Structure of the Course
Structure of the Course

- Part I. Introduction to Algorithms (Chapter 2)
- Part II. Fundamental Techniques (Chapters 4 - 6)
- Part III Advanced Algorithms (Chapters 7 - 10)
- Part IV Probabilistic Methods (Chapters 11 - 12)
Chapter 6. Dynamic Programming

6.1 The power of DNA sequence comparison

- cancer-causing oncogene
- cystic fibrosis
Chapter 6. Dynamic Programming

Chapter 6. Dynamic Programming
Chapter 6. Dynamic Programming

Chapter 6. Dynamic Programming
Chapter 6. Dynamic Programming

6.1 The power of DNA sequence comparison

- cancer-causing oncogene
- cystic fibrosis
6.2 The change problem

Given amount of money $M$, find a way to change $M$ into the smallest number of coins from denominations $c = \{c_1, c_2, \ldots, c_d\}$. 
Chapter 6. Dynamic Programming

6.2 The change problem

Given amount of money $M$, find a way to change $M$ into the smallest number of coins from denominations $c = \{c_1, c_2, \ldots, c_d\}$.

For example, $c = \{1, 5, 10, 25\}$ for the US money.
Dynamic programming seeks solutions that can be computed from solutions to subproblems;

Step 1: analysis of problem (top-down)

Assume $S$ contains minimum number of coins for $M = 77$ cents. Then there are at most 4 following scenarios to consider:

1. At least one penny coin is included in $S$; then $S$ should also contain minimum number of coins for $77 - 1 = 76$ cents;

2. At least one nickel coin is included in $S$; then $S$ should also contain minimum number of coins for $77 - 5 = 72$ cents;

3. At least one dime coin is included in $S$;

4. If one quarter coin is included in ....
Chapter 6. Dynamic Programming

Dynamic programming seeks solutions that can be computed from solutions to subproblems;

Step 1: analysis of problem (top-down)

Assume $S$ contains minimum number of coins for $M = 77$ cents. Then there are at most 4 following scenarios to consider.
Chapter 6. Dynamic Programming

Dynamic programming seeks solutions that can be computed from solutions to subproblems;

Step 1: analysis of problem (*top-down*)

Assume $S$ contains minimum number of coins for $M = 77$ cents. Then there are at most 4 following scenarios to consider:

(1) At least one penny coin is included in $S$; then $S$ should also contain minimum number of coins for $77 - 1 = 76$ cents;
Dynamic programming seeks solutions that can be computed from solutions to subproblems;

**Step 1: analysis of problem** (*top-down*)

Assume $S$ contains minimum number of coins for $M = 77$ cents. Then there are at most 4 following scenarios to consider:

1. At least one penny coin is included in $S$; then $S$ should also contain minimum number of coins for $77 - 1 = 76$ cents;

2. At least one nickel coin is included in $S$; then $S$ should also contain minimum number of coins for $77 - 5 = 72$ cents;
Dynamic programming seeks solutions that can be computed from solutions to subproblems;

**Step 1: analysis of problem (top-down)**

Assume $S$ contains minimum number of coins for $M = 77$ cents. Then there are at most 4 following scenarios to consider:

1. At least one penny coin is included in $S$; then $S$ should also contain minimum number of coins for $77 - 1 = 76$ cents;

2. At least one nickel coin is included in $S$; then $S$ should also contain minimum number of coins for $77 - 5 = 72$ cents;

3. At least one dime coin is included in $S$; ....

4. If one quarter coin is included in ....
But we do not know which one is the best option.

Step 2: define objective function and formulate recurrences (top-down) instead, we define a single numerical value on solution: the smallest number of coins used:

\[
\text{smallestNumCoins}(M) = \min \begin{cases} 
\text{smallestNumCoins}(M-1) + 1, \\
\text{smallestNumCoins}(M-5) + 1, \\
\text{smallestNumCoins}(M-10) + 1, \\
\text{smallestNumCoins}(M-25) + 1 
\end{cases}
\]
Chapter 6. Dynamic Programming

But we do not known which one is the best option.

But we can try all possible options. So the best solution on $M = 77$ is the best out (1), (2), (3), and (4).
Chapter 6. Dynamic Programming

But we do not know which one is the best option.

But we can try all possible options. So the best solution on $M = 77$ is the best out (1), (2), (3), and (4).

But it is difficult to represent 'solutions'.
Chapter 6. Dynamic Programming

But we do not know which one is the best option.

But we can try all possible options. So the best solution on $M = 77$ is the best out (1), (2), (3), and (4).

But it is difficult to represent 'solutions'.

Step 2: define objective function
and formulate recurrences (*top-down*)

\[
\begin{align*}
\text{smallestNumCoins}(M) &= \min \left\{ \text{smallestNumCoins}(M-1) + 1, \right. \\
&\phantom{=} \left. \text{smallestNumCoins}(M-5) + 1, \text{smallestNumCoins}(M-10) + 1, \text{smallestNumCoins}(M-25) + 1 \right\}
\end{align*}
\]
Chapter 6. Dynamic Programming

But we do not know which one is the best option.

But we can try all possible options. So the best solution on $M = 77$ is the best out (1), (2), (3), and (4).

But it is difficult to represent 'solutions'.

**Step 2:** define objective function
and formulate recurrences (*top-down*)

Instead, we define a single numerical value on solution: *the smallest number of coins used*:
Chapter 6. Dynamic Programming

But we do not know which one is the best option.

But we can try all possible options. So the best solution on $M = 77$ is the best out (1), (2), (3), and (4).

But it is difficult to represent 'solutions'.

Step 2: define objective function and formulate recurrences (top-down)

Instead, we define a single numerical value on solution: the smallest number of coins used:

$$\text{smallestNumCoins}(M) = \min \left\{ \text{smallestNumCoins}(M - 1) + 1, \right.$$  
$$\text{smallestNumCoins}(M - 5) + 1,$$
$$\text{smallestNumCoins}(M - 10) + 1,$$
$$\text{smallestNumCoins}(M - 25) + 1 \right\}$$

a recurrence to the numerical answer.
Chapter 6. Dynamic Programming

Step 3: computing \textit{smallestNumCoins} with an algorithm
Chapter 6. Dynamic Programming

Step 3: computing $\text{smallestNumCoins}$ with an algorithm

A straightforward top-down recursive algorithm:

$\text{StraightForwardRecursiveChanges}(M, c)$
1. if $M = 0$
2. return (0)
3. else
4. $v_{\text{min}} \leftarrow M$
5. for $i \leftarrow 1$ to $|c|$
6. if $M - c_i \geq 0$
7. $v_i \leftarrow \text{StraightForwardRecursiveChanges}(M - c_i, c) + 1$
8. if $v_i < v_{\text{min}}$
9. $v_{\text{min}} \leftarrow v_i$
10. return ($v_{\text{min}}$)

Apparently there are a lot of re-computations.
Step 3: computing *smallestNumCoins* with an algorithm

A straightforward top-down recursive algorithm:

\texttt{StraightForwardRecursiveChanges}(M, c)
1. \textbf{if} \( M = 0 \)
2. \hspace{1em} \textbf{return} \( (0) \)
3. \textbf{else}
4. \hspace{1em} \( v_{\text{min}} \leftarrow M \)
5. \hspace{1em} \textbf{for} \( i \leftarrow 1 \text{ to } |c| \)
6. \hspace{2em} \textbf{if} \( M - c_i \geq 0 \)
7. \hspace{3em} \( v_i \leftarrow \texttt{StraightForwardRecursiveChanges}(M - c_i, c) + 1 \)
8. \hspace{1em} \textbf{if} \( v_i < v_{\text{min}} \)
9. \hspace{2em} \( v_{\text{min}} \leftarrow v_i \)
10. \textbf{return} \( (v_{\text{min}}) \)

Apparently there are a lot of re-computations.
A less naive, top-down recursive algorithm by keeping a table $T_1,\ldots,M$
A less naive, top-down recursive algorithm by keeping a table $T_{1,...,M}$

Initially, $T_k = -1$, for all $k = 1, 2, \ldots, M$.

\texttt{LessNaiveRecursiveChanges}(M, c)

1. \textbf{if} $M = 0$
2. \hspace{1em} \textbf{return} 0
3. \textbf{else}
4. \hspace{1em} $v_{\text{min}} \leftarrow M$
5. \hspace{1em} \textbf{for} $i \leftarrow 1$ \textbf{to} $|c|$
6. \hspace{2em} \textbf{if} $M - c_i \geq 0$
7. \hspace{3em} \textbf{if} $T_{M-c_i} = -1$
8. \hspace{4em} $T_{M-c_i} \leftarrow \text{LessNaiveRecursiveChanges}(M - c_i, c)$
9. \hspace{3em} \textbf{if} $T_{M-c_i} + 1 < v_{\text{min}}$
10. \hspace{4em} $v_{\text{min}} \leftarrow T_{M-c_i} + 1$
11. \hspace{1em} \textbf{return}
12. \hspace{1em} $T_M \leftarrow v_{\text{min}}$

Note: $T$ is global, as a ”communication media”.
Chapter 6. Dynamic Programming

A bottom-up, iterative algorithm
Chapter 6. Dynamic Programming

A bottom-up, iterative algorithm

\textsc{DPChanges}(n, c, T)
1. \hspace{1cm} T_0 = 0
2. \hspace{1cm} \textbf{for} n \leftarrow 1 \textbf{ to } M
3. \hspace{1cm} v_{\text{min}} \leftarrow n
4. \hspace{1cm} \textbf{for} i \leftarrow 1 \textbf{ to } |c|
5. \hspace{2cm} \textbf{if} n - c_i \geq 0
6. \hspace{3cm} \textbf{if} T_{n-c_i} + 1 \leq v_{\text{min}}
7. \hspace{4cm} v_{\text{min}} \leftarrow T_{n-c_i} + 1
8. \hspace{1cm} T_n \leftarrow v_{\text{min}}
9. \hspace{1cm} \textbf{return}
Chapter 6. Dynamic Programming

Step 4: Compute a solution, not just the numerical solution
Chapter 6. Dynamic Programming

Step 4: Compute a solution, not just the numerical solution

We keep another array $coinFrom$ to record how and from which amount of money a coin was generated (the underlined parts).
Chapter 6. Dynamic Programming

Step 4: Compute a solution, not just the numerical solution

We keep another array \( coinFrom \) to record how and from which amount of money a coin was generated (the underlined parts).

\[
\text{DPChanges}(M, c, T, \text{coinFrom})
\]

1. \( T_0 = 0; \ \text{coinFrom}_0 = 0; \)
2. \( \textbf{for } n \leftarrow 1 \textbf{ to } M \)
3. \( v_{\text{min}} \leftarrow n \)
4. \( \textbf{for } i \leftarrow 1 \textbf{ to } |c| \)
5. \( \textbf{if } n - c_i \geq 0 \)
6. \( \textbf{if } T_{n-c_i} + 1 \leq v_{\text{min}} \)
7. \( v_{\text{min}} \leftarrow T_{n-c_i} + 1; \ \text{from} \leftarrow n - c_i \)
8. \( T_n \leftarrow v_{\text{min}}; \ \text{coinFrom}_n \leftarrow \text{from} \)
9. \( \text{return} \)
Example:

\[ M = 7, \ c = \{1, 2, 4\}, \ c_1 = 1, c_2 = 2, c_3 = 4 \]

the result of running algorithm DPC\textsc{HANGES}:
Example:

\[ M = 7, \ c = \{1, 2, 4\}, \ c_1 = 1, c_2 = 2, c_3 = 4 \]

the result of running algorithm \textsc{DPC changes}:

