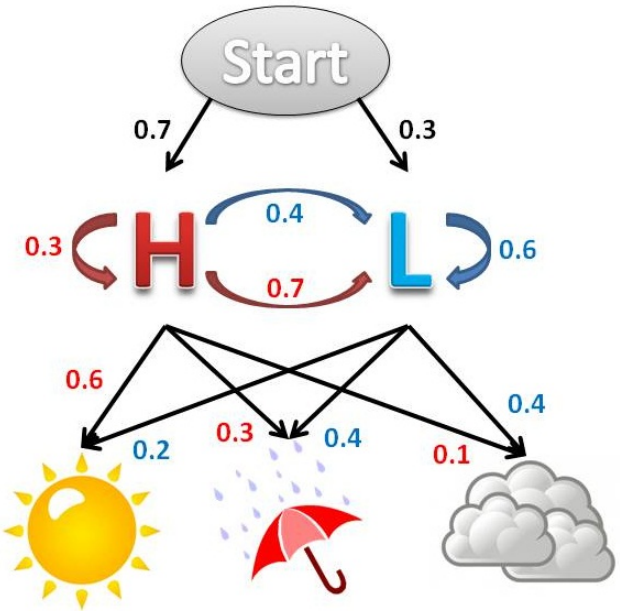
Chapter 15. Dynamic Programming

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

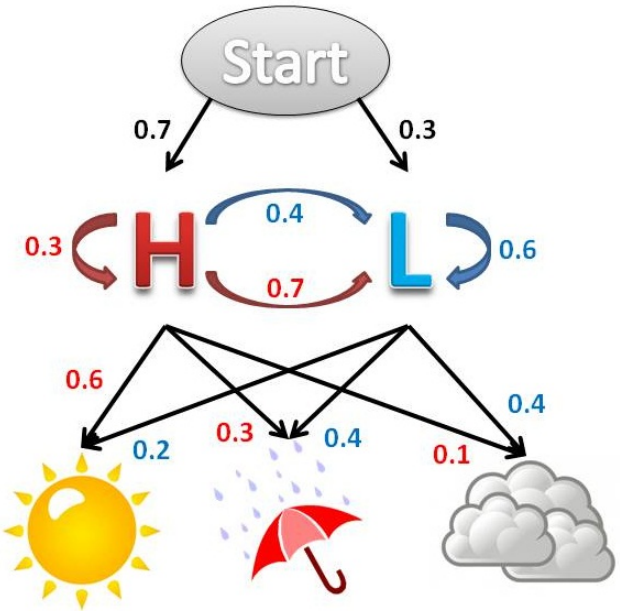


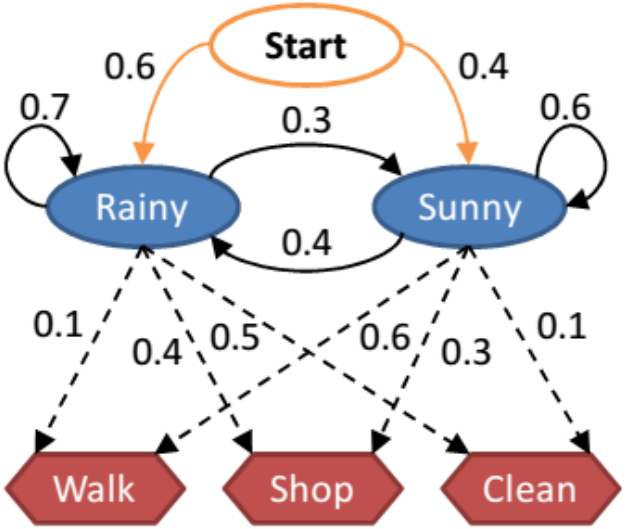
A hidden Markov model (HMM)

Chapter 15. Dynamic Programming

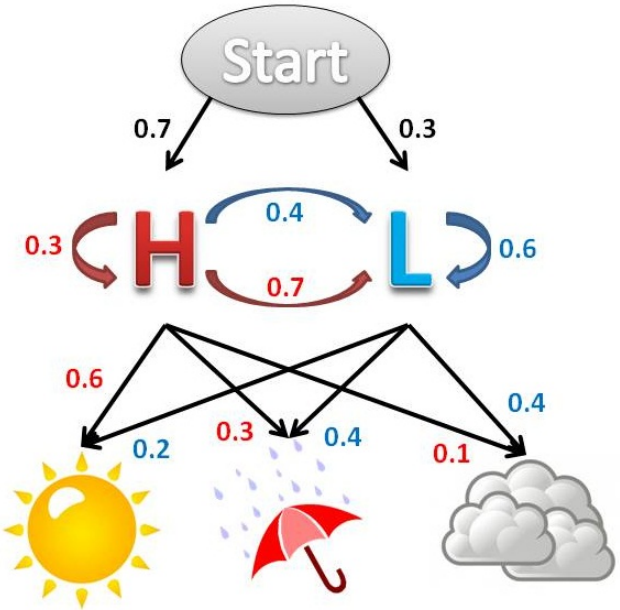


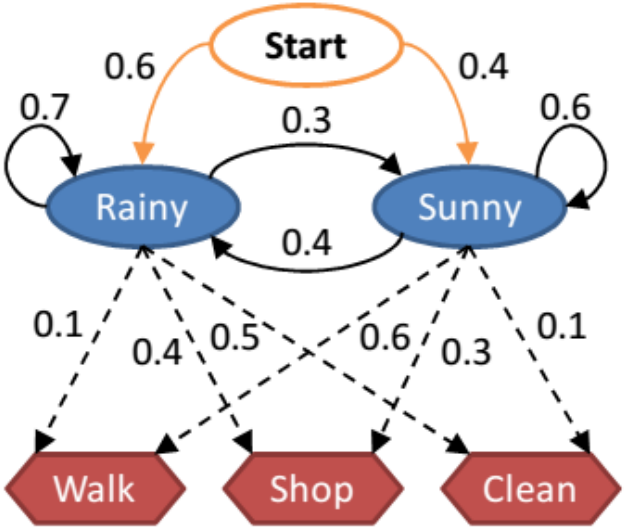
Chapter 15. Dynamic Programming

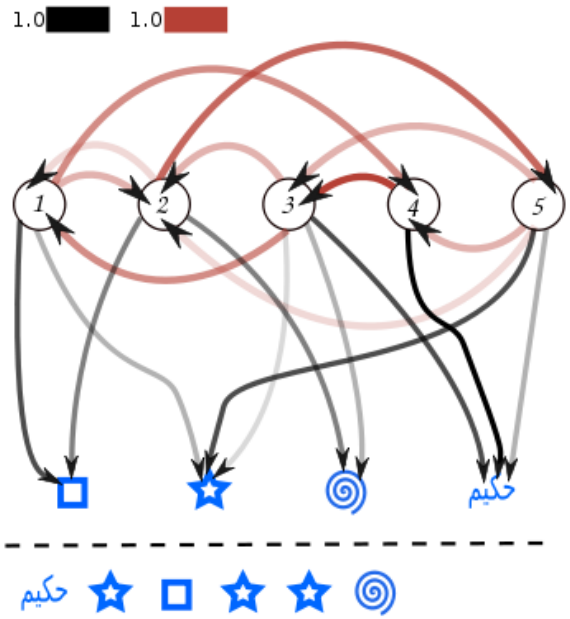




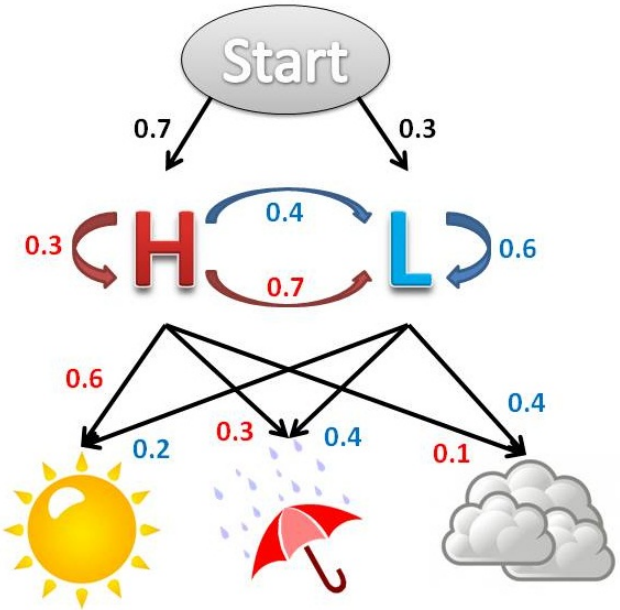
Chapter 15. Dynamic Programming

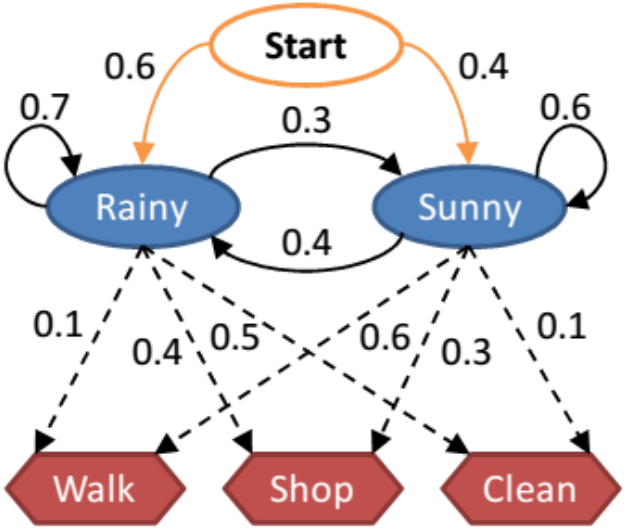


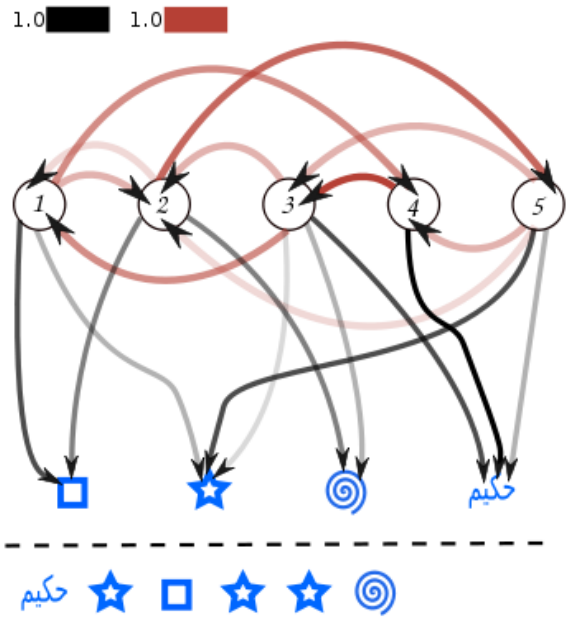


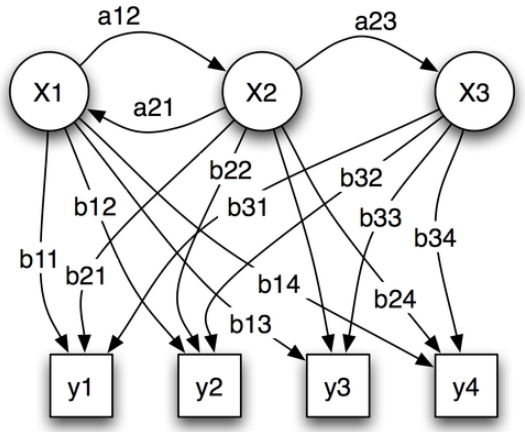


Chapter 15. Dynamic Programming



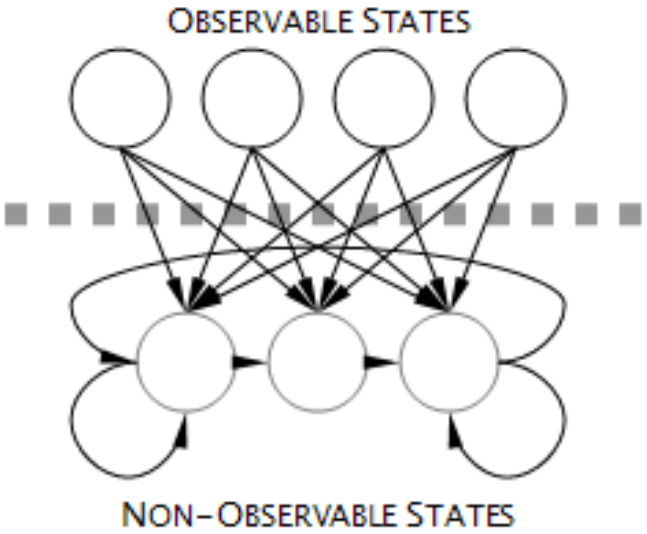






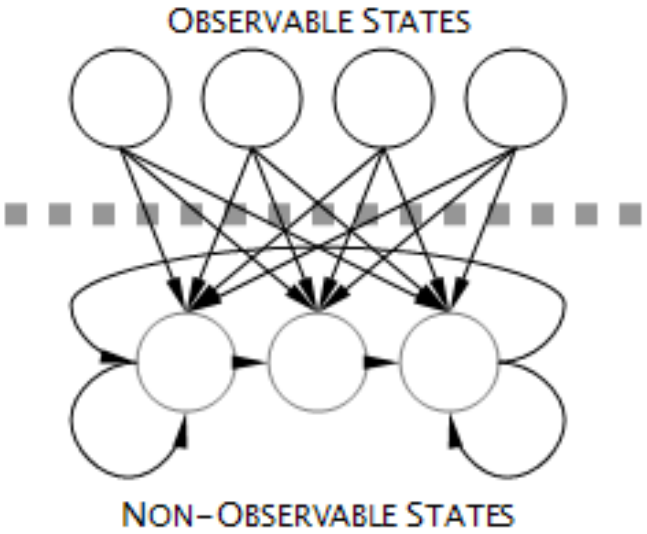
Chapter 15. Dynamic Programming

Viterbi Algorithm



Chapter 15. Dynamic Programming

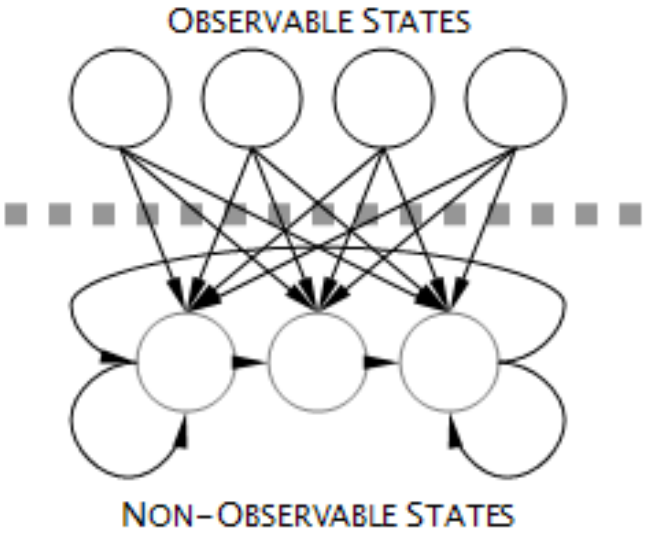
Viterbi Algorithm



- assume an HMM $\mathcal{M}$;

Chapter 15. Dynamic Programming

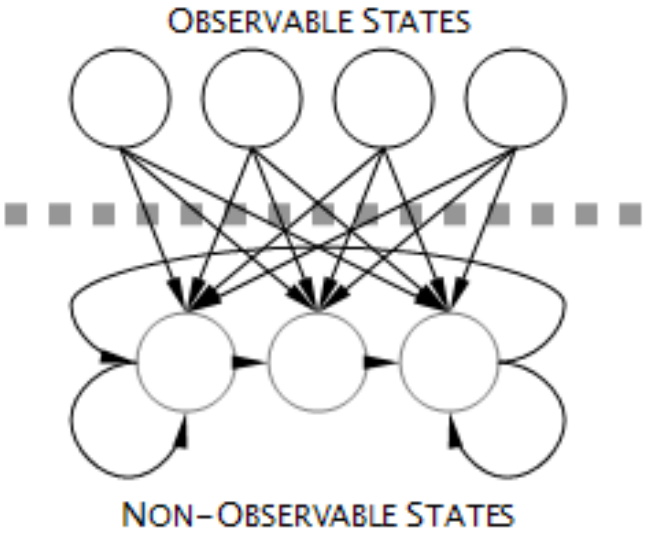
Viterbi Algorithm



- assume an HMM $\mathcal{M}$;
- decode the hidden states H^* that have produced the given (observed) data D

Chapter 15. Dynamic Programming

Viterbi Algorithm



- assume an HMM $\mathcal{M}$;
- decode the hidden states H^* that have produced the given (observed) data D by computing the maximum probability

$$H^* = \arg\max_H \{Prob(H, D \mid \mathcal{M})\}$$

to produce the given data, with dynamic programming.

Chapter 15. Dynamic Programming

A hidden Markov model (HMM) consists of (S, T, e, t), where

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;

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

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

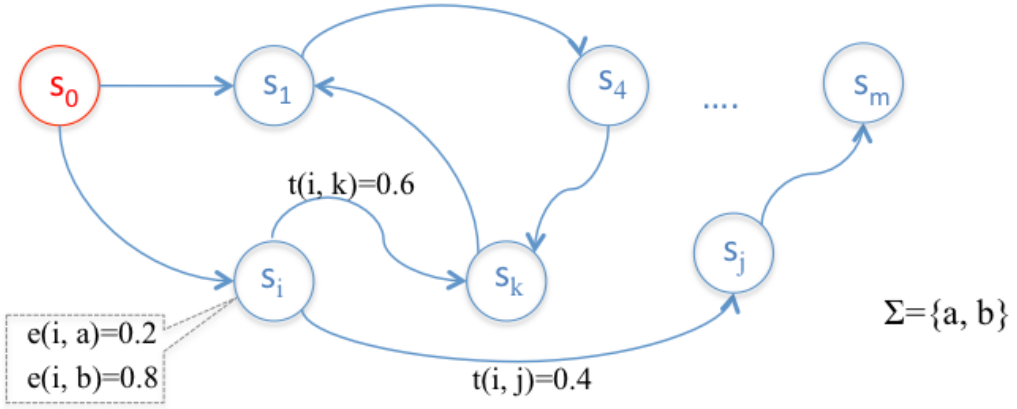
A hidden Markov model (HMM) consists of (S, T, e, t), where

- 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

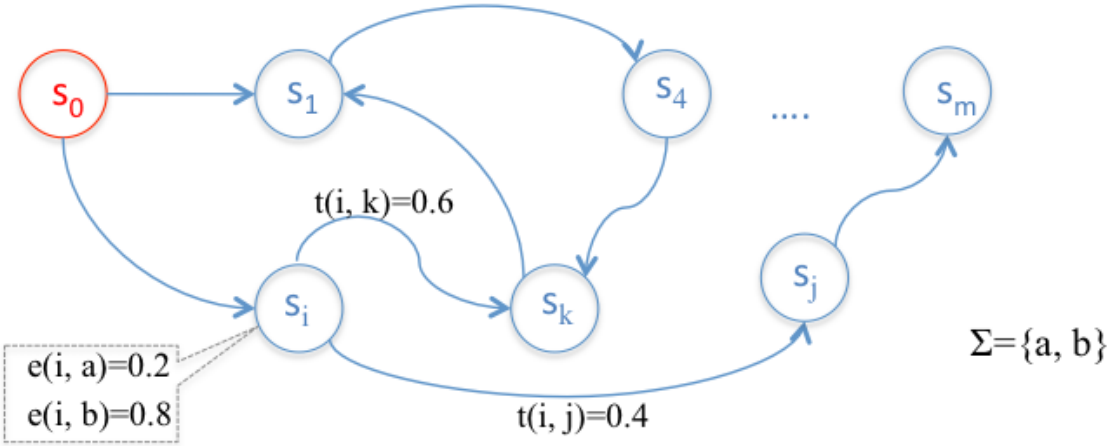
$$\sum_{a \in \Sigma} e(i, a) = 1 \quad \text{for every } i,\ 1 \leq i < m$$

- every $(s_i, s_j) \in T$ is associated with a probability distribution $t(i, j)$, such that

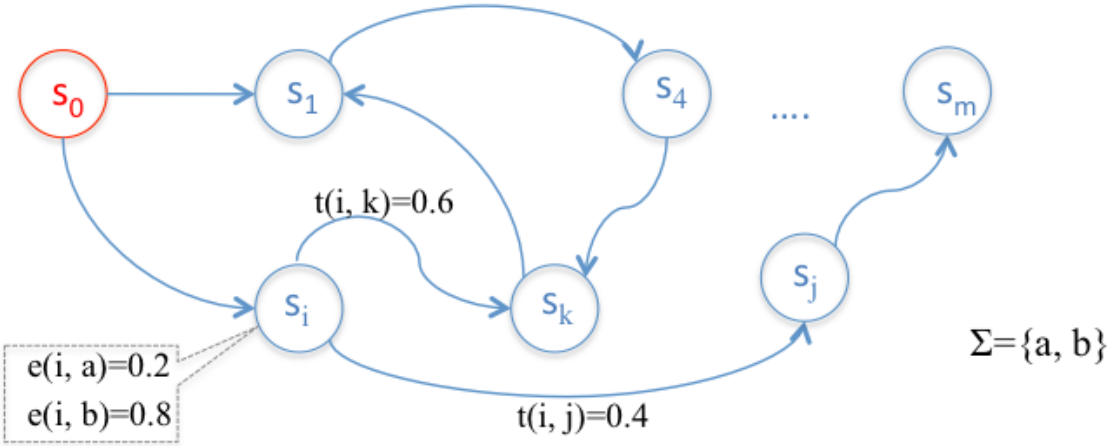
$$\sum_{1 \leq j \leq m} t(i, j) = 1 \quad \text{for every } i,\ 0 \leq i < m$$



Chapter 15. Dynamic Programming

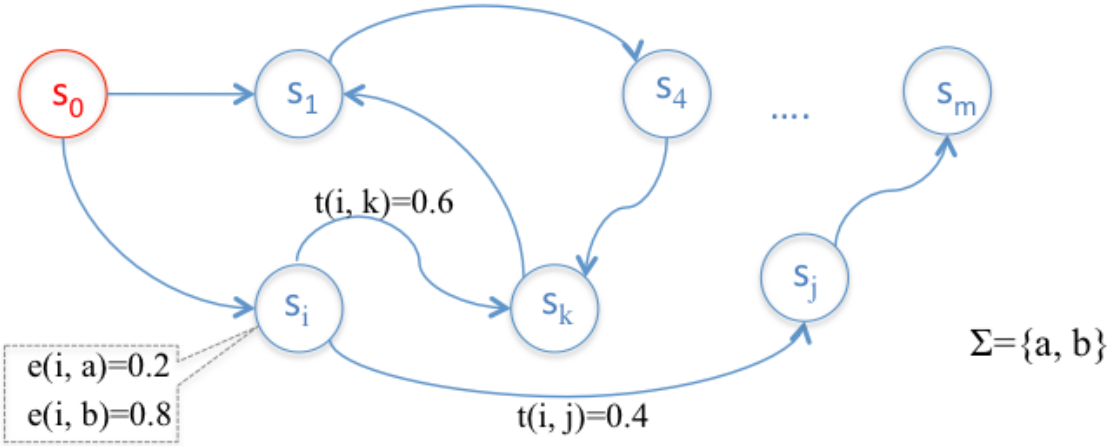


Chapter 15. Dynamic Programming



HMM decoding problem:

Chapter 15. Dynamic Programming

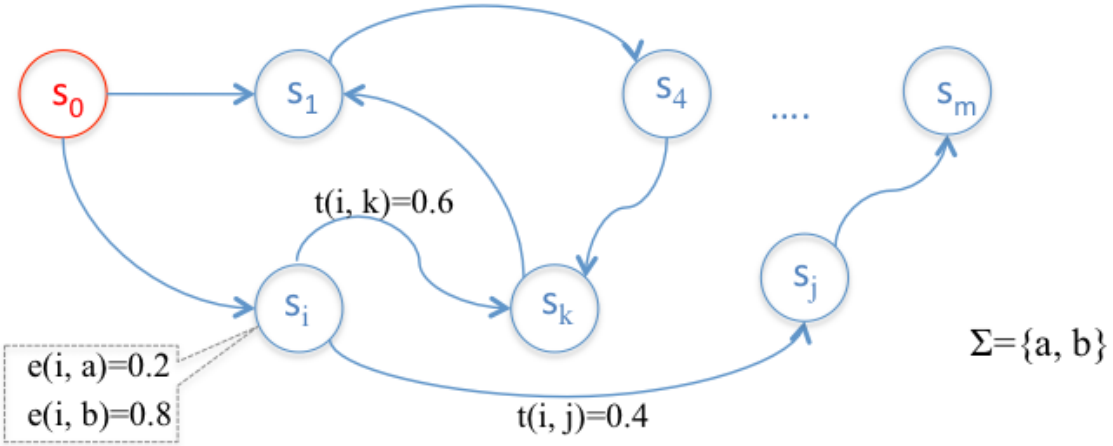


HMM decoding problem:

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

$$y^* = \arg\max_y \{Prob(y, x|\mathcal{M})\}$$

where y begins from s_0.

