CS481: Bioinformatics Algorithms

Can Alkan
EA509
calkan@cs.bilkent.edu.tr

http://www.cs.bilkent.edu.tr/~calkan/teaching/cs481/
The Change Problem

**Goal:** Convert some amount of money $M$ into given denominations, using the fewest possible number of coins

**Input:** An amount of money $M$, and an array of $d$ denominations $c = (c_1, c_2, \ldots, c_d)$, in a decreasing order of value ($c_1 > c_2 > \ldots > c_d$)

**Output:** A list of $d$ integers $i_1, i_2, \ldots, i_d$ such that

$$c_1i_1 + c_2i_2 + \ldots + c_di_d = M$$

and $i_1 + i_2 + \ldots + i_d$ is minimal
Change Problem: Example

Given the denominations 1, 3, and 5, what is the minimum number of coins needed to make change for a given value?

<table>
<thead>
<tr>
<th>Value</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min # of coins</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Only one coin is needed to make change for the values 1, 3, and 5
Given the denominations 1, 3, and 5, what is the minimum number of coins needed to make change for a given value?

<table>
<thead>
<tr>
<th>Value</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min # of coins</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

However, two coins are needed to make change for the values 2, 4, 6, 8, and 10.
Given the denominations 1, 3, and 5, what is the minimum number of coins needed to make change for a given value?

<table>
<thead>
<tr>
<th>Value</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min # of coins</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

Lastly, three coins are needed to make change for the values 7 and 9.
Change Problem: Recurrence

This example is expressed by the following recurrence relation:

$$\text{minNumCoins}(M) = \min \left\{ \begin{array}{c} \text{minNumCoins}(M-1) + 1 \\ \text{minNumCoins}(M-3) + 1 \\ \text{minNumCoins}(M-5) + 1 \end{array} \right\}$$
Given the denominations \( c: c_1, c_2, \ldots, c_d \), the recurrence relation is:

\[
\text{minNumCoins}(M) = \min \left\{ \begin{array}{l}
\text{minNumCoins}(M-c_1) + 1 \\
\text{minNumCoins}(M-c_2) + 1 \\
\vdots \\
\text{minNumCoins}(M-c_d) + 1
\end{array} \right\}
\]
Change Problem: A Recursive Algorithm

1. **RecursiveChange**($M, c, d$)
2.     if $M = 0$
3.         return 0
4.     bestNumCoins $\leftarrow$ infinity
5.     for $i \leftarrow 1$ to $d$
6.         if $M \geq c_i$
7.             numCoins $\leftarrow$ RecursiveChange($M - c_i, c, d$)
8.             if numCoins + 1 < bestNumCoins
9.                 bestNumCoins $\leftarrow$ numCoins + 1
10.     return bestNumCoins
Recursive Change Is Not Efficient

- It recalculates the optimal coin combination for a given amount of money repeatedly.

  - i.e., $M = 77$, $c = (1,3,7)$:
    - Optimal coin combo for 70 cents is computed 9 times!
The Recursive Change Tree
We Can Do Better

- We’re re-computing values in our algorithm more than once

- Save results of each computation for 0 to $M$

- This way, we can do a reference call to find an already computed value, instead of re-computing each time

- Running time $M^*d$, where $M$ is the value of money and $d$ is the number of denominations
The Change Problem: Dynamic Programming

1. **DPChange(M,c,d)**
2.  `bestNumCoins0 ← 0`
3.  for `m ← 1` to `M`  
4.     `bestNumCoinsm ← ∞`
5.     for `i ← 1` to `d`  
6.         if `m ≥ ci`  
7.             if `bestNumCoinsm-ci + 1 < bestNumCoinsm`  
8.                 `bestNumCoinsm ← bestNumCoinsm-ci + 1`  
9.         return `bestNumCoinsM`
DPChange: Example

c = (1, 3, 7)

M = 9
Imagine seeking a path (from source to sink) to travel (only eastward and southward) with the most number of attractions (*) in the Manhattan grid.
Manhattan Tourist Problem (MTP)

Imagine seeking a path (from source to sink) to travel (only eastward and southward) with the most number of attractions (*) in the Manhattan grid.
Manhattan Tourist Problem: Formulation

