Dynamic Programming: Edit Distance

Outline

- DNA Sequence Comparison: First Success Stories
- Change Problem
- Manhattan Tourist Problem
- Longest Paths in Graphs
- Sequence Alignment
- Edit Distance
- Longest Common Subsequence Problem
- Dot Matrices

DNA Sequence Comparison: First Success Story

- Finding sequence similarities with genes of known function is a common approach to infer a newly sequenced gene's function

- In 1984 Russell Doolittle and colleagues found similarities between cancer-causing gene and normal growth factor (PDGF) gene

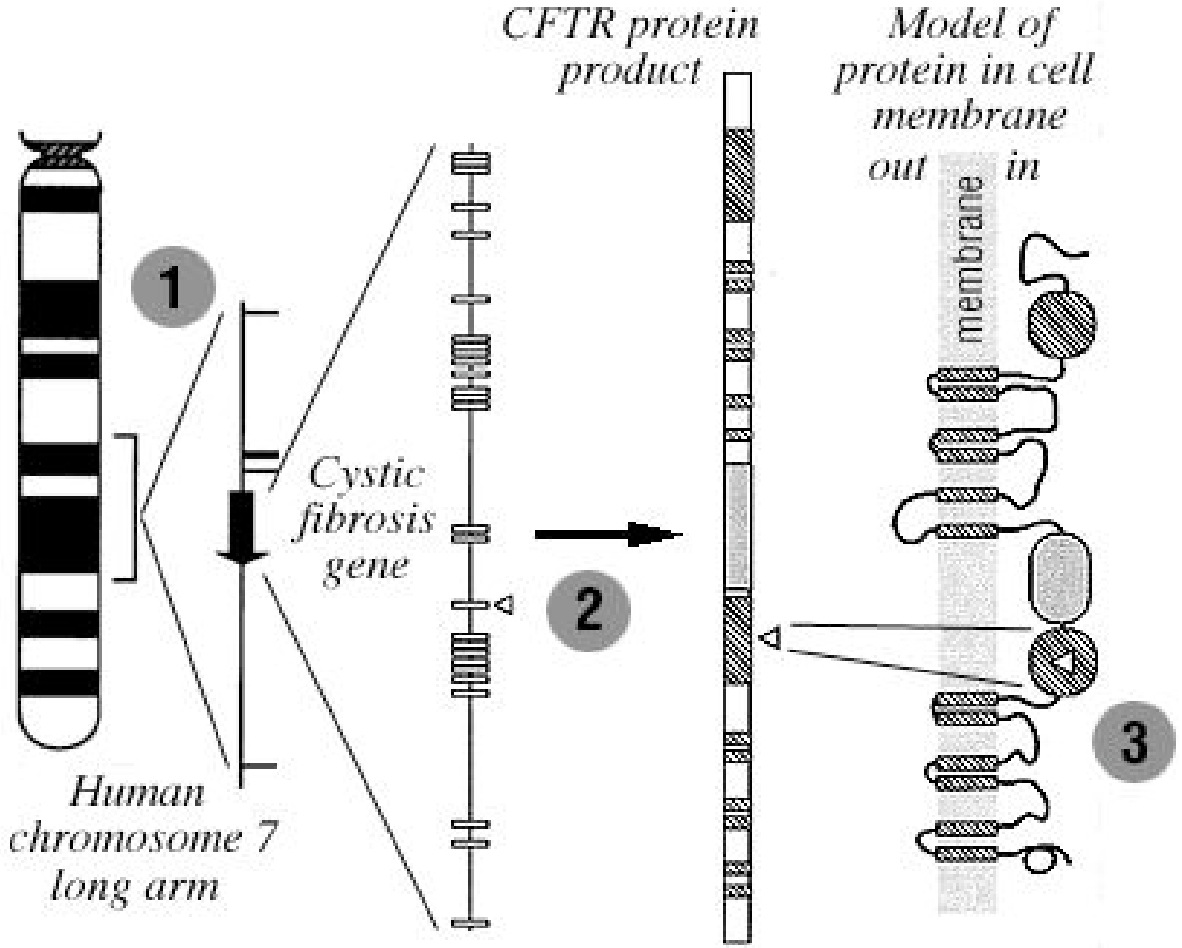
Cystic Fibrosis

- **Cystic fibrosis** (CF) is a chronic and frequently fatal genetic disease of the body's mucus glands (abnormally high level of mucus in glands). CF primarily affects the respiratory systems in children.
- Mucus is a slimy material that coats many epithelial surfaces and is secreted into fluids such as saliva

Cystic Fibrosis: Inheritance

- In early 1980s biologists hypothesized that CF is an autosomal recessive disorder caused by mutations in a gene that remained unknown till 1989
- Heterozygous carriers are asymptomatic
- Must be homozygously recessive in this gene in order to be diagnosed with CF

Cystic Fibrosis: Finding the Gene



Finding Similarities between the Cystic Fibrosis Gene and ATP binding proteins

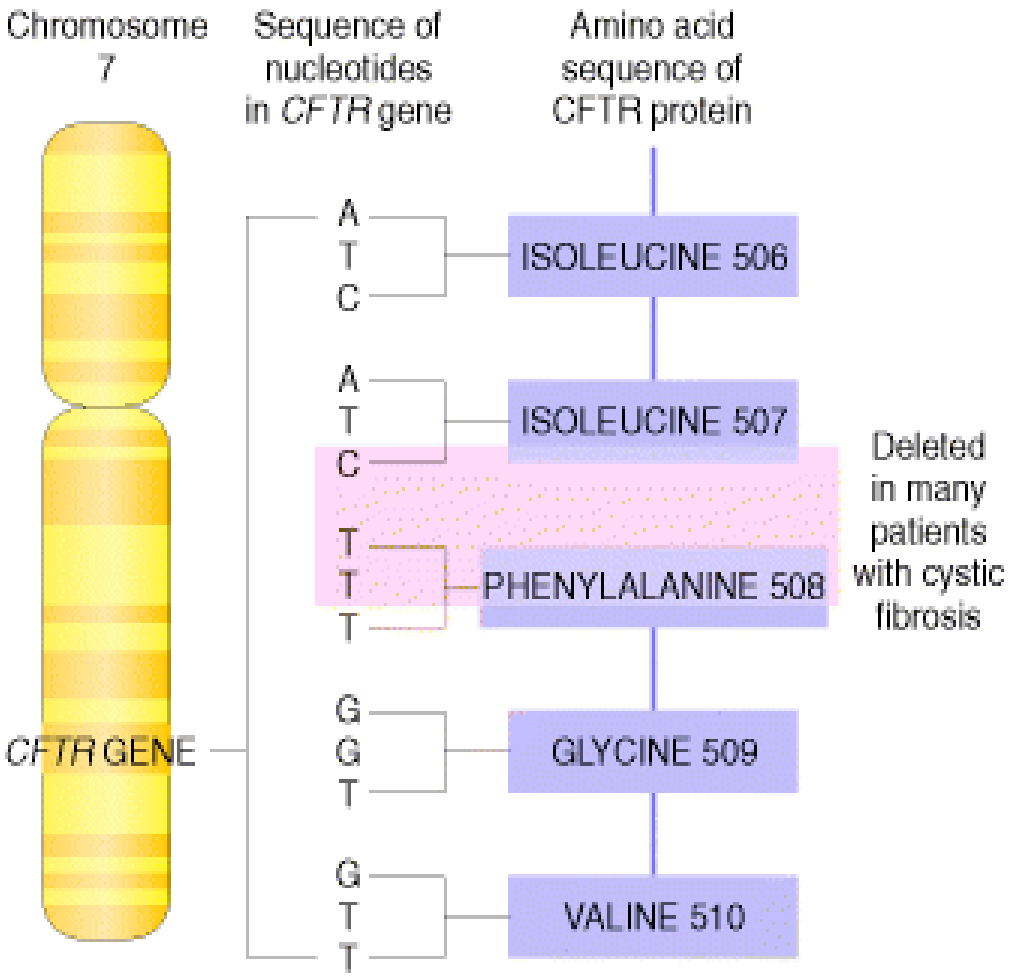
- ATP binding proteins are present on cell membrane and act as transport channel
- In 1989 biologists found similarity between the cystic fibrosis gene and ATP binding proteins
- A plausible function for cystic fibrosis gene, given the fact that CF involves sweet secretion with abnormally high sodium level