Chapter 15. Dynamic Programming

The idea of Viterbi algorithm:

Chapter 15. Dynamic Programming

The idea of Viterbi algorithm:

For any prefix $x_1 \dots x_k$, $k \geq 0$,

Chapter 15. Dynamic Programming

The idea of Viterbi algorithm:

For any prefix $x_1 \dots x_k$, $k \geq 0$, and state $s_j \in S$;

Chapter 15. Dynamic Programming

The idea of Viterbi algorithm:

For any prefix $x_1 \dots x_k$, $k \geq 0$, and state $s_j \in S$;

$$\max_{y=s_0 \dots s_j} \{Prob(y, x_1 \dots x_k | \mathcal{M})\}$$

the maximum probability of a path $s_0 \dots s_j$ to generate $x_1 \dots x_k$;

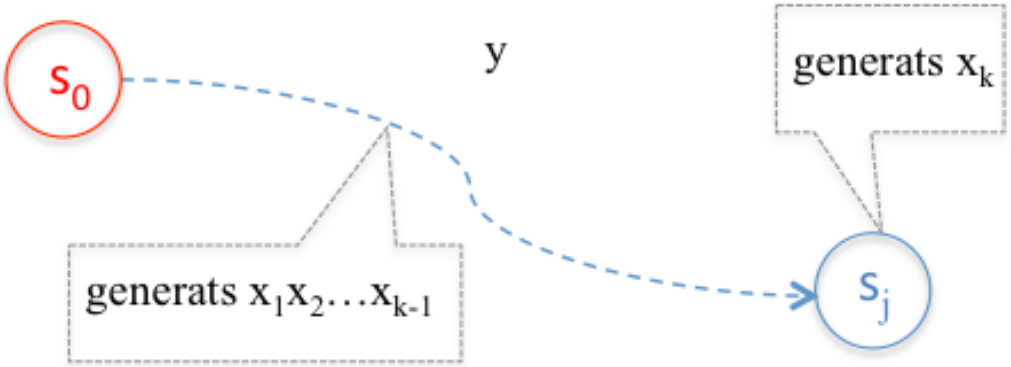
Chapter 15. Dynamic Programming

The idea of Viterbi algorithm:

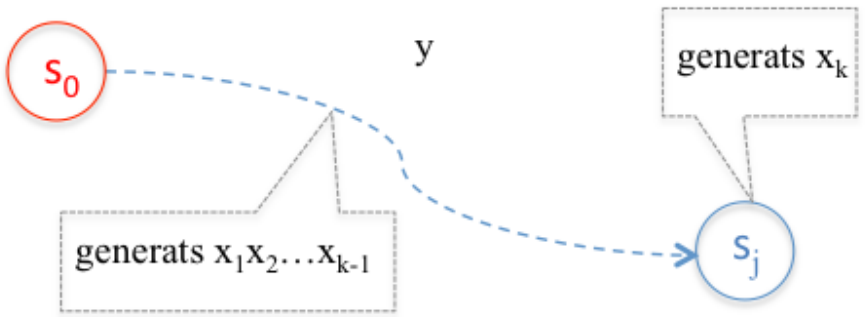
For any prefix $x_1 \dots x_k$, $k \geq 0$, and state $s_j \in S$;

$$\max_{y=s_0 \dots s_j} \{Prob(y, x_1 \dots x_k | \mathcal{M})\}$$

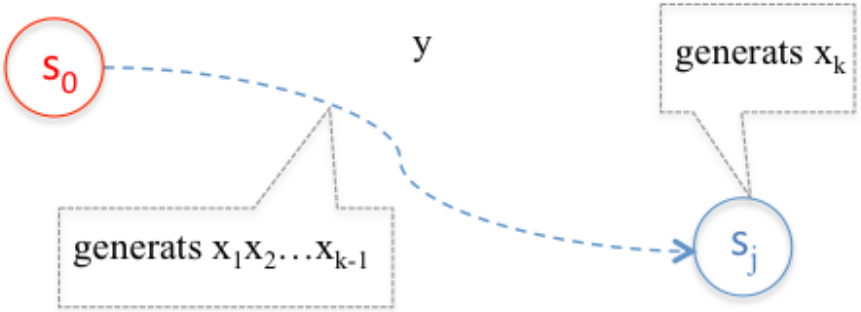
the maximum probability of a path $s_0 \dots s_j$ to generate $x_1 \dots x_k$;



Chapter 15. Dynamic Programming

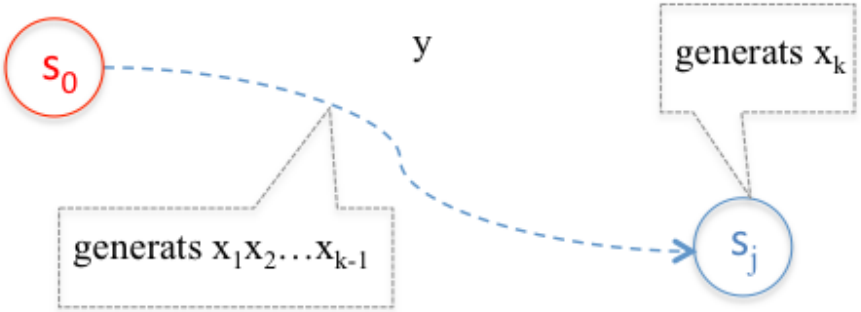


Chapter 15. Dynamic Programming

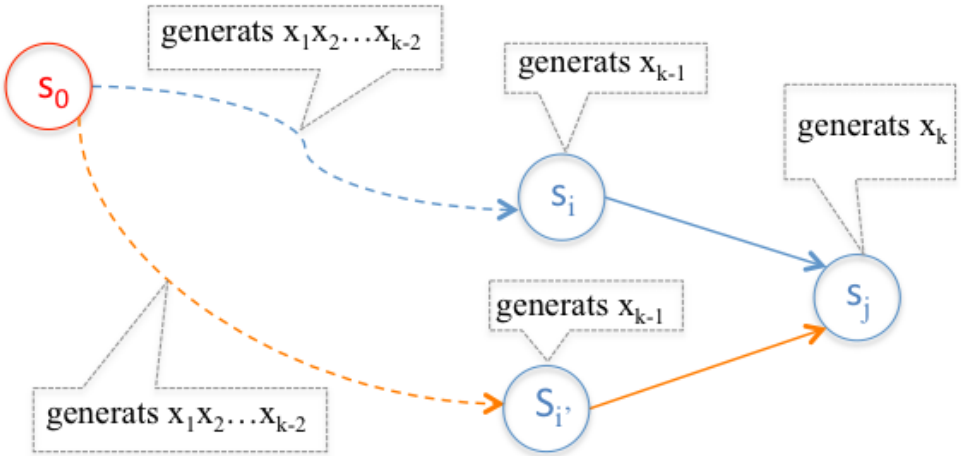


Define $f(j,k) = \max_{y=s_0\ldots s_j} \{Prob(y, x_1 \ldots x_k)\}$

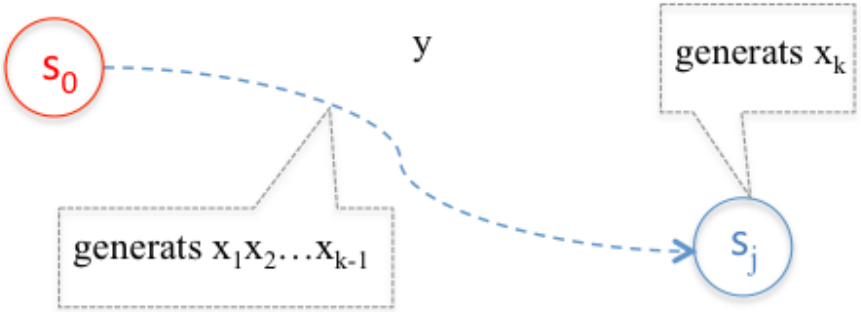
Chapter 15. Dynamic Programming



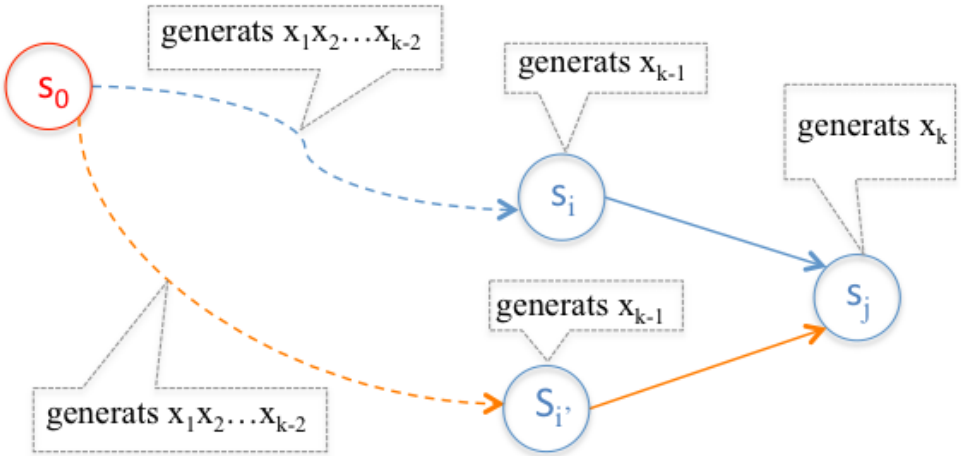
Define $f(j,k) = \max_{y=s_0\ldots s_j} \{Prob(y, x_1 \ldots x_k)\}$



Chapter 15. Dynamic Programming

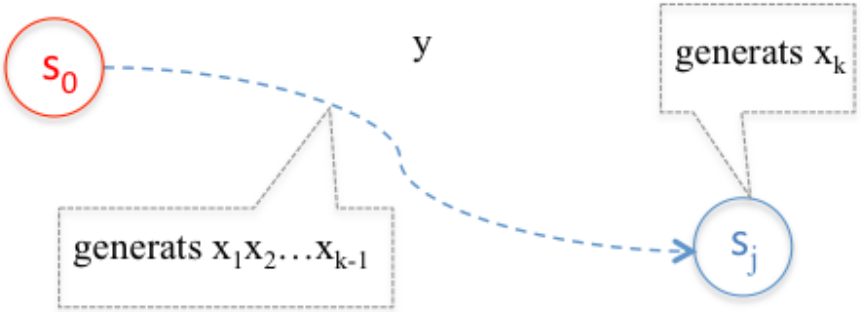


Define $f(j,k) = \max_{y=s_0\ldots s_j} \{Prob(y, x_1 \ldots x_k)\}$

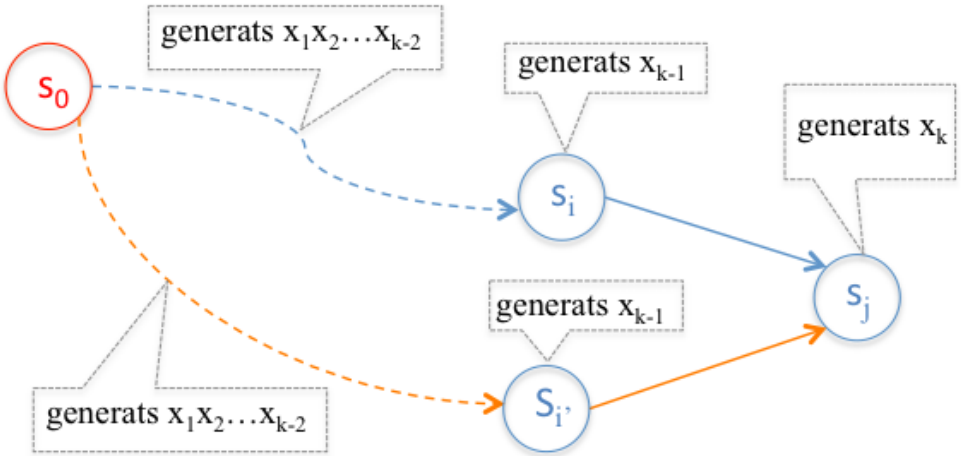


Recursively, $f(j,k) =$

Chapter 15. Dynamic Programming

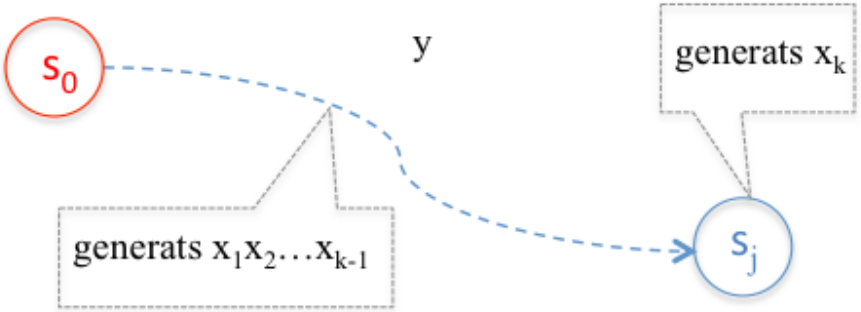


Define $f(j,k) = \max_{y=s_0\ldots s_j} \{Prob(y, x_1 \ldots x_k)\}$

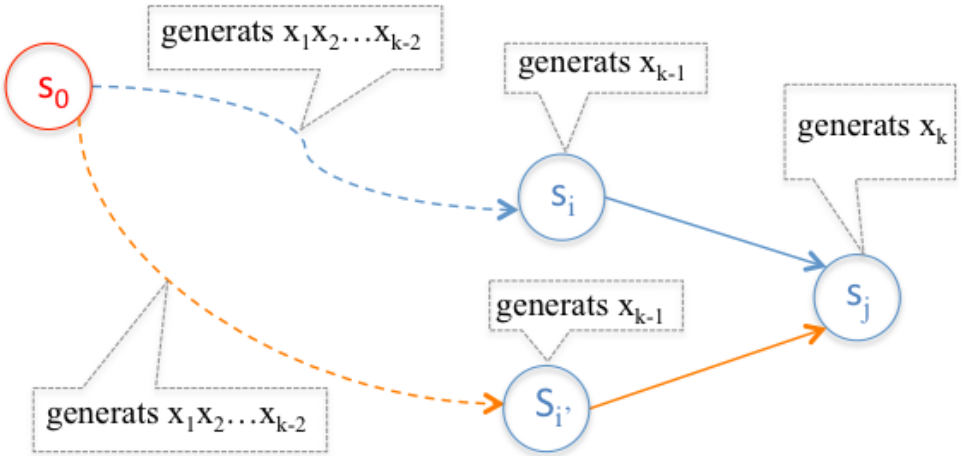


Recursively, $f(j,k) = \max_i$

Chapter 15. Dynamic Programming

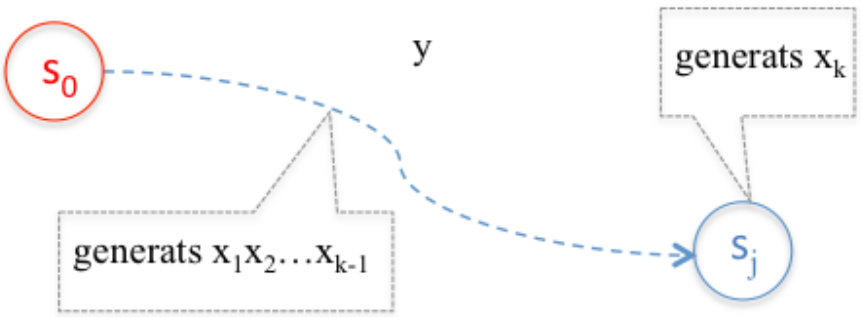


Define $f(j,k) = \max_{y=s_0\ldots s_j} \{Prob(y, x_1 \ldots x_k)\}$

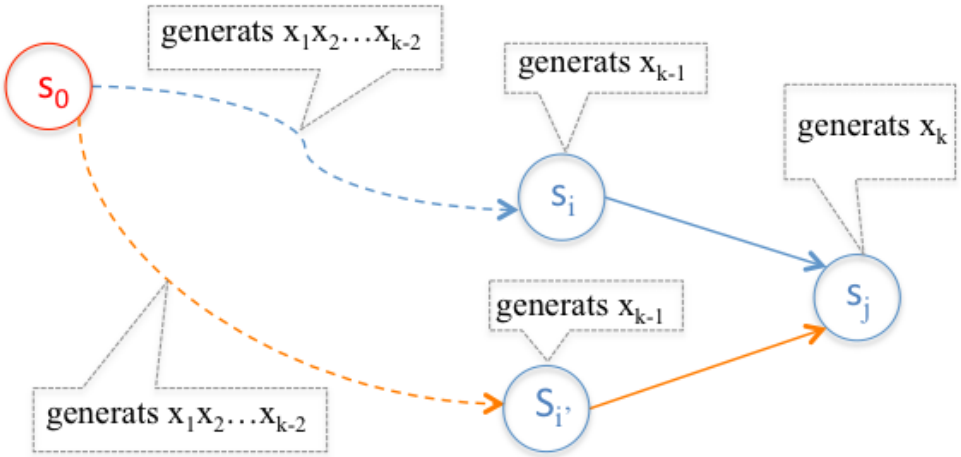


Recursively, $f(j,k) = \max_i \{f(i, k-1)$

Chapter 15. Dynamic Programming

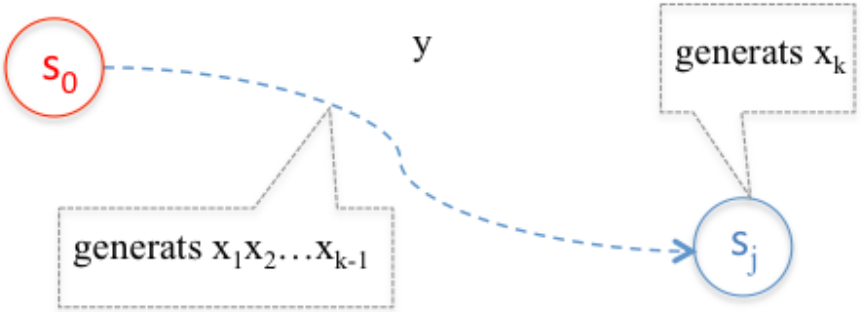


Define $f(j,k) = \max_{y=s_0\ldots s_j} \{Prob(y, x_1 \ldots x_k)\}$

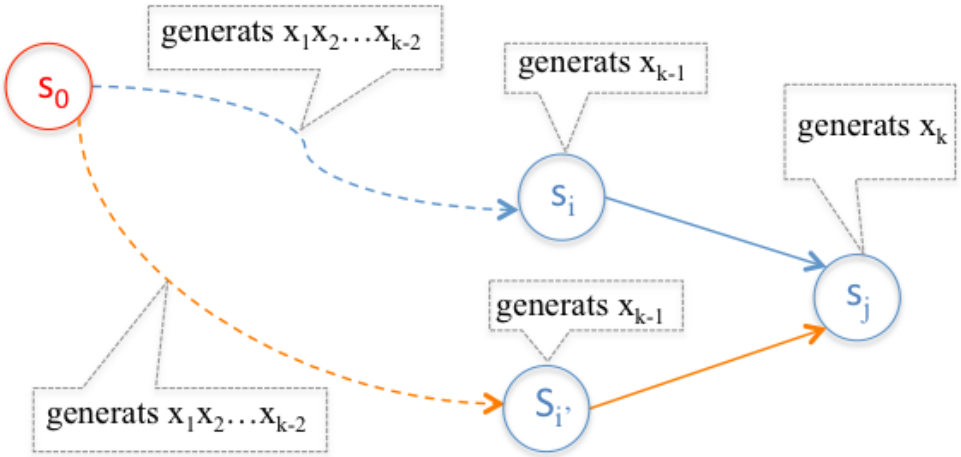


Recursively, $f(j,k) = \max_i \{f(i,k-1) \times t(i,j)$

Chapter 15. Dynamic Programming

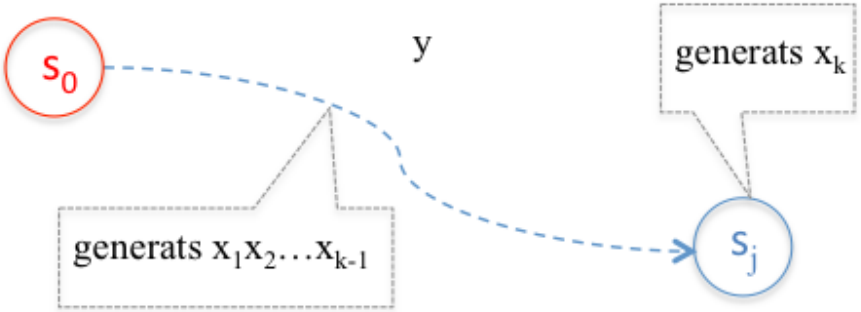


Define $f(j,k) = \max_{y=s_0\ldots s_j} \{Prob(y, x_1 \ldots x_k)\}$

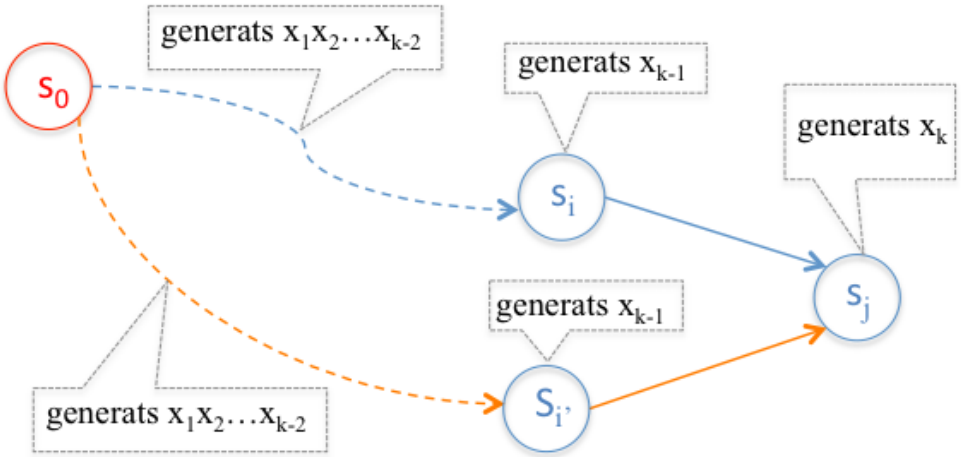


Recursively, $f(j,k) = \max_i \{f(i,k-1) \times t(i,j) \times e(j,x_k)\}$

Chapter 15. Dynamic Programming

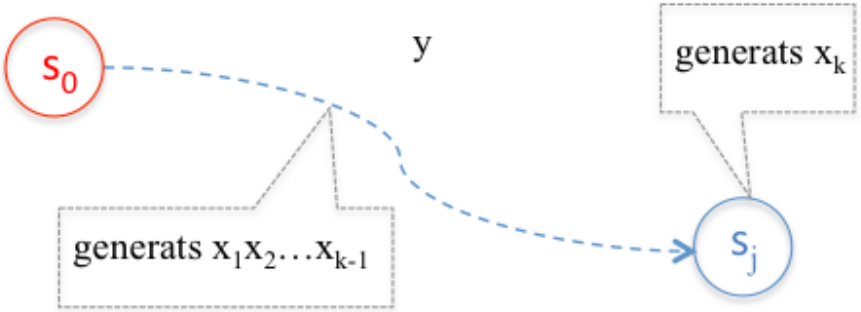


Define $f(j,k) = \max_{y=s_0\ldots s_j} \{Prob(y, x_1 \ldots x_k)\}$

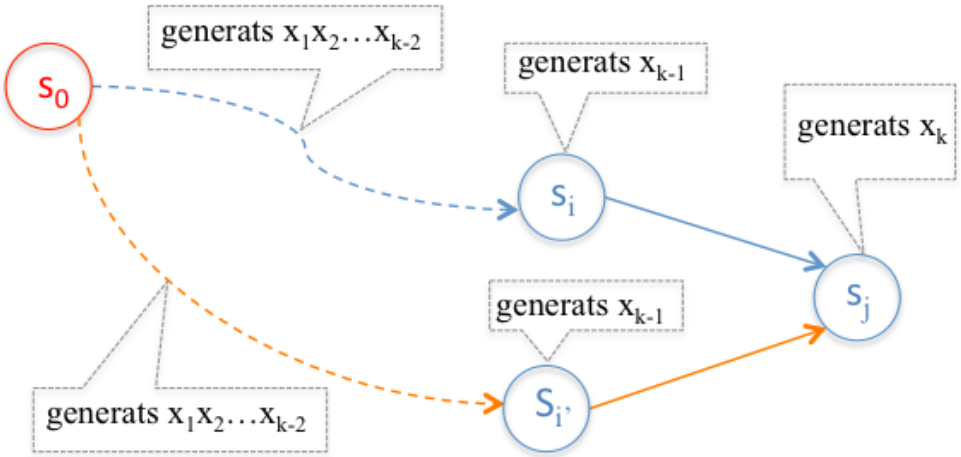


Recursively, $f(j,k) = \max_i \{f(i,k-1) \times t(i,j) \times e(j,x_k)\}$

Chapter 15. Dynamic Programming



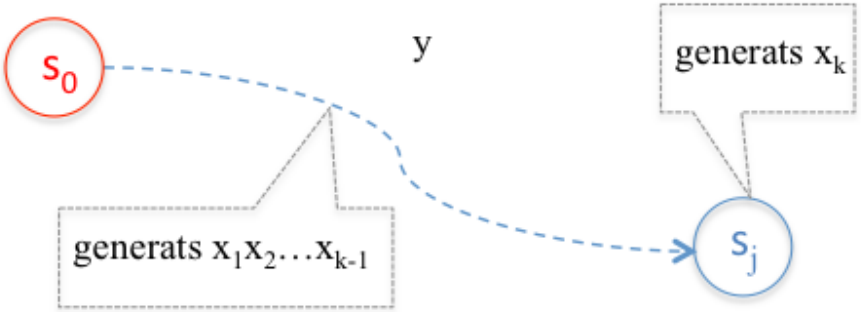
Define $f(j,k) = \max_{y=s_0\ldots s_j} \{Prob(y, x_1 \ldots x_k)\}$



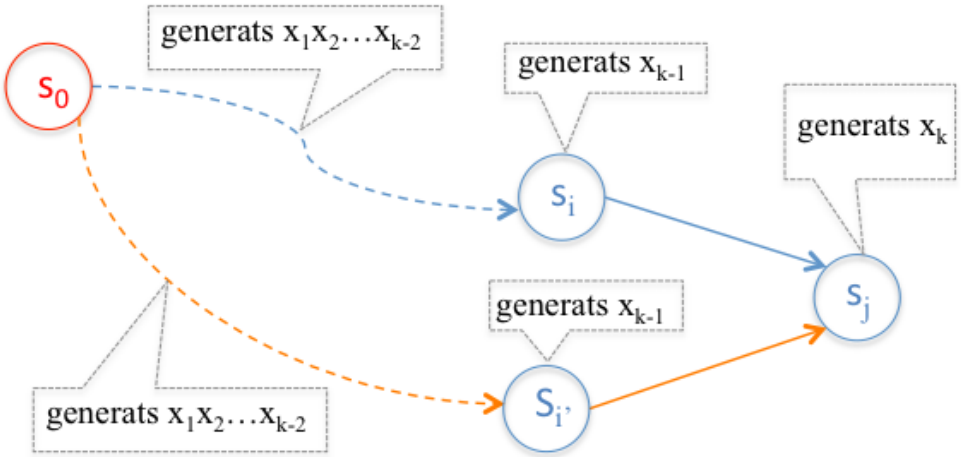
Recursively, $f(j,k) = \max_i \{f(i,k-1) \times t(i,j) \times e(j,x_k)\}$

Base case: $f(0,0) =$

Chapter 15. Dynamic Programming



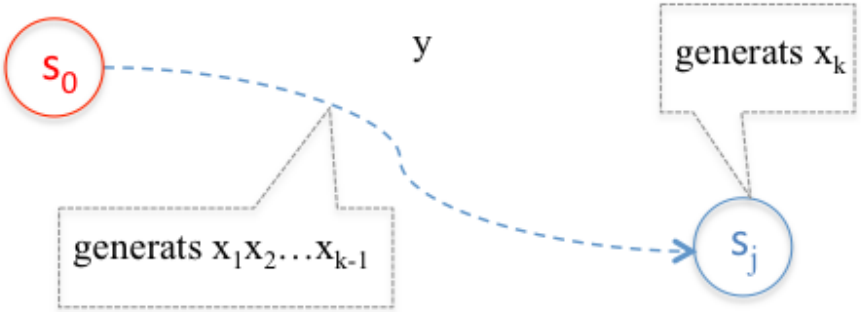
Define $f(j,k) = \max_{y=s_0\ldots s_j} \{Prob(y, x_1 \ldots x_k)\}$



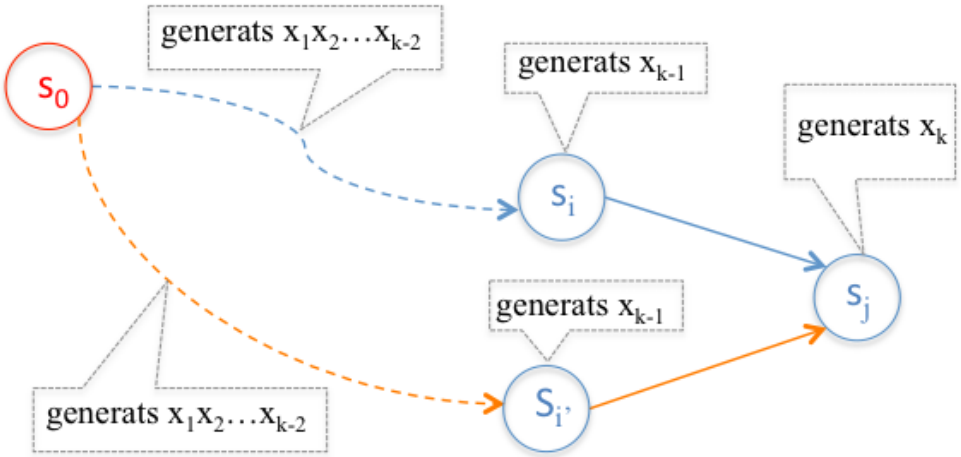
Recursively, $f(j,k) = \max_i \{f(i,k-1) \times t(i,j) \times e(j,x_k)\}$

Base case: $f(0,0) = 1$,

Chapter 15. Dynamic Programming



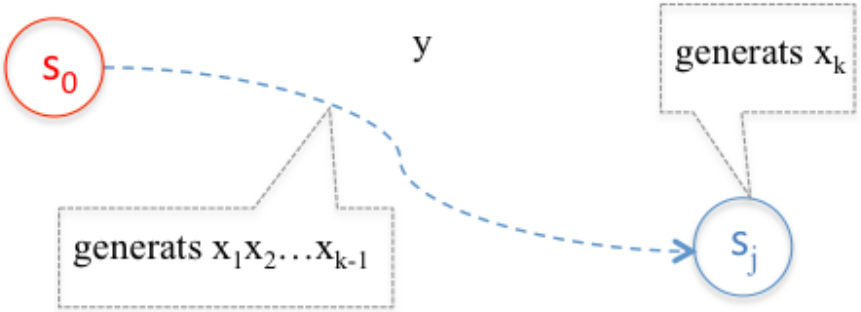
Define $f(j,k) = \max_{y=s_0\ldots s_j} \{Prob(y, x_1 \ldots x_k)\}$



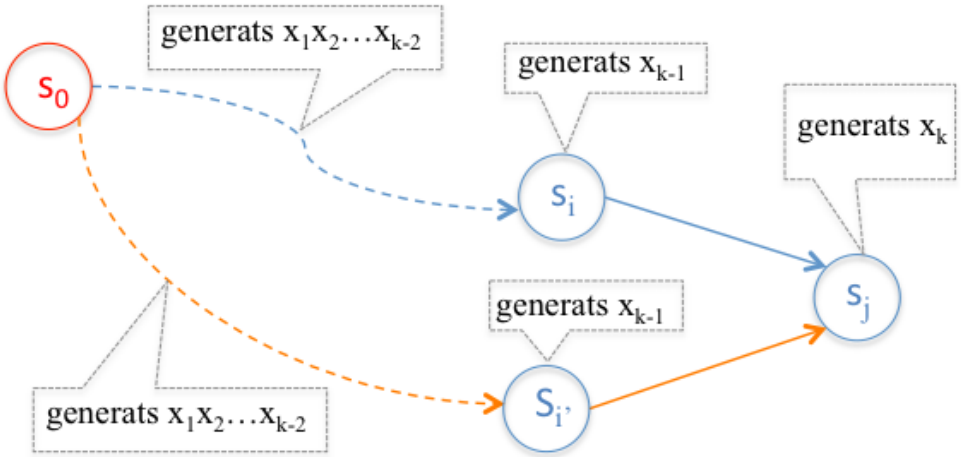
Recursively, $f(j,k) = \max_i \{f(i,k-1) \times t(i,j) \times e(j,x_k)\}$

Base case: $f(0,0) = 1, \quad f(i,0)$

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 $i \geq 1$.