<table>
<thead>
<tr>
<th>( n ) (cents)</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T ) (minimum number of coins)</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>( \text{coinFrom} ) (other than the last coin)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
Chapter 6. Dynamic Programming

6.4 Edit distance and alignments

edit distance: allowing the alignment of two sequences of different lengths
Chapter 6. Dynamic Programming

6.4 Edit distance and alignments

edit distance: allowing the alignment of two sequences of different lengths different from Hamming distance
Chapter 6. Dynamic Programming

6.4 Edit distance and alignments

edit distance: allowing the alignment of two sequences of different lengths different from Hamming distance

edit operations: substitution, insertion, and deletion
Chapter 6. Dynamic Programming

example:

- TGCATAT
- TGCATA
- TGCAT
- ATGCAT
- ATCCAT
- ATCCGAT
- ATGCATAT
- ATGCTAT
- ATCCTAT
- ATCCGAT
example:

TGCATAT by deleting last T
TGCATA by deleting last A
TGCAT by inserting A in the front
ATGCAT by substituting C for G in the third position
ATCCCAT by inserting G before the last A
ATCCGAT
**Chapter 6. Dynamic Programming**

**example:**

- TGCATAT by deleting last T
- TGCATA by deleting last A
- TGCAT by inserting A in the front
- ATGCAT by substituting C for G in the third position
- ATCCAT by inserting G before the last A
- ATCCGAT

and another series of operations:

- TGCATAT by inserting A at the front
- ATGCATAT by deleting the second A
- ATGCTAT by substituting C for G
- ATCCTAT by substituting G for the second T
- ATCCGAT
Chapter 6. Dynamic Programming

Example:

TGCATAT by deleting last T
TGCATA by deleting last A
TGCAT by inserting A in the front
ATGCAT by substituting C for G in the third position
ATCCAT by inserting G before the last A
ATCCGAT

and another series of operations:

TGCATAT by inserting A at the front
ATGCATAT by deleting the second A
ATGCTAT by substituting C for G
ATCCTAT by substituting G for the second T
ATCCGAT

These two series of operations correspond to the alignments:

-TGC-ATAT          -TGCATAT
ATCCGAT--          ATCC-GAT
Given an alignment, we can present it as a path in a grid:

AT–GTTAT–
ATCGT–A–C
Chapter 6. Dynamic Programming

Given an alignment, we can present it as a path in a grid:

AT–GTTAT–
ATCGT–A–C

(0, 0) → (1, 1) → (2, 2) → (2, 3) → (3, 4) → (4, 5) → (5, 5) → (6, 6) → (7, 6) → (7, 7)
Chapter 6. Dynamic Programming

Given an alignment, we can present it as a path in a grid:

\[
\begin{array}{c}
\text{AT-GTTAT-} \\
\text{ATCGT-A-C}
\end{array}
\]

\[
(0, 0) \rightarrow (1, 1) \rightarrow (2, 2) \rightarrow (2, 3) \rightarrow (3, 4) \rightarrow (4, 5) \rightarrow (5, 5) \rightarrow (6, 6) \rightarrow (7, 6) \rightarrow (7, 7)
\]

Given two sequences,

(1) there are more than one possible alignments;
(2) each alignment has a score (to be defined);
(3) each alignment corresponds to a path on a grid;
(4) the goal is to find a path (i.e., an alignment) with a highest score.
Chapter 6. Dynamic Programming

6.5 Longest common subsequences

A simplified scenario:
6.5 Longest common subsequences

A simplified scenario:

subsequence:
if $s = \text{ATTGCTA}$, the both AGC and ATTA are subsequences of $s$. 
6.5 Longest common subsequences

A simplified scenario:

subsequence:
if $s = \text{ATTGCTA}$, the both AGC and ATTA are subsequences of $s$.

common subsequence:
TCTA is a common subsequence of two sequences ATCTGAT and TGCATA
6.5 Longest common subsequences

A simplified scenario:

subsequence:
   if $s = \text{ATTGCTA}$, the both AGC and ATTA are subsequences of $s$.

common subsequence:
   TCTA is a common subsequence of two sequences ATCTGAT and TGCATA

Finding a common subsequence is a simple case for alignment:

AT–C–TGAT
  –TGCAT–A–

which only count the number of matches and not penalizing insertions or deletions or mismatches.
Chapter 6. Dynamic Programming

Longest Common Subsequence problem:

Find the longest subsequence common to two strings.

Input: two strings, \( v = v_1 \ldots v_n \) and \( w = w_1 \ldots w_m \);
Output: The longest common subsequence (LCS) of \( v \) and \( w \).
**Longest Common Subsequence problem:**

*Find the longest subsequence common to two strings.*

**Input:** two strings, $v = v_1 \ldots v_n$ and $w = w_1 \ldots w_m$;

**Output:** The longest common subsequence (LCS) of $v$ and $w$.

In the coin changing problem, we analyzed one coin at time. We looked at the last coin added.
Chapter 6. Dynamic Programming

Longest Common Subsequence problem:

Find the longest subsequence common to two strings.

Input: two strings, $v = v_1 \ldots v_n$ and $w = w_1 \ldots w_m$;
Output: The longest common subsequence (LCS) of $v$ and $w$.

In the coin changing problem, we analyzed one coin at time. We looked at the last coin added.

Here we analyze one character at a time. But since this problem concerns two sequences, we may analyze two characters (one on each sequence) at a time. We will look at the last characters on the sequences.
We define a numerical value to pursue, instead of pursuing directly the common subsequence:

\[
\begin{align*}
\textbf{s} & \quad 	ext{(length of LCS)} \\
\textbf{s}_{i,j} & \quad \text{be the length of an LCS between prefix string } v_1 v_2 \ldots v_i \\
& \quad \text{and prefix string } w_1 w_2 \ldots w_j. \\
\text{There are only three possibilities that would happen to } v_i \text{ and } w_j \text{ during an alignment, we have the recurrence for } \textbf{s}_{i,j}: \\
\textbf{s}_{i,j} & = \max \left\{ \textbf{s}_{i-1,j-1} + 1, \right. \\
& \left. \quad \textbf{s}_{i-1,j}, \textbf{s}_{i,j-1} \right\}, \\
& \text{if } v_i = w_j \\
\textbf{s}_{0,j} & = \textbf{s}_{i,0} = 0 \quad \text{for all } 0 \leq i \leq n \text{ and } 0 \leq j \leq m. \\
\end{align*}
\]
Chapter 6. Dynamic Programming

We define a numerical value to pursue, instead of pursuing directly the common subsequence:

Define: \( s_{i,j} \) be the length of an LCS between prefix string \( v_1v_2 \ldots v_i \) and prefix string \( w_1w_2 \ldots w_j \).

There are only three possibilities that would happen to \( v_i \) and \( w_j \) during an alignment, we have the recurrence for \( s_{i,j} \):

\[
\begin{align*}
    s_{i,j} &= \max \left\{ s_{i-1,j-1} + 1, \begin{cases} 
        s_{i-1,j} + 0, \\
        s_{i,j-1} + 0, \\
        s_{0,j} = s_{i,0} = 0
    \end{cases} \right. \\
\end{align*}
\]

for all \( 0 \leq i \leq n \) and \( 0 \leq j \leq m \).

Again, we do not want to directly implement the recurrence using the top-down recursive approaches.
Chapter 6. Dynamic Programming

We define a numerical value to pursue, instead of pursuing directly the common subsequence:

*Define:* $s_{i,j}$ be the length of an LCS between prefix string $v_1v_2\ldots v_i$ and prefix string $w_1w_2\ldots w_j$.

There are only three possibilities that would happen to $v_i$ and $w_j$ during an alignment, we have the recurrence for $s_{i,j}$:

\[
 s_{i,j} = \begin{cases} 
 s_{i-1,j-1} + 1, & \text{if } v_i = w_j \\
 s_{i-1,j}, & \text{if } v_i \neq w_j \\
 s_{i,j-1}, & \text{if } v_i \neq w_j 
\end{cases}
\]
Chapter 6. Dynamic Programming

We define a numerical value to pursue, instead of pursuing directly the common subsequence:

*Define:* $s_{i,j}$ be the length of an LCS between prefix string $v_1v_2\ldots v_i$ and prefix string $w_1w_2\ldots w_j$.

There are only three possibilities that would happen to $v_i$ and $w_j$ during an alignment, we have the recurrence for $s_{i,j}$:

$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + 1, & \text{if } v_i = w_j \\ s_{i-1,j} + 0, & \\ s_{i,j-1} + 0, & \end{cases}$$

for all $0 \leq i \leq n$ and $0 \leq j \leq m$. Again, we do not want to directly implement the recurrence using the top-down recursive approaches.
Chapter 6. Dynamic Programming

We define a numerical value to pursue, instead of pursuing directly the common subsequence:

Define: $s_{i,j}$ be the length of an LCS between prefix string $v_1v_2\ldots v_i$ and prefix string $w_1w_2\ldots w_j$.

There are only three possibilities that would happen to $v_i$ and $w_j$ during an alignment, we have the recurrence for $s_{i,j}$:

$$ s_{i,j} = \max \begin{cases} 
  s_{i-1,j-1} + 1, & \text{if } v_i = w_j \\
  s_{i-1,j} + 0, & \\
  s_{i,j-1} + 0, & 
\end{cases} $$

$s_{0,j} = s_{i,0} = 0$ for all $0 \leq i \leq n$ and $0 \leq j \leq m$. 

Again, we do not want to directly implement the recurrence using the top-down recursive approaches.
We define a numerical value to pursue, instead of pursuing directly the common subsequence:

**Define:** $s_{i,j}$ be the length of an LCS between prefix string $v_1v_2 \ldots v_i$ and prefix string $w_1w_2 \ldots w_j$.

There are only three possibilities that would happen to $v_i$ and $w_j$ during an alignment, we have the recurrence for $s_{i,j}$:

$$s_{i,j} = \max \begin{cases} 
  s_{i-1,j-1} + 1, & \text{if } v_i = w_j \\
  s_{i-1,j} + 0, & \\
  s_{i,j-1} + 0, &
\end{cases}$$

$s_{0,j} = s_{i,0} = 0$ for all $0 \leq i \leq n$ and $0 \leq j \leq m$.