**Goal:** Find the longest path in a weighted grid.

**Input:** A weighted grid $G$ with two distinct vertices, one labeled “source” and the other labeled “sink”

**Output:** A longest path in $G$ from “source” to “sink”
MTP: An Example

source

i coordinate

j coordinate

sink
MTP: Greedy Algorithm Is Not Optimal

promising start, but leads to bad choices!
MTP: Simple Recursive Program

MT(n,m)
    if n=0 or m=0
        return MT(n,m)
    x ← MT(n−1,m)+
        length of the edge from (n−1,m) to (n,m)
    y ← MT(n,m−1)+
        length of the edge from (n,m−1) to (n,m)
    return max{x,y}
MTP: Dynamic Programming

- Calculate optimal path score for each vertex in the graph
- Each vertex’s score is the maximum of the prior vertices score plus the weight of the respective edge in between
MTP: Dynamic Programming (cont’d)

source

$S_{1,1} = 4$

$S_{0,2} = 3$

$S_{2,0} = 8$
MTP: Dynamic Programming (cont’d)

source

\[ S_{3,0} = 8 \]

\[ S_{1,2} = 13 \]

\[ S_{2,1} = 9 \]
MTP: Dynamic Programming (cont’d)

source

greedy alg. fails!

S_{3,1} = 9
S_{2,2} = 12
S_{1,3} = 8
MTP: Dynamic Programming (cont’d)

The diagram illustrates a dynamic programming problem. The values are represented as weights on the edges. The source is labeled with 0, and the goal is to find the minimum path sum from the source to the sink. The path with the minimum sum is highlighted, and the values $S_{2,3} = 15$ and $S_{3,2} = 9$ are indicated.
MTP: Dynamic Programming (cont’d)

(source 0) \[ S_{3,3} = 16 \]

Done!

(showing all back-traces)
MTP: Recurrence

Computing the score for a point (i,j) by the recurrence relation:

\[
\begin{align*}
    s_{i,j} &= \max \left\{ s_{i-1,j} + \text{weight of the edge between (i-1, j) and (i, j)}, \\
                   s_{i,j-1} + \text{weight of the edge between (i, j-1) and (i, j)} \right\}
\end{align*}
\]

The running time is \( n \times m \) for a \( n \times m \) grid

\( n = \# \text{ of rows}, \ m = \# \text{ of columns} \)
Manhattan Is Not A Perfect Grid

What about diagonals?

- The score at point B is given by:

\[
 s_B = \max \left\{ s_{A_1} + \text{weight of the edge } (A_1, B), \quad s_{A_2} + \text{weight of the edge } (A_2, B), \quad s_{A_3} + \text{weight of the edge } (A_3, B) \right\}
\]
Computing the score for point $x$ is given by the recurrence relation:

$$s_x = \max_{y \in \text{Predecessors}(x)} (s_y + \text{weight of vertex (}y, x\text{) where})$$

- Predecessors ($x$) – set of vertices that have edges leading to $x$

- The running time for a graph $G(V, E)$ ($V$ is the set of all vertices and $E$ is the set of all edges) is $O(E)$ since each edge is evaluated once
Traveling in the Grid

- The only hitch is that one must decide on the order in which visit the vertices.
- By the time the vertex $x$ is analyzed, the values $s_y$ for all its predecessors $y$ should be computed – otherwise we are in trouble.
- We need to traverse the vertices in some order.
- Since Manhattan is not a perfect regular grid, we represent it as a DAG.
Longest Path in DAG Problem

- **Goal**: Find a longest path between two vertices in a weighted DAG
- **Input**: A weighted DAG G with source and sink vertices
- **Output**: A longest path in G from source to sink
Longest Path in DAG: Dynamic Programming

• Suppose vertex $v$ has indegree 3 and predecessors $\{u_1, u_2, u_3\}$

• Longest path to $v$ from source is:

$$s_v = \max \left\{ s_{u_1} + \text{weight of edge from } u_1 \text{ to } v, \right.$$
$$\left. s_{u_2} + \text{weight of edge from } u_2 \text{ to } v, s_{u_3} + \text{weight of edge from } u_3 \text{ to } v \right\}$$

In General:

$$s_v = \max_u (s_u + \text{weight of edge from } u \text{ to } v)$$
Traversing the Manhattan Grid

- 3 different strategies:
  - a) Column by column
  - b) Row by row
  - c) Along diagonals
Alignment: 2 row representation

Given 2 DNA sequences \( v \) and \( w \):

\[
\begin{align*}
 v & : \quad \text{ATCTGAT} \quad m = 7 \\
 w & : \quad \text{TGCATA} \quad n = 6
\end{align*}
\]

Alignment: 2 * \( k \) matrix (\( k > m, n \))