Chapter 15. Dynamic Programming

Algorithm $\textsc{Viterbi}(x, n, t, e, m)$

0. $T(0,0) = 1;\ P(0,0) = 0;$

Chapter 15. Dynamic Programming

Algorithm $\textsc{Viterbi}(x, n, t, e, m)$

```
T(0,0) = 1; P(0,0) = 0;
for j = 1 to m
    T(j,0) = 0;
```

Chapter 15. Dynamic Programming

Algorithm $\textsc{Viterbi}(x, n, t, e, m)$

```
T(0,0) = 1; P(0,0) = 0;
for j = 1 to m
    T(j,0) = 0;
for k = 1 to n
```

Chapter 15. Dynamic Programming

Algorithm $\textsc{Viterbi}(x, n, t, e, m)$

```
T(0,0) = 1; P(0,0) = 0;
for j = 1 to m
    T(j,0) = 0;
for k = 1 to n                    for every prefix upto kth symbol
```

Chapter 15. Dynamic Programming

Algorithm VITERBI(x, n, t, e, m)

```
T(0,0) = 1; P(0,0) = 0;
for j = 1 to m
    T(j,0) = 0;
for k = 1 to n                    for every prefix upto kth symbol
    for j = 1 to m
```

Chapter 15. Dynamic Programming

Algorithm VITERBI(x, n, t, e, m)

```
T(0,0) = 1; P(0,0) = 0;
for j = 1 to m
    T(j,0) = 0;
for k = 1 to n                  for every prefix upto kth symbol
    for j = 1 to m              for every 'last state' s_j
```

Chapter 15. Dynamic Programming

Algorithm VITERBI(x, n, t, e, m)

```
T(0,0) = 1; P(0,0) = 0;
for j = 1 to m
    T(j,0) = 0;
for k = 1 to n                  for every prefix upto kth symbol
    for j = 1 to m              for every 'last state' s_j
        premax = 0
```

Chapter 15. Dynamic Programming

Algorithm VITERBI(x, n, t, e, m)

```
T(0,0) = 1; P(0,0) = 0;
for j = 1 to m
    T(j,0) = 0;
for k = 1 to n                  for every prefix upto kth symbol
    for j = 1 to m              for every 'last state' s_j
        premax = 0              initialize the prefix probability
```

Chapter 15. Dynamic Programming

Algorithm VITERBI(x, n, t, e, m)

```
T(0,0) = 1; P(0,0) = 0;
for j = 1 to m
    T(j,0) = 0;
for k = 1 to n                  for every prefix upto kth symbol
    for j = 1 to m              for every 'last state' s_j
        premax = 0              initialize the prefix probability
        for i = 0 to m          try every 'predecessor state' s_i
```

Chapter 15. Dynamic Programming

Algorithm VITERBI(x, n, t, e, m)

```
T(0,0) = 1; P(0,0) = 0;
for j = 1 to m
    T(j,0) = 0;
for k = 1 to n                  for every prefix upto kth symbol
    for j = 1 to m              for every 'last state' s_j
        premax = 0              initialize the prefix probability
        for i = 0 to m          try every 'predecessor state' s_i
            if premax < T(i,k-1) × t(i,j) × e(j,x_k)
```

Chapter 15. Dynamic Programming

Algorithm VITERBI(x, n, t, e, m)

```
T(0,0) = 1; P(0,0) = 0;
for j = 1 to m
    T(j,0) = 0;
for k = 1 to n                  for every prefix upto kth symbol
    for j = 1 to m              for every 'last state' s_j
        premax = 0              initialize the prefix probability
        for i = 0 to m          try every 'predecessor state' s_i
            if premax < T(i,k-1) × t(i,j) × e(j,x_k)
                premax = T(i,k-1) × t(i,j) × e(j,x_k)
```

Chapter 15. Dynamic Programming

Algorithm VITERBI(x, n, t, e, m)

```
T(0,0) = 1; P(0,0) = 0;
for j = 1 to m
    T(j,0) = 0;
for k = 1 to n                  for every prefix upto kth symbol
    for j = 1 to m              for every 'last state' s_j
        premax = 0              initialize the prefix probability
        for i = 0 to m          try every 'predecessor state' s_i
            if premax < T(i,k-1) × t(i,j) × e(j,x_k)
                premax = T(i,k-1) × t(i,j) × e(j,x_k)  update probability
```

Chapter 15. Dynamic Programming

Algorithm VITERBI(x, n, t, e, m)

```
T(0,0) = 1; P(0,0) = 0;
for j = 1 to m
    T(j,0) = 0;
for k = 1 to n                  for every prefix upto kth symbol
    for j = 1 to m              for every 'last state' s_j
        premax = 0              initialize the prefix probability
        for i = 0 to m          try every 'predecessor state' s_i
            if premax < T(i,k-1) × t(i,j) × e(j,x_k)
                premax = T(i,k-1) × t(i,j) × e(j,x_k)  update probability
                P(j,k) = i
```

Chapter 15. Dynamic Programming

Algorithm VITERBI(x, n, t, e, m)

```
T(0,0) = 1; P(0,0) = 0;
for j = 1 to m
    T(j,0) = 0;
for k = 1 to n                  for every prefix upto kth symbol
    for j = 1 to m              for every 'last state' s_j
        premax = 0              initialize the prefix probability
        for i = 0 to m          try every 'predecessor state' s_i
            if premax < T(i,k-1) × t(i,j) × e(j,x_k)
                premax = T(i,k-1) × t(i,j) × e(j,x_k)  update probability
                P(j,k) = i      memorize the predecessor
```

Chapter 15. Dynamic Programming

Algorithm VITERBI(x, n, t, e, m)

```
T(0,0) = 1; P(0,0) = 0;
for j = 1 to m
    T(j,0) = 0;
for k = 1 to n                  for every prefix upto kth symbol
    for j = 1 to m              for every 'last state' s_j
        premax = 0              initialize the prefix probability
        for i = 0 to m          try every 'predecessor state' s_i
            if premax < T(i,k-1) × t(i,j) × e(j,x_k)
                premax = T(i,k-1) × t(i,j) × e(j,x_k)   update probability
                P(j,k) = i      memorize the predecessor
        T(j,k) = premax
```

Chapter 15. Dynamic Programming

Algorithm VITERBI(x, n, t, e, m)

```
T(0,0) = 1; P(0,0) = 0;
for j = 1 to m
    T(j,0) = 0;
for k = 1 to n                  for every prefix upto kth symbol
    for j = 1 to m              for every 'last state' s_j
        premax = 0              initialize the prefix probability
        for i = 0 to m          try every 'predecessor state' s_i
            if premax < T(i,k-1) × t(i,j) × e(j,x_k)
                premax = T(i,k-1) × t(i,j) × e(j,x_k)   update probability
                P(j,k) = i      memorize the predecessor
        T(j,k) = premax
laststate = 1; max = T(1,n)
```

Chapter 15. Dynamic Programming

Algorithm VITERBI(x, n, t, e, m)

```
T(0,0) = 1; P(0,0) = 0;
for j = 1 to m
    T(j,0) = 0;
for k = 1 to n                    for every prefix upto kth symbol
    for j = 1 to m                for every 'last state' s_j
        premax = 0                initialize the prefix probability
        for i = 0 to m            try every 'predecessor state' s_i
            if premax < T(i,k-1) × t(i,j) × e(j,x_k)
                premax = T(i,k-1) × t(i,j) × e(j,x_k)  update probability
                P(j,k) = i        memorize the predecessor
        T(j,k) = premax
laststate = 1; max = T(1,n)
for j = 2 to m                    identify maximum probability
```

Chapter 15. Dynamic Programming

Algorithm VITERBI(x, n, t, e, m)

```
T(0,0) = 1; P(0,0) = 0;
for j = 1 to m
    T(j,0) = 0;
for k = 1 to n                  for every prefix upto kth symbol
    for j = 1 to m              for every 'last state' s_j
        premax = 0              initialize the prefix probability
        for i = 0 to m          try every 'predecessor state' s_i
            if premax < T(i,k-1) × t(i,j) × e(j,x_k)
                premax = T(i,k-1) × t(i,j) × e(j,x_k)  update probability
                P(j,k) = i      memorize the predecessor
        T(j,k) = premax
laststate = 1; max = T(1,n)
for j = 2 to m                  identify maximum probability
    if max < T(j,n)
```

Chapter 15. Dynamic Programming

Algorithm VITERBI(x, n, t, e, m)

```
T(0,0) = 1; P(0,0) = 0;
for j = 1 to m
    T(j,0) = 0;
for k = 1 to n                    for every prefix upto kth symbol
    for j = 1 to m                for every 'last state' s_j
        premax = 0                initialize the prefix probability
        for i = 0 to m            try every 'predecessor state' s_i
            if premax < T(i,k-1) × t(i,j) × e(j,x_k)
                premax = T(i,k-1) × t(i,j) × e(j,x_k)  update probability
                P(j,k) = i        memorize the predecessor
        T(j,k) = premax
laststate = 1; max = T(1,n)
for j = 2 to m                    identify maximum probability
    if max < T(j,n)
    laststate = j;
```

Chapter 15. Dynamic Programming

Algorithm VITERBI(x, n, t, e, m)

```
T(0,0) = 1; P(0,0) = 0;
for j = 1 to m
    T(j,0) = 0;
for k = 1 to n                    for every prefix upto kth symbol
    for j = 1 to m                for every 'last state' s_j
        premax = 0                initialize the prefix probability
        for i = 0 to m            try every 'predecessor state' s_i
            if premax < T(i,k-1) × t(i,j) × e(j,x_k)
                premax = T(i,k-1) × t(i,j) × e(j,x_k)  update probability
                P(j,k) = i        memorize the predecessor
        T(j,k) = premax
laststate = 1; max = T(1,n)
for j = 2 to m                    identify maximum probability
    if max < T(j,n)
    laststate = j;
    max = T(j,n)
```

Chapter 15. Dynamic Programming

Algorithm VITERBI(x, n, t, e, m)

```
T(0,0) = 1; P(0,0) = 0;
for j = 1 to m
    T(j,0) = 0;
for k = 1 to n                  for every prefix upto kth symbol
    for j = 1 to m              for every 'last state' s_j
        premax = 0              initialize the prefix probability
        for i = 0 to m          try every 'predecessor state' s_i
            if premax < T(i,k-1) × t(i,j) × e(j,x_k)
                premax = T(i,k-1) × t(i,j) × e(j,x_k)  update probability
                P(j,k) = i      memorize the predecessor
        T(j,k) = premax
laststate = 1; max = T(1,n)
for j = 2 to m                  identify maximum probability
    if max < T(j,n)
    laststate = j;
    max = T(j,n)
return (T, P, max, laststate)
```

Chapter 15. Dynamic Programming

Algorithm VITERBI(x, n, t, e, m)

```
T(0,0) = 1; P(0,0) = 0;
for j = 1 to m
    T(j,0) = 0;
for k = 1 to n                  for every prefix upto kth symbol
    for j = 1 to m              for every 'last state' s_j
        premax = 0              initialize the prefix probability
        for i = 0 to m          try every 'predecessor state' s_i
            if premax < T(i,k-1) × t(i,j) × e(j,x_k)
                premax = T(i,k-1) × t(i,j) × e(j,x_k)  update probability
                P(j,k) = i      memorize the predecessor
        T(j,k) = premax
laststate = 1; max = T(1,n)
for j = 2 to m                  identify maximum probability
    if max < T(j,n)
    laststate = j;
    max = T(j,n)
return (T, P, max, laststate)
```

Time complexity:

Chapter 15. Dynamic Programming

Algorithm VITERBI(x, n, t, e, m)

```
T(0,0) = 1; P(0,0) = 0;
for j = 1 to m
    T(j,0) = 0;
for k = 1 to n                    for every prefix upto kth symbol
    for j = 1 to m                for every 'last state' s_j
        premax = 0                initialize the prefix probability
        for i = 0 to m            try every 'predecessor state' s_i
            if premax < T(i,k-1) × t(i,j) × e(j,x_k)
                premax = T(i,k-1) × t(i,j) × e(j,x_k)  update probability
                P(j,k) = i        memorize the predecessor
        T(j,k) = premax
laststate = 1; max = T(1,n)
for j = 2 to m                    identify maximum probability
    if max < T(j,n)
    laststate = j;
    max = T(j,n)
return (T, P, max, laststate)
```

Time complexity: $O(nm^2)$

Use the computed table P and $laststate$ to traceback the hidden states

Use the computed table P and $laststate$ to traceback the hidden states

Algorithm $\textsc{TraceHiddenStates}(P, j, k, x)$

1. **if** $j > 0$

Chapter 15. Dynamic Programming

Use the computed table P and $laststate$ to traceback the hidden states

Algorithm $\textsc{TraceHiddenStates}(P, j, k, x)$

1. **if** $j > 0$
2. $\quad\textsc{TraceHiddenStates}(P, P[j, k], k - 1, x)$

Chapter 15. Dynamic Programming

Use the computed table P and $laststate$ to traceback the hidden states

Algorithm $\textsc{TraceHiddenStates}(P, j, k, x)$

1. **if** $j > 0$
2. $\quad\textsc{TraceHiddenStates}(P, P[j, k], k - 1, x)$
3. $\quad$**print** ('State' j 'emits symbol' x_k)

Chapter 15. Dynamic Programming

Use the computed table P and $laststate$ to traceback the hidden states

Algorithm $\textsc{TraceHiddenStates}(P, j, k, x)$

```
if j > 0
    TraceHiddenStates(P, P[j, k], k − 1, x)
    print ('State' j 'emits symbol' x_k)
```

First, call $\textsc{TraceHiddenStates}(P, laststate, n, x)$

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.

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 IBRETIV

How to find objective function?
Recalled for forward, we defined

$$f(j,k) = \max_{y=s_0 \ldots s_j} \{Prob(y, x_1 \ldots x_k)\}$$

$f(j,k)$ is the maximum probability to generate prefix $x_1 \ldots x_1$ beginning from state s_0 and ending with state s_j;

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

How to find objective function?
Recalled for forward, we defined

$$f(j,k) = \max_{y=s_0 \ldots s_j} \{Prob(y, x_1 \ldots x_k)\}$$

$f(j,k)$ 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

$$b(l,k) = \max_{y=s_l \ldots s_\omega} \{Prob(y, x_k \ldots x_n)\}$$

$b(l,k)$ is the maximum probability to generate suffix $x_k \ldots x_n$ beginning from state s_l and ending with state s_ω;

Chapter 15. Dynamic Programming

- We need to amend the HMM model with a silent state s_ω;

$$b(l,k) = \max_{y=s_l \ldots s_\omega} \{Prob(y, x_k \ldots x_n)\}$$

Chapter 15. Dynamic Programming

- We need to amend the HMM model with a silent state s_ω;

$$b(l,k) = \max_{y=s_l \ldots s_\omega} \{Prob(y, x_k \ldots x_n)\}$$

- Recurrence (draw a diagram)

$$b(l,k) =$$

Chapter 15. Dynamic Programming

- We need to amend the HMM model with a silent state s_ω;

$$b(l,k) = \max_{y=s_l \ldots s_\omega} \{Prob(y, x_k \ldots x_n)\}$$

- Recurrence (draw a diagram)

$$b(l,k) = \max_i$$

Chapter 15. Dynamic Programming

- We need to amend the HMM model with a silent state s_ω;

$$b(l,k) = \max_{y=s_l \ldots s_\omega} \{Prob(y, x_k \ldots x_n)\}$$

- Recurrence (draw a diagram)

$$b(l,k) = \max_i \{b(i,k+1)$$

Chapter 15. Dynamic Programming

- We need to amend the HMM model with a silent state s_ω;

$$b(l,k) = \max_{y=s_l \ldots s_\omega} \{Prob(y, x_k \ldots x_n)\}$$

- 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_ω;

$$b(l,k) = \max_{y=s_l \ldots s_\omega} \{Prob(y, x_k \ldots x_n)\}$$

- 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

Chapter 15. Dynamic Programming

- We need to amend the HMM model with a silent state s_ω;

$$b(l,k) = \max_{y=s_l \ldots s_\omega} \{Prob(y, x_k \ldots x_n)\}$$

- 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:

Chapter 15. Dynamic Programming

- We need to amend the HMM model with a silent state s_ω;

$$b(l,k) = \max_{y=s_l \ldots s_\omega} \{Prob(y, x_k \ldots x_n)\}$$

- 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_ω;

$$b(l,k) = \max_{y=s_l \ldots s_\omega} \{Prob(y, x_k \ldots x_n)\}$$

- 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: $b(s_\omega, n+1) =$

Chapter 15. Dynamic Programming

- We need to amend the HMM model with a silent state s_ω;

$$b(l,k) = \max_{y=s_l \ldots s_\omega} \{Prob(y, x_k \ldots x_n)\}$$

- 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: $b(s_\omega, n+1) = 1$,

Chapter 15. Dynamic Programming

- We need to amend the HMM model with a silent state s_ω;

$$b(l,k) = \max_{y=s_l \ldots s_\omega} \{Prob(y, x_k \ldots x_n)\}$$

- 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_ω;

$$b(l,k) = \max_{y=s_l \ldots s_\omega} \{Prob(y, x_k \ldots x_n)\}$$

- 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** $\max_l\{b(l,1)\}$ **?**

Chapter 15. Dynamic Programming

- We need to amend the HMM model with a silent state s_ω;

$$b(l,k) = \max_{y=s_l \ldots s_\omega} \{Prob(y, x_k \ldots x_n)\}$$

- 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

<u>Computing the total probability</u> (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

$$Prob(x|\mathcal{M})$$

Chapter 15. Dynamic Programming

<u>Computing the total probability</u> (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 $\mathcal{M}$, and sequence $x \in \Sigma^*$, what is the probability

$$Prob(x|\mathcal{M}) = \sum_y Prob(y, x|\mathcal{M})$$

- consider prefix, $x_1 \ldots x_k$ and state s_j, define:

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

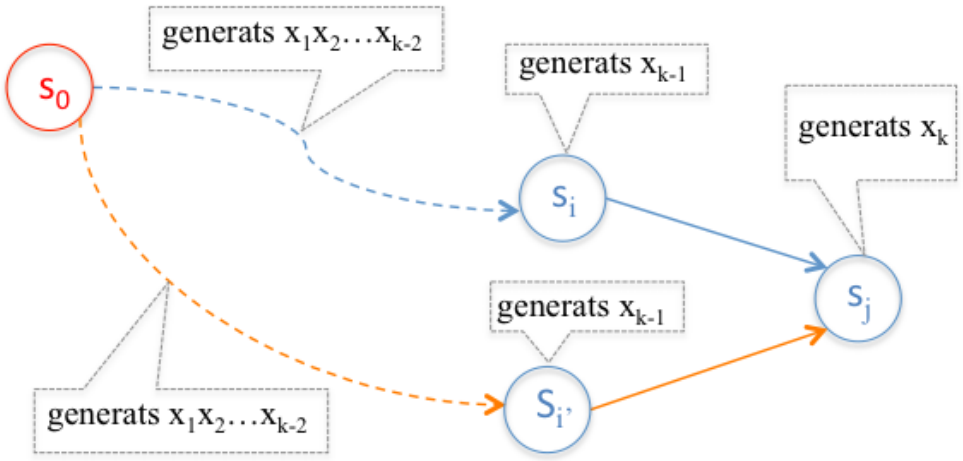
$$Prob(x|\mathcal{M}) = \sum_y Prob(y, x|\mathcal{M})$$

- consider prefix, $x_1 \ldots x_k$ and state s_j, 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.

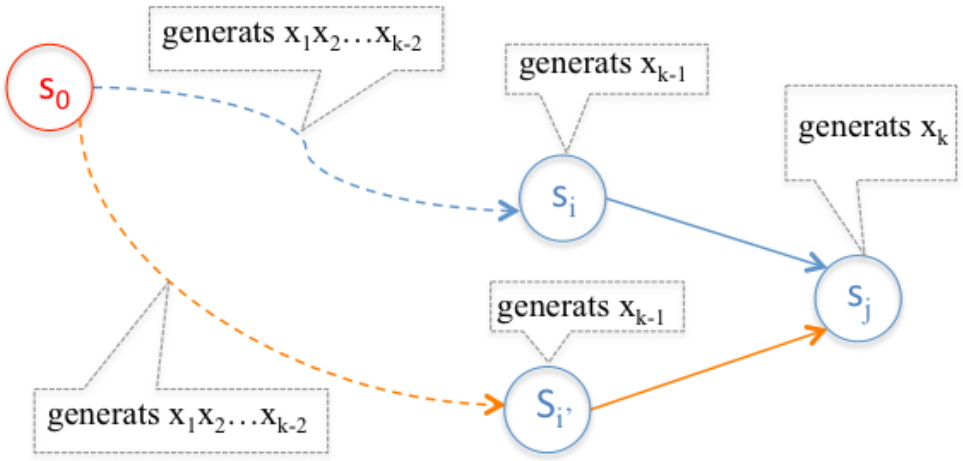
Chapter 15. Dynamic Programming

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.



Chapter 15. Dynamic Programming

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.



then

$$p(j,k) = \sum_i p(i,k-1) \times t(i,j) \times e(j,x_k)$$

with base cases: $p(0,0) = 1$, and $p(j,0) = 0$ for all $j \geq 1$.

Implementation of the total probability algorithm

Implementation of the total probability algorithm

- almost the same as for Viterbi algorithm.

Chapter 15. Dynamic Programming

Implementation of the total probability algorithm

- almost the same as for Viterbi algorithm.
- applications, for example

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

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

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

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

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)

Chapter 15. Dynamic Programming

Longest Common Subsequence (LCS)

$$x = \texttt{ACCGGTCGAGTGCG}$$
$$y = \texttt{GTCGTTCGGATGCCC}$$

Chapter 15. Dynamic Programming

Longest Common Subsequence (LCS)

$$x = \texttt{ACCGGTCGAGTGCG}$$
$$y = \texttt{GTCGTTCGGATGCCC}$$

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)

$$x = \texttt{ACCGGTCGAGTGCG}$$
$$y = \texttt{GTCGTTCGGATGCCC}$$

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)

$x =$ ACCGGTCGAGTGCG
$y =$ 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

```
ACCGGTCGAGTGCG
  CGTCGATGC   ⟵ common sequence
GTCGTTCGGATGCCC
```

Chapter 15. Dynamic Programming

Longest Common Subsequence (LCS)

$x =$ ACCGGTCGAGTGCG
$y =$ 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

```
ACCGGTCGAGTGCG
   CGTCGATGC   ⟵ common sequence
GTCGTTCGGATGCCC
```

significance of common sequence

Chapter 15. Dynamic Programming

Longest Common Subsequence (LCS)

$x = \texttt{ACCGGTCGAGTGCG}$
$y = \texttt{GTCGTTCGGATGCCC}$

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

```
ACCGGTCGAGTGCG
   CGTCGATGC   ⟵ common sequence
GTCGTTCGGATGCCC
```

significance of common sequence

```
ACCGGTC GAGTGCG
   || || || |||
   CG TC GA TGC   ⟵ common sequence
   || || || |||
GTCGTTCGGA TGCCC
```

Chapter 15. Dynamic Programming

Let $X = x_1 x_2 \cdots x_m$ be a sequence, $x_i \in \Sigma$.

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

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 *subsequence* of X
if there are indexes $i_1 < i_2 < \cdots, < i_k$ of X such that

$z_j = x_{i_j}$ for $j = 1, \cdots, k$.

```
        34  6789 11 12 13
X =ACCGGTCGAGTGCG
        ||  ||||  |||
Z =     CG  TCGA  TGC
        12  3456  789
```

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 $j = 1, \cdots, k$.

```
        34  6789 11 12 13
X =ACCGGTCGAGTGCG
        ||  ||||  |||
Z =     CG  TCGA  TGC
        12  3456  789
```

the corresponding indexes in X are:

$$3 < 4 < 6 < 7 < 8 < 9 < 11 < 12 < 13$$

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

Z is a *common subsequence* of X and Y
if Z is a subsequence of X and Z is a subsequence of Y.

```
ACCGGTC GAGTGCG
  || || || |||
  CG TC GA TGC  ⟵ common sequence
  || || || |||
GTCGTTCGGA TGCCC
```

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.

```
ACCGGTC GAGTGCG
  || || || |||
  CG TC GA TGC  ⟵ common sequence
  || || || |||
GTCGTTCGGA TGCCC
```

<u>Longest Common Subsequence</u> (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 $length(Z)$ is the maximum.

Chapter 15. Dynamic Programming

Step 1: Analyze the LCS problem to see if it has the desired features:

Chapter 15. Dynamic Programming

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

```
ACCGGTC GAGTGCG
  || || || |||
  CG TC GA TGC
  || || || |||
GTCGTTCGGA TGCCC
```

Chapter 15. Dynamic Programming

Step 1: Analyze the LCS problem to see if it has the desired features:

- optimal substructure
- overlapping subproblems

```
ACCGGTC GAGTGCG
  || || || |||
  CG TC GA TGC
  || || || |||
GTCGTTCGGA TGCCC
```

We examine prefixes $x_1 x_2 \ldots x_i$, $i \leq m$
and $y_1 y_2 \ldots y_j$, $j \leq n$

Chapter 15. Dynamic Programming

Step 1: Analyze the LCS problem to see if it has the desired features:

- optimal substructure
- overlapping subproblems

```
ACCGGTC GAGTGCG
  || || || |||
  CG TC GA TGC
  || || || |||
GTCGTTCGGA TGCCC
```

We examine prefixes $x_1 x_2 \ldots x_i$, $i \leq m$
and $y_1 y_2 \ldots y_j$, $j \leq n$

- how is the LCS of $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_j$ related to the LCS of shorter prefixes of X and Y?

Chapter 15. Dynamic Programming

$$x_1 \ldots x_{i-1} x_i$$
$$y_1 \cdots y_{j-1} y_j$$

Chapter 15. Dynamic Programming

$$x_1 \ldots x_{i-1} x_i$$
$$y_1 \ldots y_{j-1} y_j$$

$$\boxed{x_1 \ldots x_{i-1}} x_i$$
$$\boxed{y_1 \ldots y_{j-1}} y_j$$

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

$$x_1 \ldots x_{i-1} x_i$$
$$y_1 \ldots y_{j-1} y_j$$

$$\boxed{x_1 \ldots x_{i-1}} x_i$$
$$\boxed{y_1 \ldots y_{j-1}} y_j$$

if $x_i = y_j$, include x_i in the LCS, reducing to LCS for $x_1 \ldots x_{i-1}$ and $y_1 \ldots y_{j-1}$

$$\boxed{x_1 \ldots x_{i-1} x_i}$$
$$\boxed{y_1 \ldots y_{j-1}} y_j$$

$$\boxed{x_1 \ldots x_{i-1}} x_i$$
$$\boxed{y_1 \ldots y_{j-1} y_j}$$

if $x_i \neq y_j$, should we abandon either?