Cystic Fibrosis: Mutation Analysis

If a high % of cystic fibrosis (CF) patients have a certain mutation in the gene and the normal patients don't, then that could be an indicator of a mutation that is related to CF

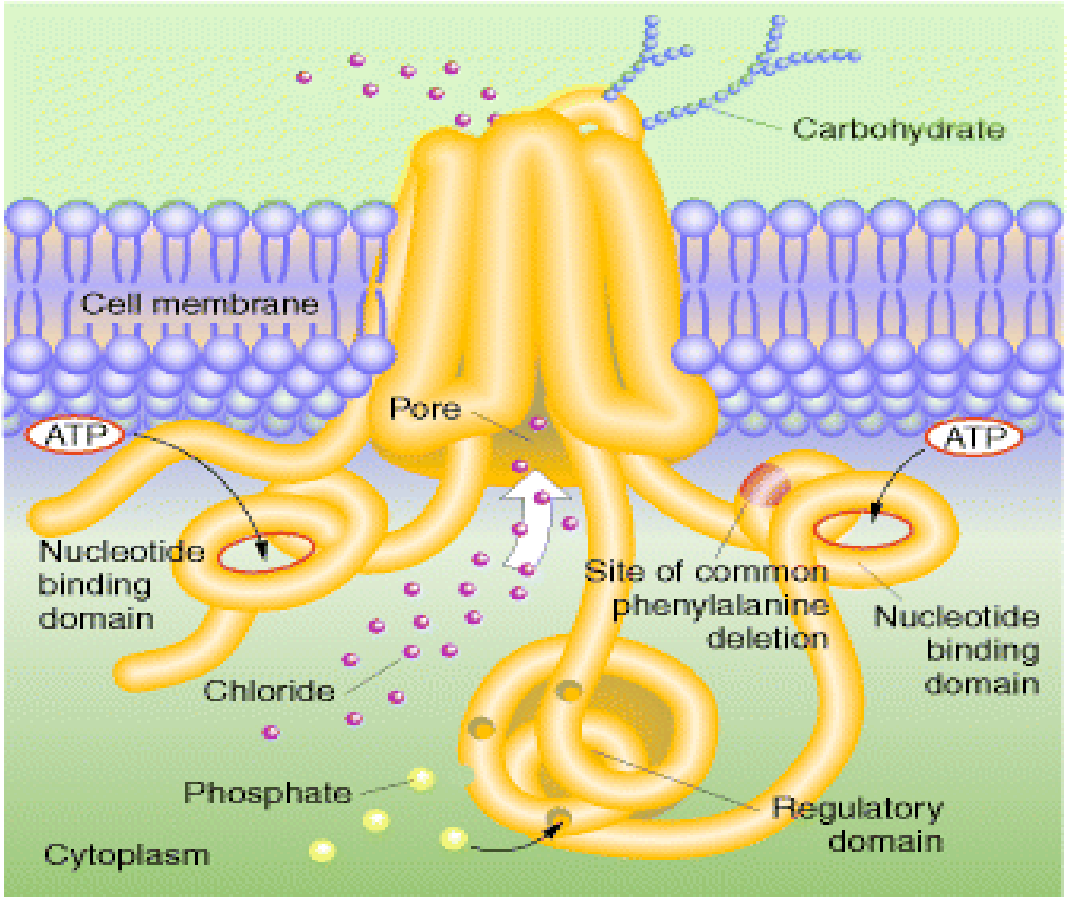
A certain mutation was found in 70% of CF patients, convincing evidence that it is a predominant genetic diagnostics marker for CF

Cystic Fibrosis and CFTR Gene :



Cystic Fibrosis and the CFTR Protein

- **CFTR (Cystic Fibrosis Transmembrane conductance Regulator)** protein is acting in the cell membrane of epithelial cells that secrete mucus
- These cells line the airways of the nose, lungs, the stomach wall, etc.



Mechanism of Cystic Fibrosis

- The **CFTR protein** (1480 amino acids) regulates a chloride ion channel
- Adjusts the "wateriness" of fluids secreted by the cell
- Those with cystic fibrosis are missing one single amino acid in their CFTR
- Mucus ends up being too thick, affecting many organs

Bring in the Bioinformaticians

- Gene similarities between two genes with known and unknown function alert biologists to some possibilities
- Computing a similarity score between two genes tells how likely it is that they have similar functions
- Dynamic programming is a technique for revealing similarities between genes
- The ***Change Problem*** is a good problem to introduce the idea of dynamic programming

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, …, 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, …, i_d such that

$$c_1 i_1 + c_2 i_2 + \ldots + c_d i_d = M$$

and $i_1 + i_2 + \ldots + i_d$ is minimal

Change Problem: Example

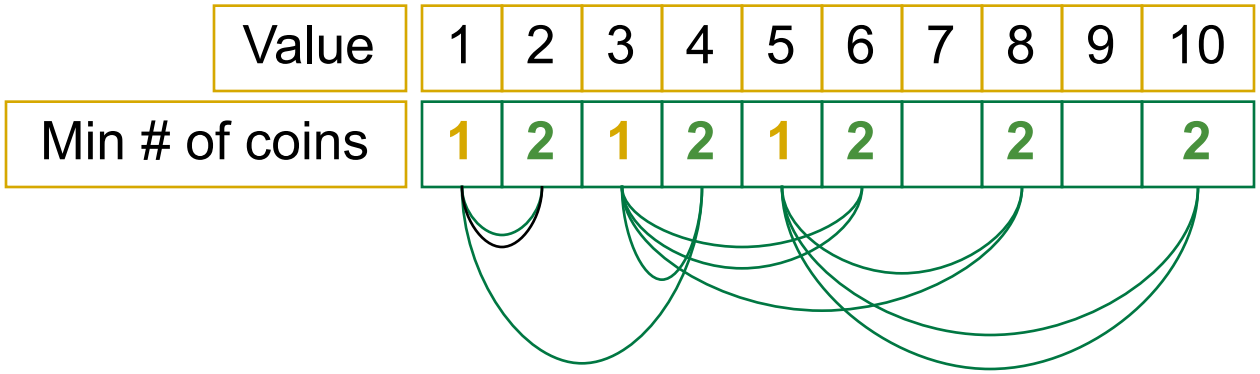
Given the denominations 1, 3, and 5, what is the minimum number of coins needed to make change for a given value?

Value	1	2	3	4	5	6	7	8	9	10
Min # of coins	1		1		1					

Only one coin is needed to make change for the values 1, 3, and 5

Change Problem: Example (cont'd)

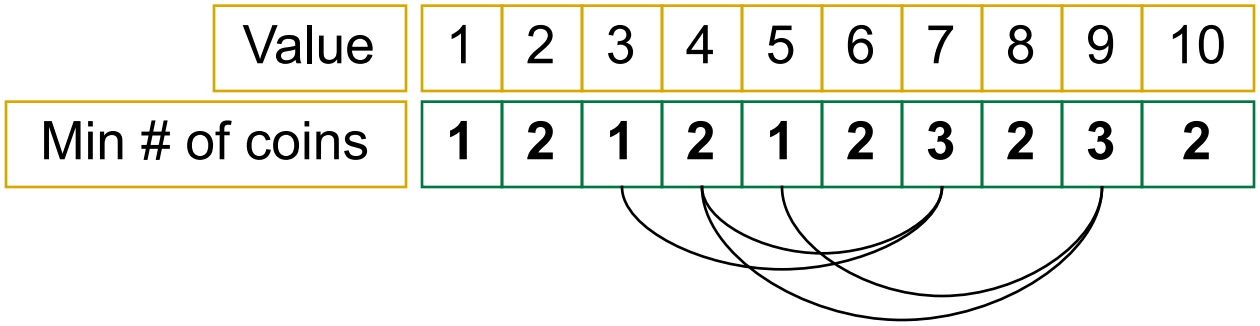
Given the denominations 1, 3, and 5, what is the minimum number of coins needed to make change for a given value?