Again, we do not want to directly implement the recurrence using the top-down recursive approaches.
LCS\((v, w)\)
1. for \(i \leftarrow 0\) to \(n\)
2. \(s_{i, 0} \leftarrow 0\)
3. for \(j \leftarrow 0\) to \(m\)
4. \(s_{0, j} \leftarrow 0\)
5. for \(i \leftarrow 1\) to \(n\)
6. for \(j \leftarrow 1\) to \(m\)
7. if \(v_i = w_j\)
8. \(a \leftarrow 1\)
9. else
10. \(a \leftarrow -\infty\)
11. if \(s_{i-1, j-1} + a > \max\{s_{i, j-1}, s_{i-1, j}\}\)
12. \(s_{i, j} \leftarrow s_{i-1, j-1} + 1; \quad b_{i, j} \leftarrow \searrow\)
13. else
14. if \(s_{i, j-1} > \max\{s_{i-1, j-1} + a, s_{i-1, j}\}\)
15. \(s_{i, j} \leftarrow s_{i, j-1}; \quad b_{i, j} \leftarrow \leftarrow\)
16. else
17. \(s_{i, j} \leftarrow s_{i-1, j}; \quad b_{i, j} \leftarrow \uparrow\)
18. return
Chapter 6. Dynamic Programming

Figure 6.14 on page 173 (left table for LCS).

<table>
<thead>
<tr>
<th>table</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T )</td>
<td>0</td>
<td>T</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1 A</td>
<td>0</td>
<td>0↑</td>
<td>0↑</td>
<td>0↑</td>
<td>1↖</td>
<td>1←</td>
<td>1←</td>
</tr>
<tr>
<td>2 T</td>
<td>0</td>
<td>1↖</td>
<td>1←</td>
<td>1←</td>
<td>1↑</td>
<td>2↖</td>
<td>2←</td>
</tr>
<tr>
<td>3 C</td>
<td>0</td>
<td>1↑</td>
<td>1↑</td>
<td>2↖</td>
<td>2←</td>
<td>2↑</td>
<td>2↑</td>
</tr>
<tr>
<td>4 T</td>
<td>0</td>
<td>1↖</td>
<td>1↑</td>
<td>2↑</td>
<td>2↑</td>
<td>3↖</td>
<td>3←</td>
</tr>
<tr>
<td>5 G</td>
<td>0</td>
<td>1↑</td>
<td>2↖</td>
<td>2↑</td>
<td>2↑</td>
<td>3↑</td>
<td>3↑</td>
</tr>
<tr>
<td>6 A</td>
<td>0</td>
<td>1↑</td>
<td>2↑</td>
<td>2↑</td>
<td>3↖</td>
<td>3↑</td>
<td>4↖</td>
</tr>
<tr>
<td>7 T</td>
<td>0</td>
<td>1↖</td>
<td>2↑</td>
<td>2↑</td>
<td>3↑</td>
<td>4↖</td>
<td>4↑</td>
</tr>
</tbody>
</table>

In notation of alignment:

\[ \text{A T - C - T G A T} \]
\[ \text{- T G C A T - A -} \]
\[ \text{↑↖←↖←↖↑↖↑} \]

The LCS is between the two sequences is TCTA
**Chapter 6. Dynamic Programming**

Figure 6.14 on page 173 (left table for LCS).

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>T</strong></td>
<td>0</td>
<td>T</td>
<td>G</td>
<td>C</td>
<td>A</td>
<td>T</td>
<td>A</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1 A</td>
<td>0</td>
<td>0↑</td>
<td>0↑</td>
<td>0↑</td>
<td>1↖</td>
<td>1←</td>
<td>1←</td>
</tr>
<tr>
<td>2 T</td>
<td>0</td>
<td>1↖</td>
<td>1←</td>
<td>1←</td>
<td>1↑</td>
<td>2↖</td>
<td>2←</td>
</tr>
<tr>
<td>3 C</td>
<td>0</td>
<td>1↑</td>
<td>1↑</td>
<td>2↖</td>
<td>2←</td>
<td>2↑</td>
<td>2↑</td>
</tr>
<tr>
<td>4 T</td>
<td>0</td>
<td>1↖</td>
<td>1↑</td>
<td>2↑</td>
<td>2↑</td>
<td>3↖</td>
<td>3←</td>
</tr>
<tr>
<td>5 G</td>
<td>0</td>
<td>1↑</td>
<td>2↖</td>
<td>2↑</td>
<td>2↑</td>
<td>3↑</td>
<td>3↑</td>
</tr>
<tr>
<td>6 A</td>
<td>0</td>
<td>1↑</td>
<td>2↑</td>
<td>2↑</td>
<td>3↖</td>
<td>3↑</td>
<td>4↖</td>
</tr>
<tr>
<td>7 T</td>
<td>0</td>
<td>1↖</td>
<td>2↑</td>
<td>2↑</td>
<td>3↑</td>
<td>4↖</td>
<td>4↑</td>
</tr>
</tbody>
</table>

In notation of alignment:

\[
\begin{align*}
A & T - C - T G A T \\
- & T G C A T - A - \\
\uparrow & \leftarrow \leftarrow \leftarrow \uparrow \uparrow
\end{align*}
\]

The LCS is between the two sequences is TCTA
Retrieve the corresponding LCS from table $b_{i,j}$. 

```python
PrintLCS(b,v,i,j)
1. if $i = 0$ or $j = 0$
2. return
3. if $b_{i,j} = \begin{cases} \\
4. PrintLCS(b,v,i-1,j-1) \\
5. print v_i \\
6. \end{cases}$
7. else
8. if $b_{i,j} = \begin{cases} \\
9. PrintLCS(b,v,i-1,j) \\
10. \end{cases}$
11. else
12. PrintLCS(b,v,i,j-1)
```
Retrieve the corresponding LCS from table $b_{i,j}$.

The following recursive function prints the found LCS between $v_1v_2 \ldots v_i$ and string $w_1w_2 \ldots w_j$:

\begin{verbatim}
PRINTLCS(b, v, i, j)
1. if $i = 0$ or $j = 0$
2.     return
3. if $b_{i,j} = '↖'$
4.     PRINTLCS(b, v, i - 1, j - 1)
5.     print $v_i$
6. else
7.     if $b_{i,j} = '↑'$
8.         PRINTLCS(b, v, i - 1, j)
9.     else
10.    PRINTLCS(b, v, i, j - 1)
\end{verbatim}
Chapter 6. Dynamic Programming

The LCS computes the similarity between two sequences, thus to maximize the length.

On the other hand, edit distance is to measure the similarity between the two using distance, thus to minimize the score.

\[ d_{i,j} = \min \begin{cases} 
  d_{i-1,j-1}, & \text{if } v_i = w_j \\
  d_{i-1,j} + 1, & \\
  d_{i,j-1} + 1,
\end{cases} \]

For alignment, usually we are looking for higher scores.
Chapter 6. Dynamic Programming

6.6 Global pairwise sequence alignment

We need scores for matches, substitutions, deletions and insertions.

1. Scoring matrices $\delta$, $4 \times 4$ for nucleic acids and $20 \times 20$ for proteins, which include scores for matches and substitutions.

2. For insertion and deletion (indel, $'$), a penalty is applied.

3. If the penalty is uniform for every gap, thus linear, then it can be built into the scoring matrices, resulting in $5 \times 5$ and $21 \times 21$ matrices.

4. But often the gap penalty is not uniform. For example, affine gap penalty is defined as $o + e (l - 1)$ for $l$ consecutive gaps, where $o$ is the gap opening penalty and $e$ is the gap extension penalty.
6.6 Global pairwise sequence alignment

We need scores for matches, substitutions, deletions and insertions.

1. Scoring matrices $\delta$, $4 \times 4$ for nucleic acids and $20 \times 20$ for proteins, which include scores for matches and substitutions.
Chapter 6. Dynamic Programming

6.6 Global pairwise sequence alignment

We need scores for matches, substitutions, deletions and insertions.

(1) Scoring matrices $\delta$, $4 \times 4$ for nucleic acids and $20 \times 20$ for proteins, which include scores for matches and substitutions.

(2) For insertion and deletion (indel, ‘-‘), a penalty is applied.
6.6 Global pairwise sequence alignment

We need scores for matches, substitutions, deletions and insertions.

(1) Scoring matrices \(\delta\), \(4 \times 4\) for nucleic acids and \(20 \times 20\) for proteins, which include scores for matches and substitutions.

(2) For insertion and deletion (\textit{indel}, \(\text{'}-\text{'}\)), a penalty is applied.

(3) If the penalty is uniform for every gap, thus linear, then it can be built into the scoring matrices, resulting in \(5 \times 5\) and \(21 \times 21\) matrices.
6.6 Global pairwise sequence alignment

We need scores for matches, substitutions, deletions and insertions.

(1) Scoring matrices $\delta$, $4 \times 4$ for nucleic acids and $20 \times 20$ for proteins, which include scores for matches and substitutions.

(2) For insertion and deletion (indel, ‘−’), a penalty is applied.

(3) If the penalty is uniform for every gap, thus linear, then it can be built into the scoring matrices, resulting in $5 \times 5$ and $21 \times 21$ matrices.

(4) But often the gap penalty is not uniform. For example, affine gap penalty is defined as $o + e(l - 1)$ for $l$ consecutive gaps, where $o$ is the gap opening penalty and $e$ is the gap extension penalty.
Chapter 6. Dynamic Programming

Global Alignment Problem

Define:

$$s_{i,j}$$ to be the maximum score to align $$v_1...v_i$$ and $$w_1...w_j$$,
given scoring matrix $$\delta$$.

Like LCS, we do not know which of the three scenarios is the best, so
$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + \delta(v_i, w_j), \\ s_{i-1,j} + \delta(v_i, -), \\ s_{i,j-1} + \delta(-, w_j) \end{cases}$$
Chapter 6. Dynamic Programming

**Global Alignment Problem**

*Find the best alignment between two strings under a given scoring matrix.*

**Input:** String $v$, $w$ and a scoring matrix $\delta$.

**Output:** An alignment of $v$ and $w$ whose score (as defined by the matrix $\delta$) is the maximum among possible alignments of $v$ and $w$. 

Define:

$$s_{i,j} \text{ to be the maximum score to align } v_1 \ldots v_i \text{ and } w_1 \ldots w_j,$$

given scoring matrix $\delta$.

Like LCS, we do not know which of the three scenarios is the best, so

$$s_{i,j} = \max \begin{cases} 
  s_{i-1,j-1} + \delta(v_i, w_j) \\
  s_{i-1,j} + \delta(v_i, -) \\
  s_{i,j-1} + \delta(-, w_j) 
\end{cases}$$

where $\delta(v_i, -)$ is the gap penalty for aligning $v_i$ to a gap, etc..
Chapter 6. Dynamic Programming

**Global Alignment Problem**

*Find the best alignment between two strings under a given scoring matrix.*

**Input:** String $v$, $w$ and a scoring matrix $\delta$.

**Output:** An alignment of $v$ and $w$ whose score (as defined by the matrix $\delta$) is the maximum among possible alignments of $v$ and $w$.

Define: $s_{i,j}$ to be the maximum score to align $v_1 \ldots v_i$ and $w_1 \ldots w_j$, given scoring matrix $\delta$. 

Like LCS, we do not know which of the three scenarios is the best, so $s_{i,j} = \max \begin{cases} s_{i-1,j-1} + \delta(v_i, w_j) \\ s_{i-1,j} + \delta(v_i, -) \\ s_{i,j-1} + \delta(-, w_j) \end{cases}$.
Chapter 6. Dynamic Programming

**Global Alignment Problem**

*Find the best alignment between two strings under a given scoring matrix.*

**Input:** String $v$, $w$ and a scoring matrix $\delta$.

**Output:** An alignment of $v$ and $w$ whose score (as defined by the matrix $\delta$) is the maximum among possible alignments of $v$ and $w$.