Chapter 15. Dynamic Programming

$$x_1 \ldots x_{i-1} x_i$$
$$y_1 \ldots y_{j-1} y_j$$

$$\boxed{x_1 \ldots x_{i-1}} x_i$$
$$\boxed{y_1 \ldots y_{j-1}} y_j$$

if $x_i = y_j$, include x_i in the LCS, reducing to LCS for $x_1 \ldots x_{i-1}$ and $y_1 \ldots y_{j-1}$

$$\boxed{x_1 \ldots x_{i-1} x_i} \qquad \boxed{x_1 \ldots x_{i-1}} x_i$$
$$\boxed{y_1 \ldots y_{j-1}} y_j \qquad \boxed{y_1 \ldots y_{j-1} y_j}$$

if $x_i \neq y_j$, should we abandon either? two options

(1) keep x_i but abandon y_j
resulting in prefixes: $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_{j-1}$, or

Chapter 15. Dynamic Programming

$$x_1 \ldots x_{i-1} x_i$$
$$y_1 \ldots y_{j-1} y_j$$

$$\boxed{x_1 \ldots x_{i-1}} x_i$$
$$\boxed{y_1 \ldots y_{j-1}} y_j$$

if $x_i = y_j$, include x_i in the LCS, reducing to LCS for $x_1 \ldots x_{i-1}$ and $y_1 \ldots y_{j-1}$

$$\boxed{x_1 \ldots x_{i-1} x_i}$$
$$\boxed{y_1 \ldots y_{j-1}} y_j$$

$$\boxed{x_1 \ldots x_{i-1}} x_i$$
$$\boxed{y_1 \ldots y_{j-1} y_j}$$

if $x_i \neq y_j$, should we abandon either? two options

(1) keep x_i but abandon y_j
resulting in prefixes: $x_1 x_2 \ldots x_i$ 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 $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_j$, 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 $LCS(i, j) = concat(\, LCS(i-1, j-1), x_i)$

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 $LCS(i, j) = concat(\, LCS(i-1, j-1), x_i)$
- if 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.)

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 $LCS(i, j) = concat(\, LCS(i-1, j-1), x_i)$
- if 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.

Step 2: define objective function and formulate recurrence

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 $y_1 \ldots y_j$.

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

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 $y_1 \ldots y_j$. then

$$l(i,j) = \begin{cases} l(i-1,j-1)+1 & \text{if } x_i = y_j; \\ \max \begin{cases} l(i,j-1); \\ l(i-1,j); \end{cases} & \text{if } x_i \neq y_j \end{cases}$$

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 $y_1 \ldots y_j$. then

$$l(i,j) = \begin{cases} l(i-1,j-1)+1 & \text{if } x_i = y_j; \\ \max \begin{cases} l(i,j-1); \\ l(i-1,j); \end{cases} & \text{if } x_i \neq y_j \end{cases}$$

the value of $l(i,j)$ depends on values of $l(i-1,j-1)$, $l(i,j-1)$, and $l(i-1,j)$.

Step 2: define objective function and formulate recurrence

Define $l(i,j)$ to be the length of the LCS for prefixes $x_1 \dots x_i$ and $y_1 \dots y_j$. then

$$l(i,j) = \begin{cases} l(i-1,j-1)+1 & \text{if } x_i = y_j; \\ \max \begin{cases} l(i,j-1); \\ l(i-1,j); \end{cases} & \text{if } x_i \neq y_j \end{cases}$$

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

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 $x_1 \dots x_i$ and $y_1 \dots y_j$. then

$$l(i,j) = \begin{cases} l(i-1,j-1)+1 & \text{if } x_i = y_j; \\ \max \begin{cases} l(i,j-1); \\ l(i-1,j); \end{cases} & \text{if } x_i \neq y_j \end{cases}$$

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 $0 \leq j \leq n$

Chapter 15. Dynamic Programming

Step 3: bottom-up table building for function $l(i, j)$.

		G	C	C	C	T	A	G	C	G
G										
C										
G										
C										
A										
A										
T										
G										

Chapter 15. Dynamic Programming

Step 3: bottom-up table building for function $l(i, j)$.

		G	C	C	C	T	A	G	C	G
	0	0	0	0	0	0	0	0	0	0
G	0									
C	0									
G	0									
C	0									
A	0									
A	0									
T	0									
G	0									

Chapter 15. Dynamic Programming

Step 3: bottom-up table building for function $l(i, j)$.

		G	C	C	C	T	A	G	C	G
	0	0	0	0	0	0	0	0	0	0
G	0	1	1	1	1	1	1	1	1	1
C	0	1	2	2	2	2	2	2	2	2
G	0	1	2	2	2	2	2	3	3	3
C	0	1	2	3						
A	0									
A	0									
T	0									
G	0									

Chapter 15. Dynamic Programming

Step 3: bottom-up table building for function $l(i, j)$.

		G	C	C	C	T	A	G	C	G
	0	0	0	0	0	0	0	0	0	0
G	0	1	1	1	1	1	1	1	1	1
C	0	1	2	2	2	2	2	2	2	2
G	0	1	2	2	2	2	2	3	3	3
C	0	1	2	3						
A	0									
A	0									
T	0									
G	0									

LCS result between prefixes `GCCCTAGC` and `GCG`:

```
GCCCTAGC
G  C  G
```

Chapter 15. Dynamic Programming

Algorithm LCS(x, y)

1. $m = length(x)$, $n = length(y)$;

Chapter 15. Dynamic Programming

Algorithm LCS(x, y)

1. $m = length(x)$, $n = length(y)$;
2. **for** $i = 0$ **to** m

Chapter 15. Dynamic Programming

Algorithm LCS(x, y)

```
m = length(x), n = length(y);
for i = 0 to m
    T[i, 0] = 0
```

Chapter 15. Dynamic Programming

Algorithm LCS(x, y)

```
m = length(x), n = length(y);
for i = 0 to m
    T[i, 0] = 0
for j = 0 to n
```

Chapter 15. Dynamic Programming

Algorithm LCS(x, y)

```
m = length(x), n = length(y);
for i = 0 to m
    T[i, 0] = 0
for j = 0 to n
    T[0, j] = 0
```

Chapter 15. Dynamic Programming

Algorithm LCS(x, y)

```
m = length(x), n = length(y);
for i = 0 to m
    T[i, 0] = 0
for j = 0 to n
    T[0, j] = 0
for i = 1 to m
```

Chapter 15. Dynamic Programming

```
Algorithm LCS(x, y)
    m = length(x), n = length(y);
    for i = 0 to m
        T[i, 0] = 0
    for j = 0 to n
        T[0, j] = 0
    for i = 1 to m
        for j = 1 to n
```

Chapter 15. Dynamic Programming

```
Algorithm LCS(x, y)
    m = length(x), n = length(y);
    for i = 0 to m
        T[i, 0] = 0
    for j = 0 to n
        T[0, j] = 0
    for i = 1 to m
        for j = 1 to n
            if x_i = y_j
```

Chapter 15. Dynamic Programming

```
Algorithm LCS(x, y)
    m = length(x), n = length(y);
    for i = 0 to m
        T[i, 0] = 0
    for j = 0 to n
        T[0, j] = 0
    for i = 1 to m
        for j = 1 to n
            if x_i = y_j
                T[i, j] = T[i − 1, j − 1] + 1; P[i, j] = ‘ ↖ ‘
```

Chapter 15. Dynamic Programming

```
Algorithm LCS(x, y)
    m = length(x), n = length(y);
    for i = 0 to m
        T[i, 0] = 0
    for j = 0 to n
        T[0, j] = 0
    for i = 1 to m
        for j = 1 to n
            if x_i = y_j
                T[i, j] = T[i − 1, j − 1] + 1; P[i, j] = ‘ ↖ ‘
            else if T[i, j − 1] > T[i − 1, j]
```

Chapter 15. Dynamic Programming

```
Algorithm LCS(x, y)
    m = length(x), n = length(y);
    for i = 0 to m
        T[i, 0] = 0
    for j = 0 to n
        T[0, j] = 0
    for i = 1 to m
        for j = 1 to n
            if x_i = y_j
                T[i, j] = T[i − 1, j − 1] + 1; P[i, j] = ‘ ↖ ‘
            else if T[i, j − 1] > T[i − 1, j]
                        T[i, j] = T[i, j − 1]; P[i, j] = ‘ ← ‘
```

Chapter 15. Dynamic Programming

```
Algorithm LCS(x, y)
    m = length(x), n = length(y);
    for i = 0 to m
        T[i, 0] = 0
    for j = 0 to n
        T[0, j] = 0
    for i = 1 to m
        for j = 1 to n
            if x_i = y_j
                T[i, j] = T[i − 1, j − 1] + 1; P[i, j] = ‘ ↖ ‘
            else if T[i, j − 1] > T[i − 1, j]
                        T[i, j] = T[i, j − 1]; P[i, j] = ‘ ← ‘
                else T[i, j] = T[i − 1, j]; P[i, j] = ‘ ↑ ‘
```

Chapter 15. Dynamic Programming

```
Algorithm LCS(x, y)
    m = length(x), n = length(y);
    for i = 0 to m
        T[i, 0] = 0
    for j = 0 to n
        T[0, j] = 0
    for i = 1 to m
        for j = 1 to n
            if x_i = y_j
                T[i, j] = T[i − 1, j − 1] + 1; P[i, j] = ‘ ↖ ‘
            else if T[i, j − 1] > T[i − 1, j]
                        T[i, j] = T[i, j − 1]; P[i, j] = ‘ ← ‘
                else T[i, j] = T[i − 1, j]; P[i, j] = ‘ ↑ ‘
    return (T, P)
```

Chapter 15. Dynamic Programming

```
Algorithm LCS(x, y)
    m = length(x), n = length(y);
    for i = 0 to m
        T[i, 0] = 0
    for j = 0 to n
        T[0, j] = 0
    for i = 1 to m
        for j = 1 to n
            if x_i = y_j
                T[i, j] = T[i − 1, j − 1] + 1; P[i, j] = ‘ ↖ ‘
            else if T[i, j − 1] > T[i − 1, j]
                        T[i, j] = T[i, j − 1]; P[i, j] = ‘ ← ‘
                else T[i, j] = T[i − 1, j]; P[i, j] = ‘ ↑ ‘
    return (T, P)
```

Time complexity: $O(mn)$

Chapter 15. Dynamic Programming

Traceback the longest common subsequence

either recursive or iterative algorithm

Traceback the longest common subsequence

either recursive or iterative algorithm

Algorithm $\textsc{PrintLCS}(P, x, i, j)$

Chapter 15. Dynamic Programming

Traceback the longest common subsequence

either recursive or iterative algorithm

Algorithm $\textsc{PrintLCS}(P, x, i, j)$

1. **if** $(i > 0) \wedge (j > 0)$

Traceback the longest common subsequence

either recursive or iterative algorithm

Algorithm $\textsc{PrintLCS}(P, x, i, j)$

1. **if** $(i > 0) \wedge (j > 0)$
2. **if** $P[i, j] = '\nwarrow'$

Traceback the longest common subsequence

either recursive or iterative algorithm

Algorithm $\textsc{PrintLCS}(P, x, i, j)$

```
if (i > 0) ∧ (j > 0)
    if P[i, j] = '↖'
        PrintLCS(P, x, i − 1, j − 1)
```

Traceback the longest common subsequence

either recursive or iterative algorithm

Algorithm $\textsc{PrintLCS}(P, x, i, j)$

```
if (i > 0) ∧ (j > 0)
    if P[i, j] = '↖'
        PrintLCS(P, x, i − 1, j − 1)
        Print (x_i)
```

Traceback the longest common subsequence

either recursive or iterative algorithm

Algorithm $\textsc{PrintLCS}(P, x, i, j)$

```
if (i > 0) ∧ (j > 0)
    if P[i, j] = '↖'
        PrintLCS(P, x, i − 1, j − 1)
        Print (x_i)
    else if P[i, j] = '←'
```

Traceback the longest common subsequence

either recursive or iterative algorithm

Algorithm $\textsc{PrintLCS}(P, x, i, j)$

```
if (i > 0) ∧ (j > 0)
    if P[i, j] = '↖'
        PrintLCS(P, x, i − 1, j − 1)
        Print (x_i)
    else if P[i, j] = '←'
            PrintLCS(P, x, i, j − 1)
```

Traceback the longest common subsequence

either recursive or iterative algorithm

Algorithm $\textsc{PrintLCS}(P, x, i, j)$

```
if (i > 0) ∧ (j > 0)
    if P[i, j] = '↖'
        PrintLCS(P, x, i − 1, j − 1)
        Print (x_i)
    else if P[i, j] = '←'
            PrintLCS(P, x, i, j − 1)
        else PrintLCS(P, x, i − 1, j)
```

Traceback the longest common subsequence

either recursive or iterative algorithm

Algorithm $\textsc{PrintLCS}(P, x, i, j)$

```
if (i > 0) ∧ (j > 0)
    if P[i, j] = '↖'
        PrintLCS(P, x, i − 1, j − 1)
        Print (x_i)
    else if P[i, j] = '←'
            PrintLCS(P, x, i, j − 1)
        else PrintLCS(P, x, i − 1, j)
else return ()
```

Chapter 15. Dynamic Programming

Pairwise Alignment

Chapter 15. Dynamic Programming

Pairwise Alignment

```
ACCGGTCGAGTGCG
  CGTCGATGC   ⟵ common sequence
GTCGTTCGGATGCCC
```

Chapter 15. Dynamic Programming

Pairwise Alignment

```
ACCGGTCGAGTGCG
  CGTCGATGC   ⟵ common sequence
GTCGTTCGGATGCCC
```

biologically more meaningful view:

```
ACCGGTC GAGTGCG
  || || || |||
  CG TC GA TGC   ⟵ common sequence
  || || || |||
GTCGTTCGGA TGCCC
```

Pairwise Alignment

```
ACCGGTCGAGTGCG
  CGTCGATGC   ⟵ common sequence
GTCGTTCGGATGCCC
```

biologically more meaningful view:

```
ACCGGTC GAGTGCG
  || || || |||
  CG TC GA TGC   ⟵ common sequence
  || || || |||
GTCGTTCGGA TGCCC
```

blue letters are conserved;
red letters are mutated;
purple letters are inserted or deleted;

Chapter 15. Dynamic Programming

Alignment between two sequences:

Chapter 15. Dynamic Programming

Alignment between two sequences:

```
ACCGGTC_GAGTGCG_
  || || || |||
  CG TC GA TGC  ⟵ common sequence
  || || || |||
GTCGTTCGGA_TGCCC
```

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**;

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;

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

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

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)

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 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_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 score
$(-2)+(-2)+5+5+(-2)+5+5+(-6)+5+5+(-6)+5+5+5+(-2)+(-6)$
LCS is a special case

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 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

```
FOS_RAT     MMFSGFNADYEASSSRCSSASPAGDSLSYYHSPADSFSSMGSPVNTQDFCADLSVSSANF 60
FOS_MOUSE   MMFSGFNADYEASSSRCSSASPAGDSLSYYHSPADSFSSMGSPVNTQDFCADLSVSSANF 60
FOS_CHICK   MMYQGFAGEYEAPSSRCSSASPAGDSLTYYPSPADSFSSMGSPVNSQDFCTDLAVSSANF 60
FOSB_MOUSE  -MFQAFPGDYDS-GSRCSS-SPSAESQ--YLSSVDSFGSPPTAAASQE-CAGLGEMPGSF 54
FOSB_HUMAN  -MFQAFPGDYDS-GSRCSS-SPSAESQ--YLSSVDSFGSPPTAAASQE-CAGLGEMPGSF 54
Consensus    *:..* .:*:: .***** **:.:*   * *..***.*  :.. :*: *:.*.  ...*
```

- Homology reveals regulatory structure (E. Coli promoters)

```
tyr tRNA    TCTCAACGTAACACTTTACAGCGGCG··CGTCATTTGATATGATGC·GCCCCGCTTCCCGATAAGGG
rrn D1      GATCAAAAAAATACTTGTGCAAAAAA··TTGGGATCCCTATAATGCGCCTCCGTTGAGACGACAACG
rrn X1      ATGCATTTTTCCGCTTGTCTTCCTGA··GCCGACTCCCTATAATGCGCCTCCATCGACACGGCGGAT
rrn (DXE)2  CCTGAAATTCAGGGTTGACTCTGAAA··GAGGAAAGCGTAATATAC·GCCACCTCGCGACAGTGAGC
rrn E1      CTGCAATTTTTCTATTGCGGCCTGCG··GAGAACTCCCTATAATGCGCCTCCATCGACACGGCGGAT
rrn A1      TTTTAAATTTCCTCTTGTCAGGCCGG··AATAACTCCCTATAATGCGCCACCACTGACACGGAACAA
rrn A2      GCAAAAATAAATGCTTGACTCTGTAG··CGGGAAGGCGTATTATGC·ACACCCCGCGCCGCTGAGAA
λ PR        TAACACCGTGCGTGTTGACTATTTTA·CCTCTGGCGGTGATAATGG··TTGCATGTACTAAGGAGGT
λ PL        TATCTCTGGCGGTGTTGACATAAATA·CCACTGGCGGTGATACTGA··GCACATCAGCAGGACGCAC
T7 A3       GTGAAACAAAACGGTTGACAACATGA·AGTAAACACGGTACGATGT·ACCACATGAAACGACAGTGA
T7 A1       TATCAAAAAGAGTATTGACTTAAAGT·CTAACCTATAGGATACTTA·CAGCCATCGAGAGGGACACG
T7 A2       ACGAAAAACAGGTATTGACAACATGAAGTAACATGCAGTAAGATAC·AAATCGCTAGGTAACACTAG
fd VIII     GATACAAATCTCCGTTGTACTTTGTT··TCGCGCTTGGTATAATCG·CTGGGGGTCAAAGATGAGTG
                          −35                        −10                  +1
```

Chapter 15. Dynamic Programming

Pairwise alignement problem:

INPUT: sequences $x = x_1 x_2 \ldots x_m$, $y = y_1 y_2 \ldots y_n$

OUTPUT: x' and y', which are x and y padded with '_'s, respectively such that $\sum_{i=1}^{l} \mathcal{S}(x'_i, y'_i)$ is the maximum, where $l = |x'| = |y'|$ for some given scoring function $\mathcal{S}$.

How to solve it?

Chapter 15. Dynamic Programming

Pairwise alignement problem:

INPUT: sequences $x = x_1 x_2 \ldots x_m$, $y = y_1 y_2 \ldots y_n$

OUTPUT: x' and y', which are x and y padded with '_'s, respectively such that $\sum_{i=1}^{l} \mathcal{S}(x'_i, y'_i)$ is the maximum, where $l = |x'| = |y'|$ for some given scoring function $\mathcal{S}$.

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 alignement problem:

INPUT: sequences $x = x_1 x_2 \ldots x_m$, $y = y_1 y_2 \ldots y_n$

OUTPUT: x' and y', which are x and y padded with '_'s, respectively such that $\sum_{i=1}^{l} \mathcal{S}(x'_i, y'_i)$ is the maximum, where $l = |x'| = |y'|$ for some given scoring function $\mathcal{S}$.

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) $\begin{array}{l} x_1 \ldots x_{i-1} x_i \\ y_1 \ldots y_{j-1} y_j \end{array}$ (2) $\begin{array}{l} x_1 \ldots x_{i-1} x_i \\ y_1 \ldots y_{j-1} y_j \end{array}$ (3) $\begin{array}{l} x_1 \ldots x_{i-1} x_i \\ y_1 \ldots y_{j-1} y_j \end{array}$

Chapter 15. Dynamic Programming

Pairwise alignement problem:

INPUT: sequences $x = x_1 x_2 \ldots x_m$, $y = y_1 y_2 \ldots y_n$

OUTPUT: x' and y', which are x and y padded with '_'s, respectively such that $\sum_{i=1}^{l} \mathcal{S}(x'_i, y'_i)$ is the maximum, where $l = |x'| = |y'|$ for some given scoring function $\mathcal{S}$.

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) $\begin{array}{l} x_1 \ldots x_{i-1} x_i \\ y_1 \ldots y_{j-1} y_j \end{array}$ (2) $\begin{array}{l} x_1 \ldots x_{i-1} x_i \\ y_1 \ldots y_{j-1} y_j \end{array}$ (3) $\begin{array}{l} x_1 \ldots x_{i-1} x_i \\ y_1 \ldots y_{j-1} y_j \end{array}$

case (1) contributes to a match/mismatch score

Chapter 15. Dynamic Programming

Pairwise alignement problem:

INPUT: sequences $x = x_1 x_2 \ldots x_m$, $y = y_1 y_2 \ldots y_n$

OUTPUT: x' and y', which are x and y padded with '_'s, respectively such that $\sum_{i=1}^{l} \mathcal{S}(x'_i, y'_i)$ is the maximum, where $l = |x'| = |y'|$ for some given scoring function $\mathcal{S}$.

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) $\begin{array}{l} x_1 \ldots x_{i-1} x_i \\ y_1 \ldots y_{j-1} y_j \end{array}$ (2) $\begin{array}{l} x_1 \ldots x_{i-1} x_i \\ y_1 \ldots y_{j-1} y_j \end{array}$ (3) $\begin{array}{l} x_1 \ldots x_{i-1} x_i \\ y_1 \ldots y_{j-1} y_j \end{array}$

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

Chapter 15. Dynamic Programming

Scoring function for pairwise alignment

Eexample (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

Eexample (1)

+5 for conservation columns, (reward)
-2 for substitution columns, (penalty)
-6 for delete/insert columns, (penalty)

Example (2)

	A	C	G	T	-
A	5	-1	-2	-1	-4
C	-1	5	-3	-2	-4
G	-2	-3	5	-2	-4
T	-1	-2	-2	5	-4
-	-4	-4	-4	-4	*

Chapter 15. Dynamic Programming

Scoring function for pairwise alignment

Eexample (1)

+5 for conservation columns, (reward)
-2 for substitution columns, (penalty)
-6 for delete/insert columns, (penalty)

Example (2)

	A	C	G	T	-
A	5	-1	-2	-1	-4
C	-1	5	-3	-2	-4
G	-2	-3	5	-2	-4
T	-1	-2	-2	5	-4
-	-4	-4	-4	-4	*

Why sum of column scores?

Chapter 15. Dynamic Programming

Scoring function for pairwise alignment

Eexample (1)

+5 for conservation columns, (reward)
-2 for substitution columns, (penalty)
-6 for delete/insert columns, (penalty)

Example (2)

	A	C	G	T	-
A	5	-1	-2	-1	-4
C	-1	5	-3	-2	-4
G	-2	-3	5	-2	-4
T	-1	-2	-2	5	-4
-	-4	-4	-4	-4	*

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

$$A(i,j) = \begin{cases} concat(A(i-1, j-1), \begin{smallmatrix} x_i \\ y_i \end{smallmatrix}) & \text{if this has the longest length;} \\ \\ \end{cases}$$

Chapter 15. Dynamic Programming

As for the LCS problem, we can define

$A(i,j)$ is the maximum score alignment between $x_1 \dots x_i$ and $y_1 \dots y_j$, then

$$A(i,j) = \begin{cases} concat(A(i-1,j-1), \begin{smallmatrix} x_i \\ y_i \end{smallmatrix}) & \text{if this has the longest length;} \\ contat(A(i,j-1), \begin{smallmatrix} - \\ y_j \end{smallmatrix}) & \text{if this has the longest length;} \\ \end{cases}$$

Chapter 15. Dynamic Programming

As for the LCS problem, we can define

$A(i,j)$ is the maximum score alignment between $x_1 \dots x_i$ and $y_1 \dots y_j$, then

$$A(i,j) = \begin{cases} concat(A(i-1,j-1), \begin{smallmatrix} x_i \\ y_i \end{smallmatrix}) & \text{if this has the longest length;} \\ contat(A(i,j-1), \begin{smallmatrix} - \\ y_j \end{smallmatrix}) & \text{if this has the longest length;} \\ contat(A(i-1,j), \begin{smallmatrix} x_i \\ - \end{smallmatrix}) & \text{if this has the longest length;} \end{cases}$$

Chapter 15. Dynamic Programming

As for the LCS problem, we can define

$A(i,j)$ is the maximum score alignment between $x_1 \dots x_i$ and $y_1 \dots y_j$, then

$$A(i,j) = \begin{cases} concat(A(i-1,j-1), \begin{smallmatrix} x_i \\ y_i \end{smallmatrix}) & \text{if this has the longest length;} \\ contat(A(i,j-1), \begin{smallmatrix} - \\ y_j \end{smallmatrix}) & \text{if this has the longest length;} \\ contat(A(i-1,j), \begin{smallmatrix} x_i \\ - \end{smallmatrix}) & \text{if this has the longest length;} \end{cases}$$

Instead, we define objective function $S(i,j)$ to be the maximum alignment score between $x_1 \dots x_i$ and $y_1 \dots y_j$

$$S(i,j) = \max \begin{cases} S(i-1,j-1) + score(x_i, y_j) & i \geq 1, j \geq 1; \\ \\ \end{cases}$$

Chapter 15. Dynamic Programming

As for the LCS problem, we can define

$A(i,j)$ is the maximum score alignment between $x_1 \dots x_i$ and $y_1 \dots y_j$, then

$$A(i,j) = \begin{cases} concat(A(i-1,j-1), \begin{smallmatrix} x_i \\ y_i \end{smallmatrix}) & \text{if this has the longest length;} \\ contat(A(i,j-1), \begin{smallmatrix} - \\ y_j \end{smallmatrix}) & \text{if this has the longest length;} \\ contat(A(i-1,j), \begin{smallmatrix} x_i \\ - \end{smallmatrix}) & \text{if this has the longest length;} \end{cases}$$

Instead, we define objective function $S(i,j)$ to be the maximum alignment score between $x_1 \dots x_i$ and $y_1 \dots y_j$

$$S(i,j) = \max \begin{cases} S(i-1,j-1) + score(x_i, y_j) & i \geq 1, j \geq 1; \\ S(i,j-1) + score(-, y_j) & j \geq 1; \end{cases}$$

Chapter 15. Dynamic Programming

As for the LCS problem, we can define

$A(i,j)$ is the maximum score alignment between $x_1 \dots x_i$ and $y_1 \dots y_j$, then

$$A(i,j) = \begin{cases} concat(A(i-1,j-1), \begin{smallmatrix} x_i \\ y_i \end{smallmatrix}) & \text{if this has the longest length;} \\ contat(A(i,j-1), \begin{smallmatrix} - \\ y_j \end{smallmatrix}) & \text{if this has the longest length;} \\ contat(A(i-1,j), \begin{smallmatrix} x_i \\ - \end{smallmatrix}) & \text{if this has the longest length;} \end{cases}$$

Instead, we define objective function $S(i,j)$ to be the maximum alignment score between $x_1 \dots x_i$ and $y_1 \dots y_j$

$$S(i,j) = \max \begin{cases} S(i-1,j-1) + score(x_i, y_j) & i \geq 1, j \geq 1; \\ S(i,j-1) + score(-, y_j) & j \geq 1; \\ S(i-1,j) + score(x_i, -) & i \geq 1; \end{cases}$$

scoring function $score$

	A	C	G	T	-
A	5	-1	-2	-1	-4
C	-1	5	-3	-2	-4
G	-2	-3	5	-2	-4
T	-1	-2	-2	5	-4
-	-4	-4	-4	-4	*

Chapter 15. Dynamic Programming

As for the LCS problem, we can define

$A(i,j)$ is the maximum score alignment between $x_1 \dots x_i$ and $y_1 \dots y_j$, then

$$A(i,j) = \begin{cases} concat(A(i-1,j-1), \begin{smallmatrix} x_i \\ y_i \end{smallmatrix}) & \text{if this has the longest length;} \\ contat(A(i,j-1), \begin{smallmatrix} - \\ y_j \end{smallmatrix}) & \text{if this has the longest length;} \\ contat(A(i-1,j), \begin{smallmatrix} x_i \\ - \end{smallmatrix}) & \text{if this has the longest length;} \end{cases}$$

Instead, we define objective function $S(i,j)$ to be the maximum alignment score between $x_1 \dots x_i$ and $y_1 \dots y_j$

$$S(i,j) = \max \begin{cases} S(i-1,j-1) + score(x_i, y_j) & i \geq 1, j \geq 1; \\ S(i,j-1) + score(-, y_j) & j \geq 1; \\ S(i-1,j) + score(x_i, -) & i \geq 1; \end{cases}$$

scoring function $score$

	A	C	G	T	-
A	5	-1	-2	-1	-4
C	-1	5	-3	-2	-4
G	-2	-3	5	-2	-4
T	-1	-2	-2	5	-4
-	-4	-4	-4	-4	*

exercise: Table filling and traceback

Chapter 15. Dynamic Programming

Affine gap penalty: a challenge for DP

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

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 $op > ext$.