However, two coins are needed to make change for the values 2, 4, 6, 8, and 10.

Change Problem: Example (cont'd)

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



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:

$$minNumCoins(M) = \textbf{min of} \begin{cases} minNumCoins(M-1) + 1 \\ minNumCoins(M-3) + 1 \\ minNumCoins(M-5) + 1 \end{cases}$$

Change Problem: Recurrence (cont'd)

Given the denominations $\boldsymbol{c}$: $c_1, c_2, \ldots, c_{\boldsymbol{d}}$, the recurrence relation is:

$$minNumCoins(M) = \textbf{min of} \begin{cases} minNumCoins(M-c_1) + 1 \\ minNumCoins(M-c_2) + 1 \\ \ldots \\ minNumCoins(M-c_d) + 1 \end{cases}$$

Change Problem: A Recursive Algorithm

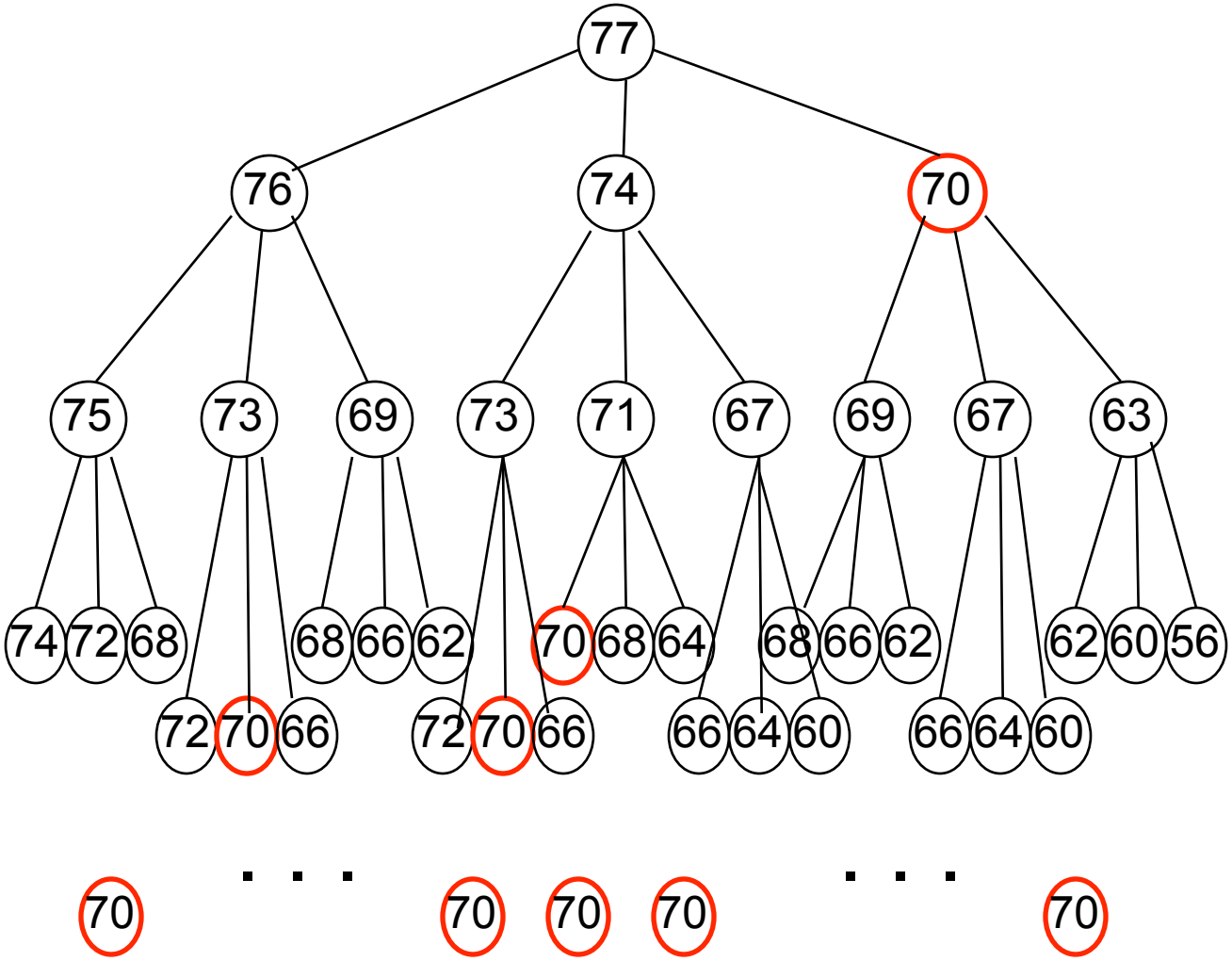
```
1.  RecursiveChange(M,c,d)
2.    if M = 0
3.      return 0
4.    bestNumCoins ß infinity
5.    for i ß 1 to d
6.      if M ≥ c_i
7.        numCoins ß RecursiveChange(M – c_i , c, d)
8.        if numCoins + 1 < bestNumCoins
9.          bestNumCoins ß numCoins + 1
10.   return bestNumCoins
```

RecursiveChange 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 RecursiveChange Tree