Define: $s_{i,j}$ to be the maximum score to align $v_1 \ldots v_i$ and $w_1 \ldots w_j$, given scoring matrix $\delta$.

Like LCS, we do not know which of the three scenarios is the best, so

$$s_{i,j} = \max \left\{ s_{i-1,j-1} + \delta(v_i, w_j), s_{i-1,j} + \delta(v_i, -), s_{i,j-1} + \delta(-, w_j) \right\}$$

where $\delta(v_i, -)$ is the gap penalty for aligning $v_i$ to a gap, etc..
Chapter 6. Dynamic Programming

**GLOBALPAIRWISEALIGNMENT**(v, w)

0. \( s_{0,0} = 0 \)
1. **for** \( i \leftarrow 1 \) to \( n \) \hspace{1cm} initializing the first column
2. \( s_{i,0} \leftarrow \sum_{k=1}^{i} \delta(v_k, -) \)
3. **for** \( j \leftarrow 1 \) to \( m \) \hspace{1cm} initializing the first row
4. \( s_{0,j} \leftarrow \sum_{k=1}^{j} \delta(-, w_k) \)
5. **for** \( i \leftarrow 1 \) to \( n \)
6. **for** \( j \leftarrow 1 \) to \( m \) \hspace{1cm} filling entries for the matrix
7. if \( s_{i-1,j-1} + \delta(v_i, w_j) > \max\{s_{i,j-1} + \delta(-, w_j), s_{i-1,j} + \delta(v_i, -)\} \)
8. \( s_{i,j} \leftarrow s_{i-1,j-1} + \delta(v_i, w_j); \quad b_{i,j} \leftarrow '\langle' \)
9. **else**
10. if \( s_{i-1,j-1} + \delta(-, w_j) > \max\{s_{i-1,j-1} + \delta(v_i, w_j), s_{i-1,j} + \delta(v_i, -)\} \)
11. \( s_{i,j} \leftarrow s_{i-1,j-1} + \delta(-, w_j); \quad b_{i,j} \leftarrow '\langle' \)
12. **else**
13. \( s_{i,j} \leftarrow s_{i-1,j} + \delta(v_i, -); \quad b_{i,j} \leftarrow '\uparrow' \)
14. **return**
6.9 Alignment with affine gap penalty

So far, we have adopted the gap penalties that are “column-independent”, such as $\delta(x, -)$. 
Chapter 6. Dynamic Programming

6.9 Alignment with affine gap penalty

So far, we have adopted the gap penalties that are “column-independent”, such as \( \delta(x, -) \).

When \( \delta(x, -) \) is a constant \(-\gamma\), where \( \gamma > 0 \) fixed regardless of \( x \), we can replace the recurrence for global alignment

\[
s_{i,j} = \max \begin{cases} 
  s_{i-1,j-1} + \delta(v_i, w_j) \\
  s_{i-1,j} + \delta(v_i, -) \\
  s_{i,j-1} + \delta(-, w_j)
\end{cases}
\]
6.9 Alignment with affine gap penalty

So far, we have adopted the gap penalties that are “column-independent”, such as $\delta(x, -)$.

When $\delta(x, -)$ is a constant $-\gamma$, where $\gamma > 0$ fixed regardless of $x$, we can replace the recurrence for global alignment

$$s_{i,j} = \max \begin{cases} 
  s_{i-1,j-1} + \delta(v_i, w_j) \\
  s_{i-1,j} - \gamma \\
  s_{i,j-1} - \gamma
\end{cases}$$

with

$$s_{i,j} = \max \begin{cases} 
  s_{i-1,j-1} + \delta(v_i, w_j) \\
  s_{i-1,j} - \gamma \\
  s_{i,j-1} - \gamma
\end{cases}$$
Chapter 6. Dynamic Programming

Now if gap penalty for a gap of length $l$ (i.e., number of single gaps) is defined as

$$\rho + (l - 1)\sigma$$

where $\rho > 0$ is a gap opening penalty and $\sigma > 0$ is gap extension penalty.
Chapter 6. Dynamic Programming

Now if gap penalty for a gap of length $l$ (i.e., number of single gaps) is defined as

$$\rho + (l - 1)\sigma$$

where $\rho > 0$ is a gap opening penalty and $\sigma > 0$ is gap extension penalty.

We cannot simply replace $-\gamma$ with $-\rho - (l - 1)\sigma$ because $\gamma$ is for just one single gap.
Chapter 6. Dynamic Programming

Solution-1: we consider all possible gap situations
Solution-1: we consider all possible gap situations

\[
s_{i,j} = \max \left\{ s_{i-1,j-1} + \delta(v_i, w_j), \max_{1 \leq l \leq j} \left\{ s_{i,j-l} - \rho - (l - 1)\sigma \right\}, \max_{1 \leq l \leq i} \left\{ s_{i-l,j} - \rho - (l - 1)\sigma \right\} \right\}
\]

Does it work? Yes. But how much time would it take to build the DP table? \( O(n^2) \) time is needed for GlobalPairwiseAlignment. The above recurrence would required \( O(n^3) \) time.
Chapter 6. Dynamic Programming

Solution-1: we consider all possible gap situations

$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + \delta(v_i, w_j) \\ \max_{1 \leq l \leq j} \{s_{i,j-l} - \rho - (l - 1)\sigma\} \\ \max_{1 \leq l \leq i} \{s_{i-l,j} - \rho - (l - 1)\sigma\} \end{cases}$$

Does it work?
Chapter 6. Dynamic Programming

Solution-1: we consider all possible gap situations

\[
s_{i,j} = \max \begin{cases} 
  s_{i-1,j-1} + \delta(v_i, w_j) \\
  \max_{1 \leq l \leq j} \{s_{i,j-l} - \rho - (l - 1)\sigma\} \\
  \max_{1 \leq l \leq i} \{s_{i-l,j} - \rho - (l - 1)\sigma\}
\end{cases}
\]

Does it work?

Yes.
Solution-1: we consider all possible gap situations

\[ s_{i,j} = \max \left\{ \begin{array}{l} s_{i-1,j-1} + \delta(v_i, w_j) \\ \max_{1 \leq l \leq j} \{s_{i,j-l} - \rho - (l - 1)\sigma\} \\ \max_{1 \leq l \leq i} \{s_{i-l,j} - \rho - (l - 1)\sigma\} \end{array} \right\} \]

Does it work?
Yes.

But how much time would it take to build the DP table?

\( O(n^2) \) time is needed for GlobalPairwiseAlignment.

The above recurrence would required \( O(n^3) \) time.
Solution-1: we consider all possible gap situations

\[
s_{i,j} = \max \left\{ \begin{array}{ll}
  s_{i-1,j-1} + \delta(v_i, w_j) \\
  \max_{1 \leq l \leq j} \{s_{i,j-l} - \rho - (l - 1)\sigma\} \\
  \max_{1 \leq l \leq i} \{s_{i-l,j} - \rho - (l - 1)\sigma\}
\end{array} \right\}
\]

Does it work?

Yes.

But how much time would it take to build the DP table?

\(O(n^2)\) time is needed for \texttt{GLOBALPAIRWISEALIGNMENT}. 
Chapter 6. Dynamic Programming

Solution-1: we consider all possible gap situations

\[
s_{i,j} = \max \begin{cases}
  s_{i-1,j-1} + \delta(v_i, w_j) \\
  \max_{1 \leq l \leq j} \{ s_{i,j-l} - \rho - (l - 1)\sigma \} \\
  \max_{1 \leq l \leq i} \{ s_{i-l,j} - \rho - (l - 1)\sigma \}
\end{cases}
\]

Does it work?

Yes.

But how much time would it take to build the DP table?

\(O(n^2)\) time is needed for \texttt{GLOBALPAIRWISEALIGNMENT}.

The above recurrence would required \(O(n^3)\) time.
Solution-2: actually gap opening and gap extension penalties can be handled separately. For this, we need multiple recurrence-based DP techniques.
Solution-2: actually gap opening and gap extension penalties can be handled separately. For this, we need multiple recurrence-based DP techniques.

\[ s_{i,j} = \max \left\{ s_{i-1,j-1} + \delta(v_i, w_j), I_{i,j}, D_{i,j} \right\} \]
Chapter 6. Dynamic Programming

Solution-2: actually gap opening and gap extension penalties can be handled separately. For this, we need multiple recurrence-based DP techniques.

\[ s_{i,j} = \max \begin{cases} 
  s_{i-1,j-1} + \delta(v_i, w_j) \\
  I_{i,j} \\
  D_{i,j} 
\end{cases} \]

where \( I_{i,j} \) is the score of optimal alignment between \( v_1 \ldots v_i \) and \( w_1 \ldots w_j \) for which \( w_j \) is inserted, and
Solution-2: actually gap opening and gap extension penalties can be handled separately. For this, we need multiple recurrence-based DP techniques.

\[
s_{i,j} = \max \begin{cases} 
    s_{i-1,j-1} + \delta(v_i, w_j) \\
    I_{i,j} \\
    D_{i,j}
\end{cases}
\]

where \( I_{i,j} \) is the score of optimal alignment between \( v_1 \ldots v_i \) and \( w_1 \ldots w_j \) for which \( w_j \) is inserted, and

\( D_{i,j} \) the score of optimal alignment between \( v_1 \ldots v_i \) and \( w_1 \ldots w_j \) for which \( v_i \) is deleted.
Chapter 6. Dynamic Programming

Thus,

\[ I_{i,j} = \max \left\{ I_{i,j-1} - \sigma \quad \text{extending the gap} \right\} \]
\[ s_{i,j-1} - \rho \quad \text{closing the gap} \]
Thus,

\[ I_{i,j} = \max \begin{cases} I_{i,j-1} - \sigma \\ s_{i,j-1} - \rho \end{cases} \text{ extending the gap} \]

\[ D_{i,j} = \max \begin{cases} D_{i-1,j} - \sigma \\ s_{i-1,j} - \rho \end{cases} \text{ closing the gap} \]
Chapter 6. Dynamic Programming

Thus,

\[ I_{i,j} = \max \begin{cases} \ I_{i,j-1} - \sigma \\ s_{i,j-1} - \rho \end{cases} \text{ extending the gap} \]

\[ D_{i,j} = \max \begin{cases} \ D_{i-1,j} - \sigma \\ s_{i-1,j} - \rho \end{cases} \text{ extending the gap} \]

We would need to computed three tables, one for each of \( s_{i,j}, I_{i,j} \) and \( D_{i,j} \).
Thus,

\[ I_{i,j} = \max \begin{cases} I_{i,j-1} - \sigma & \text{extending the gap} \\ s_{i,j-1} - \rho & \text{closing the gap} \end{cases} \]

\[ D_{i,j} = \max \begin{cases} D_{i-1,j} - \sigma & \text{extending the gap} \\ s_{i-1,j} - \rho & \text{closing the gap} \end{cases} \]

We would need to computed three tables, one for each of \( s_{i,j}, I_{i,j} \) and \( D_{i,j} \).