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:

$$S(i,j) = \max \begin{cases} S(i-1,j-1) + score(x_i, y_j) \\ \\ \end{cases}$$

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

penalty $= -op - ext(l-1)$ for a gap covering l positions, where $op > ext$.

how to modify recurrences:

$$S(i,j) = \max \begin{cases} S(i-1,j-1) + score(x_i, y_j) \\ S(i,j-1) + score(-, y_j) \\ \end{cases}$$

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

penalty $= -op - ext(l-1)$ for a gap covering l positions, where $op > ext$.

how to modify recurrences:

$$S(i,j) = \max \begin{cases} S(i-1,j-1) + score(x_i, y_j) \\ S(i,j-1) + score(-, y_j) \\ S(i-1,j) + score(x_i, -) \end{cases}$$

Chapter 15. Dynamic Programming

Issue: we do not know how long a gap is.

$$S(i,j) = \max \begin{cases} S(i-1,j-1) + score(x_i, y_j) \\ S(i,j-1) + score(-, y_j) \\ S(i-1,j) + score(x_i, -) \end{cases}$$

Solution 1: we can try all different length l

$$S(i,j) = \max \begin{cases} S(i-1,j-1) + score(x_i, y_j) \\ \\ \end{cases}$$

Chapter 15. Dynamic Programming

Issue: we do not know how long a gap is.

$$S(i,j) = \max \begin{cases} S(i-1,j-1) + score(x_i, y_j) \\ S(i,j-1) + score(-, y_j) \\ S(i-1,j) + score(x_i, -) \end{cases}$$

Solution 1: we can try all different length l

$$S(i,j) = \max \begin{cases} S(i-1,j-1) + score(x_i, y_j) \\ \max_{1 \le l \le j} S(i, j-l) - op - (l-1) \times ext \\ \end{cases}$$

Chapter 15. Dynamic Programming

Issue: we do not know how long a gap is.

$$S(i,j) = \max \begin{cases} S(i-1,j-1) + score(x_i, y_j) \\ S(i,j-1) + score(-, y_j) \\ S(i-1,j) + score(x_i, -) \end{cases}$$

Solution 1: we can try all different length l

$$S(i,j) = \max \begin{cases} S(i-1,j-1) + score(x_i, y_j) \\ \max_{1 \le l \le j} S(i, j-l) - op - (l-1) \times ext \\ \max_{1 \le l \le i} S(i-l, j) - op - (l-1) \times ext \end{cases}$$

Chapter 15. Dynamic Programming

Issue: we do not know how long a gap is.

$$S(i,j) = \max \begin{cases} S(i-1,j-1) + score(x_i, y_j) \\ S(i,j-1) + score(-, y_j) \\ S(i-1,j) + score(x_i, -) \end{cases}$$

Solution 1: we can try all different length l

$$S(i,j) = \max \begin{cases} S(i-1,j-1) + score(x_i, y_j) \\ \max_{1 \le l \le j} S(i, j-l) - op - (l-1) \times ext \\ \max_{1 \le l \le i} S(i-l, j) - op - (l-1) \times ext \end{cases}$$

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.

$$S(i,j) = \max \begin{cases} S(i-1,j-1) + score(x_i, y_j) \\ S(i,j-1) + score(-, y_j) \\ S(i-1,j) + score(x_i, -) \end{cases}$$

Solution 1: we can try all different length l

$$S(i,j) = \max \begin{cases} S(i-1,j-1) + score(x_i, y_j) \\ \max_{1 \le l \le j} S(i, j-l) - op - (l-1) \times ext \\ \max_{1 \le l \le i} S(i-l, j) - op - (l-1) \times ext \end{cases}$$

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.

$$S(i,j) = \max \begin{cases} S(i-1,j-1) + score(x_i, y_j) \\ S(i,j-1) + score(-, y_j) \\ S(i-1,j) + score(x_i, -) \end{cases}$$

Solution 1: we can try all different length l

$$S(i,j) = \max \begin{cases} S(i-1,j-1) + score(x_i, y_j) \\ \max_{1 \le l \le j} S(i, j-l) - op - (l-1) \times ext \\ \max_{1 \le l \le i} S(i-l, j) - op - (l-1) \times ext \end{cases}$$

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.

$$S(i,j) = \max \begin{cases} S(i-1,j-1) + score(x_i, y_j) \\ S(i,j-1) + score(-, y_j) \\ S(i-1,j) + score(x_i, -) \end{cases}$$

Solution 1: we can try all different length l

$$S(i,j) = \max \begin{cases} S(i-1,j-1) + score(x_i, y_j) \\ \max_{1 \le l \le j} S(i, j-l) - op - (l-1) \times ext \\ \max_{1 \le l \le i} S(i-l, j) - op - (l-1) \times ext \end{cases}$$

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

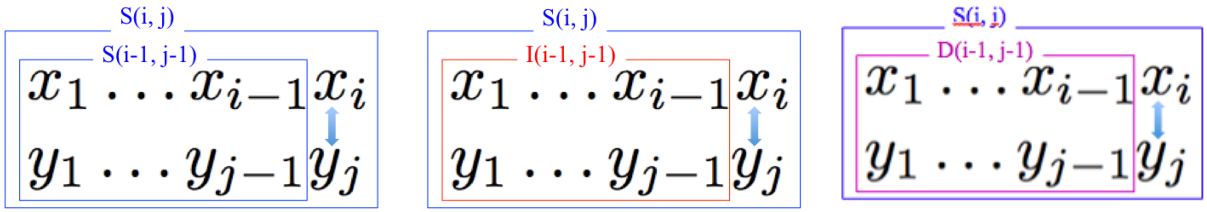
So we want to do recursion "one step at a time":

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

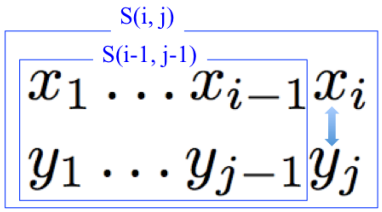


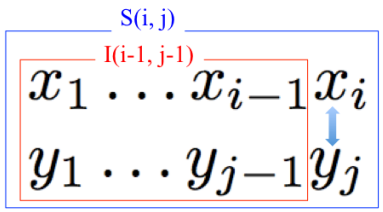
Chapter 15. Dynamic Programming

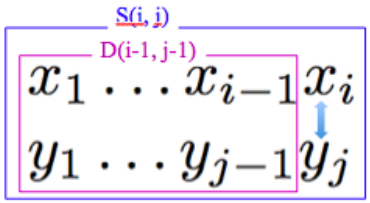
So we want to do recursion "one step at a time":

but at the same time, differentiate different "states"
where the two tails are

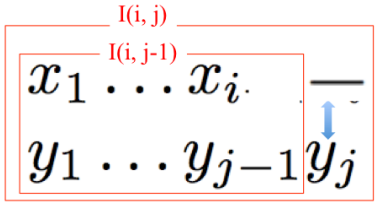
Substitute

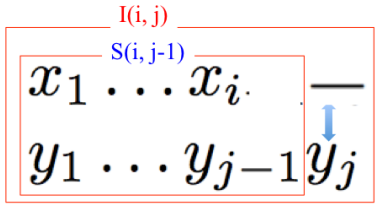






Insertion



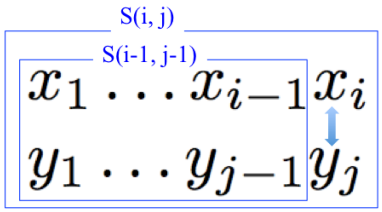


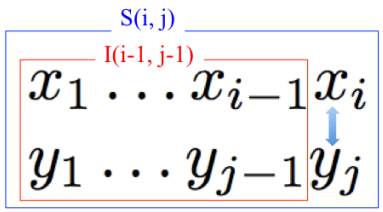
Chapter 15. Dynamic Programming

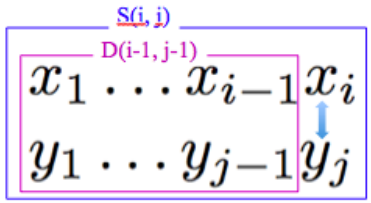
So we want to do recursion "one step at a time":

but at the same time, differentiate different "states"
where the two tails are

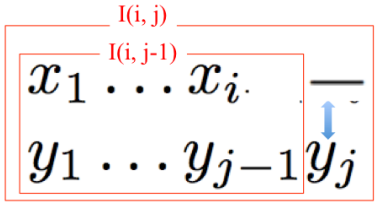
Substitute

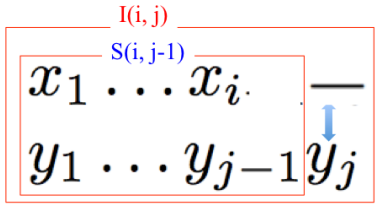




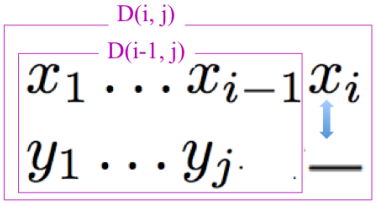


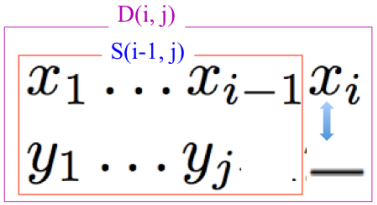
Insertion





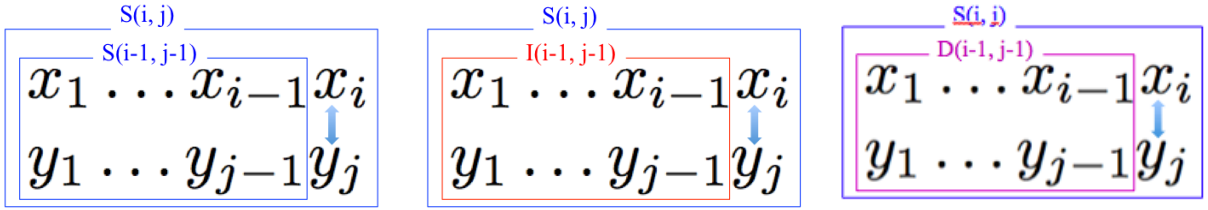
Deletion





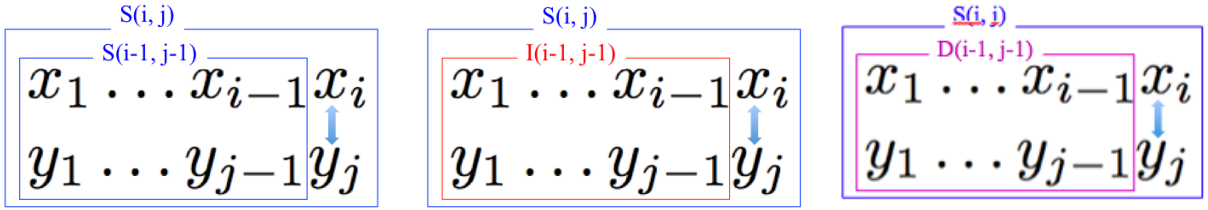
Chapter 15. Dynamic Programming

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.



Chapter 15. Dynamic Programming

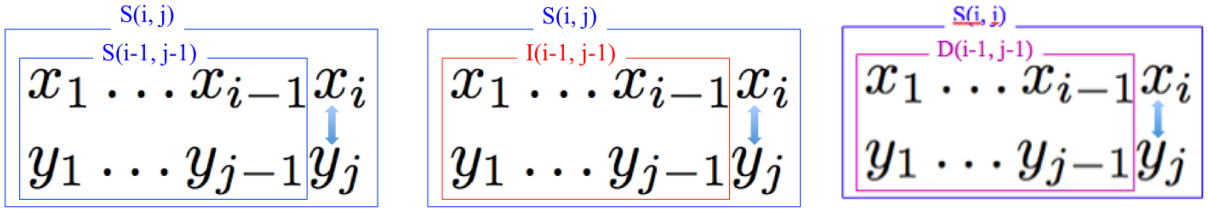
Define $S(i,j)$ to be the maximum score of alignment between $x_1 \ldots x_i$ and $y_1 \ldots y_j$ with x_i being aligned to y_j.



$$S(i,j) = \max \begin{cases} S(i-1,j-1) + score(x_i, y_j) \\ I(i-1,j-1) + score(x_i, y_j) \\ D(i-1,j-1) + score(x_i, y_j) \end{cases}$$

Chapter 15. Dynamic Programming

Define $S(i,j)$ to be the maximum score of alignment between $x_1 \dots x_i$ and $y_1 \dots y_j$ with x_i being aligned to y_j.

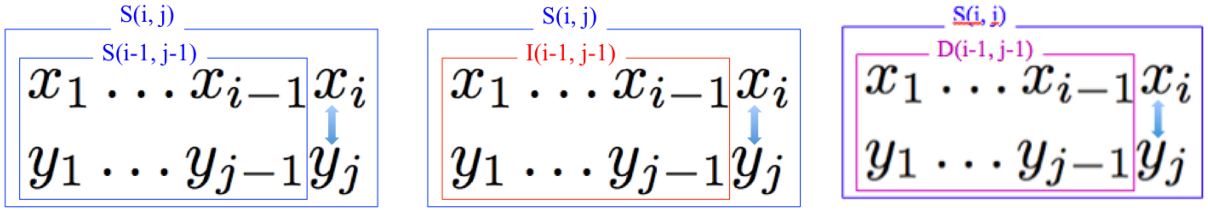


$$S(i,j) = \max \begin{cases} S(i-1,j-1) + score(x_i, y_j) \\ I(i-1,j-1) + score(x_i, y_j) \\ D(i-1,j-1) + score(x_i, y_j) \end{cases}$$

Define $I(i,j)$ to be the maximum score of alignment between $x_1 \dots x_i$ and $y_1 \dots y_j$ with y_j being insertion.

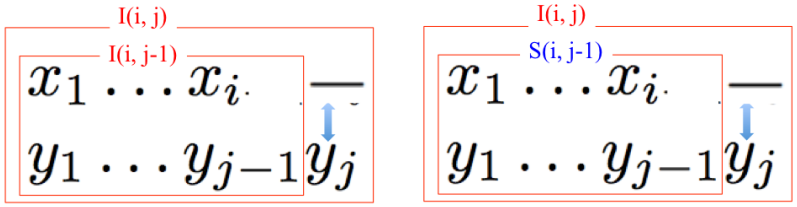
Chapter 15. Dynamic Programming

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.



$$S(i,j) = \max \begin{cases} S(i-1,j-1) + score(x_i, y_j) \\ I(i-1,j-1) + score(x_i, y_j) \\ D(i-1,j-1) + score(x_i, y_j) \end{cases}$$

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.



$$I(i,j) = \max \begin{cases} I(i,j-1) - ext \\ S(i,j-1) - op \end{cases}$$

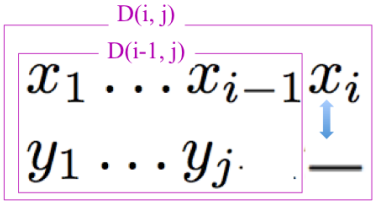
Chapter 15. Dynamic Programming

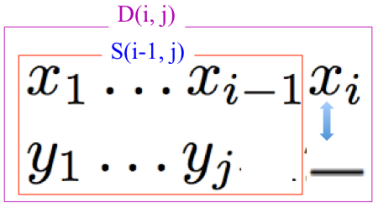
Chapter 15. Dynamic Programming

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.

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.

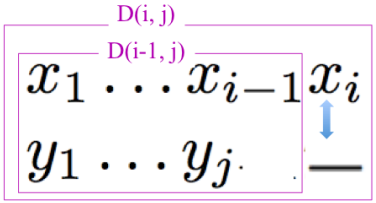


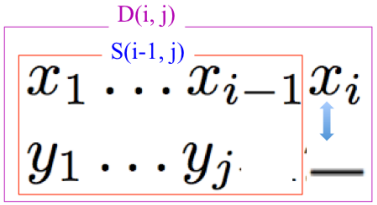


$$D(i, j) = \max \begin{cases} D(i-1, j) - ext \\ S(i-1, j) - op \end{cases}$$

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.



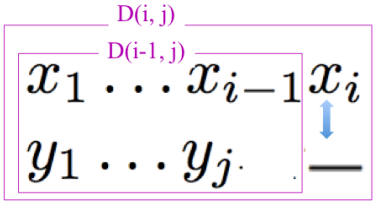


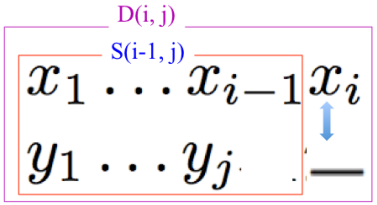
$$D(i, j) = \max \begin{cases} D(i-1, j) - ext \\ S(i-1, j) - op \end{cases}$$

- 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.





$$D(i, j) = \max \begin{cases} D(i-1, j) - ext \\ S(i-1, j) - op \end{cases}$$

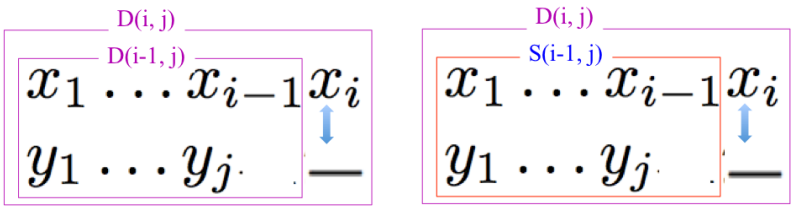
- Note that there is no deletion right after insertion, neither insertion right after deletion. why?

```
ACA_GTA
ACCT__A
```

```
ACAGTA
ACCT_A
```

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.



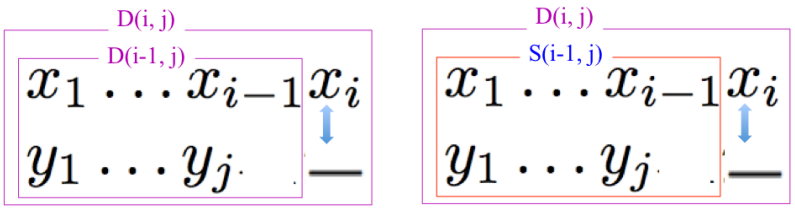
$$D(i,j) = \max \begin{cases} D(i-1,j) - ext \\ S(i-1,j) - op \end{cases}$$

- 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.



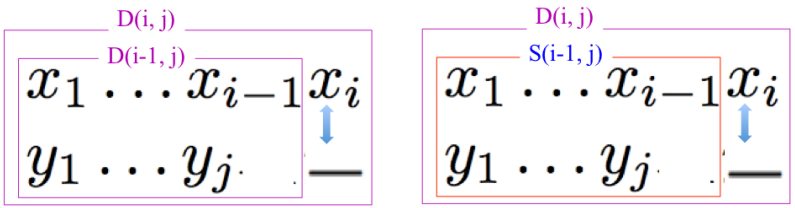
$$D(i,j) = \max \begin{cases} D(i-1,j) - ext \\ S(i-1,j) - op \end{cases}$$

- Note that there is no deletion right after insertion, neither insertion right after deletion. why?

- What is the desired value ? $\max \begin{cases} S(m,n) \\ I(m,n) \\ D(m,n) \end{cases}$

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.



$$D(i,j) = \max \begin{cases} D(i-1,j) - ext \\ S(i-1,j) - op \end{cases}$$

- Note that there is no deletion right after insertion, neither insertion right after deletion. why?

```
ACA_GTA
ACCT__A
```

```
ACAGTA
ACCT_A
```

- What is the desired value ? $\max \begin{cases} S(m,n) \\ I(m,n) \\ D(m,n) \end{cases}$
- How to fill the table(s) and to traceback the optimal alignment?

Chapter 15. Dynamic Programming

DP applied to summation problems

DP applied to summation problems

Unmatched Socks Problem

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.

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 $n = 5$, $k = 1$

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 $n = 5$, $k = 1$

$$P_1 =$$

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 $n = 5$, $k = 1$

$$P_1 = \frac{\text{number of color-not-matched pairs}}{\text{number of total pairs}}$$

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 $n = 5$, $k = 1$

$$P_1 = \frac{\text{number of color-not-matched pairs}}{\text{number of total pairs}} = \frac{(10 \times 8)/2}{}$$

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 $n = 5$, $k = 1$

$$P_1 = \frac{\text{number of color-not-matched pairs}}{\text{number of total pairs}} = \frac{(10 \times 8)/2}{\binom{10}{2}} =$$

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 $n = 5$, $k = 1$

$$P_1 = \frac{\text{number of color-not-matched pairs}}{\text{number of total pairs}} = \frac{(10 \times 8)/2}{\binom{10}{2}} = \frac{40}{45} =$$

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 $n = 5$, $k = 1$

$$P_1 = \frac{\text{number of color-not-matched pairs}}{\text{number of total pairs}} = \frac{(10 \times 8)/2}{\binom{10}{2}} = \frac{40}{45} = \frac{8}{9}$$

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 $n = 5$, $k = 2$

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 $n = 5$, $k = 2$

$$P_2 = P_1$$

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 $n = 5$, $k = 2$

$$P_2 = P_1 \times (P_u + P_m + P_{u,m}) =$$

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 $n = 5$, $k = 2$

$$P_2 = P_1 \times (P_u + P_m + P_{u,m}) = \frac{8}{9} \times \left(\frac{1}{\binom{8}{2}} + \frac{(6 \times 4)/2}{\binom{8}{2}} + \frac{2 \times 6}{\binom{8}{2}} \right) =$$

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 $n = 5$, $k = 2$

$$P_2 = P_1 \times (P_u + P_m + P_{u,m}) = \frac{8}{9} \times \left(\frac{1}{\binom{8}{2}} + \frac{(6 \times 4)/2}{\binom{8}{2}} + \frac{2 \times 6}{\binom{8}{2}} \right) = \frac{8}{9} \times \frac{25}{28}$$

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 $n = 5$, $k = 2$

$$P_2 = P_1 \times (P_u + P_m + P_{u,m}) = \frac{8}{9} \times \left(\frac{1}{\binom{8}{2}} + \frac{(6 \times 4)/2}{\binom{8}{2}} + \frac{2 \times 6}{\binom{8}{2}} \right) = \frac{8}{9} \times \frac{25}{28}$$

$$P_3 = P_2 \times ??$$

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 kth 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 kth day.

Let u_k be the number of unmatched socks *after* selection on the kth 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 kth day.

Let u_k be the number of unmatched socks *after* selection on the kth day.

Note that

$$2m_k + u_k = 2n - 2k, \quad \text{for every } k \leq n$$

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 kth day.

Let u_k be the number of unmatched socks *after* selection on the kth 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

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 kth day.

Let u_k be the number of unmatched socks *after* selection on the kth 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

$$P_k(m_k, u_k) = \sum \begin{cases} P_{k-1}(m_k + 2, u_k - 2) \times \frac{(m_k+2)(2m_k+2)}{\binom{2m_k+2+u_k}{2}} \\ P_{k-1}(m_k + 1, u_k) \times \frac{2(m_k+1)u_k}{\binom{2m_k+2+u_k}{2}} \\ P_{k-1}(m_k, u_k + 2) \times \frac{\binom{u_k+2}{2}}{\binom{2m_k+2+u_k}{2}} \end{cases}$$

Chapter 15. Dynamic Programming

Knapsack Problem

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

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

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:

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

- 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

- either increase value by v_n, reduce available space from B to B-s_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

- 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

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

- either increase value by v_n, reduce available space from B to B-s_n, reduce the number of items by 1,
- 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;

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

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

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

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

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

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,

$$V(0, X) = 0,$$

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,

$$V(0, X) = 0, \; V(k, 0) = 0$$

- What does the table look like?

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,

$$V(0, X) = 0, \; V(k, 0) = 0$$

- What does the table look like? How to fill it out?

Chapter 15. Dynamic Programming

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

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

base case,

$$V(0, X) = 0, \; V(k, 0) = 0$$

- 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

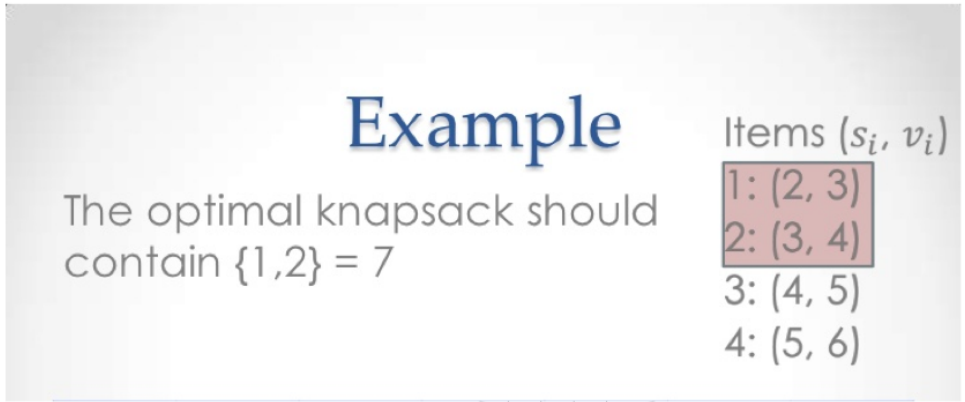
Items (s_i, v_i)

1: (2, 3)
2: (3, 4)
3: (4, 5)
4: (5, 6)

i\s	0	1	2	3	4	5
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

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

Chapter 15. Dynamic Programming



i\s	0	1	2	3	4	5
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

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

- 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

We have seen a number of examples with "forward DP" solutions.

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).

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.

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.

<u>Matrix Chain Multiplication problem</u>

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.

<u>Matrix Chain Multiplication problem</u>

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

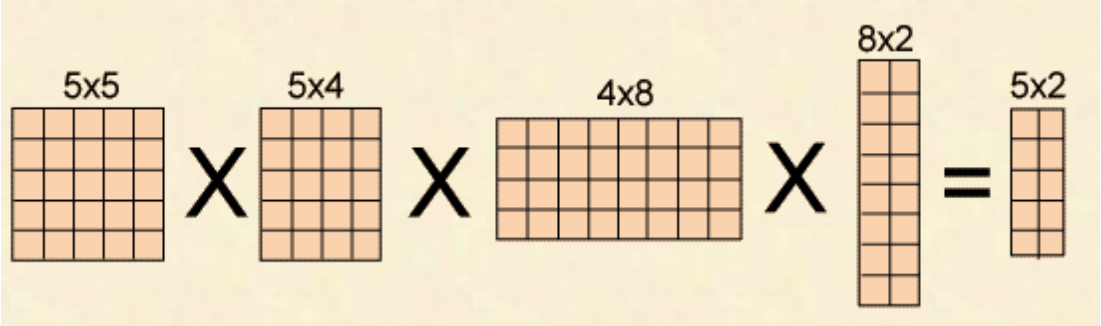
We have seen a number of examples with "forward DP" solutions.

(Most of these problems are solvable with "backward DP" as well).

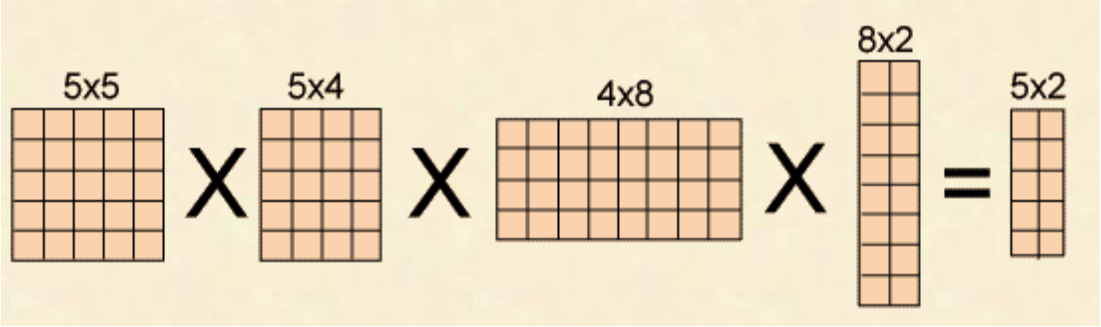
We now consider an example with an "inside-out DP" solution.