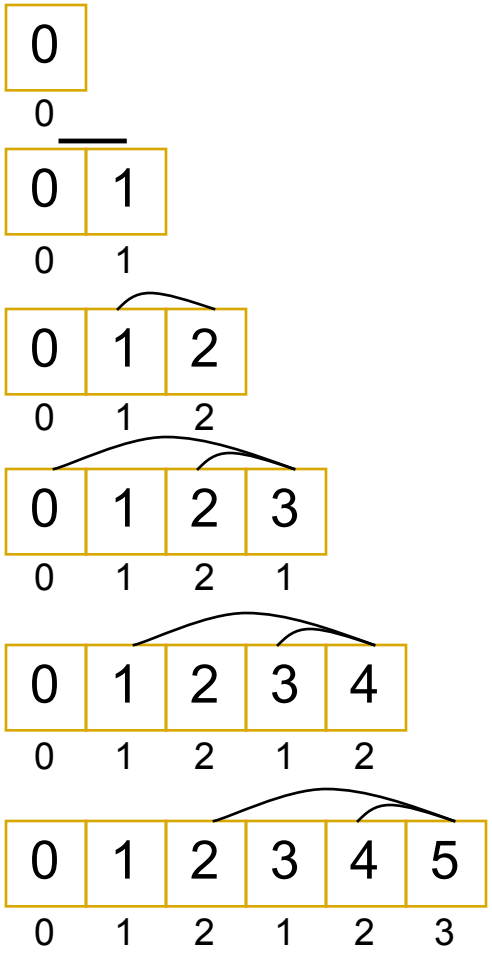
We Can Do Better

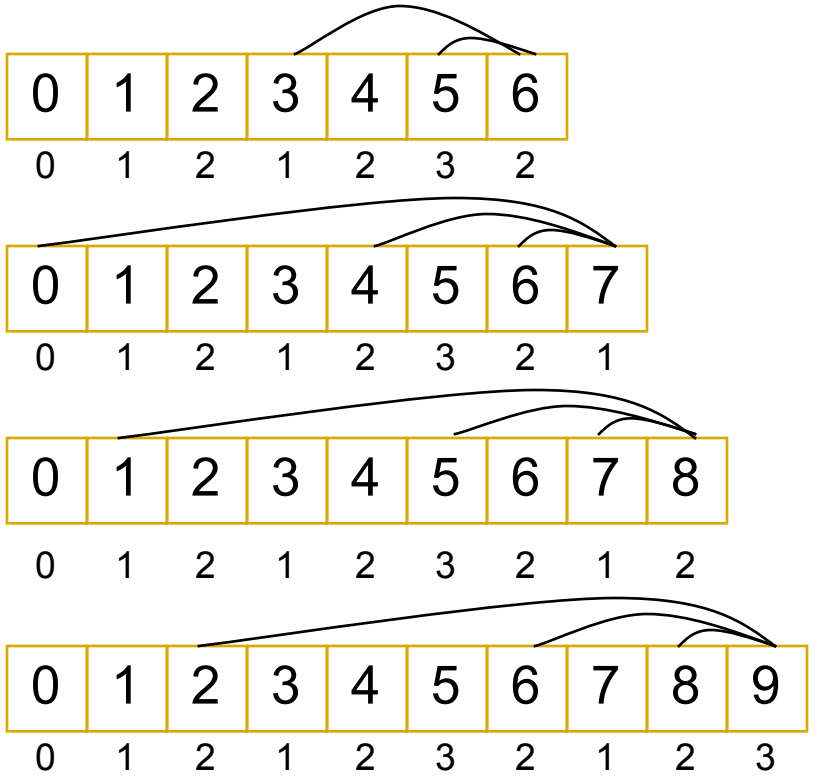
- We're re-computing values in our algorithm more than once
- Save results of each computation for 0 to ***M***
- This way, we can do a reference call to find an already computed value, instead of re-computing each time
- Running time ***M*d***, where ***M*** is the value of money and ***d*** is the number of denominations

The Change Problem: Dynamic Programming

- DPChange(*M*,*c*,*d*)
- $bestNumCoins_0$ ß 0
- for *m* ß 1 to *M*
- $bestNumCoins_m$ ß infinity
- for *i* ß 1 to *d*
- if $m \geq c_i$
- if $bestNumCoins_{m - c_i} + 1 < bestNumCoins_m$
- $bestNumCoins_m$ ß $bestNumCoins_{m - c_i} + 1$
- return $bestNumCoins_M$

DPChange: Example



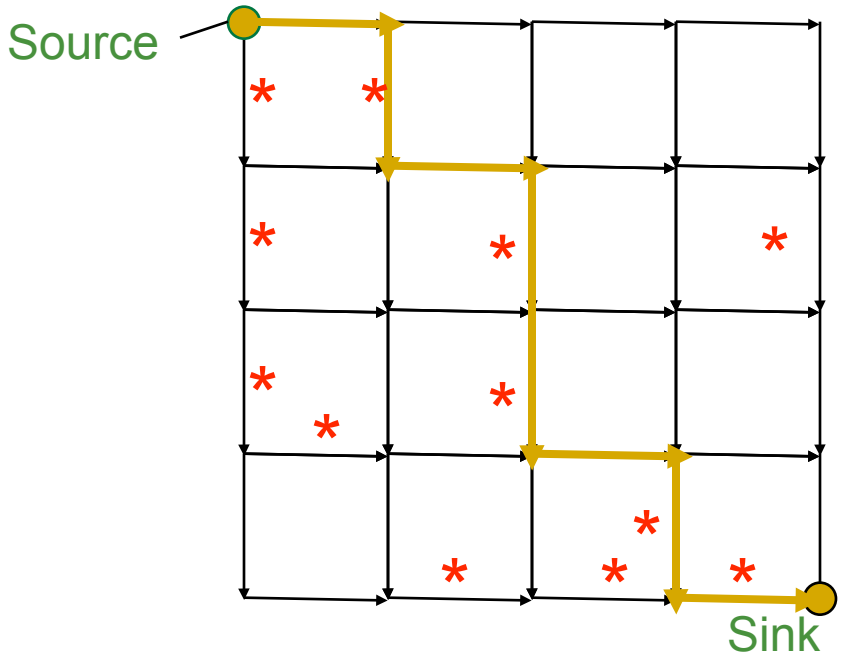


$\boldsymbol{c} = (1,3,7)$

$\boldsymbol{M} = 9$

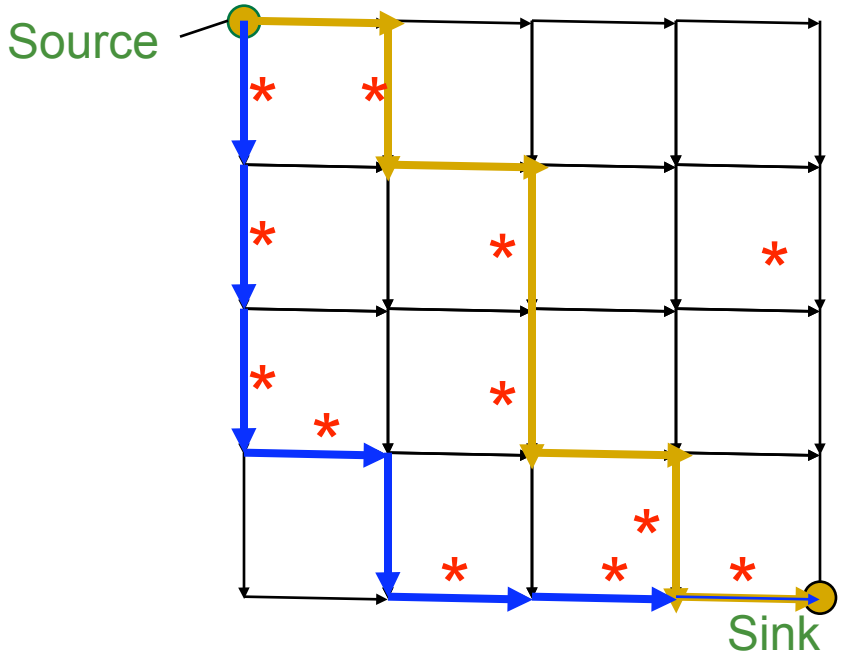
Manhattan Tourist Problem (MTP)

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 (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