Each table can be computed in time \( O(n^2) \).
6.7 Scoring alignment

Consider alignment ATTGTTAT- with a simple model. Assume $p(T, C)$ to be the probability that $T$ aligns to $C$, then the score of (column 3) aligning $T$ with $C$ can be defined as the ratio:

$$\frac{p(T, C)}{q(T)q(C)}$$

The denominator is the probability for $T$ and $C$ to occur independently. If the score for column 3 is greater than 1, it means $T$ and $C$ are evolutionarily related; otherwise unrelated.
6.7 Scoring alignment

Consider alignment

ATTGTTAT-
ATCGT-A-C

with a simple model. Assume \( p(T, C) \) to be probability that \( T \) aligns to \( C \),
6.7 Scoring alignment

Consider alignment

\[
\begin{array}{c}
\text{ATTGTTAT-} \\
\text{ATCGT-A-C}
\end{array}
\]

with a simple model. Assume \( p(T, C) \) to be probability that \( T \) aligns to \( C \),
then the score of (column 3) aligning \( T \) with \( C \) can be defined as the ratio:

\[
\frac{p(T, C)}{q(T)q(C)}
\]

the denominator is the probability for \( T \) and \( C \) to occur independently.
Chapter 6. Dynamic Programming

6.7 Scoring alignment

Consider alignment

ATTGTTAT–
ATCGT–A–C

with a simple model. Assume \( p(T, C) \) to be probability that \( T \) aligns to \( C \), then the score of (column 3) aligning \( T \) with \( C \) can be defined as the ratio:

\[
\frac{p(T, C)}{q(T)q(C)}
\]

the denominator is the probability for \( T \) and \( C \) to occur independently.

If the score for column 3 is greater than 1, it means \( T \) and \( C \) are evolutionarily related; otherwise unrelated.
Alignment probability is \( \prod_{k=1}^{r} \frac{p(\bar{v}_k, \bar{w}_k)}{q(\bar{v}_k)q(\bar{w}_k)} \), i.e., product over \( r \) columns.
Alignment probability is \( \prod_{k=1}^{r} \frac{p(\bar{v}_k, \bar{w}_k)}{q(\bar{v}_k)q(\bar{w}_k)} \), i.e., product over \( r \) columns.

Taken the logarithm, the score becomes either positive or negative, and the product becomes summation,

which the sum of column scores!
Chapter 6. Dynamic Programming

PAM (point accepted mutations) matrices

Examine closely related protein sequences for mutation rates.
Chapter 6. Dynamic Programming

PAM (point accepted mutations) matrices

Examine closely related protein sequences for mutation rates.
Chapter 6. Dynamic Programming

One PAM: the amount of time in which an “average” protein mutates 1% of its amino acids.

Let $f(i, j)$ be the frequency that amino acids $i$ and $j$ are aligned.
Chapter 6. Dynamic Programming

One PAM: the amount of time in which an “average” protein mutates 1% of its amino acids.

Let \( f(i, j) \) be the frequency that amino acids \( i \) and \( j \) are aligned.

Let \( f(i) \) be the frequency of amino acid \( i \). Then \( \frac{f(i, j)}{f(i)f(j)} \) is to measure \( i \) and \( j \) are aligned as opposed to they occur independently.
Chapter 6. Dynamic Programming

One PAM: the amount of time in which an “average” protein mutates 1% of its amino acids.

Let \( f(i, j) \) be the frequency that amino acids \( i \) and \( j \) are aligned.

Let \( f(i) \) be the frequency of amino acid \( i \). Then \( \frac{f(i, j)}{f(i)f(j)} \) is to measure \( i \) and \( j \) are aligned as oppose to they occur independently.

Let \( f(i, j) = f(i)f(j|i) \), by considering \( f \) as a probability.
One PAM: the amount of time in which an “average” protein mutates 1% of its amino acids.

Let \( f(i, j) \) be the frequency that amino acids \( i \) and \( j \) are aligned.

Let \( f(i) \) be the frequency of amino acid \( i \). Then \( \frac{f(i, j)}{f(i)f(j)} \) is to measure \( i \) and \( j \) are aligned as oppose to they occur independently.

Let \( f(i, j) = f(i)f(j|i) \), by considering \( f \) as a probability.

Let \( g(i, j) = f(j|i) \) is the probability of mutating to \( j \) given amino acid \( i \).

So the measure is \( \frac{f(i, j)}{f(i)f(j)} = \frac{f(j|i)}{f(j)} = \frac{g(i, j)}{f(j)} \).
Chapter 6. Dynamic Programming

One PAM: the amount of time in which an “average” protein mutates 1% of its amino acids.

Let \( f(i, j) \) be the frequency that amino acids \( i \) and \( j \) are aligned.

Let \( f(i) \) be the frequency of amino acid \( i \). Then \( \frac{f(i, j)}{f(i)f(j)} \) is to measure \( i \) and \( j \) are aligned as oppose to they occur independently.

Let \( f(i, j) = f(i)f(j|i) \), by considering \( f \) as a probability.

Let \( g(i, j) = f(j|i) \) is the probability of mutating to \( j \) given amino acid \( i \).

So the measure is \( \frac{f(i, j)}{f(i)f(j)} = \frac{f(j|i)}{f(j)} = \frac{g(i, j)}{f(j)} \)

Taken logarithm, the measure becomes

\[
\log \frac{g(i, j)}{f(j)} = \log \frac{observed \ frequency}{expected \ frequency}
\]
PAM 1 matrix

The \((i,j)\) entry in the PAM 1 matrix is \(\log \frac{g(i,j)}{f(j)}\).
PAM 1 matrix

The \((i, j)\) entry in the PAM 1 matrix is \(\log \frac{g(i, j)}{f(j)}\).

Define \(G\) to be the matrix containing entries \(g(i, j)\). Then
PAM 1 matrix

The \((i, j)\) entry in the PAM 1 matrix is \(\log \frac{g(i, j)}{f(j)}\).

Define \(G\) to be the matrix containing entries \(g(i, j)\). Then

\[
PAM 1 \text{ matrix} = \log\left(\frac{G}{f(j)}\right),\text{ where } f(j) \text{ is applied to the } j\text{th column.}
\]
PAM 1 matrix

The \((i, j)\) entry in the PAM 1 matrix is \(\log \frac{g(i, j)}{f(j)}\).

Define \(G\) to be the matrix containing entries \(g(i, j)\). Then

\[
PAM 1\ \text{matrix} = \log(\frac{G}{f(j)}), \text{ where } f(j) \text{ is applied to the } j\text{th column.}
\]

Define \(G^n\) to be the matrix obtained from \(G\) by multiplying itself \(n\) times.
PAM 1 matrix

The \((i,j)\) entry in the PAM 1 matrix is \(\log \frac{g(i,j)}{f(j)}\).

Define \(G\) to be the matrix containing entries \(g(i,j)\). Then

\[
\text{PAM 1 matrix} = \log \left( \frac{G}{f(j)} \right), \text{ where } f(j) \text{ is applied to the } j\text{th column.}
\]

Define \(G^n\) to be the matrix obtained from \(G\) by multiplying itself \(n\) times.

\[
\text{PAM } n \text{ matrix is } \log \left( \frac{G^n}{f(j)} \right), \text{ where } f(j) \text{ is applied to } j\text{th column.}
\]
Chapter 6. Dynamic Programming

PAM 1 matrix

The \((i, j)\) entry in the PAM 1 matrix is \(\log \frac{g(i, j)}{f(j)}\).

Define \(G\) to be the matrix containing entries \(g(i, j)\). Then

\[
PAM \ 1 \ matrix = \log(G/f(j)), \text{ where } f(j) \text{ is applied to the } j\text{th column.}
\]

Define \(G^n\) to be the matrix obtained from \(G\) by multiplying itself \(n\) times.

\[
PAM \ n \ matrix \ is \ \log(G^n/f(j)), \text{ where } f(j) \text{ is applied to } j\text{th column.}
\]

But what does \(G^n\) mean and what does PAM \(n\) mean?
Chapter 6. Dynamic Programming

<table>
<thead>
<tr>
<th>$G$</th>
<th>$X$</th>
<th>$Y$</th>
<th>$Z$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X$</td>
<td>$a$</td>
<td>$b$</td>
<td>$c$</td>
</tr>
<tr>
<td>$Y$</td>
<td>$d$</td>
<td>$e$</td>
<td>$f$</td>
</tr>
<tr>
<td>$Z$</td>
<td>$g$</td>
<td>$h$</td>
<td>$i$</td>
</tr>
</tbody>
</table>

Assume $G$

Consider $G$

$g(X, Y) = b$
Assume $G$:

<table>
<thead>
<tr>
<th>$G$</th>
<th>$X$</th>
<th>$Y$</th>
<th>$Z$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X$</td>
<td>$a$</td>
<td>$b$</td>
<td>$c$</td>
</tr>
<tr>
<td>$Y$</td>
<td>$d$</td>
<td>$e$</td>
<td>$f$</td>
</tr>
<tr>
<td>$Z$</td>
<td>$g$</td>
<td>$h$</td>
<td>$i$</td>
</tr>
</tbody>
</table>

$g(X, Y) = b$

Consider $G^2$:

<table>
<thead>
<tr>
<th>$G^2$</th>
<th>$X$</th>
<th>$Y$</th>
<th>$Z$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X$</td>
<td>...</td>
<td>$(ab + be + ch)$</td>
<td>...</td>
</tr>
<tr>
<td>$Y$</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$Z$</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Chapter 6. Dynamic Programming

Assume $G$

\[
\begin{array}{c|ccc}
G & X & Y & Z \\
\hline
X & a & b & c \\
Y & d & e & f \\
Z & g & h & i \\
\end{array}
\]

$g(X, Y) = b$

Consider $G^2$

\[
\begin{array}{c|ccc}
G^2 & X & Y & Z \\
\hline
X & \ldots & (ab + be + ch) & \ldots \\
Y & \ldots & \ldots & \ldots \\
Z & \ldots & \ldots & \ldots \\
\end{array}
\]

where

\[
\begin{align*}
ab &= g(X, X)g(X, Y) \\
br &= g(X, Y)g(Y, Y) \\
ch &= g(X, Z)g(Z, Y)
\end{align*}
\]
Chapter 6. Dynamic Programming

Assume $G$

<table>
<thead>
<tr>
<th></th>
<th>$X$</th>
<th>$Y$</th>
<th>$Z$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X$</td>
<td>$a$</td>
<td>$b$</td>
<td>$c$</td>
</tr>
<tr>
<td>$Y$</td>
<td>$d$</td>
<td>$e$</td>
<td>$f$</td>
</tr>
<tr>
<td>$Z$</td>
<td>$g$</td>
<td>$h$</td>
<td>$i$</td>
</tr>
</tbody>
</table>