Matrix Chain Multiplication problem

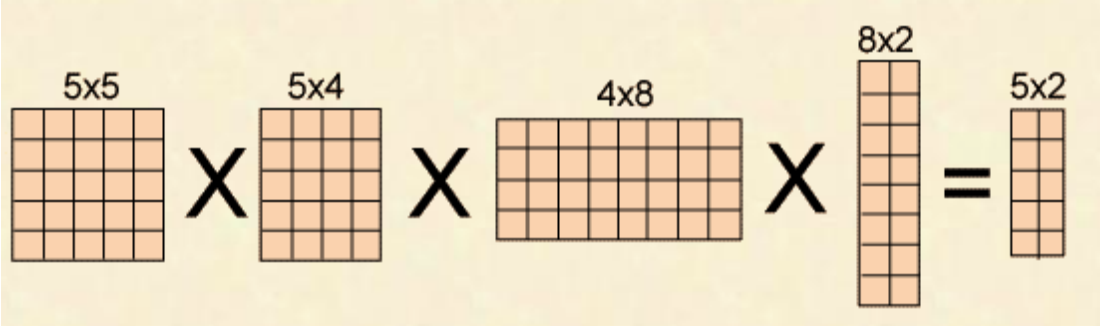
INPUT: matrices $A_1, \cdots, A_n$, where A_i has dimension $p_{i-1} \times p_i$;
OUTPUT: a parenthesization by which the product $A_1 \times A_2 \times \cdots \times A_n$ uses the minimum number of scalar multiplications.



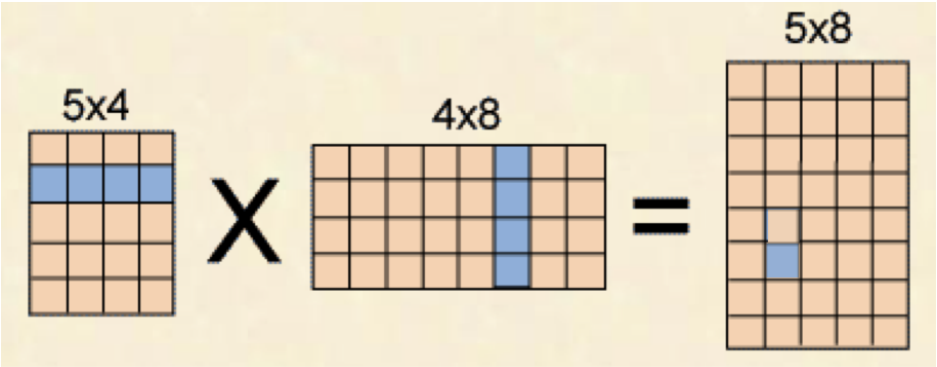
Chapter 15. Dynamic Programming



Chapter 15. Dynamic Programming



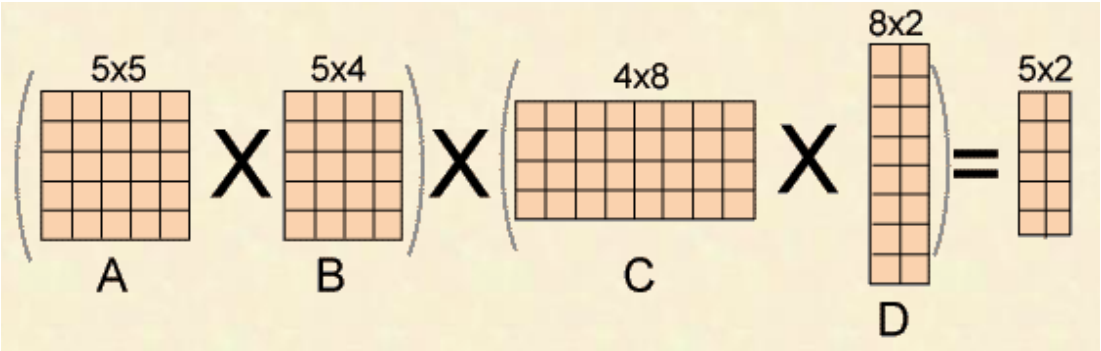
scalable multiplications



$$= 5 \times 4 \times 8$$

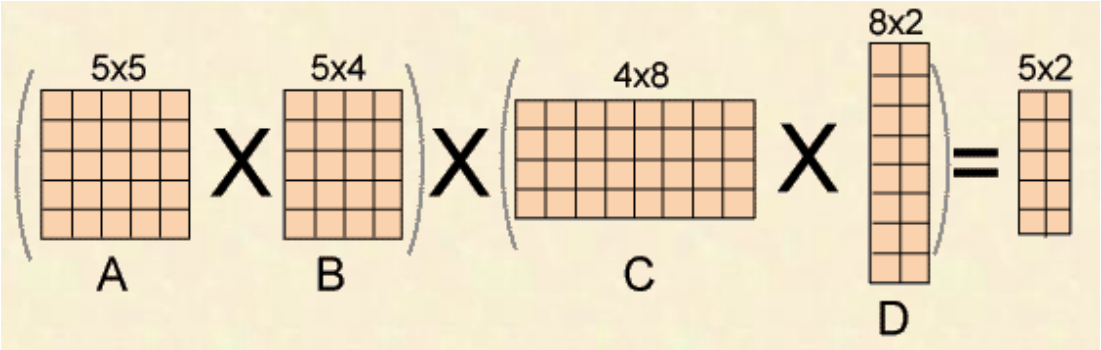
Chapter 15. Dynamic Programming

Many possible parenthesizations



Chapter 15. Dynamic Programming

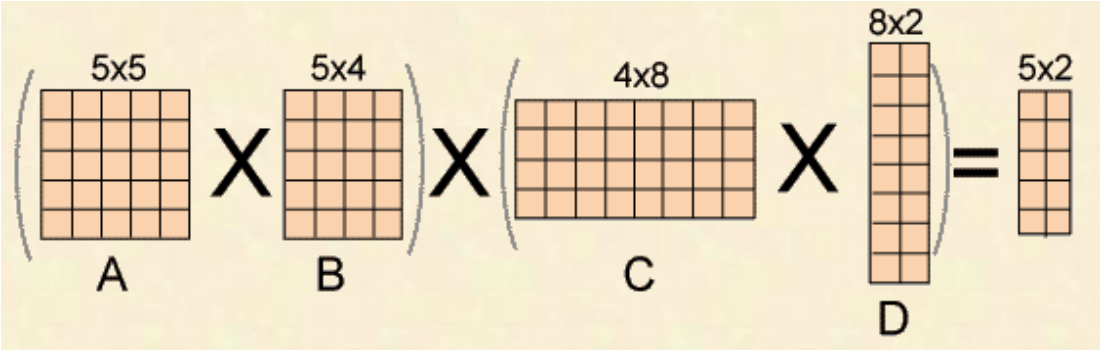
Many possible parenthesizations



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



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

Note: The number of all possible parenthesizations is

$$P(n) = \Sigma_{k=1}^{n-1} P(k)P(n-k), \text{ Catalan number}$$

Chapter 15. Dynamic Programming

Step 1: identify *optimal substructure*:
Optimal solution in terms of optimal solutions to subproblems:

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

$$(A_i \cdots A_k)(A_{k+1} \cdots A_j)$$

$k = i, i+1, \cdots, j-1$.

(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

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

$$f(i,j) = \min_{i \le k < j} \{f(i,k) + f(k+1,j) + p_{i-1}p_k p_j\}$$

where base case is

$f(i,j) = 0$ when $i = j$, 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

	1	2	3	4	5	6
1	0 _	20 1	35 2	53 3	95 4	99 1
2		0 _	60 2	132 3	270 3	95 2
3			0 _	90 3	231 3	75 3
4				0 _	126 4	60 4
5					0 _	42 5
6						0 _

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 $\textsc{MatrixChainOrder}(p)$

1. $n = length[p] - 1;$

Chapter 15. Dynamic Programming

Algorithm $\textsc{MatrixChainOrder}(p)$

1. $n = length[p] - 1$;
2. **for** $i = 1$ **to** n

Chapter 15. Dynamic Programming

Algorithm $\textsc{MatrixChainOrder}(p)$

```
n = length[p] - 1;
for i = 1 to n
    M[i, i] = 0;
```

Algorithm $\textsc{MatrixChainOrder}(p)$

```
n = length[p] - 1;
for i = 1 to n
    M[i, i] = 0;
for l = 2 to n
```

Algorithm $\textsc{MatrixChainOrder}(p)$

```
n = length[p] - 1;
for i = 1 to n
    M[i, i] = 0;
for l = 2 to n
    for i = 1 to n - l + 1
```

Algorithm $\textsc{MatrixChainOrder}(p)$

```
n = length[p] - 1;
for i = 1 to n
    M[i, i] = 0;
for l = 2 to n
    for i = 1 to n - l + 1
        j = i + l - 1;
```

Algorithm $\textsc{MatrixChainOrder}(p)$

```
n = length[p] − 1;
for i = 1 to n
    M[i, i] = 0;
for l = 2 to n
    for i = 1 to n − l + 1
        j = i + l − 1;
        M[i, j] = ∞;
```

```
Algorithm MatrixChainOrder(p)
    n = length[p] − 1;
    for i = 1 to n
      M[i, i] = 0;
    for l = 2 to n
      for i = 1 to n − l + 1
        j = i + l − 1;
        M[i, j] = ∞;
        for k = i to j − 1
```

```
Algorithm MatrixChainOrder(p)
    n = length[p] − 1;
    for i = 1 to n
      M[i, i] = 0;
    for l = 2 to n
      for i = 1 to n − l + 1
        j = i + l − 1;
        M[i, j] = ∞;
        for k = i to j − 1
            q = M[i, k] + M[k + 1, j] + p[i − 1]p[k]p[j]
```

```
Algorithm MatrixChainOrder(p)
    n = length[p] − 1;
    for i = 1 to n
      M[i, i] = 0;
    for l = 2 to n
      for i = 1 to n − l + 1
        j = i + l − 1;
        M[i, j] = ∞;
        for k = i to j − 1
            q = M[i, k] + M[k + 1, j] + p[i − 1]p[k]p[j]
          if q < M[i, j]
```

```
Algorithm MatrixChainOrder(p)
    n = length[p] − 1;
    for i = 1 to n
      M[i, i] = 0;
    for l = 2 to n
      for i = 1 to n − l + 1
        j = i + l − 1;
        M[i, j] = ∞;
        for k = i to j − 1
            q = M[i, k] + M[k + 1, j] + p[i − 1]p[k]p[j]
          if q < M[i, j]
              M[i, j] = q
```

```
Algorithm MatrixChainOrder(p)
    n = length[p] − 1;
    for i = 1 to n
      M[i, i] = 0;
    for l = 2 to n
      for i = 1 to n − l + 1
        j = i + l − 1;
        M[i, j] = ∞;
        for k = i to j − 1
            q = M[i, k] + M[k + 1, j] + p[i − 1]p[k]p[j]
          if q < M[i, j]
              M[i, j] = q
              S[i, j] = k
```

```
Algorithm MatrixChainOrder(p)
    n = length[p] − 1;
    for i = 1 to n
      M[i, i] = 0;
    for l = 2 to n
      for i = 1 to n − l + 1
        j = i + l − 1;
        M[i, j] = ∞;
        for k = i to j − 1
            q = M[i, k] + M[k + 1, j] + p[i − 1]p[k]p[j]
          if q < M[i, j]
              M[i, j] = q
              S[i, j] = k
return (M, S)
```

Algorithm $\textsc{MatrixChainOrder}(p)$

```
n = length[p] − 1;
for i = 1 to n
  M[i, i] = 0;
for l = 2 to n
  for i = 1 to n − l + 1
    j = i + l − 1;
    M[i, j] = ∞;
    for k = i to j − 1
        q = M[i, k] + M[k + 1, j] + p[i − 1]p[k]p[j]
      if q < M[i, j]
          M[i, j] = q
          S[i, j] = k
return (M, S)
```

Time complexity: $O(n^2 \times n)$

Step 4. obtain the optimization parenthesization

Step 4. obtain the optimization parenthesization

Algorithm $\textsc{MatrixChainParenthesization}(i, j, S)$

```
if i < j
    k = s[i, j]
    print (i, j, " : ", k)
    MatrixChainParenthesization(i, k, S)
    MatrixChainParenthesization(k + 1, j, S)
return
```

Step 4. obtain the optimization parenthesization

Algorithm $\textsc{MatrixChainParenthesization}(i, j, S)$

```
if i < j
    k = s[i, j]
    print (i, j, " : ", k)
    MatrixChainParenthesization(i, k, S)
    MatrixChainParenthesization(k + 1, j, S)
return
```

Usage: **call** $\textsc{MatrixChainParenthesization}(1, n, S)$

Step 4. obtain the optimization parenthesization

Algorithm $\textsc{MatrixChainParenthesization}(i, j, S)$

```
if i < j
    k = s[i, j]
    print (i, j, " : ", k)
    MatrixChainParenthesization(i, k, S)
    MatrixChainParenthesization(k + 1, j, S)
return
```

Usage: **call** $\textsc{MatrixChainParenthesization}(1, n, S)$

Time complexity for $\textsc{MatrixChainParenthesization}$:

Step 4. obtain the optimization parenthesization

Algorithm $\textsc{MatrixChainParenthesization}(i, j, S)$

```
if i < j
    k = s[i, j]
    print (i, j, " : ", k)
    MatrixChainParenthesization(i, k, S)
    MatrixChainParenthesization(k + 1, j, S)
return
```

Usage: **call** $\textsc{MatrixChainParenthesization}(1, n, S)$

Time complexity for $\textsc{MatrixChainParenthesization}$: $O(n)$.

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

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

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

Chapter 16. Greedy Algorithms

<u>Activity-selection problem</u>

INPUT: n activities $S = \{a_1, \ldots, a_n\}$, each with a start time s_i and finish time f_i, $1 \leq i \leq n$

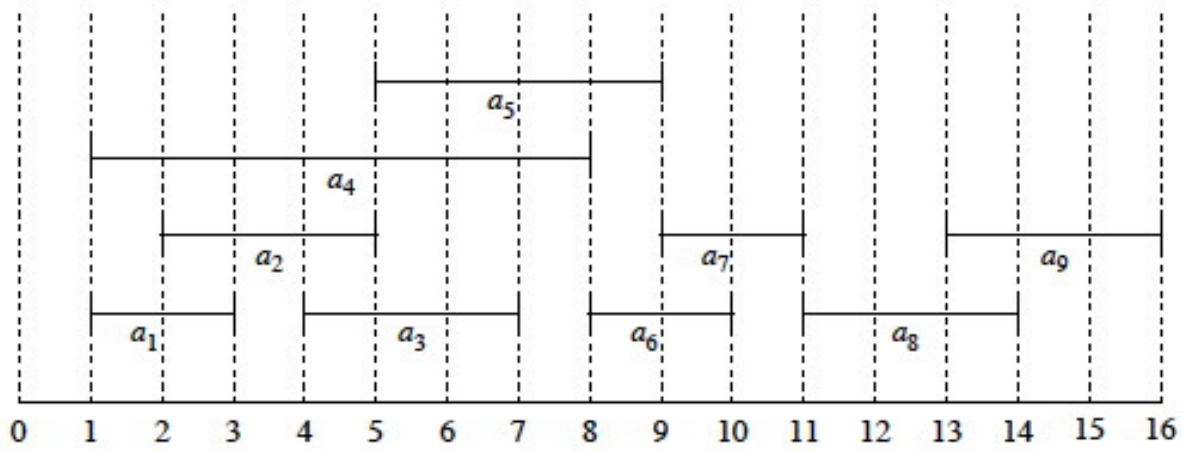
OUTPUT: subset $A \subseteq A$ of mutually compatible activities such that $|A|$ is the maximum.

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 $A \subseteq A$ of mutually compatible activities such that $|A|$ is the maximum.



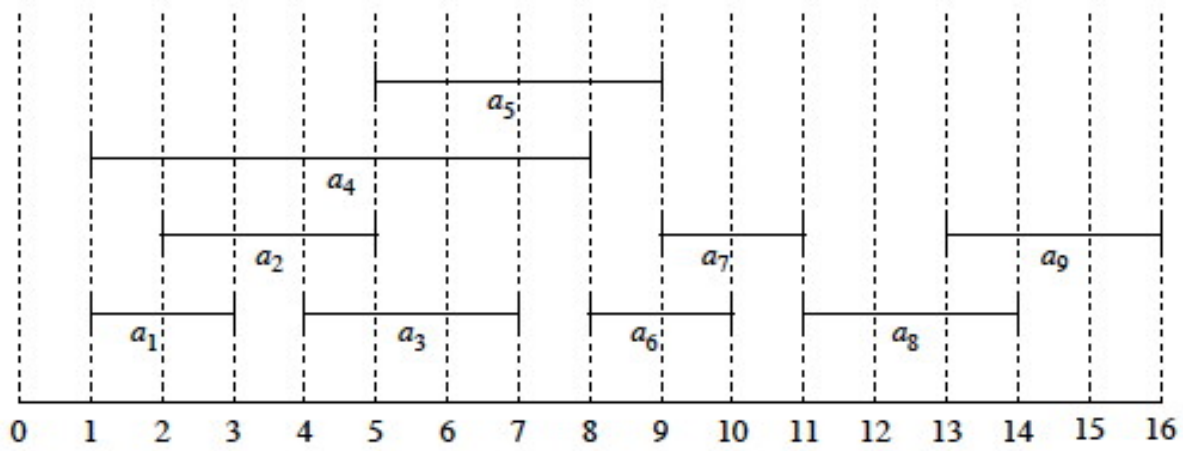
$A = \{a_1, a_3, a_6, a_8\}$, or

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 $A \subseteq A$ of mutually compatible activities such that $|A|$ is the maximum.



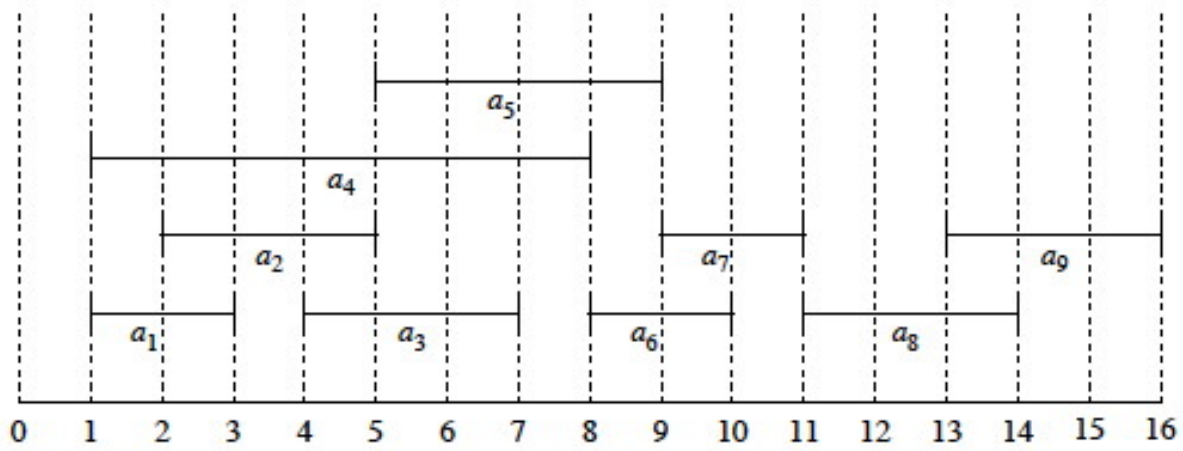
$A = \{a_1, a_3, a_6, a_8\}$, or $A = \{a_2, a_5, a_7, a_8\}$, or

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 $A \subseteq A$ of mutually compatible activities such that $|A|$ is the maximum.



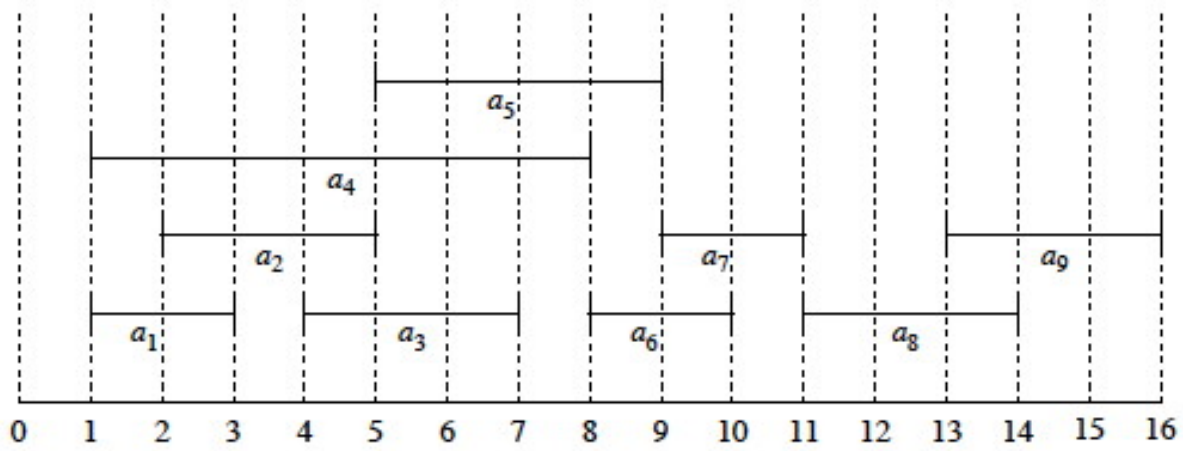
$A = \{a_1, a_3, a_6, a_8\}$, or $A = \{a_2, a_5, a_7, a_8\}$, or $A = \{a_2, a_5, a_7, a_9\}$,

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 $A \subseteq A$ of mutually compatible activities such that $|A|$ is the maximum.



$A = \{a_1, a_3, a_6, a_8\}$, or $A = \{a_2, a_5, a_7, a_8\}$, or $A = \{a_2, a_5, a_7, a_9\}$, $\ldots$.

Chapter 16. Greedy Algorithms

A dynamic programming solution

Chapter 16. Greedy Algorithms

A dynamic programming solution

Step 1 analysis of problem

A dynamic programming solution

Step 1 analysis of problem

- like the Knapsack Problem to consider the first k items and exhaustively consider to include a_k or not to set A

A dynamic programming solution

Step 1 analysis of problem

- like the Knapsack Problem to consider the first k items and exhaustively consider to include a_k or not to set A

  but how to put the activities in order?

A dynamic programming solution

Step 1 analysis of problem

- like the Knapsack Problem to consider the first k items and exhaustively consider to include a_k or not to set A

  but how to put the activities in order?
  perhaps there is no "forward DP" for this problem.

A dynamic programming solution

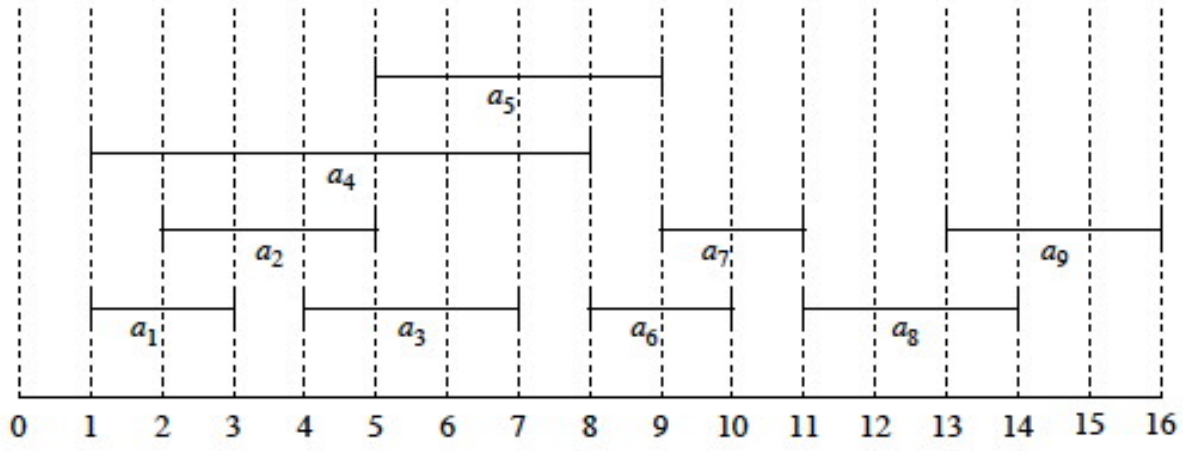
Step 1 analysis of problem

- like the Knapsack Problem to consider the first k items and exhaustively consider to include a_k or not to set A

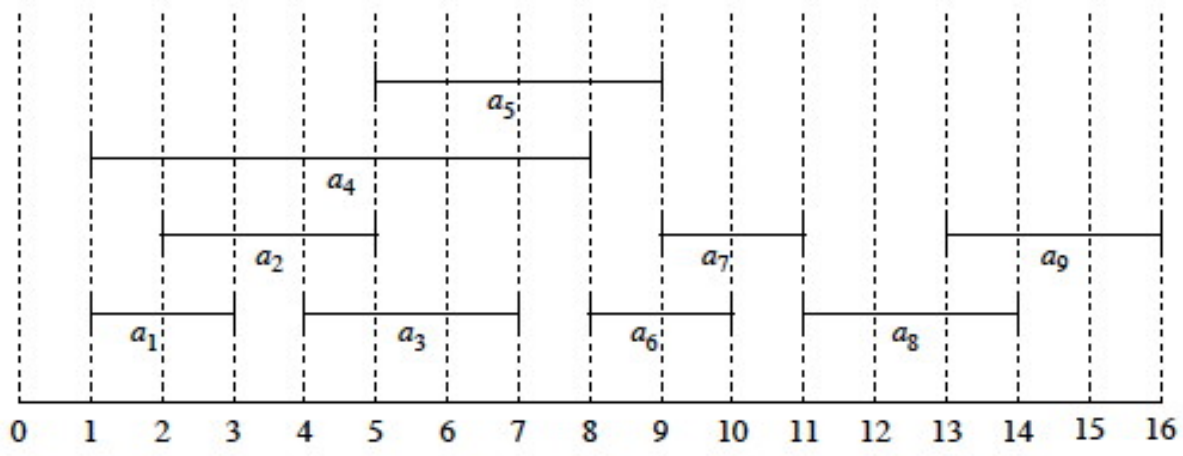
  but how to put the activities in order?
  perhaps there is no "forward DP" for this problem.

- instead, like Matrix Chain Multiplication problem,
  we consider "inside-out DP" method
  exhaustively consider to **include** a_k, for some k.