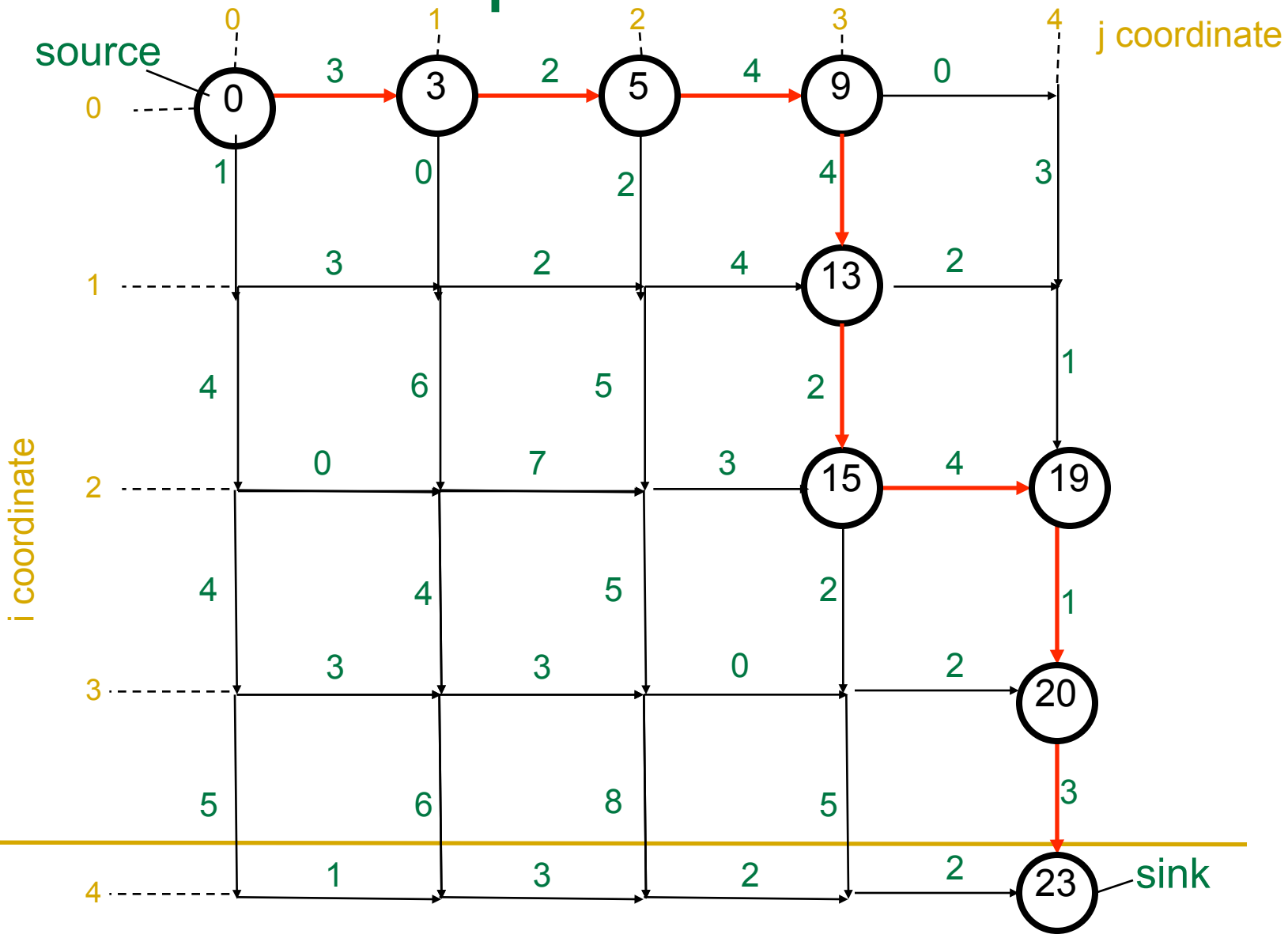
Manhattan Tourist Problem: Formulation

Goal: Find the longest path in a weighted grid.

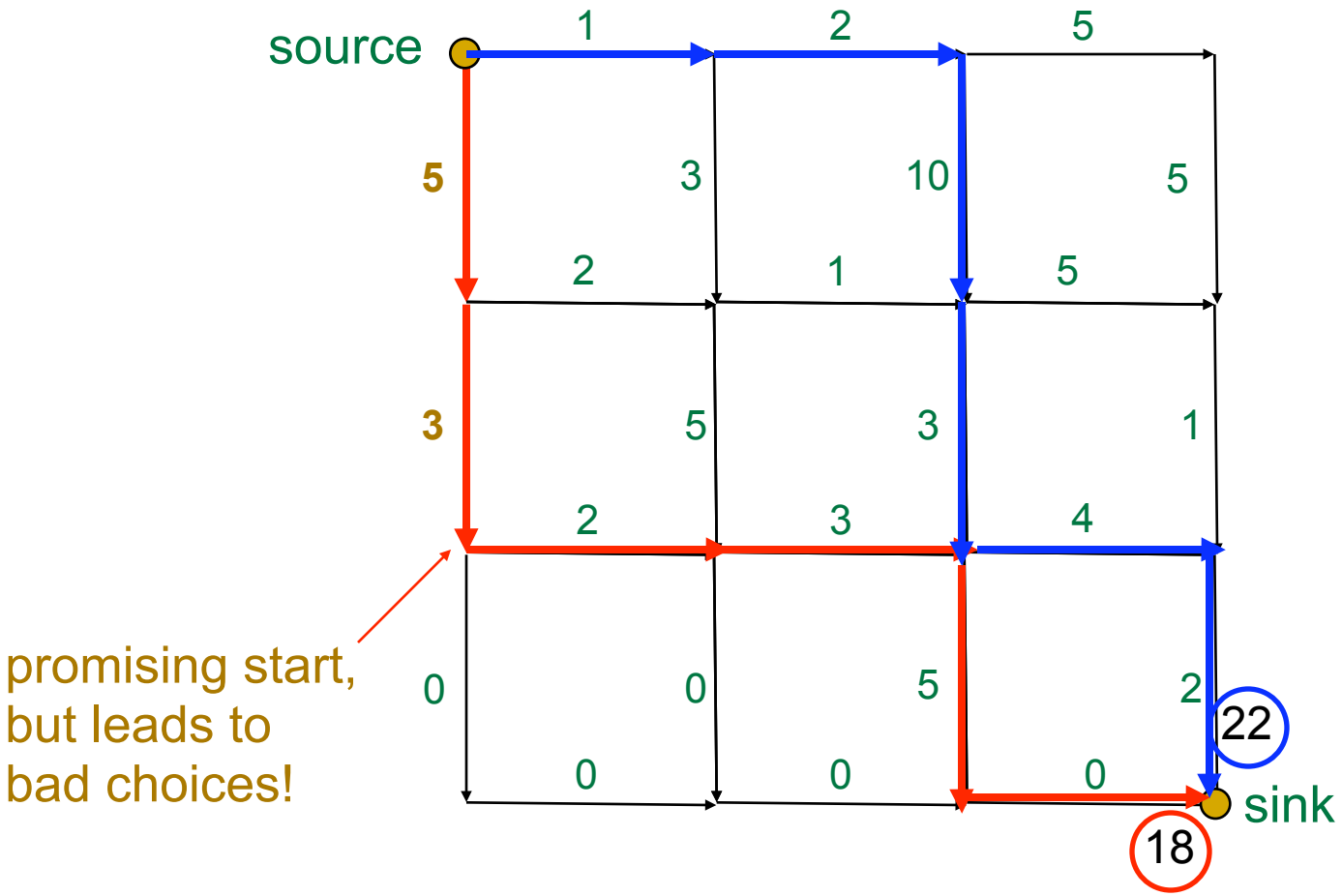
Input: A weighted grid ***G*** with two distinct vertices, one labeled "*source*" and the other labeled "*sink*"

Output: A longest path in ***G*** from "*source*" to "*sink*"

MTP: An Example



MTP: Greedy Algorithm Is Not Optimal



MTP: Simple Recursive Program

```
MT(n,m)
  if n=0  or m=0
    return MT(n,m)
  x ß MT(n-1,m)+
            length of the edge from (n- 1,m) to (n,m)
  y ß MT(n,m-1)+
            length of the edge from (n,m-1) to (n,m)
  return max{x,y}
```

MTP: Simple Recursive Program

MT(*n*,*m*)