$g(X, Y) = b$

Consider $G^2$

<table>
<thead>
<tr>
<th></th>
<th>$X$</th>
<th>$Y$</th>
<th>$Z$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X$</td>
<td>$\ldots$</td>
<td>$(ab + be + ch)$</td>
<td>$\ldots$</td>
</tr>
<tr>
<td>$Y$</td>
<td>$\ldots$</td>
<td>$\ldots$</td>
<td>$\ldots$</td>
</tr>
<tr>
<td>$Z$</td>
<td>$\ldots$</td>
<td>$\ldots$</td>
<td>$\ldots$</td>
</tr>
</tbody>
</table>

where

\[
ab = g(X, X) g(X, Y) \\
be = g(X, Y) g(Y, Y) \\
ch = g(X, Z) g(Z, Y)
\]

These are probabilities of two step mutations from $X$ to $Y$. So $n$ is a multiple of the time unit.
Chapter 6. Dynamic Programming

Assume $G$

<table>
<thead>
<tr>
<th></th>
<th>$X$</th>
<th>$Y$</th>
<th>$Z$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X$</td>
<td>$a$</td>
<td>$b$</td>
<td>$c$</td>
</tr>
<tr>
<td>$Y$</td>
<td>$d$</td>
<td>$e$</td>
<td>$f$</td>
</tr>
<tr>
<td>$Z$</td>
<td>$g$</td>
<td>$h$</td>
<td>$i$</td>
</tr>
</tbody>
</table>

$g(X, Y) = b$

Consider $G^2$

<table>
<thead>
<tr>
<th></th>
<th>$X$</th>
<th>$Y$</th>
<th>$Z$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X$</td>
<td>...</td>
<td>$(ab + be + ch)$</td>
<td>...</td>
</tr>
<tr>
<td>$Y$</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$Z$</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

where

\[
ab = g(X, X)g(X, Y) \\
be = g(X, Y)g(Y, Y) \\
ch = g(X, Z)g(Z, Y)
\]

These are probabilities of two step mutations from $X$ to $Y$. So $n$ is a multiple of the time unit.

http://www.bioinformatics.nl/tools/pam.html
Chapter 6. Dynamic Programming

BLOSUM (blocks of amino acid substitution) matrices
Chapter 6. Dynamic Programming

BLOSUM (blocks of amino acid substitution) matrices

Scores within a BLOSUM are log-odds scores that measure, in an alignment, the logarithm for the ratio of the likelihood of two amino acids appearing with a biological sense and the likelihood of the same amino acids appearing by chance.
Chapter 6. Dynamic Programming

BLOSUM (blocks of amino acid substitution) matrices

Scores within a BLOSUM are log-odds scores that measure, in an alignment, the logarithm for the ratio of the likelihood of two amino acids appearing with a biological sense and the likelihood of the same amino acids appearing by chance.

The matrices are based on the minimum percentage identity of the aligned protein sequence used in calculating them.
BLOSUM (blocks of amino acid substitution) matrices

Scores within a BLOSUM are log-odds scores that measure, in an alignment, the logarithm for the ratio of the likelihood of two amino acids appearing with a biological sense and the likelihood of the same amino acids appearing by chance.

The matrices are based on the minimum percentage identity of the aligned protein sequence used in calculating them.

Every possible identity or substitution is assigned a score based on its observed frequencies in the alignment of related proteins.
Chapter 6. Dynamic Programming

```plaintext
# Matrix made by matblas from blosum62.ijj
# * column uses minimum score
# BLOSUM Clustered Scoring Matrix in 1/2 Bit Units
# Blocks Database = /data/blocks_5.0/blocks.dat
# Cluster Percentage: >= 62
# Entropy =  0.6979, Expected = -0.5209

|   | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | *
| A | 4 | -1 | -2 | -2 | 0 | -1 | -1 | 0 | -2 | -1 | -1 | -1 | -1 | -1 | -2 | -1 | 1 | 0 | -3 | -2 | 0 | -2 | -1 | 0 | -1 | 4 |
| R | -1 | 5 | 0 | -2 | -3 | 1 | 0 | -2 | 0 | -3 | -2 | 2 | -1 | -3 | -2 | -1 | -1 | -3 | -2 | -3 | -1 | 0 | -1 | -4 | 4 |
| N | -2 | 0 | 6 | 1 | -3 | 0 | 0 | 0 | 1 | -3 | -3 | 0 | -2 | -3 | -2 | 1 | 0 | -4 | -2 | -3 | 3 | 0 | -1 | -4 | 4 |
| D | -2 | -2 | 1 | 6 | -3 | 0 | 2 | -1 | -1 | -3 | -4 | -1 | -3 | -3 | -1 | 0 | -1 | -4 | -3 | -3 | 4 | 1 | -1 | -4 | 4 |
| C | 0 | -3 | -3 | -1 | 9 | -3 | -4 | -3 | -3 | -1 | -1 | -3 | -1 | -2 | -3 | -1 | -1 | -2 | -2 | -1 | -3 | -3 | -2 | -4 | 4 |
| Q | -1 | 1 | 0 | 0 | -3 | 5 | 2 | -2 | 0 | -3 | -2 | 1 | 0 | -3 | -1 | 0 | -1 | -2 | -1 | -2 | 0 | 3 | -1 | -4 | 4 |
| E | -1 | 0 | 0 | 0 | 2 | -4 | 2 | 5 | -2 | 0 | -3 | -1 | 1 | -2 | -3 | -1 | 0 | -1 | -3 | -2 | 1 | 4 | -1 | -4 | 4 |
| G | 0 | -2 | 0 | -1 | -3 | -2 | -2 | 6 | -2 | -4 | -4 | -2 | -3 | -3 | -2 | 0 | -2 | -2 | -3 | -3 | -1 | -2 | -1 | -4 | 4 |
| H | -2 | 0 | 1 | -1 | -3 | 0 | 0 | -2 | 0 | -3 | -2 | -1 | -2 | -1 | -2 | -1 | -2 | -2 | -2 | -3 | 0 | 0 | -1 | -4 | 4 |
| I | -1 | -1 | -3 | -3 | -3 | -1 | -3 | -4 | -3 | 4 | 2 | -3 | 1 | 0 | -3 | -2 | -1 | -3 | -1 | -3 | -3 | -3 | -1 | -4 | 4 |
| L | -1 | -2 | -3 | -4 | -1 | -2 | -3 | -4 | -3 | 2 | -2 | -3 | 2 | 0 | -3 | -2 | -1 | -2 | -1 | -2 | 1 | 1 | -4 | -3 | -1 | 4 |
| K | -1 | 2 | 0 | -1 | -3 | 1 | 1 | 2 | -1 | -3 | -2 | 5 | -1 | -3 | 1 | 0 | -1 | -3 | -2 | -2 | 0 | 1 | -1 | -4 | 4 |
| M | -1 | -1 | -2 | -3 | -1 | 0 | -2 | -3 | -2 | 1 | 2 | -1 | 5 | 0 | -2 | -1 | -1 | -1 | -1 | 1 | 3 | -1 | -4 | 4 |
| F | -2 | -3 | -3 | -3 | -2 | -3 | -3 | -1 | 1 | 0 | -3 | 0 | 0 | -6 | -4 | -2 | -2 | 1 | 3 | -1 | -3 | -3 | -1 | 4 |
| P | -1 | -2 | -2 | -1 | -3 | -1 | -1 | -2 | -2 | -3 | -3 | -1 | -2 | -4 | 7 | -1 | -1 | -4 | -3 | -2 | -2 | -1 | -2 | -4 | 4 |
| S | 1 | -1 | 1 | 0 | -1 | 0 | 0 | 0 | 1 | -2 | -2 | 0 | -1 | -2 | -1 | 4 | 1 | -3 | -2 | -2 | 0 | 0 | 0 | -4 | 4 |
| T | 0 | -1 | 0 | -1 | -1 | -1 | -2 | -2 | -1 | -1 | -1 | -2 | -1 | 1 | 5 | -2 | -2 | 0 | -1 | -1 | 0 | 0 | -4 | 4 |
| W | -3 | -3 | -4 | -4 | -2 | -2 | -3 | -2 | -3 | -2 | -3 | -1 | 1 | -4 | -3 | -2 | 1 | 1 | 2 | -3 | -4 | -3 | -2 | -4 | 4 |
| Y | -2 | -2 | -2 | -2 | -3 | -2 | -2 | -3 | 2 | -1 | -1 | -2 | -1 | 3 | -3 | -2 | -2 | 2 | 7 | -1 | -3 | -2 | -4 | 4 |
| V | 0 | -3 | -3 | -3 | -1 | -2 | -2 | -3 | -3 | 3 | 1 | 2 | 1 | -1 | -2 | -2 | 0 | -3 | -1 | 4 | -3 | -2 | -1 | -4 | 4 |
| B | -2 | -1 | 3 | 4 | -3 | 0 | 1 | -1 | 0 | -3 | -4 | 0 | -3 | -3 | -2 | 0 | -1 | -4 | -3 | -3 | 4 | 1 | -1 | -4 | 4 |
| Z | -1 | 0 | 0 | 1 | -3 | 3 | 4 | -2 | 0 | -3 | -3 | 1 | -1 | -3 | -1 | 0 | -1 | -3 | -2 | -2 | 1 | 4 | -1 | -4 | 4 |
| X | 0 | -1 | -1 | -2 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | 0 | -2 | -1 | -1 | -1 | -1 | -1 | -4 | 4 |
| * | -4 | -4 | -4 | -4 | -4 | -4 | -4 | -4 | -4 | -4 | -4 | -4 | -4 | -4 | -4 | -4 | -4 | -4 | -4 | -4 | -4 | -4 | -4 | -4 | 4 |
```
6.8 Local sequence alignment

To find conserved regions between two sequences that are not necessarily similar overall.
6.8 Local sequence alignment

To find conserved regions between two sequences that are not necessarily similar overall.

A situation that global alignment is not appropriate.

..........xxxxxxxxxx
xxxxxxxxxxxxx...........

where xxxxxxxxxx is a conserved motif.

Also see Figure 6.16
Chapter 6. Dynamic Programming

A *local alignment* between two sequences

\[ v = v_1 \ldots v_n \text{ and } w = w_1 \ldots w_m \]
Chapter 6. Dynamic Programming

A *local alignment* between two sequences

\[ v = v_1 \ldots v_n \text{ and } w = w_1 \ldots w_m \]

is a global alignment between two substrings

\[ v_{a} \ldots v_{b} \text{ and } w_{c} \ldots w_{d}, \text{ of } v \text{ and } w \text{ respectively,} \]

that achieves the best alignment score among all such indexes \( a, b, c, d, \)

\[ 1 \leq a \leq b \leq n \text{ and } 1 \leq c \leq d \leq m. \]
Chapter 6. Dynamic Programming

LOCAL SEQUENCE ALIGNMENT

Find the best local alignment between two strings.
Chapter 6. Dynamic Programming

**Local sequence alignment**

*Find the best local alignment between two strings.*

**Input:** Strings $v$ and $w$ and a scoring matrix $\delta$,

**Output:** Substrings of $v$ and $w$ whose global alignment, as defined by $\delta$, is the maximum among all global alignments of all substrings of $v$ and $w$. 
Chapter 6. Dynamic Programming

How to solve this ‘seemingly the same problem’ as global alignment?
Chapter 6. Dynamic Programming

How to solve this 'seemingly the same problem' as global alignment?

(1) The global alignment algorithm actually computes all "semi-global alignments".
How to solve this 'seemingly the same problem' as global alignment?