Chapter 16. Greedy Algorithms

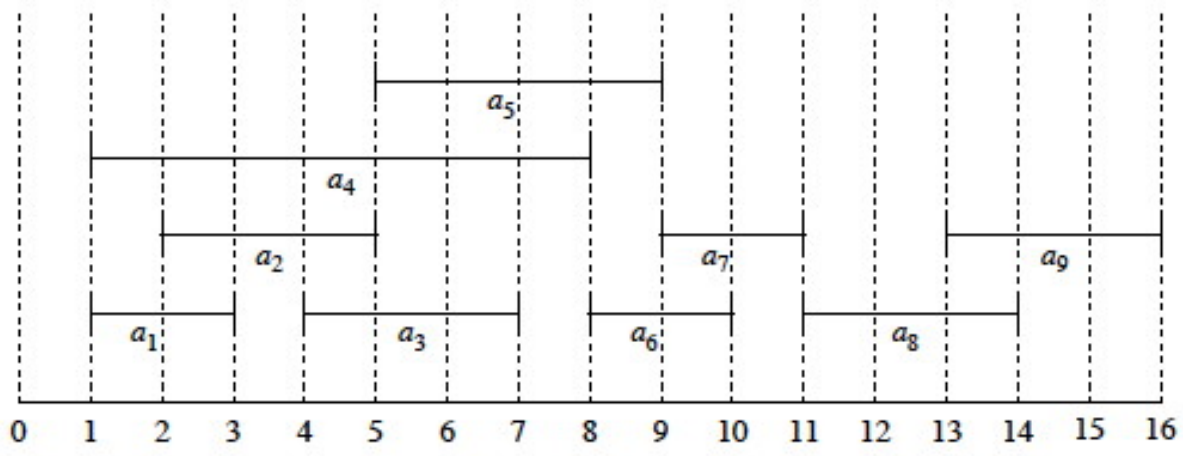


Chapter 16. Greedy Algorithms



For example, selecting a_3 would result in two subsets of activities to further
consider: $\{a_1\}$ and $\{a_6, a_7, a_8, a_9\}$;

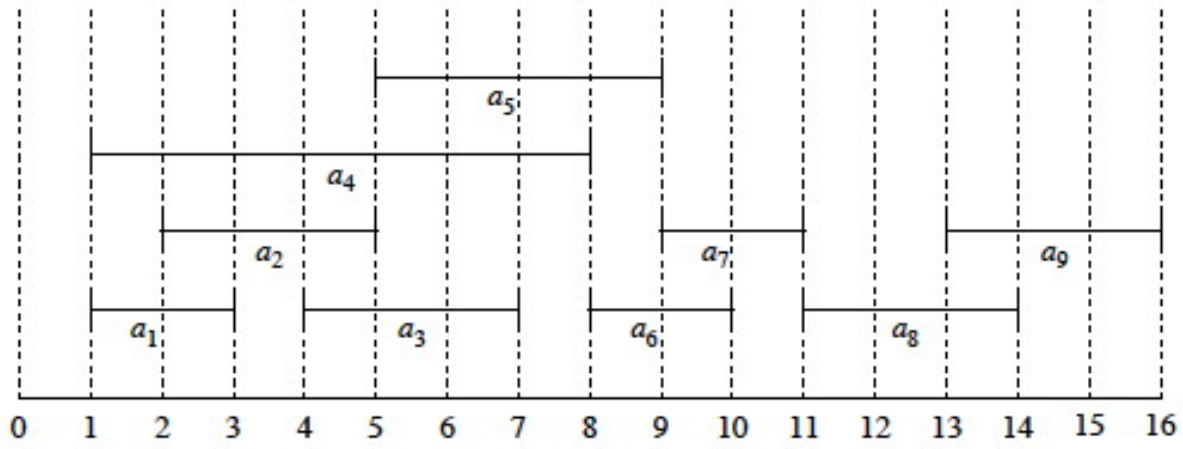
Chapter 16. Greedy Algorithms



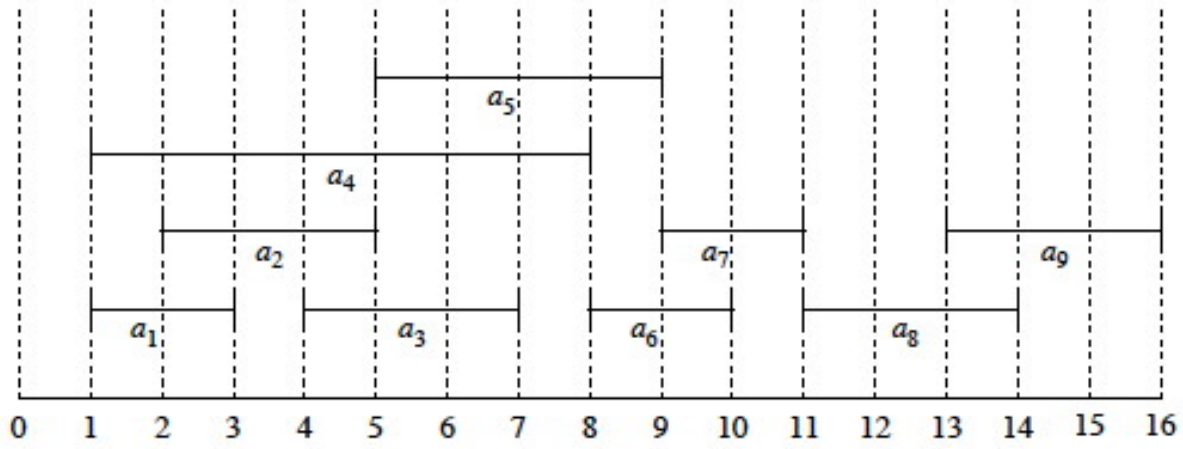
For example, selecting a_3 would result in two subsets of activities to further
consider: $\{a_1\}$ and $\{a_6, a_7, a_8, a_9\}$;

selecting a_5 would result in two subsets of activities to further consider: $\{a_1, a_2\}$ and $\{a_7, a_8, a_9\}$;

Chapter 16. Greedy Algorithms

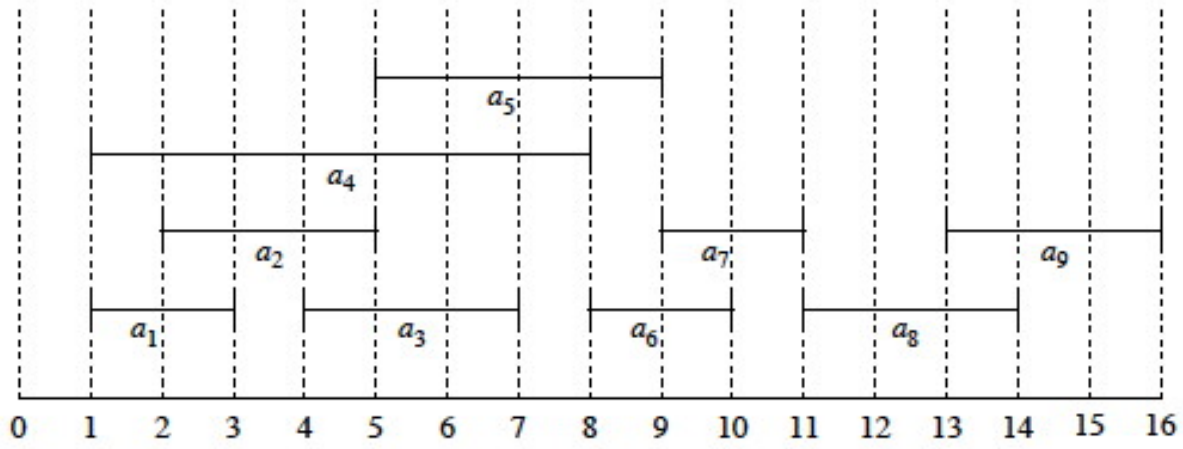


Chapter 16. Greedy Algorithms



Define subset $S_{i,j} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$

Chapter 16. Greedy Algorithms

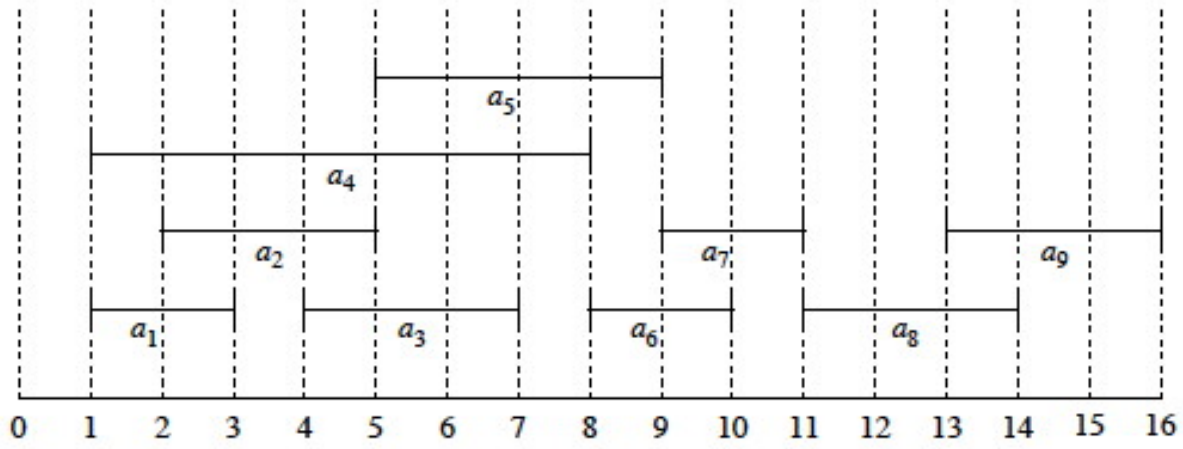


Define subset $S_{i,j} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$

e.g.,

$$S_{2,9} = \{a_k \in S : f_2 \leq s_k < f_k \leq s_9\} =$$

Chapter 16. Greedy Algorithms

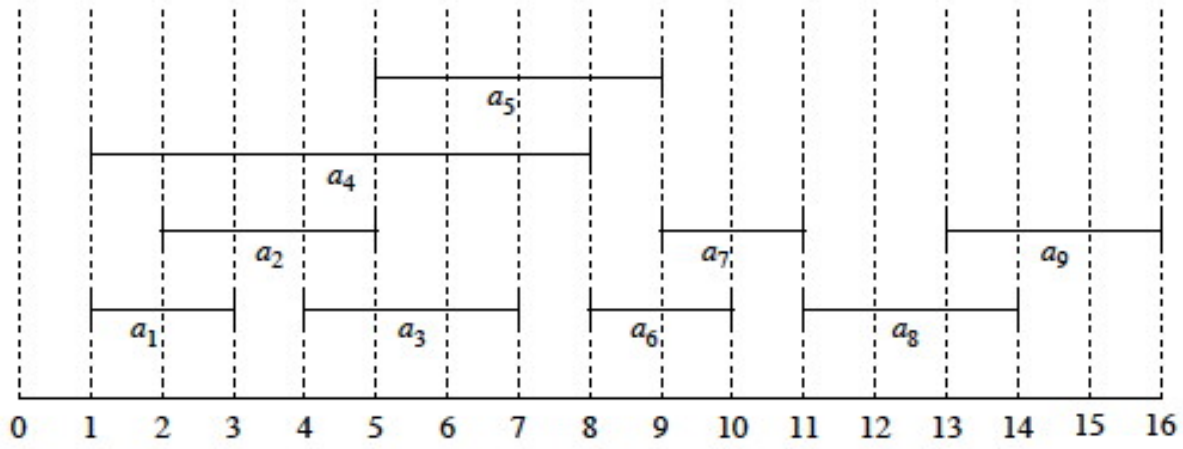


Define subset $S_{i,j} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$

e.g.,

$$S_{2,9} = \{a_k \in S : f_2 \leq s_k < f_k \leq s_9\} = \{a_5, a_6, a_7\}$$

Chapter 16. Greedy Algorithms



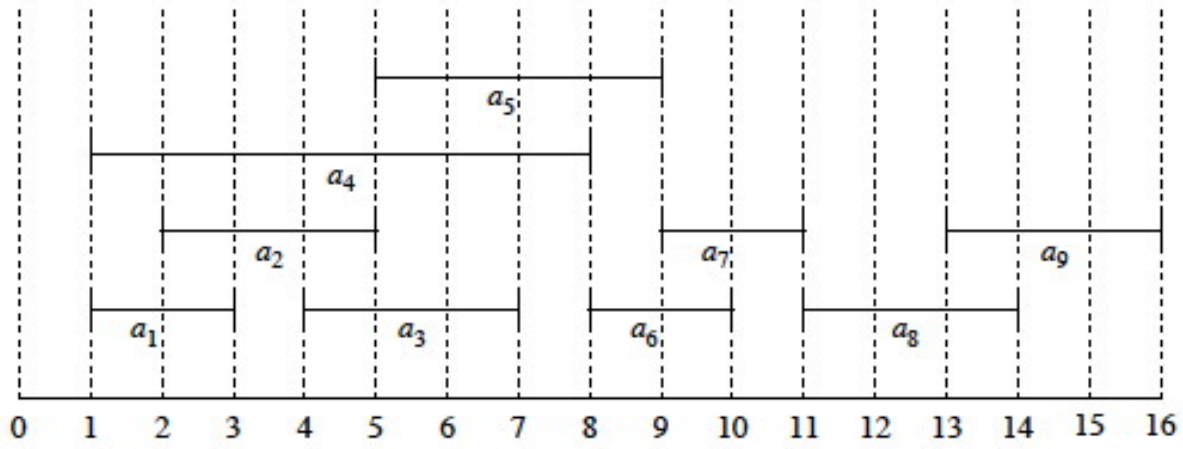
Define subset $S_{i,j} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$

e.g.,

$S_{2,9} = \{a_k \in S : f_2 \leq s_k < f_k \leq s_9\} = \{a_5, a_6, a_7\}$

$S_{3,8} = \{a_6, a_7\}$.

Chapter 16. Greedy Algorithms



Define subset $S_{i,j} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$

e.g.,

$S_{2,9} = \{a_k \in S : f_2 \leq s_k < f_k \leq s_9\} = \{a_5, a_6, a_7\}$

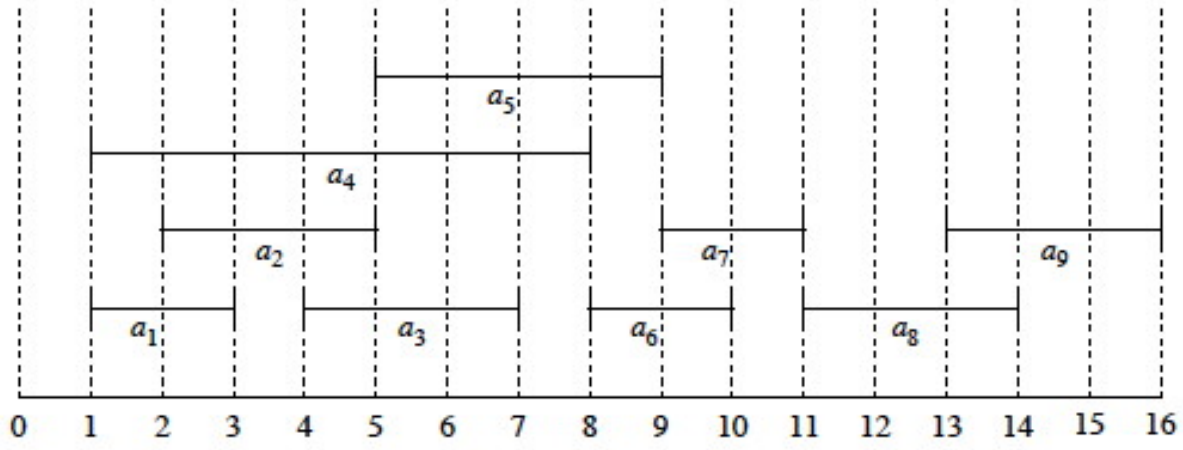
$S_{3,8} = \{a_6, a_7\}$.

introducing dummy activities:

a_0 with f_0 =the earliest start time of all, and

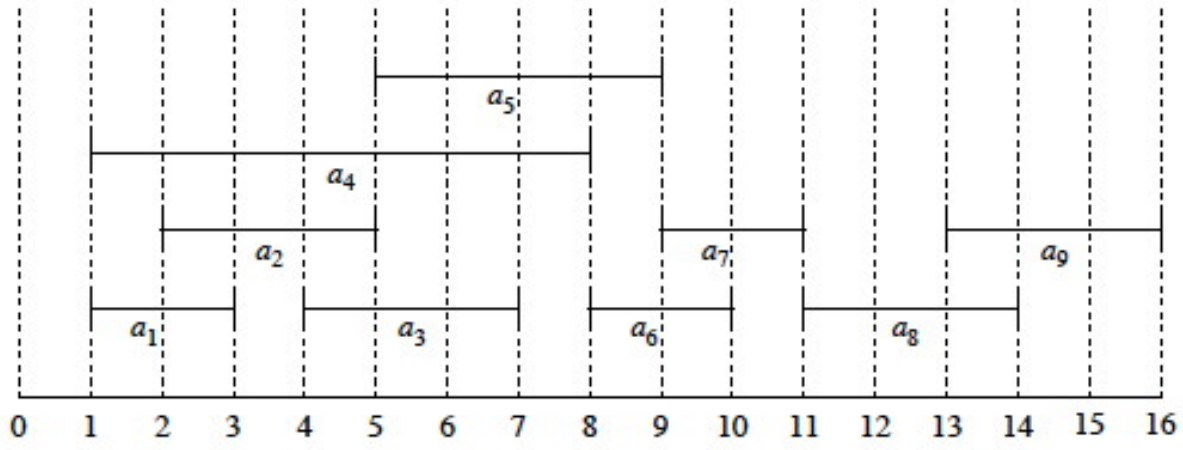
a_{n+1} with s_{n+1} = latest finish time of all.

Chapter 16. Greedy Algorithms



Define subset $S_{i,j} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$

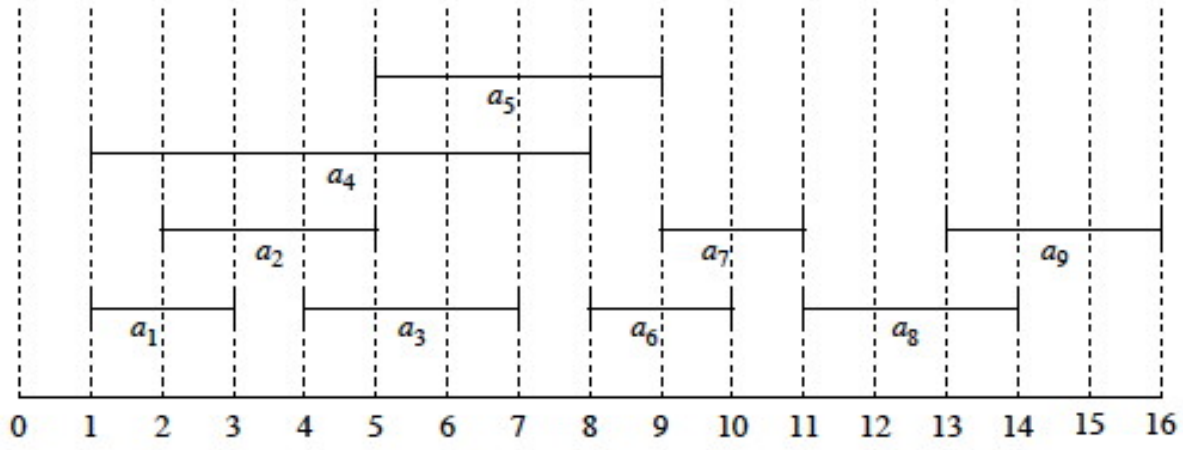
Chapter 16. Greedy Algorithms



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.

Chapter 16. Greedy Algorithms



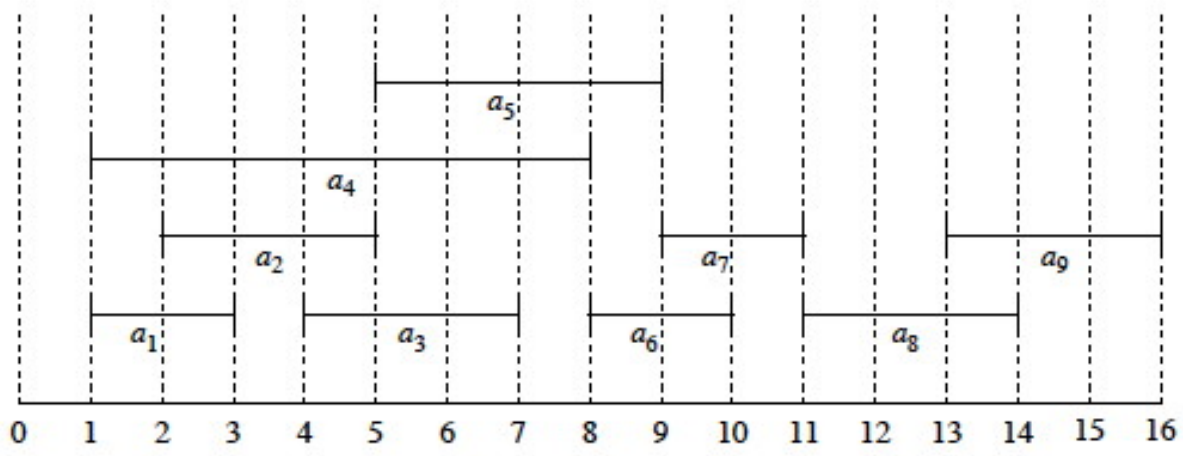
Define subset $S_{i,j} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$

Define solution $A_{i,j} \subseteq S_{i,j}$ is the maximum size set of compatible activities.

Then

$$|A_{i,j}| = \max_{a_k \in S_{ij}} \{|A_{i,k} \cup \{a_k\} \cup A_{k,j}|\}$$

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.

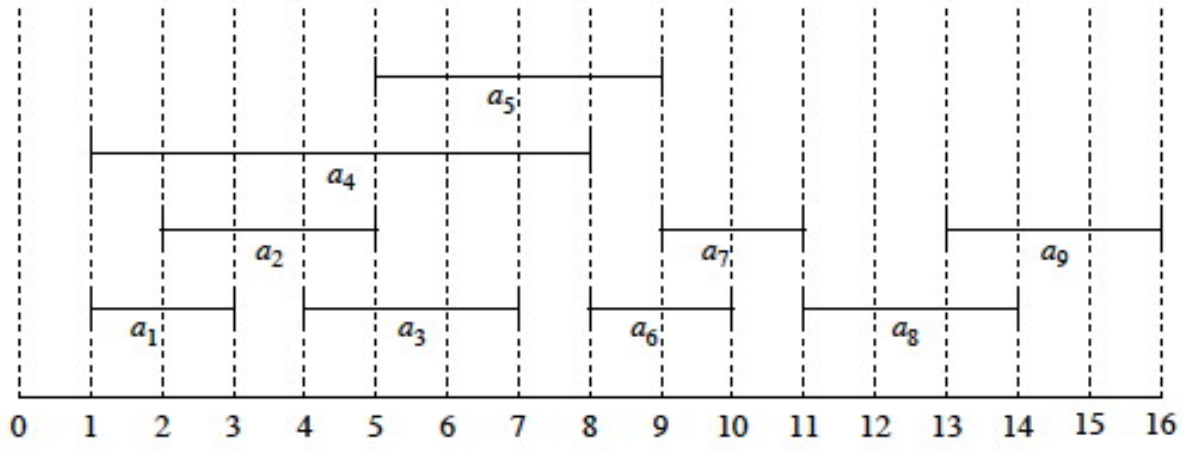
Then

$$|A_{i,j}| = \max_{a_k \in S_{ij}} \{|A_{i,k} \cup \{a_k\} \cup A_{k,j}|\}$$

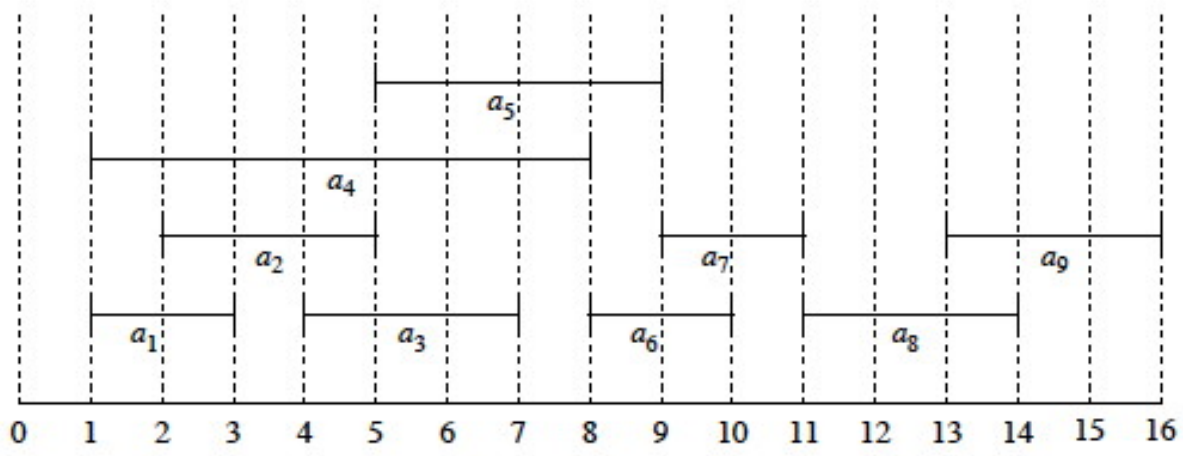
e.g.,

$$|A_{0,10}| = \max \begin{cases} \dots \\ |A_{0,6} \cup \{a_6\} \cup A_{6,10}|, \text{ where } \quad S_{0,6} = \{a_1, \dots, a_4\} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad S_{6,10} = \{a_8, a_9\} \\ \dots \end{cases}$$

Chapter 16. Greedy Algorithms



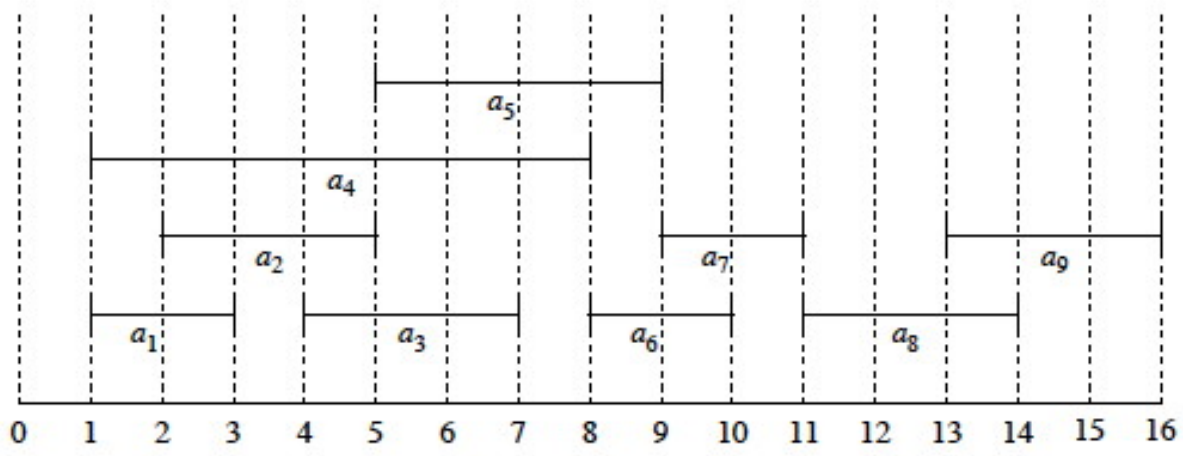
Chapter 16. Greedy Algorithms



Instead of go forward with the 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

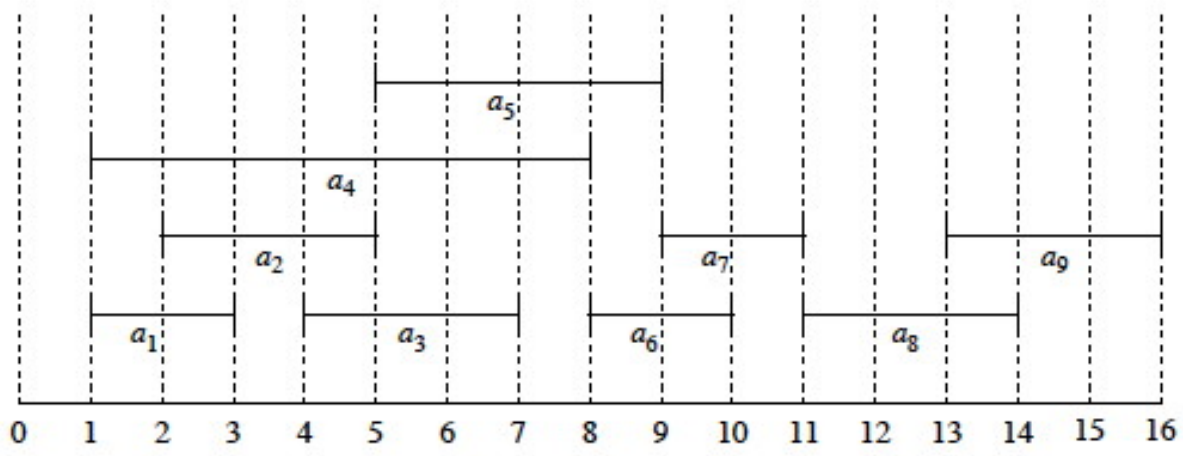


Instead of go forward with the DP, we wonder if all but one **of the subproblems** could be made empty.

$$|A_{i,j}| = \max_{a_k \in S_{i,j}} \{|A_{i,k} \cup \{a_k\} \cup A_{k,j}|\}$$

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 $l \neq k$

Chapter 16. Greedy Algorithms



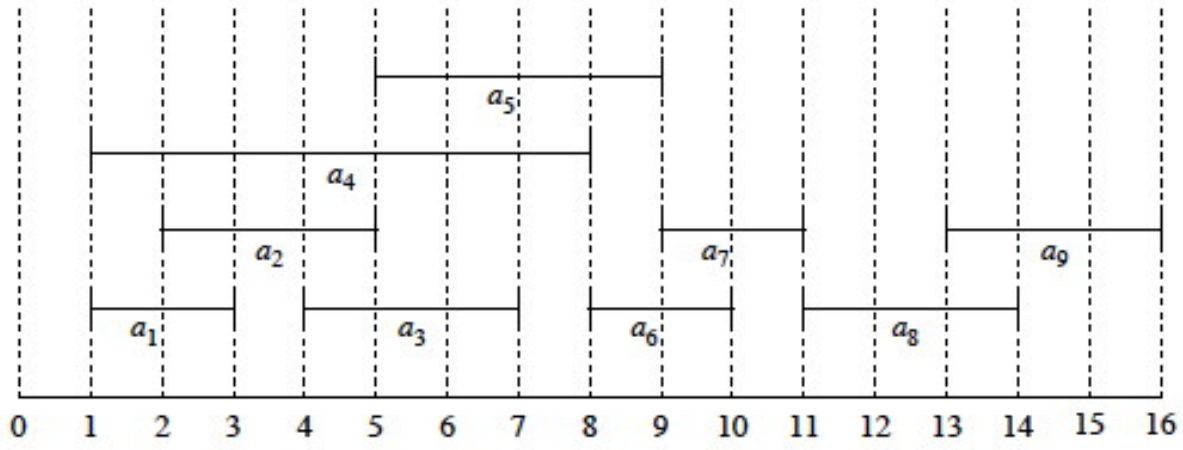
Instead of go forward with the DP, we wonder if all but one **of the subproblems** could be made empty.

$$|A_{i,j}| = \max_{a_k \in S_{i,j}} \{|A_{i,k} \cup \{a_k\} \cup A_{k,j}|\}$$

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.