x ß *MT(n-1,m)*+

length of the edge from *(n- 1,m) to (n,m)*

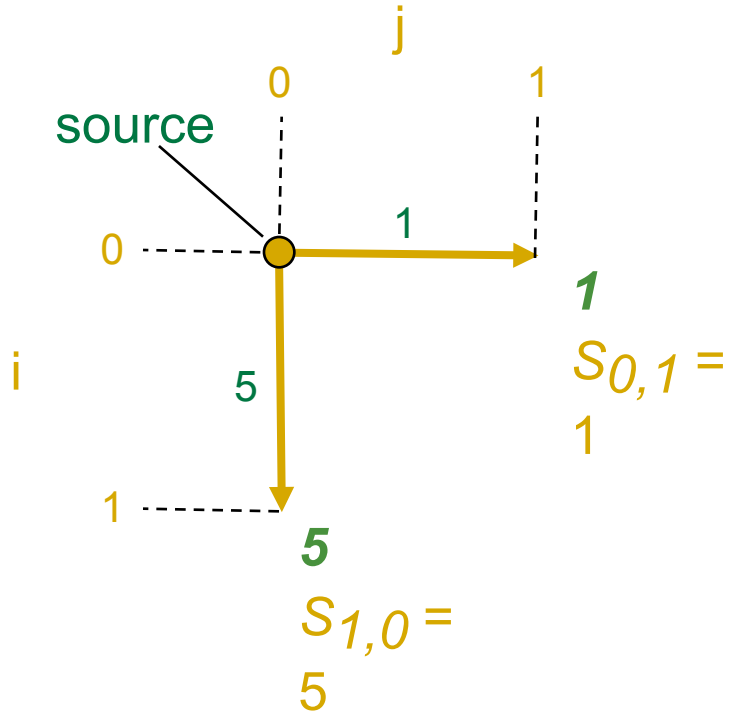
y ß *MT(n,m-1)*+

length of the edge from *(n,m-1) to (n,m)*

return *min{x,y}*

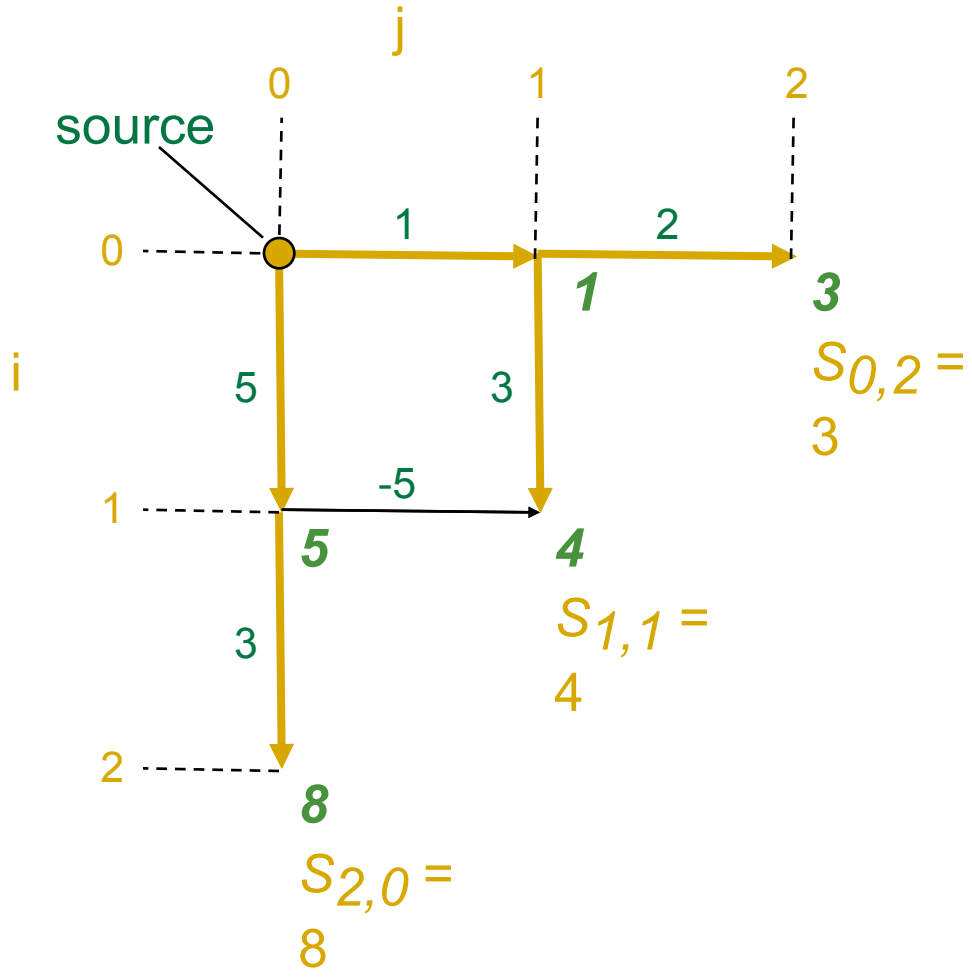
What's wrong with this approach?

MTP: Dynamic Programming

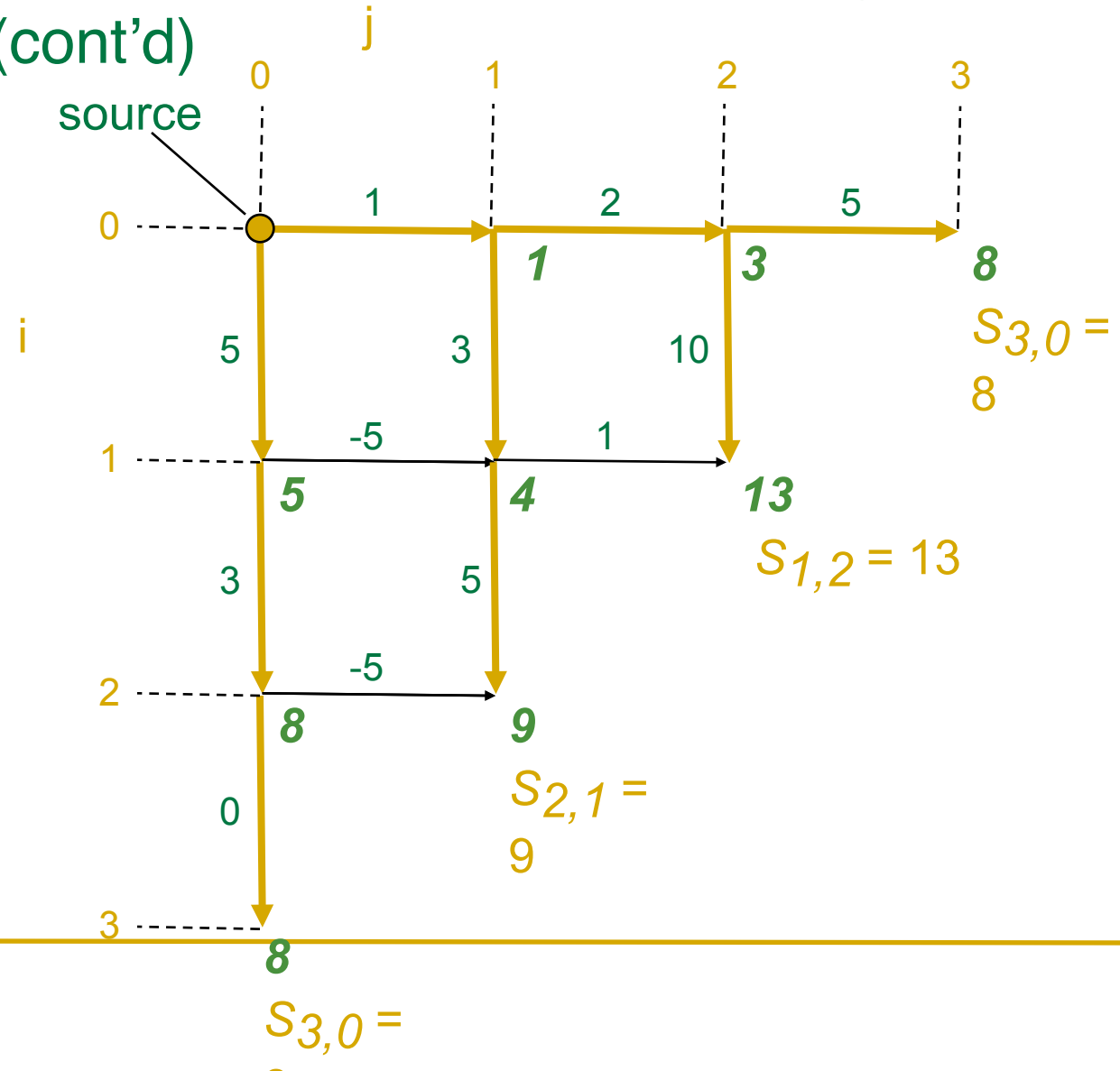


- Calculate optimal path score for each vertex in the graph
- Each vertex's score is the maximum of the prior vertices score plus the weight of the respective edge in between