(1) The global alignment algorithm actually computes all “semi-global alignments”.

That is, it computes all best alignment scores for prefix substrings $v_1 \ldots v_i$ and $w_1 \ldots w_j$ for all indexes $i, j$. 
(2) For local alignment, we would also like to drop any alignment 'head'
that has been penalized.

\[
\begin{align*}
  v_1 \ldots v_h \ldots \ldots \ldots v_k \ldots v_i \\
  w_1 \ldots w_p \ldots \ldots \ldots w_q \ldots w_j \\
\end{align*}
\]

\[\Rightarrow | \text{neg} |<--
\]

score
(2) For local alignment, we would also like to drop any alignment 'head'
that has been penalized.

\[ v_{1} \ldots v_{h} \ldots v_{k} \ldots v_{i} \]
\[ w_{1} \ldots w_{p} \ldots w_{q} \ldots w_{j} \]

\[-\rightarrow|\text{neg}|\leftarrow\]
\[\text{score}\]

which can be achieved by following recurrence:

\[
s_{i,j} = \max \begin{cases} 
  s_{i-1,j-1} + \delta(v_{i},w_{j}) \\
  s_{i-1,j} + \delta(v_{i},-) \\
  s_{i,j-1} + \delta(-,w_{j}) \\
  0 
\end{cases}
\]
(3) For local alignment, we would like to drop any alignment 'tail' that has a negative score.

\[
v_1 \ldots v_k \ldots v_i \\
w_1 \ldots w_q \ldots w_j
\]

\[\rightarrow | \text{neg} | \leftarrow \]

score
(3) For local alignment, we would like to drop any alignment 'tail' that has a negative score.

\[ \begin{align*}
  v_1 & \ldots \ldots \ldots \ldots v_k \ldots \ldots \ldots \ldots \ldots v_i \\
  w_1 & \ldots \ldots \ldots \ldots w_q \ldots \ldots \ldots \ldots \ldots w_j \\
  \rightarrow & \neg \leftarrow
\end{align*} \]

This can be accomplished by tracing the DP table from the highest value cell.
6.10 Multiple alignment

Option-1: assume a $k$-dimensional scoring matrix. (unrealistic)

Option-2: Sum-of-Pair (SP) scoring scheme: the score of a multiple alignment is computed as the sum of the scores between every pair of the aligned sequences.

Option-3: entropy approach: the score of a multiple alignment is computed as the sum of entropies of all aligned columns. The entropy for a column $i$ is computed as
\[ \sum_{x \in \{A, C, G, T\}} f_i x \log f_i x \] where $f_i x$ is the frequency of residue $x$ in column $i$. 
Chapter 6. Dynamic Programming

6.10 Multiple alignment

To compare more than two sequences.

How to score an alignment (a column) involving $k$ sequences?
Chapter 6. Dynamic Programming

6.10 Multiple alignment

To compare more than two sequences.

How to score an alignment (a column) involving \( k \) sequences?

Option-1: assume a \( k \)-dimensional scoring matrix. (unrealistic)
Chapter 6. Dynamic Programming

6.10 Multiple alignment

To compare more than two sequences.

How to score an alignment (a column) involving \( k \) sequences?

Option-1: assume a \( k \)-dimensional scoring matrix. (unrealistic)

Option-2: Sum-of-Pair (SP) scoring scheme:
- the score of a multiple alignment is computed as the sum of the scores between every pair of the aligned sequences.

Option-3: entropy approach:
- the score of a multiple alignment is computed as the sum of entropies of all aligned columns.

The entropy for a column \( i \) is computed as

\[
\sum_{x \in \{A, C, G, T\}} f_i x \log f_i x
\]

where \( f_i x \) is the frequency of residue \( x \) in column \( i \).
6.10 Multiple alignment

To compare more than two sequences.

How to score an alignment (a column) involving \( k \) sequences?

**Option-1:** assume a \( k \)-dimensional scoring matrix. (unrealistic)

**Option-2:** *Sum-of-Pair* (SP) scoring scheme:
the score of a multiple alignment is computed as the sum of the scores between every pair of the aligned sequences.

**Option-3:** entropy approach:
the score of a multiple alignment is computed as the sum of entropies of all aligned columns.

The entropy for a column \( i \) is computed as

\[
\sum_{x \in \{A,C,G,T\}} f^i_x \log f^i_x
\]

where \( f^i_x \) is the frequency of residue \( x \) in column \( i \).
A dynamic programming solution for multiple alignment.
Chapter 6. Dynamic Programming

A dynamic programming solution for multiple alignment.

Assume we have 3 sequences \( u, v, w \) to align. Consider aligning prefixes:

\[
\begin{align*}
u_1 \ldots u_i \\
v_1 \ldots v_j \\
w_1 \ldots w_k
\end{align*}
\]
A dynamic programming solution for multiple alignment.

Assume we have 3 sequences $u, v, w$ to align. Consider aligning prefixes:

$u_1 \ldots u_i$
$v_1 \ldots v_j$
$w_1 \ldots w_k$

$$s_{i,j,k} = \max \begin{cases} 
    s_{i-1,j-1,k-1} + \delta(u_i, v_j, w_k) \\
    s_{i,j-1,k-1} + \delta(-, v_j, w_k) \\
    s_{i-1,j,k-1} + \delta(u_i, -, w_k) \\
    s_{i-1,j-1,k} + \delta(u_i, v_j, -) \\
    s_{i,j,k-1} + \delta(-, -, w_k) \\
    s_{i,j-1,k} + \delta(-, v_j, -) \\
    s_{i-1,j,k} + \delta(u_i, -, -) 
\end{cases}$$
Chapter 6. Dynamic Programming

A dynamic programming solution for multiple alignment.

Assume we have 3 sequences $u, v, w$ to align. Consider aligning prefixes:

\[ u_1 \ldots u_i \]
\[ v_1 \ldots v_j \]
\[ w_1 \ldots w_k \]

\[
s_{i,j,k} = \max \begin{cases} 
    s_{i-1,j-1,k-1} + \delta(u_i, v_j, w_k) \\
    s_{i,j-1,k-1} + \delta(\_, v_j, w_k) \\
    s_{i-1,j,k-1} + \delta(u_i, \_, w_k) \\
    s_{i-1,j-1,k} + \delta(u_i, v_j, \_) \\
    s_{i,j,k-1} + \delta(\_, \_, w_k) \\
    s_{i,j-1,k} + \delta(\_, v_j, \_) \\
    s_{i-1,j,k} + \delta(u_i, \_, \_) 
\end{cases}
\]

Inefficient: $7 \times n^3$ time. In general,
Chapter 6. Dynamic Programming

A dynamic programming solution for multiple alignment.

Assume we have 3 sequences $u, v, w$ to align. Consider aligning prefixes:

$u_1 \ldots u_i$
$v_1 \ldots v_j$
$w_1 \ldots w_k$

$s_{i,j,k} = \max \left\{ \begin{array}{l}
  s_{i-1,j-1,k-1} + \delta(u_i, v_j, w_k) \\
  s_{i,j-1,k-1} + \delta(\underline{--}, v_j, w_k) \\
  s_{i-1,j,k-1} + \delta(u_i, \underline{--}, w_k) \\
  s_{i-1,j-1,k} + \delta(u_i, v_j, \underline{--}) \\
  s_{i,j,k-1} + \delta(\underline{--}, \underline{--}, w_k) \\
  s_{i,j-1,k} + \delta(\underline{--}, v_j, \underline{--}) \\
  s_{i-1,j,k} + \delta(u_i, \underline{--}, \underline{--})
\end{array} \right\}$

Inefficient: $7 \times n^3$ time. In general,

$O(n^m 2^{m-1})$ time and $O(n^m)$ size table for $m$ sequences of length $n$. 
Chapter 6. Dynamic Programming

Heuristic algorithms for multiple alignment

- Note that use a collection of pairwise alignments may not work
- Find a small set of 'core' sequences and align other to them
  - E.g., center-star algorithm
  - E.g., CLUSTAL
- the "once gap, forever gap" strategy
Chapter 6. Dynamic Programming

Heuristic algorithms for multiple alignment

Typically **progressive** approaches:
Chapter 6. Dynamic Programming

Heuristic algorithms for multiple alignment

Typically **progressive** approaches:

- Note that use a collection of pairwise alignments may not work
Heuristic algorithms for multiple alignment

Typically **progressive** approaches:

- Note that use a collection of pairwise alignments may not work
- Find a small set of ‘core’ sequences and align other to them
Chapter 6. Dynamic Programming

Heuristic algorithms for multiple alignment

Typically progressive approaches:

- Note that use a collection of pairwise alignments may not work

- Find a small set of ‘core’ sequences and align other to them

  E.g., center-star algorithm
  E.g., CLUSTAL

the “once gap, forever gap” strategy
6.10.1/2 HMM and Dynamic Programming Solutions

A Markov Model characterizes stochastic processes that assume following Markov property.

The “oblivious” property, i.e, the conditional probability distribution of future states of a stochastic process depends only upon the present state, not on the sequence of events that preceded it.
A Markov Model over alphabet $\Sigma$ consists of a set $S$ of states and transitions $T$ between states, such that
A Markov Model over alphabet \( \Sigma \) consists of a set \( S \) of states and transitions \( T \) between states, such that

(1) With a probability distribution, each state can emit symbols in \( \Sigma \);

\[
\text{for every } s \in S, \sum_{x \in \Sigma} p(s, x) = 1
\]
A Markov Model over alphabet $\Sigma$ consists of a set $S$ of states and transitions $T$ between states, such that

(1) With a probability distribution, each state can emit symbols in $\Sigma$;

\[
\text{for every } s \in S, \sum_{x \in \Sigma} p(s, x) = 1
\]

(2) With a probability distribution, there are transitions from each state to all other states in the model;

\[
\text{for every } s \in S, \sum_{t \in S} q(s, t) = 1
\]
Chapter 6. Dynamic Programming

Definitions:

Let $M$ be a Markov model over alphabet $\Sigma$. Let $X = x_1 \ldots x_n$ be a sequence over $\Sigma$, i.e., $x_i \in \Sigma$. Let $\pi = s_1 \ldots s_n$ be a sequence of states taken from $S$ (i.e., $\pi$ is called a path). Then the probability for $M$ to generate symbol sequence $X$ with the path $\pi$ is

$$p(X, \pi | M) = p(s_1, x_1) \cdot q(s_1, s_2) \cdot p(s_2, x_2) \cdots p(s_{n-1}, x_{n-1}) \cdot q(s_{n-1}, s_n) \cdot p(s_n, x_n)$$
Chapter 6. Dynamic Programming

Definitions:

Let $M$ be a Markov model over alphabet $\Sigma$.
Let $X = x_1 \ldots x_n$ be a sequence over $\Sigma$, i.e., $x_i \in \Sigma$. 
Chapter 6. Dynamic Programming

Definitions:

Let $M$ be a Markov model over alphabet $\Sigma$.
Let $X = x_1 \ldots x_n$ be a sequence over $\Sigma$, i.e., $x_i \in \Sigma$.