Chapter 16. Greedy Algorithms



Instead of go forward with the DP, we wonder if all but one **of the subproblems** could be made empty.

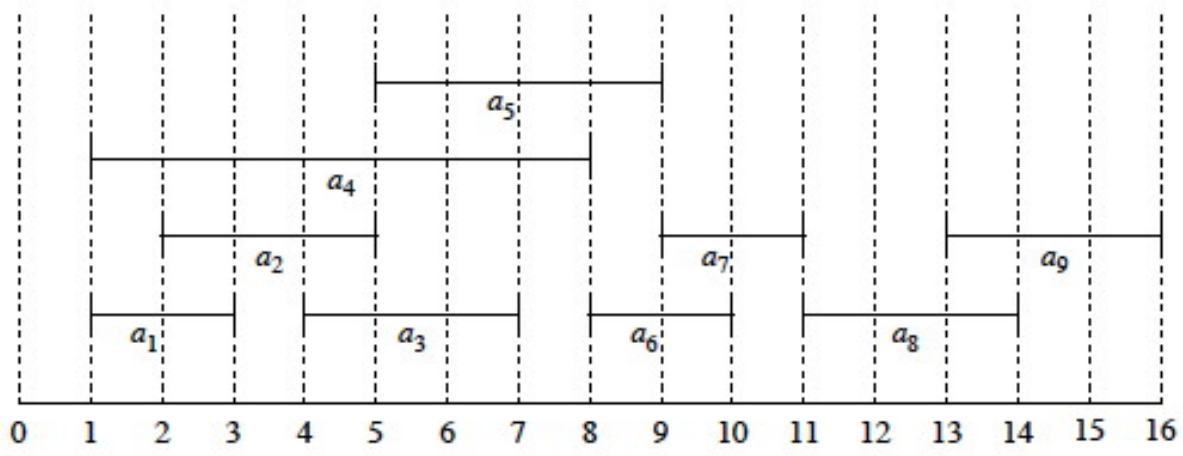
$$|A_{i,j}| = \max_{a_k \in S_{i,j}} \{|A_{i,k} \cup \{a_k\} \cup A_{k,j}|\}$$

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**

Chapter 16. Greedy Algorithms



Instead of go forward with the 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

Activity Selection problem has the greedy-choice property.

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 be the activity of the earliest finish time, then a_k is in some optimal 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 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**

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 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 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

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 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")

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 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")

Let $A_{i,j}$ be an optimal 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 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")

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.

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 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")

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 $A_{i,j}$.

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 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")

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 $S_{i,j}$.

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 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")

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 $S_{i,j}$.

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 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")

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

$$|A_{i,j}| = \max_{a_k \in S_{ij}} \{|A_{i,k} \cup \{a_k\} \cup A_{k,j}|\}$$

we can derive greedy algorithms for the Activity Selection problem.

Chapter 16. Greedy Algorithms

Assume that the activities are sorted according to their finish times.
$f_1 \leq f_2 \leq \cdots \leq f_n$.

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 $s_0 = f_0 = s_1$)

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 $s_{n+1} = f_{n+1} = f_n$).

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 $\textsc{Activity-Selection}(s, f, k, n)$

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. $\quad m = k + 1$

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. $m = k + 1$
2. **while** $m \leq n$ and $s_m < f_k$

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 $\textsc{Activity-Selection}(s, f, k, n)$

```
m = k + 1
while m ≤ n and s_m < f_k
    m = m + 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 $\textsc{Activity-Selection}(s, f, k, n)$

```
m = k + 1
while m ≤ n and s_m < f_k
    m = m + 1
if 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 $\textsc{Activity-Selection}(s, f, k, n)$

```
m = k + 1
while m ≤ n and s_m < f_k
    m = m + 1
if m ≤ n
    return {a_m} ∪ 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 $\textsc{Activity-Selection}(s, f, k, n)$

```
m = k + 1
while m ≤ n and s_m < f_k
    m = m + 1
if m ≤ n
    return {a_m} ∪ ACTIVITY-SELECTION(s, f, m, n)
else return (∅)
```

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 $\textsc{Activity-Selection}(s, f, k, n)$

```
m = k + 1
while m ≤ n and s_m < f_k
    m = m + 1
if m ≤ n
    return {a_m} ∪ ACTIVITY-SELECTION(s, f, m, n)
else return (∅)
```

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 $\textsc{Activity-Selection}(s, f, k, n)$

```
m = k + 1
while m ≤ n and s_m < f_k
    m = m + 1
if m ≤ n
    return {a_m} ∪ ACTIVITY-SELECTION(s, f, m, n)
else return (∅)
```

First called with $\textsc{Activity-Selection}(s, f_0, n)$

Time complexity: $O(n)$.

Chapter 16. Greedy Algorithms

Fractional Knapsack Problem:

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

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 $0 < f_i \leq 1$,

Chapter 16. Greedy Algorithms

<u>Fractional Knapsack Problem</u>:

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.

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

$$K(S, X) = \max_{0 < f_i \leq 1, i \in S}$$

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

$$K(S, X) = \max_{0 < f_i \leq 1, i \in S} \{f_i v_i + K(S, X - f_i s_i)\}$$

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

$$K(S, X) = \max_{0 < f_i \leq 1, i \in S} \{f_i v_i + K(S, X - f_i s_i)\}$$

- 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

$$K(S, X) = \max_{0 < f_i \leq 1, i \in S} \{f_i v_i + K(S, X - f_i s_i)\}$$

- 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

$$K(S, X) = \max_{0 < f_i \leq 1, i \in S} \{f_i v_i + K(S, X - f_i s_i)\}$$

- 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

$$K(S, X) = \max_{0 < f_i \leq 1, i \in S} \{f_i v_i + K(S, X - f_i s_i)\}$$

- 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.

Theorem: Fractional Knapsack problem has the greedy-choice property.

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 $f_i s_i \leq X$.

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 $f_i s_i \leq X$.

(This is left as an exercise)

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

Chapter 16. Greedy Algorithms

Huffman Code

compressing data using binary bits
code: a compressing scheme, fixed length vs variable length code

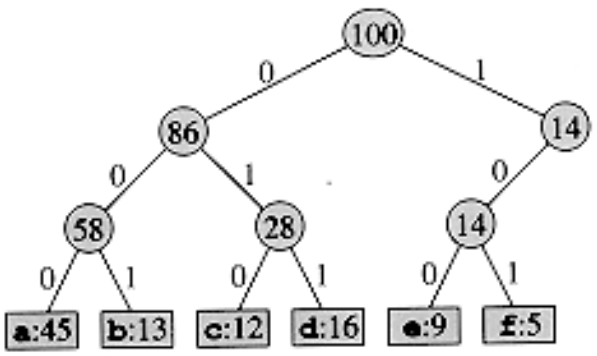
character	a	b	c	d	e	f
frequency	.45	.13	.12	.16	.09	.05
fixed length code	000	001	010	011	100	101
variable length code	0	101	100	111	1101	1100

Chapter 16. Greedy Algorithms

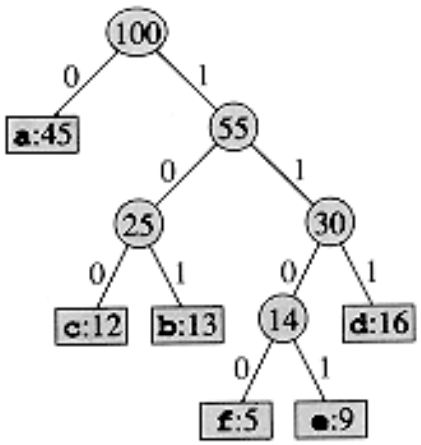
Huffman Code

compressing data using binary bits
code: a compressing scheme, fixed length vs variable length code

character	a	b	c	d	e	f
frequency	.45	.13	.12	.16	.09	.05
fixed length code	000	001	010	011	100	101
variable length code	0	101	100	111	1101	1100



(a)



(b)

Chapter 16. Greedy Algorithms

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:

Chapter 16. Greedy Algorithms

prefix code: a prefix of a codeword cannot be another codeword.

The Optimal Prefix Code Problem:

INPUT: character set C and frequencies f;

Chapter 16. Greedy Algorithms

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

Chapter 16. Greedy Algorithms

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

$$B(T) = \sum_{c \in C} f(c) d_T(c) \text{ achieves the minimum}$$

where $d_T(c)$ is the depth of character c in tree T.

Chapter 16. Greedy Algorithms

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

$$B(T) = \sum_{c \in C} f(c) d_T(c) \text{ achieves the minimum}$$

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.

Chapter 16. Greedy Algorithms

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

$$B(T) = \sum_{c \in C} f(c) d_T(c) \text{ achieves the minimum}$$

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).

Chapter 16. Greedy Algorithms

Algorithm $\textsc{Huffman}(C, f)$

Chapter 16. Greedy Algorithms

Algorithm $\textsc{Huffman}(C, f)$

1. $n = |C|$

Chapter 16. Greedy Algorithms

Algorithm $\textsc{Huffman}(C, f)$

1. $n = |C|$
2. $Q = C$

Chapter 16. Greedy Algorithms

Algorithm $\textsc{Huffman}(C, f)$

```
n = |C|
Q = C
for i = 1 to n − 1
```

Algorithm $\textsc{Huffman}(C, f)$

```
n = |C|
Q = C
for i = 1 to n − 1
    newnode(z)
```

Algorithm $\textsc{Huffman}(C, f)$

```
n = |C|
Q = C
for i = 1 to n − 1
    newnode(z)
    x = EXTRACT-MIN(Q)        extract two least frequent characters
```

Algorithm $\textsc{Huffman}(C, f)$

```
n = |C|
Q = C
for i = 1 to n − 1
    newnode(z)
    x = EXTRACT-MIN(Q)        extract two least frequent characters
    z.leftchild = x
```

Algorithm $\textsc{Huffman}(C, f)$

```
n = |C|
Q = C
for i = 1 to n − 1
    newnode(z)
    x = EXTRACT-MIN(Q)        extract two least frequent characters
    z.leftchild = x
    y = EXTRACT-MIN(Q)
```

Algorithm $\textsc{Huffman}(C, f)$

```
n = |C|
Q = C
for i = 1 to n − 1
    newnode(z)
    x = EXTRACT-MIN(Q)        extract two least frequent characters
    z.leftchild = x
    y = EXTRACT-MIN(Q)
    z.rightchild = y
```

Algorithm $\textsc{Huffman}(C, f)$

```
n = |C|
Q = C
for i = 1 to n − 1
    newnode(z)
    x = EXTRACT-MIN(Q)        extract two least frequent characters
    z.leftchild = x
    y = EXTRACT-MIN(Q)
    z.rightchild = y
    f(z) = f(x) + f(y)
```

Algorithm $\textsc{Huffman}(C, f)$

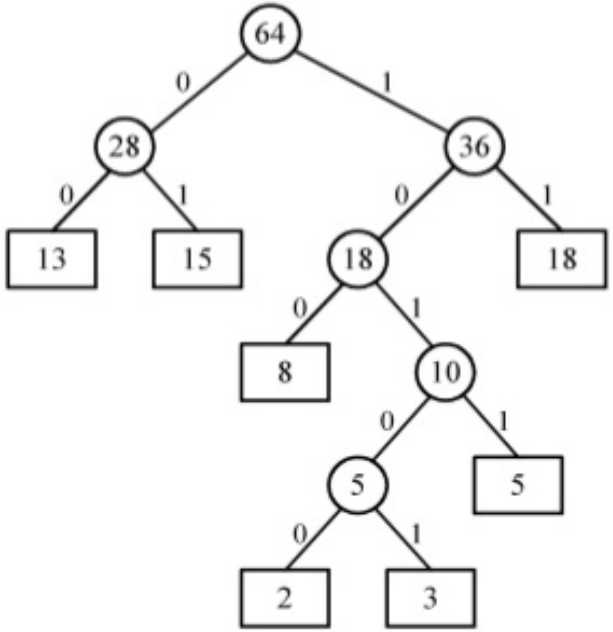
```
n = |C|
Q = C
for i = 1 to n − 1
    newnode(z)
    x = EXTRACT-MIN(Q)        extract two least frequent characters
    z.leftchild = x
    y = EXTRACT-MIN(Q)
    z.rightchild = y
    f(z) = f(x) + f(y)
    INSERT(Q, z)
```

Algorithm $\textsc{Huffman}(C, f)$

```
n = |C|
Q = C
for i = 1 to n − 1
    newnode(z)
    x = EXTRACT-MIN(Q)        extract two least frequent characters
    z.leftchild = x
    y = EXTRACT-MIN(Q)
    z.rightchild = y
    f(z) = f(x) + f(y)
    INSERT(Q, z)
return (EXTRACT-MIN(Q))
```

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

Correctness of Huffman's algorithm

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.

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.

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.

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!

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.

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.

We use the following swapping strategy to obtain another prefix code T' for C.

Without loss of generality, assume $\{a, b\} \cap \{x, y\} = \phi$.

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'.

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'.

$$B(T) - B(T') = \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c)$$

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'.

$$B(T) - B(T') = \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c)$$

$$= f(a) d_T(a) + f(b) d_T(b) + f(x) d_T(x) + f(y) d_T(y)$$

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'.

$$B(T) - B(T') = \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c)$$

$$= f(a)d_T(a) + f(b)d_T(b) + f(x)d_T(x) + f(y)d_T(y)$$

$$-f(a)d_{T'}(a) - f(b)d_{T'}(b) - f(x)d_{T'}(x) - f(y)d_{T'}(y)$$

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'.

$$
\begin{aligned}
B(T) - B(T') &= \sum_{c \in C} f(c) d_T(c) - \textstyle\sum_{c \in C} f(c) d_{T'}(c) \\
&= f(a)d_T(a) + f(b)d_T(b) + f(x)d_T(x) + f(y)d_T(y) \\
&\qquad - f(a)d_{T'}(a) - f(b)d_{T'}(b) - f(x)d_{T'}(x) - f(y)d_{T'}(y) \\
&= (f(a) - f(x))d_T(a) + (f(x) - f(a))d_T(x) + (f(b) - f(y))d_T(b)
\end{aligned}
$$

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'.

$$
\begin{aligned}
B(T) - B(T') &= \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c) \\
&= f(a)d_T(a) + f(b)d_T(b) + f(x)d_T(x) + f(y)d_T(y) \\
&\qquad - f(a)d_{T'}(a) - f(b)d_{T'}(b) - f(x)d_{T'}(x) - f(y)d_{T'}(y) \\
&= (f(a) - f(x))d_T(a) + (f(x) - f(a))d_T(x) + (f(b) - f(y))d_T(b) \\
&\qquad + (f(y) - f(b))d_T(y)
\end{aligned}
$$

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'.

$$
\begin{aligned}
B(T) - B(T') &= \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c) \\
&= f(a)d_T(a) + f(b)d_T(b) + f(x)d_T(x) + f(y)d_T(y) \\
&\qquad - f(a)d_{T'}(a) - f(b)d_{T'}(b) - f(x)d_{T'}(x) - f(y)d_{T'}(y) \\
&= (f(a) - f(x))d_T(a) + (f(x) - f(a))d_T(x) + (f(b) - f(y))d_T(b) \\
&\qquad + (f(y) - f(b))d_T(y) \\
&= (f(a) - f(x))(d_T(a) - d_T(x)) + (f(b) - f(y))(d_T(b) - d_T(y)) \geq 0
\end{aligned}
$$

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'.

$$
\begin{aligned}
B(T) - B(T') &= \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c) \\
&= f(a)d_T(a) + f(b)d_T(b) + f(x)d_T(x) + f(y)d_T(y) \\
&\qquad - f(a)d_{T'}(a) - f(b)d_{T'}(b) - f(x)d_{T'}(x) - f(y)d_{T'}(y) \\
&= (f(a) - f(x))d_T(a) + (f(x) - f(a))d_T(x) + (f(b) - f(y))d_T(b) \\
&\qquad + (f(y) - f(b))d_T(y) \\
&= (f(a) - f(x))(d_T(a) - d_T(x)) + (f(b) - f(y))(d_T(b) - d_T(y)) \geq 0
\end{aligned}
$$

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 $\{a, b\} \cap \{x, y\} = \phi$.

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'.

$$
\begin{aligned}
B(T) - B(T') &= \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c) \\
&= f(a)d_T(a) + f(b)d_T(b) + f(x)d_T(x) + f(y)d_T(y) \\
&\qquad - f(a)d_{T'}(a) - f(b)d_{T'}(b) - f(x)d_{T'}(x) - f(y)d_{T'}(y) \\
&= (f(a) - f(x))d_T(a) + (f(x) - f(a))d_T(x) + (f(b) - f(y))d_T(b) \\
&\qquad + (f(y) - f(b))d_T(y) \\
&= (f(a) - f(x))(d_T(a) - d_T(x)) + (f(b) - f(y))(d_T(b) - d_T(y)) \geq 0
\end{aligned}
$$

which implies $B(T) \geq B(T')$. So T' is also an optimal code for C.

So T' satisfies the lemma.

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.

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 $f(z) = f(x) + f(y)$.

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.

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

$$\begin{aligned} B(T) &= \sum_{c \in C} f(c) d_T(c) = \sum_{c \in C - \{x,y\}} f(c) d_T(c) + f(x) d_T(x) + f(y) d_T(y) \\ &= \sum_{c \in C - \{x,y\}} f(c) d_T(c) + [f(x) + f(y)] d_T(x) \end{aligned}$$

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

$$\begin{aligned} B(T) &= \sum_{c \in C} f(c) d_T(c) = \sum_{c \in C - \{x,y\}} f(c) d_T(c) + f(x) d_T(x) + f(y) d_T(y) \\ &= \sum_{c \in C - \{x,y\}} f(c) d_T(c) + [f(x) + f(y)] d_T(x) \\ &= \sum_{c \in C - \{x,y\}} f(c) d_{T'}(c) + [f(x) + f(y)](d_{T'}(z) + 1) \end{aligned}$$

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

$$\begin{aligned} B(T) &= \textstyle\sum_{c \in C} f(c) d_T(c) = \sum_{c \in C - \{x,y\}} f(c) d_T(c) + f(x) d_T(x) + f(y) d_T(y) \\ &= \textstyle\sum_{c \in C - \{x,y\}} f(c) d_T(c) + [f(x) + f(y)] d_T(x) \\ &= \textstyle\sum_{c \in C - \{x,y\}} f(c) d_{T'}(c) + [f(x) + f(y)](d_{T'}(z) + 1) \\ &= \textstyle\sum_{c \in C - \{x,y\} \cup \{z\}} f(c) d_{T'}(c) + f(x) + f(y) \end{aligned}$$

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

$$\begin{aligned}
B(T) &= \textstyle\sum_{c \in C} f(c) d_T(c) = \sum_{c \in C - \{x,y\}} f(c) d_T(c) + f(x) d_T(x) + f(y) d_T(y) \\
&= \textstyle\sum_{c \in C - \{x,y\}} f(c) d_T(c) + [f(x) + f(y)] d_T(x) \\
&= \textstyle\sum_{c \in C - \{x,y\}} f(c) d_{T'}(c) + [f(x) + f(y)](d_{T'}(z) + 1) \\
&= \textstyle\sum_{c \in C - \{x,y\} \cup \{z\}} f(c) d_{T'}(c) + f(x) + f(y) \\
&= \textstyle\sum_{c \in C'} f(c) d_{T'}(c) + f(x) + f(y) = B(T') + f(x) + f(y)
\end{aligned}$$

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

$$\begin{aligned}
B(T) &= \textstyle\sum_{c \in C} f(c) d_T(c) = \sum_{c \in C - \{x,y\}} f(c) d_T(c) + f(x) d_T(x) + f(y) d_T(y) \\
&= \textstyle\sum_{c \in C - \{x,y\}} f(c) d_T(c) + [f(x) + f(y)] d_T(x) \\
&= \textstyle\sum_{c \in C - \{x,y\}} f(c) d_{T'}(c) + [f(x) + f(y)](d_{T'}(z) + 1) \\
&= \textstyle\sum_{c \in C - \{x,y\} \cup \{z\}} f(c) d_{T'}(c) + f(x) + f(y) \\
&= \textstyle\sum_{c \in C'} f(c) d_{T'}(c) + f(x) + f(y) = B(T') + f(x) + f(y)
\end{aligned}$$

Now if T is not optimal for C, assume optimal prefix code S for C such that $B(S) < B(T)$.

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.

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

$$B(S') < B(T) - f(x) - f(y) = B(T') + f(x) + f(y) - f(x) - f(y) = B(T')$$

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

$$B(S') < B(T) - f(x) - f(y) = B(T') + f(x) + f(y) - f(x) - f(y) = B(T')$$

This implies T' is not optimal code for C'. Contradict.

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

$$B(S') < B(T) - f(x) - f(y) = B(T') + f(x) + f(y) - f(x) - f(y) = B(T')$$

This implies T' is not optimal code for C'. Contradict.

So T should be optimal for C.

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

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.

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 $D = \{v_1, \ldots, v_m\}$

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

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

$$\sum_{i=1}^{m} x_i v_i = n \text{ to minimize } (\sum_{i=1}^{m} x_i)$$

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

$$c(k) = \min \left\{ \right.$$

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

$$c(k) = \min \begin{cases} c(k - v_1) + 1 & k \geq v_1 \\ & \end{cases}$$

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

$$c(k) = \min \begin{cases} c(k - v_1) + 1 & k \geq v_1 \\ c(k - v_2) + 1 & k \geq v_2 \\ & \end{cases}$$

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

$$c(k) = \min \begin{cases} c(k - v_1) + 1 & k \geq v_1 \\ c(k - v_2) + 1 & k \geq v_2 \\ \dots & \\ & \end{cases}$$

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

$$c(k) = \min \begin{cases} c(k - v_1) + 1 & k \geq v_1 \\ c(k - v_2) + 1 & k \geq v_2 \\ \dots & \\ c(k - v_m) + 1 & k \geq v_m \end{cases}$$

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

$$c(k) = \min \begin{cases} c(k - v_1) + 1 & k \geq v_1 \\ c(k - v_2) + 1 & k \geq v_2 \\ \dots & \\ c(k - v_m) + 1 & k \geq v_m \end{cases}$$

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

$$c(k) = \min \begin{cases} c(k - v_1) + 1 & k \geq v_1 \\ c(k - v_2) + 1 & k \geq v_2 \\ \dots & \\ c(k - v_m) + 1 & k \geq v_m \end{cases}$$

- base case: $c(v_1) = 1$,

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

$$c(k) = \min \begin{cases} c(k - v_1) + 1 & k \geq v_1 \\ c(k - v_2) + 1 & k \geq v_2 \\ \dots & \\ c(k - v_m) + 1 & k \geq v_m \end{cases}$$

- base case: $c(v_1) = 1, \dots, c(v_m) = 1$.

Chapter 16. Greedy Algorithms

Chapter 16. Greedy Algorithms

A greedy algorithm

$$c(k) = \min \begin{cases} c(k - v_1) + 1 & k \geq v_1 \\ c(k - v_2) + 1 & k \geq v_2 \\ \dots \\ c(k - v_m) + 1 & k \geq v_m \end{cases}$$

only need to consider one of the denominations

Chapter 16. Greedy Algorithms

A greedy algorithm

$$c(k) = \min \begin{cases} c(k - v_1) + 1 & k \geq v_1 \\ c(k - v_2) + 1 & k \geq v_2 \\ \dots \\ c(k - v_m) + 1 & k \geq v_m \end{cases}$$

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.

Chapter 16. Greedy Algorithms

A greedy algorithm

$$c(k) = \min \begin{cases} c(k - v_1) + 1 & k \geq v_1 \\ c(k - v_2) + 1 & k \geq v_2 \\ \dots \\ c(k - v_m) + 1 & k \geq v_m \end{cases}$$

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.

Chapter 16. Greedy Algorithms

A greedy algorithm

$$c(k) = \min \begin{cases} c(k - v_1) + 1 & k \geq v_1 \\ c(k - v_2) + 1 & k \geq v_2 \\ \dots \\ c(k - v_m) + 1 & k \geq v_m \end{cases}$$

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.

but not all coin denominations have such property,

Chapter 16. Greedy Algorithms

A greedy algorithm

$$c(k) = \min \begin{cases} c(k - v_1) + 1 & k \geq v_1 \\ c(k - v_2) + 1 & k \geq v_2 \\ \dots \\ c(k - v_m) + 1 & k \geq v_m \end{cases}$$

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.

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.

Proof: Assume A is the optimal solution for n. Consider various cases

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.

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 $5 \in A$.

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.

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 $B = A - \{1, 1, 1, 1, 1, 1, 1, 1, 1, 1\} \cup \{10\}$,

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 $B = A - \{1, 1, 1, 1, 1, 1, 1, 1, 1, 1\} \cup \{10\}$, 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 $5 \notin A$, 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 $B = A - \{1,1,1,1,1,1,1,1,1,1\} \cup \{10\}$, 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 $5 \notin A$, 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 $B = A - \{1,1,1,1,1,1,1,1,1,1\} \cup \{10\}$, 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.

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 $B = A - \{1,1,1,1,1,1,1,1,1,1\} \cup \{10\}$, 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 $10 \in A$;

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 $B = A - \{1,1,1,1,1,1,1,1,1,1\} \cup \{10\}$, 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 $B = A - \{5,5\} \cup \{10\}$,

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 $B = A - \{1,1,1,1,1\} \cup \{5\}$. $|B| < |A|$, 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 $B = A - \{1,1,1,1,1,1,1,1,1,1\} \cup \{10\}$, 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 $10 \in A$;

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 $B = A - \{1,1,1,1,1\} \cup \{5\}$. $|B| < |A|$, 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 $B = A - \{1,1,1,1,1,1,1,1,1,1\} \cup \{10\}$, 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 $10 \in A$;

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.

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.

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,

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, ...

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, $|C| > 1$

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 $25 \in A$;

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 $m = n - 10$.

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),

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 $25 \in A$.

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.

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 $k = n - 2 \times 10$.

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,

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 $\{10, 10, 10\} \subseteq A$,

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 $25 \notin A$, 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 $25 \in A$;

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 $B = A - \{10, 10, 10\} \cup \{25, 5\}$, $|B| < |A|$, which is impossible. So $25 \in A$;

    Otherwise, $5 \geq k < 10$, according to I, 5 is in the solution for k.

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 $B = A - \{10, 10, 10\} \cup \{25, 5\}$, $|B| < |A|$, 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, ... 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

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

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

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

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

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

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

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

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

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:

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)

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)

Review for Parts I, II, and III

Part II: Sorting and order statistics

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

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 kth smallest element

Review for Parts I, II, and III

Part III. Exhaustive search, DP and greedy algorithms

Review for Parts I, II, and III

Part III. Exhaustive search, DP and greedy algorithms

- exhaustive search by

Review for Parts I, II, and III

Part III. Exhaustive search, DP and greedy algorithms

- exhaustive search by
  simple enumeration of all possible solutions

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

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 $T(n) = T(n-1) + T(n-1-m)$, $m \geq 1$.

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 $T(n) = T(n-1) + T(n-1-m)$, $m \geq 1$.

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

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)

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

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

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.

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'

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

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

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
  proof of greedy-choice property (via swapping method)

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
  proof of greedy-choice property (via swapping method)
  efficient algorithms