<table>
<thead>
<tr>
<th>letters of ( v )</th>
<th>A</th>
<th>T</th>
<th>--</th>
<th>G</th>
<th>T</th>
<th>T</th>
<th>A</th>
<th>T</th>
<th>--</th>
</tr>
</thead>
<tbody>
<tr>
<td>letters of ( w )</td>
<td>A</td>
<td>T</td>
<td>C</td>
<td>G</td>
<td>T</td>
<td>--</td>
<td>A</td>
<td>--</td>
<td>C</td>
</tr>
</tbody>
</table>

- 5 matches
- 2 insertions
- 2 deletions
Aligning DNA Sequences

\[ V = ATCTGATG \]
\[ W = TGCATAC \]

\( n = 8 \)
\( m = 7 \)

4 matches
1 mismatch
2 insertions
3 deletions
Longest Common Subsequence (LCS) – Alignment without Mismatches

- Given two sequences
  \[ v = v_1 v_2 \ldots v_m \text{ and } w = w_1 w_2 \ldots w_n \]
- The LCS of \( v \) and \( w \) is a sequence of positions in
  \[ v: 1 \leq i_1 < i_2 < \ldots < i_t \leq m \]
  and a sequence of positions in
  \[ w: 1 \leq j_1 < j_2 < \ldots < j_t \leq n \]
  such that \( i_t \)-th letter of \( v \) equals to \( j_t \)-letter of \( w \) and \( t \) is maximal
LCS: Example

<table>
<thead>
<tr>
<th>i coords:</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>2</th>
<th>3</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>elements of v</td>
<td>A</td>
<td>T</td>
<td>--</td>
<td>C</td>
<td>--</td>
<td>T</td>
<td>G</td>
<td>A</td>
<td>T</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>elements of w</td>
<td>--</td>
<td>T</td>
<td>G</td>
<td>C</td>
<td>A</td>
<td>T</td>
<td>--</td>
<td>A</td>
<td>--</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>j coords:</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

(0,0) \rightarrow (1,0) \rightarrow (2,1) \rightarrow (2,2) \rightarrow (3,3) \rightarrow (3,4) \rightarrow (4,5) \rightarrow (5,5) \rightarrow (6,6) \rightarrow (7,6) \rightarrow (8,7)

Matches shown in red

positions in v: 2 < 3 < 4 < 6 < 8
positions in w: 1 < 3 < 5 < 6 < 7

Every common subsequence is a path in 2-D grid
LCS Problem as Manhattan Tourist Problem
Edit Graph for LCS Problem
Every path is a common subsequence.

Every diagonal edge adds an extra element to common subsequence.

LCS Problem: Find a path with maximum number of diagonal edges.
Computing LCS

Let $v_i = \text{prefix of } v \text{ of length } i: \ v_1 \ldots v_i$

and $w_j = \text{prefix of } w \text{ of length } j: \ w_1 \ldots w_j$

The length of LCS$(v_i, w_j)$ is computed by:

$$s_{i, j} = \max \begin{cases} 
    s_{i-1, j} \\
    s_{i, j-1} \\
    s_{i-1, j-1} + 1 \text{ if } v_i = w_j
\end{cases}$$
Computing LCS (cont’d)

\[ s_{i,j} = \text{MAX} \begin{cases} 
  s_{i-1,j} + 0 \\
  s_{i,j-1} + 0 \\
  s_{i-1,j-1} + 1, \quad \text{if } v_i = w_j 
\end{cases} \]
Every Path in the Grid Corresponds to an Alignment

\[
\begin{array}{cccccc}
 & W & A & T & C & G \\
V & & 0 & 1 & 2 & 3 & 4 \\
0 & & & & & & \\
1 & & & & & & \\
2 & & & & & & \\
3 & & & & & & \\
4 & & & & & & \\
\end{array}
\]

\[
V = \text{AT} - \text{GT} \\
W = \text{ATCG} - \\
0 1 2 3 4 4
\]
DISTANCE BETWEEN STRINGS
Aligning Sequences without Insertions and Deletions: Hamming Distance

Given two DNA sequences \( v \) and \( w \):

\[
\begin{align*}
v & : \textcolor{red}{AT} \textcolor{red}{AT} \textcolor{red}{AT} \textcolor{red}{AT} \textcolor{red}{AT} \\
w & : \textcolor{brown}{TATATATAA}
\end{align*}
\]

- The Hamming distance: \( d_H(v, w) = 8 \) is large but the sequences are very similar.
Aligning Sequences with Insertions and Deletions

By shifting one sequence over one position:

\[ v : \text{ATATATATA} \quad \text{--} \quad w : \text{--TATATATA} \]