Let $\pi = s_1 \ldots s_n$ be a sequence of states taken from $S$, i.e., $s_i \in S$ ($\pi$ is called a path).
Chapter 6. Dynamic Programming

Definitions:

Let $M$ be a Markov model over alphabet $\Sigma$.
Let $X = x_1 \ldots x_n$ be a sequence over $\Sigma$, i.e., $x_i \in \Sigma$.

Let $\pi = s_1 \ldots s_n$ be a sequence of states taken from $S$, i.e., $s_i \in S$ ($\pi$ is called a path).

Then the probability for $M$ to generate symbol sequence $X$ with the path $\pi$ is

\[
p(X, \pi | M) = p(s_1, x_1)q(s_1, s_2)p(s_2, x_2) \ldots p(s_{n-1}, x_{n-1})q(s_{n-1}, s_n)p(s_n, x_n)
= \prod_{k=1}^{n-1} p(s_k, x_k)q(s_k, s_{k+1}) \times p(s_n, x_n)
\]
Chapter 6. Dynamic Programming

Uses of HMMs

1. Modeling specific classes of sequences
   e.g., profile HMM for motifs
Chapter 6. Dynamic Programming

Uses of HMMs

1. Modeling specific classes of sequences
   e.g., profile HMM for motifs
2. Modeling general classes of sequences
   typically for prediction
Chapter 6. Dynamic Programming

Fundamental algorithms with HMMs

1. decoding (for prediction, discrimination)
Chapter 6. Dynamic Programming

Fundamental algorithms with HMMs

1. decoding (for prediction, discrimination)
2. computing likelihood (for model fitness)
Chapter 6. Dynamic Programming

Fundamental algorithms with HMMs

1. decoding (for prediction, discrimination)
2. computing likelihood (for model fitness)
3. learning (for building models)
HMM Decoding Problem:
Chapter 6. Dynamic Programming

HMM Decoding Problem:

Given a Markov model $M$ and a sequence $X$ over the alphabet $\Sigma$, find the path $\pi^*$ such that

$$p(X, \pi^* | M) \text{ achieves the maximum.}$$
Chapter 6. Dynamic Programming

HMM Decoding Problem:

Given a Markov model $M$ and a sequence $X$ over the alphabet $\Sigma$, find the path $\pi^*$ such that

$$p(X, \pi^* | M) \text{ achieves the maximum.}$$

That is to find an optimal path $\pi^*$ such that

$$\pi^* = \arg \max_{\pi} \{ p(X, \pi | M) \}$$
Chapter 6. Dynamic Programming

Dynamic programming to compute $\pi^*$ such that

$$p(X, \pi^* | M)$$

achieves the maximum.
Chapter 6. Dynamic Programming

Dynamic programming to compute $\pi^*$ such that

$$p(X, \pi^*|M)$$ achieves the maximum.

If we consider the model $M$ to be a “generic” sequence
then the task is to find the “best alignment” between $X$ and $M$
Chapter 6. Dynamic Programming

Dynamic programming to compute $\pi^*$ such that

\[ p(X, \pi^* | M) \] achieves the maximum.

If we consider the model $M$ to be a “generic” sequence
then the task is to find the “best alignment” between $X$ and $M$

That is: to “align” $x_i$ on $X$ with state $s_j$
Dynamic programming to compute $\pi^*$ such that

$$p(X, \pi^*|M)$$ achieves the maximum.

If we consider the model $M$ to be a “generic” sequence then the task is to find the “best alignment” between $X$ and $M$.

That is: to “align” $x_i$ on $X$ with state $s_j$.

The “score” is $p(s_j, x_i)$ for this “column”.
Chapter 6. Dynamic Programming

Consider a more general algorithm Viterbi Algorithm:

Computing the maximum probability for $M$ to generate prefix $x_1x_2 \ldots x_i$

such that symbol $x_i$ is emitted by state $s_j$. 
Chapter 6. Dynamic Programming

Consider a more general algorithm Viterbi Algorithm:

Computing the maximum probability for $M$ to generate prefix $x_1 x_2 \ldots x_i$

such that symbol $x_i$ is emitted by state $s_j$.

We define such probability to be $V_{i,j}$
Chapter 6. Dynamic Programming

Consider a more general algorithm Viterbi Algorithm:

Computing the maximum probability for \( M \) to generate prefix \( x_1x_2 \ldots x_i \)

such that symbol \( x_i \) is emitted by state \( s_j \).

We define such probability to be \( V_{i,j} \)

Then we have the following recurrence:

\[
V_{i,j} = \max_{s_k \in S} \{ V_{i-1,k} \times q(s_k, s_j) \times p(s_j, x_i) \}
\]
Consider a more general algorithm \textbf{Viterbi Algorithm}:

Computing the maximum probability for $M$ to generate prefix $x_1 x_2 \ldots x_i$

such that symbol $x_i$ is emitted by state $s_j$.

We define such probability to be $V_{i,j}$

Then we have the following recurrence:

$$V_{i,j} = \max_{s_k \in S} \{ V_{i-1,k} \times q(s_k, s_j) \times p(s_j, x_i) \}$$

If stochastic processes always begin from state $s_b$, then:

$$V_{1,b} = 1 \times p(s_b, x_1)$$

$$V_{1,a} = 0 \quad \text{for every } a \neq b$$
Chapter 6. Dynamic Programming

dynamic programming to compute $G_{i,j}$ consisting 4 steps:
dynamic programming to compute $G_{i,j}$
consisting 4 steps:

1. problem analysis
2. objective function and recurrence formulation
3. iterative algorithm implementation
4. tracing back the solution (path)
Use an HMM to construct a sequence profile
called a *profile-HMM*
Chapter 6. Dynamic Programming

Use an HMM to construct a sequence profile called a *profile-HMM*

element:

```
CGGGGCTatccagctGGGTCGTCACATTCCCCTTTTCGATA
TTTGAGGGTGCCCAATAAgggcaactCCAAAGCGGACAAA
GGatggatctGATGCCGTTTGAGCGACCTAAATCAACGGGCC
AAGGaagcaaccCCAGGAGCGCCTTTGCTGGTTCTACCTG
CTAAAAGATTATAATGTCGCTCCttggaactTC
CTGTACATCATGCTGCatgccattTTCAAC
TACATGATCTTTTGatggcactTGGATGATGAGGGAATGC
```

___________
Chapter 6. Dynamic Programming

8 columns for match
Chapter 6. Dynamic Programming

8 columns for match

What about insertions and deletions?
8 columns for match

What about insertions and deletions?

Profile-HMM definition:

A profile-HMM consists of the following states:
- begin and end states
- match states $M_j, j = 1, 2, \ldots, m$
- insert states $I_j, j = 1, 2, \ldots, m$
- delete states $D_j, j = 1, 2, \ldots, m$
Chapter 6. Dynamic Programming
Chapter 6. Dynamic Programming

Viterbi algorithm is still usable for computing $\pi^*$ but needs to be revised a little.
Chapter 6. Dynamic Programming

Viterbi algorithm is still usable for computing $\pi^*$ but needs to be revised a little.

Define:

$V_{i,j}^M$ is the optimal probability the HMM produces prefix $x_1 \ldots x_i$ ending at state $M_j$. 
Viterbi algorithm is still usable for computing $\pi^*$ but needs to be revised a little.

Define:

$V_{i,j}^M$ is the optimal probability the HMM produces prefix $x_1 \ldots x_i$ ending at state $M_j$.

$V_{i,j}^I$ is the optimal probability the HMM produces prefix $x_1 \ldots x_i$ ending at state $I_j$. 
Viterbi algorithm is still usable for computing $\pi^*$ but needs to be revised a little.

Define:

$V_{i,j}^M$ is the optimal probability the HMM produces prefix $x_1 \ldots x_i$ ending at state $M_j$.

$V_{i,j}^I$ is the optimal probability the HMM produces prefix $x_1 \ldots x_i$ ending at state $I_j$.

$D_{i,j}^I$ is the optimal probability the HMM produces prefix $x_1 \ldots x_i$ ending at state $D_j$. 
Chapter 6. Dynamic Programming

Recurrences for $V_{i,j}^M$:

Base cases?

And recurrences for $V_{i,j}^I$, $V_{i,j}^D$?
Recurrences for $V_{i,j}^M$:

{$V_{i,j}^M = \max\{ V_{i-1,j-1}^M q(M_{j-1}, M_j) p(M_j, x_i), \quad V_{i-1,j-1}^I q(I_{j-1}, M_j) p(M_j, x_i), \quad V_{i-1,j-1}^D q(D_{j-1}, M_j) p(M_j, x_i) \}$
Chapter 6. Dynamic Programming

Recurrences for $V_{i,j}^M$:

$$V_{i,j}^M = \max \{$$

$$V_{i-1,j-1}^M q(M_{j-1}, M_j) p(M_j, x_i),$$

$$V_{i-1,j-1}^I q(I_{j-1}, M_j) p(M_j, x_i),$$

$$V_{i-1,j-1}^D q(D_{j-1}, M_j) p(M_j, x_i)$$

$$\}$$

Base cases?
Recurrences for $V_{i,j}^M$:

$$V_{i,j}^M = \max\{$$

$$V_{i-1,j-1}^M q(M_{j-1}, M_j) p(M_j, x_i),$$

$$V_{i-1,j-1}^I q(I_{j-1}, M_j) p(M_j, x_i),$$

$$V_{i-1,j-1}^D q(D_{j-1}, M_j) p(M_j, x_i)$$

$$\}$$

Base cases?

And recurrences for $V^I, V^D$?
To set up a profile-HMM:

- obtain a multiple alignment of training data;
- determine the number of match states;
- compute emission prob distribution for every match state;
- determine the number insert states;
- compute emission prob distribution for every insert state;
- determine the number of delete states;
- determine the transition probability distributions;
Chapter 6. Dynamic Programming

To set up a profile-HMM:

- obtain a multiple alignment of training data;
- determine the number of match states;
- compute emission prob distribution for every match state;
- determine the number insert states;
- compute emission prob distribution for every insert state;
- determine the number of delete states;
- determine the transition probability distributions;

Resolve the over fitting issue with pseudo-counts
Chapter 6. Dynamic Programming

Example of profile-HMM:

`atccag-ct`
`gggcaa-ct`
`atggat-ct`
`a-gcaatcc`
`ttggaa-ct`
`atgccat-ct`
`atggca-ct`
Example of profile-HMM:

atccag-ct
gggcaaa-ct
atggat-ct
a-gcaatcc
ttggaa-ct
atgcca-tt
atggca-ct

1. how to determine match columns and insert columns (for consensus)
2. deletion is w.r.t. match
3. when there is no pseudo counts
4. to avoid overfitting
Using a profile HMM (of a family of sequences) to search for new members on genomes/databases:
Using a profile HMM (of a family of sequences) to search for new members on genomes/databases:

- construct a profile-HMM
- develop Viterbi algorithm
- choose a scanning window size
- post-process results
6.11 Gene Prediction

Identification of protein coding genes in genome sequences.

Statistical approaches

based on statistical features surrounding genes
6.11 Gene Prediction

Identification of protein coding genes in genome sequences.

Statistical approaches

based on statistical features surrounding genes

Similarity-based approaches

based on similarities of genes across different species