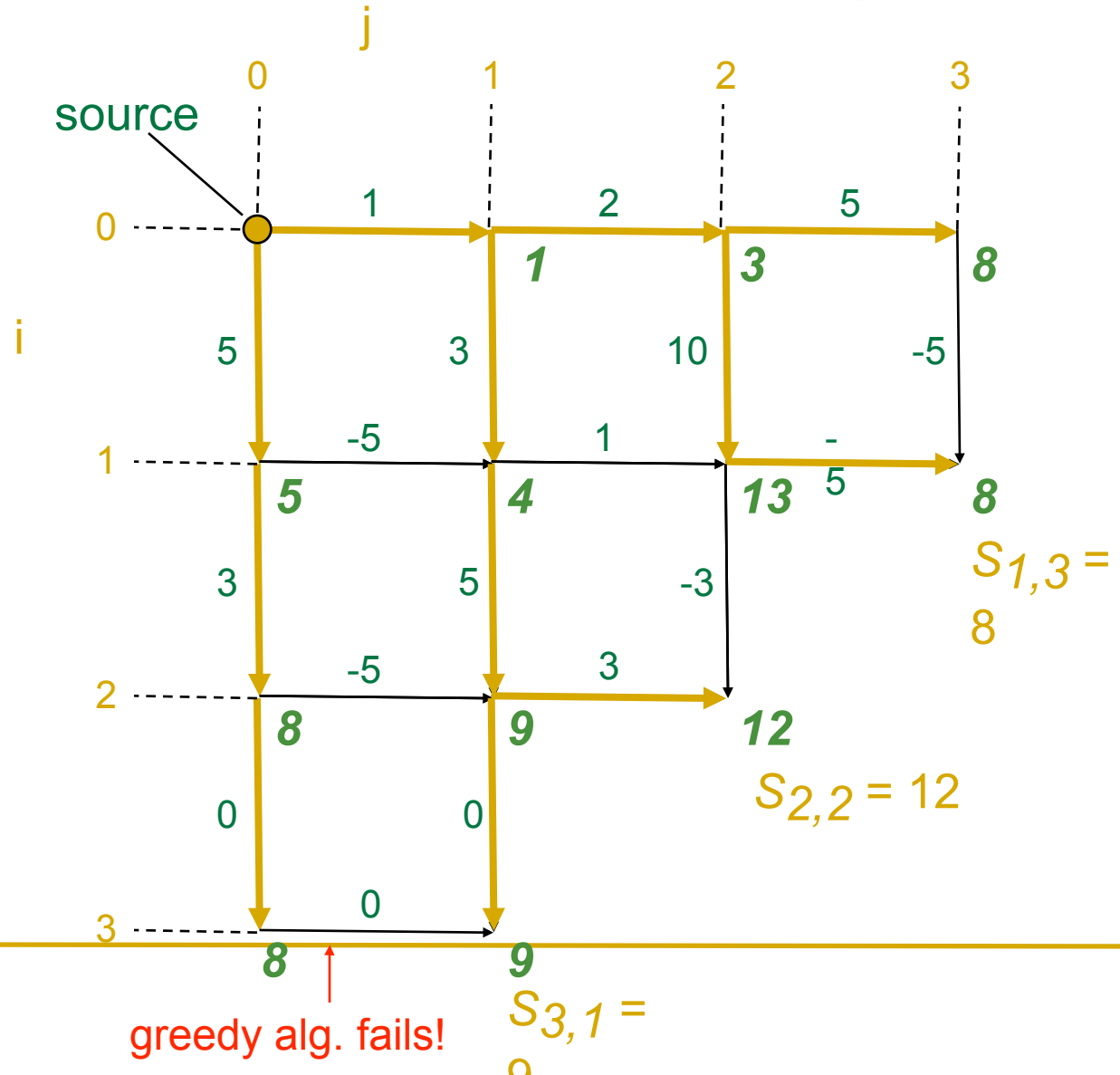
MTP: Dynamic Programming (cont'd)



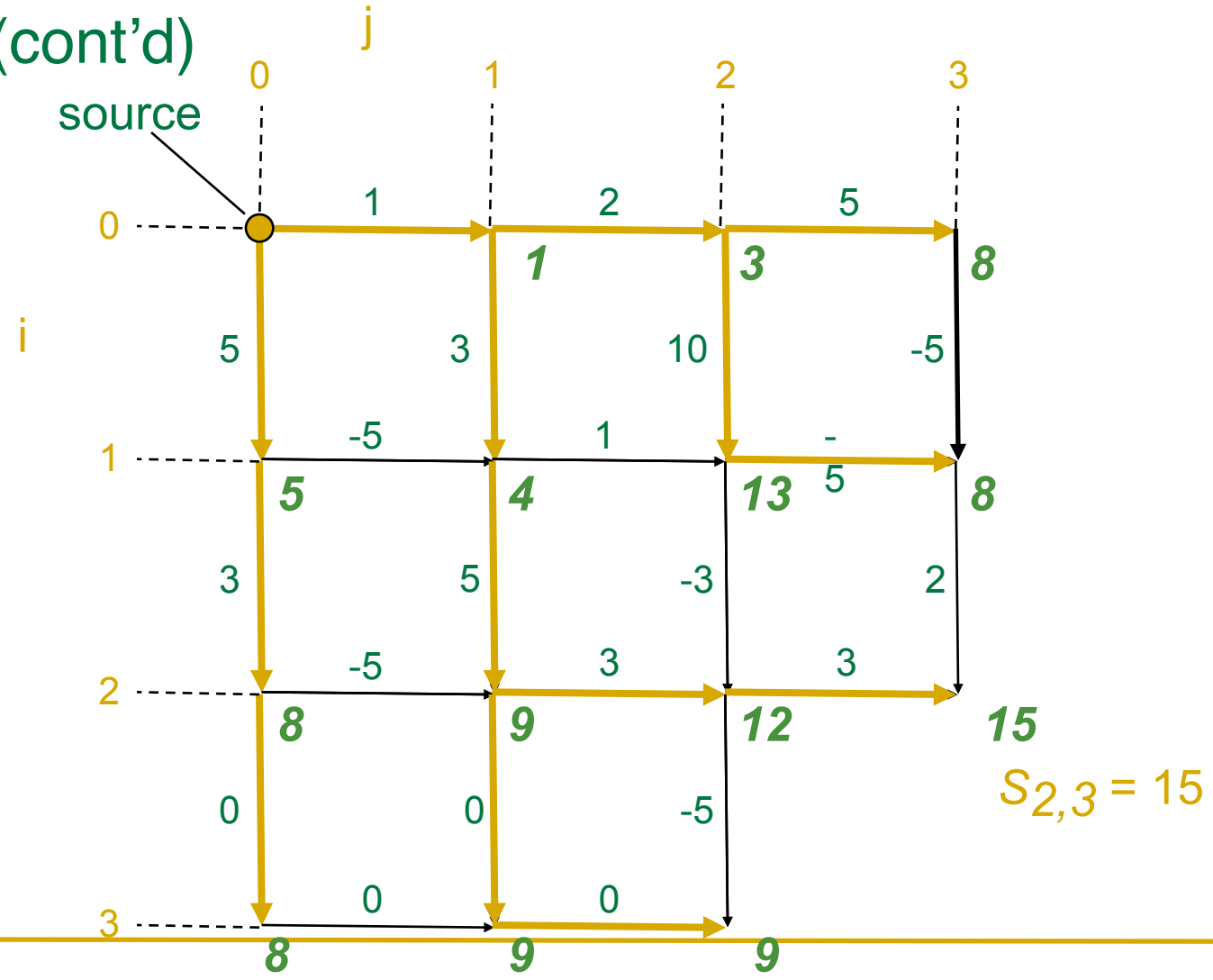
MTP: Dynamic Programming (cont'd)



MTP: Dynamic Programming (cont'd)



MTP: Dynamic Programming (cont'd)