- The edit distance: \( d_H(v, w) = 2 \).
- Hamming distance neglects insertions and deletions in DNA
Edit Distance

Levenshtein (1966) introduced edit distance between two strings as the minimum number of elementary operations (insertions, deletions, and substitutions) to transform one string into the other

\[ d(v,w) = \text{MIN number of elementary operations to transform } v \rightarrow w \]
Edit Distance vs Hamming Distance

Hamming distance always compares
  i-th letter of v with
  i-th letter of w

\[ v = \text{ATATATATAT} \]
\[ w = \text{TATATATATA} \]

Hamming distance:
\[ d(v, w) = 8 \]

Computing Hamming distance is a trivial task.
Edit Distance vs Hamming Distance

Hamming distance always compares $i$-th letter of $v$ with $i$-th letter of $w$

$V = \text{ATATATAT}$

$W = \text{TATATATA}$

Hamming distance:

$d(v, w) = 8$

Computing Hamming distance is a trivial task

Edit distance may compare $i$-th letter of $v$ with $j$-th letter of $w$

$V = \text{- ATATATAT}$

$W = \text{TATATATA}$

Edit distance:

$d(v, w) = 2$

Computing edit distance is a non-trivial task
Edit Distance vs Hamming Distance

Hamming distance always compares $i$-th letter of $v$ with $i$-th letter of $w$

$V = \text{ATATATA}$

$W = \text{TATATATA}$

Hamming distance:

$d(v, w) = 8$

Edit distance may compare $i$-th letter of $v$ with $j$-th letter of $w$

$V = \text{-ATATATA}$

$W = \text{TATATATA}$

Edit distance:

$d(v, w) = 2$

(one insertion and one deletion)

How to find what $j$ goes with what $i$ ???
Edit Distance: Example

TGCATAT \rightarrow ATCCGAT in 5 steps

TGCATAT \rightarrow (delete last T)
TGCATA \rightarrow (delete last A)
TGCAT \rightarrow (insert A at front)
ATGCAT \rightarrow (substitute C for 3rd G)
ATCCAT \rightarrow (insert G before last A)
ATCCGAT \rightarrow (Done)
Edit Distance: Example

TGCATAT $\rightarrow$ ATCCGAT in 5 steps

- TGCATAT $\rightarrow$ (delete last T)
- TGCATA $\rightarrow$ (delete last A)
- TGCAT $\rightarrow$ (insert A at front)
- ATGCAT $\rightarrow$ (substitute C for 3\textsuperscript{rd} G)
- ATCCAT $\rightarrow$ (insert G before last A)
- ATCCGAT (Done)

What is the edit distance? 5?
Edit Distance: Example (cont’d)

TGCATAT → ATCCGAT in 4 steps

TGCATAT  →  (insert A at front)
ATGCATAT  →  (delete 6\textsuperscript{th} T)
ATGCA\textcolor{red}{A} TA  →  (substitute G for 5\textsuperscript{th} A)
ATGCG\textcolor{red}{T}A  →  (substitute C for 3\textsuperscript{rd} G)
ATCCG\textcolor{red}{G}AT  (Done)
**Edit Distance: Example (cont’d)**

TGCATAT $\rightarrow$ ATCCGAT in 4 steps

TGCATAT $\rightarrow$ (insert A at front)
ATGCATAT $\rightarrow$ (delete 6\(^{th}\) T)
ATGCATA $\rightarrow$ (substitute G for 5\(^{th}\) A)
ATGCGTA $\rightarrow$ (substitute C for 3\(^{rd}\) G)
ATCCGAT (Done)

Can it be done in 3 steps???
The Alignment Grid

- Every alignment path is from source to sink
Alignment as a Path in the Edit Graph

- Corresponding path -

(0,0), (1,1), (2,2), (2,3), (3,4), (4,5), (5,5), (6,6), (7,6), (7,7)
Alignments in Edit Graph (cont’d)

and $\rightarrow$ represent indels in $v$ and $w$ with score 0.

$\downarrow$ represent matches with score 1.

- The score of the alignment path is 5.
Alignment as a Path in the Edit Graph

Every path in the edit graph corresponds to an alignment:
Alignment as a Path in the Edit Graph

Old Alignment

\[
\begin{align*}
v &= \text{AT\_GTTAT}\_ \\
w &= \text{ATCGT\_A\_C} \\
012345677
\end{align*}
\]

New Alignment

\[
\begin{align*}
v &= \text{AT\_GTTAT}\_ \\
w &= \text{ATCG\_TA\_C} \\
012345667
\end{align*}
\]
Alignment as a Path in the Edit Graph

\[ v = \text{AT\_GT\_TAT\_} \]

\[ w = \text{ATCGT\_A\_C} \]

\[ (0,0), (1,1), (2,2), (2,3), (3,4), (4,5), (5,5), (6,6), (7,6), (7,7) \]
Alignment: Dynamic Programming

\[
\begin{align*}
    s_{i,j} &= \max \begin{cases} 
        s_{i-1,j-1} + 1 & \text{if } v_i = w_j \\
        s_{i-1,j} & \\
        s_{i,j-1} & 
    \end{cases} \\
\end{align*}
\]
Dynamic Programming Example

Initialize 1\(^{st}\) row and 1\(^{st}\) column to be all zeroes.

Or, to be more precise, initialize 0\(^{th}\) row and 0\(^{th}\) column to be all zeroes.
Dynamic Programming Example

\[ S_{i,j} = \max \left( S_{i-1,j-1} + \text{value from NW +1, if } v_i = w_j, S_{i-1,j} \right) \]

\[ S_{i,j} \leftarrow \text{value from North (top)} \]

\[ S_{i,j} \leftarrow \text{value from West (left)} \]
Alignment: Backtracking

Arrows show where the score originated from.

- if from the top
- if from the left
- if \( v_i = w_j \)
Find a match in row and column 2.

i=2, j=2,5 is a match (T).

j=2, i=4,5,7 is a match (T).

Since $v_i = w_j$, $s_{i,j} = s_{i-1,j-1} + 1$

$$s_{2,2} = [s_{1,1} = 1] + 1$$
$$s_{2,5} = [s_{1,4} = 1] + 1$$
$$s_{4,2} = [s_{3,1} = 1] + 1$$
$$s_{5,2} = [s_{4,1} = 1] + 1$$
$$s_{7,2} = [s_{6,1} = 1] + 1$$
Backtracking Example

Continuing with the dynamic programming algorithm gives this result.
Alignment: Dynamic Programming

\[ s_{i, j} = \begin{cases} 
  s_{i-1, j-1} + 1 & \text{if } v_i = w_j \\
  \max & \\
  s_{i-1, j} \\
  s_{i, j-1} &
\end{cases} \]
Alignment: Dynamic Programming

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

This recurrence corresponds to the Manhattan Tourist problem (three incoming edges into a vertex) with all horizontal and vertical edges weighted by zero.
LCS Algorithm

1. \textbf{Levenshtein}(v,w)
2. \hspace{1em} \textbf{for} \ i \leftarrow 1 \ \textbf{to} \ n
3. \hspace{2em} s_{i,0} \leftarrow 0
4. \hspace{1em} \textbf{for} \ j \leftarrow 1 \ \textbf{to} \ m
5. \hspace{2em} s_{0,j} \leftarrow 0
6. \hspace{1em} \textbf{for} \ i \leftarrow 1 \ \textbf{to} \ n
7. \hspace{2em} \textbf{for} \ j \leftarrow 1 \ \textbf{to} \ m
8. \hspace{3em} s_{i,j} \leftarrow \max\left\{ s_{i-1,j}, s_{i,j-1}, s_{i-1,j-1} + 1, \text{ if } v_i = w_j \right\}
9. \hspace{3em} b_{i,j} \leftarrow \left\{ \begin{array}{ll}
\text{"↑" if } s_{i,j} = s_{i-1,j} \\
\text{"←" if } s_{i,j} = s_{i,j-1} \\
\text{" " if } s_{i,j} = s_{i-1,j-1} + 1
\end{array} \right.
10. \hspace{1em} \textbf{return} \ \ (s_{n,m}, b)
Now What?

- LCS($v,w$) created the alignment grid
- Now we need a way to read the best alignment of $v$ and $w$
- Follow the arrows backwards from sink
1. **PrintLCS** (*b,v,i,j*)
2. if \( i = 0 \) or \( j = 0 \)
3. return
4. if \( b_{i,j} = "\uparrow" \)
5. **PrintLCS** (*b,v,i−1,j−1*)
6. print \( v_i \)
7. else
8. if \( b_{i,j} = "\downarrow" \)
9. **PrintLCS** (*b,v,i−1,j*)
10. else
11. **PrintLCS** (*b,v,i,j−1*)
LCS Runtime

- It takes $O(nm)$ time to fill in the $n \cdot m$ dynamic programming matrix.