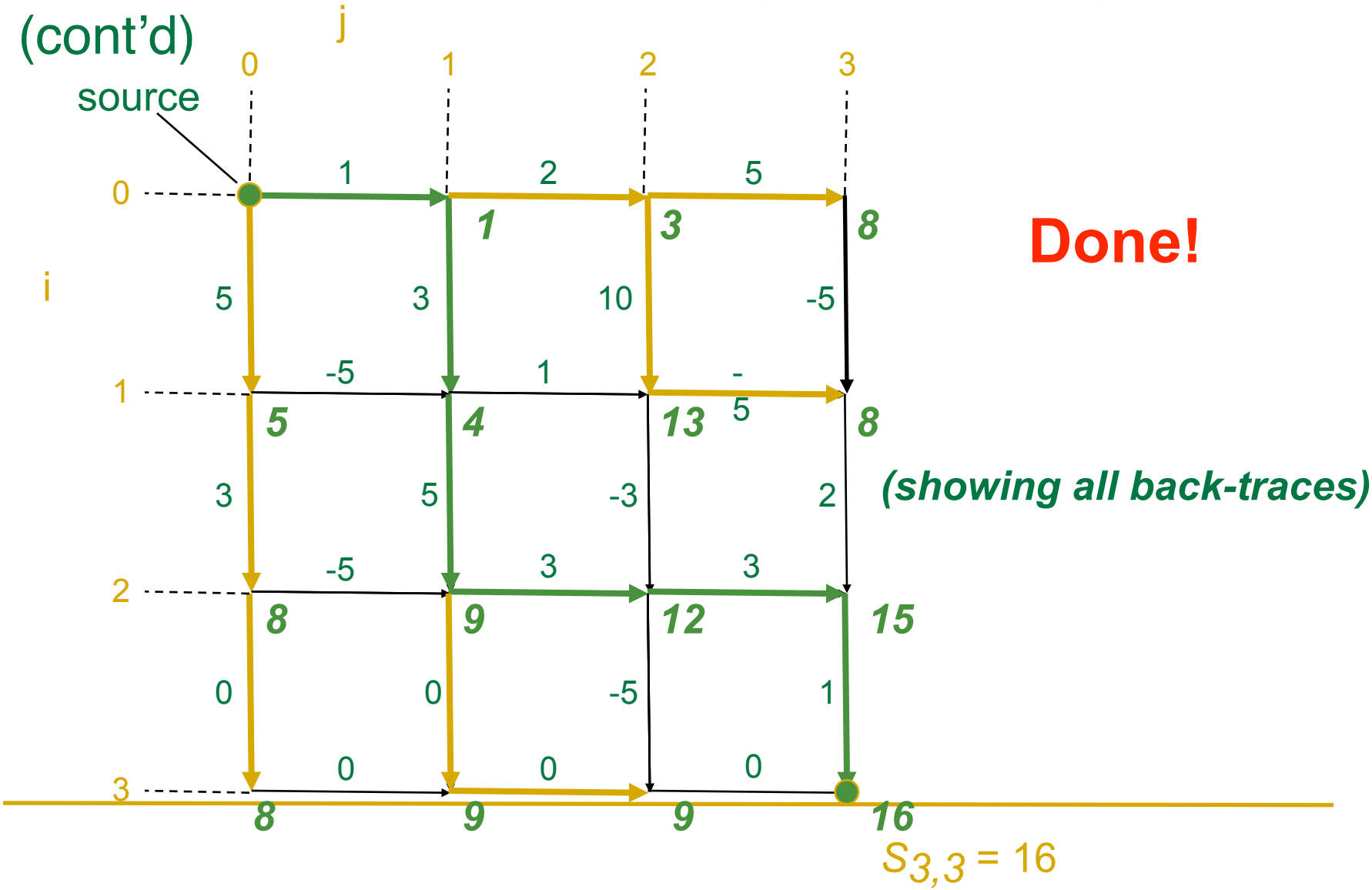


$S_{2,3} = 15$

$S_{3,2} =$

MTP: Dynamic Programming

(cont'd)



Done!

(showing all back-traces)

$S_{3,3} = 16$

MTP: Recurrence

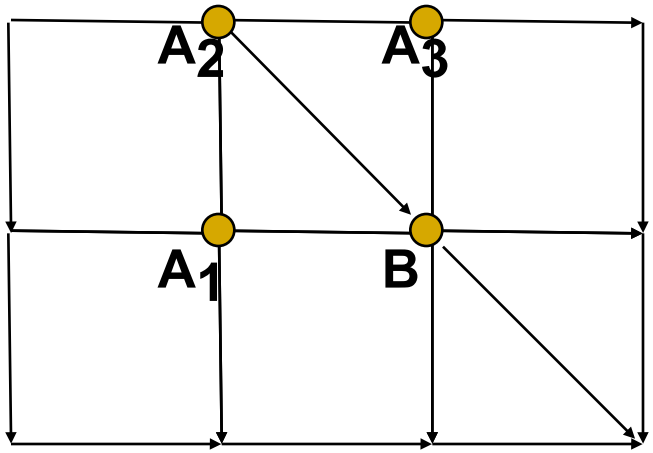
Computing the score for a point *(i,j)* by the recurrence relation:

$$s_{i,j} = \max \begin{cases} s_{i-1,j} + \text{weight of the edge between } (i-1, j) \text{ and } (i, j) \\ s_{i,j-1} + \text{weight of the edge between } (i, j-1) \text{ and } (i, j) \end{cases}$$

The running time is ***n x m*** for a ***n*** by ***m*** grid

(***n*** = # of rows, ***m*** = # of columns)

Manhattan Is Not A Perfect Grid



What about diagonals?

- The score at point B is given by:

$$s_B = \max \text{ of } \begin{cases} s_{A1} + \text{weight of the edge } (A_1, B) \\ s_{A2} + \text{weight of the edge } (A_2, B) \\ s_{A3} + \text{weight of the edge } (A_3, B) \end{cases}$$

Manhattan Is Not A Perfect Grid (cont'd)

Computing the score for point ***x*** is given by the recurrence relation:

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

- 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

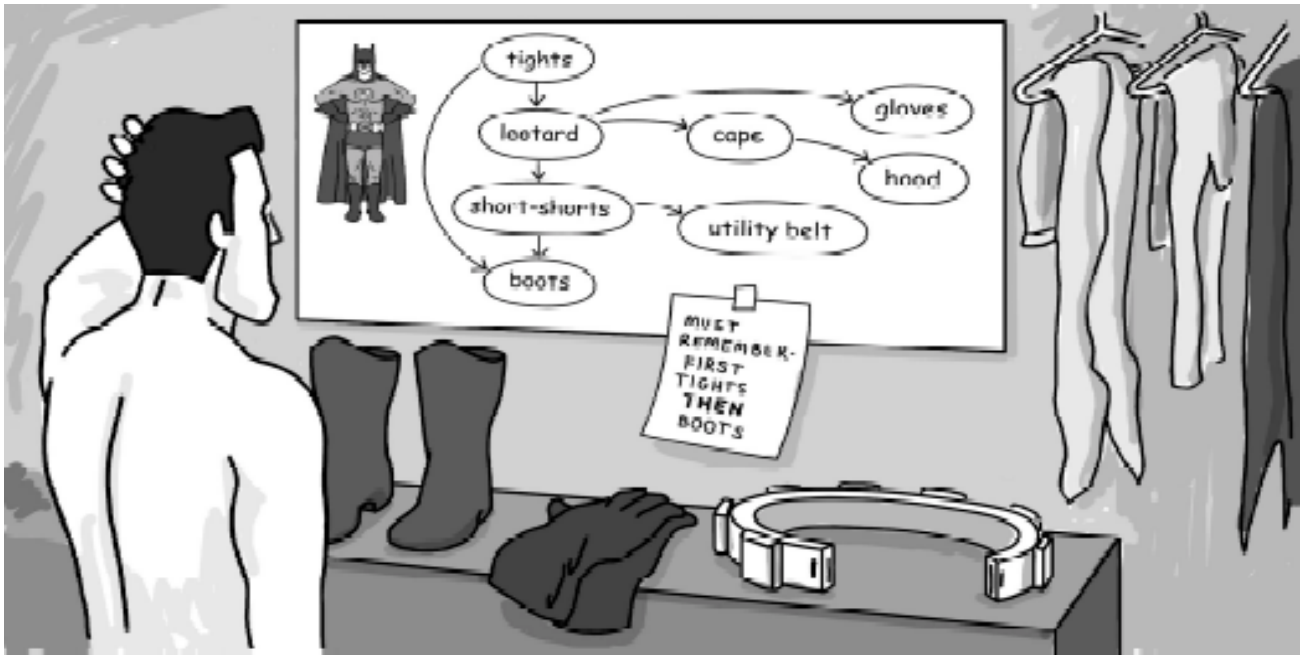
Traveling in the Grid

- 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
- Try to find such order for a directed cycle

???

DAG: Directed Acyclic Graph

- Since Manhattan is not a perfect regular grid, we represent it as a DAG
- DAG for *Dressing in the morning* problem

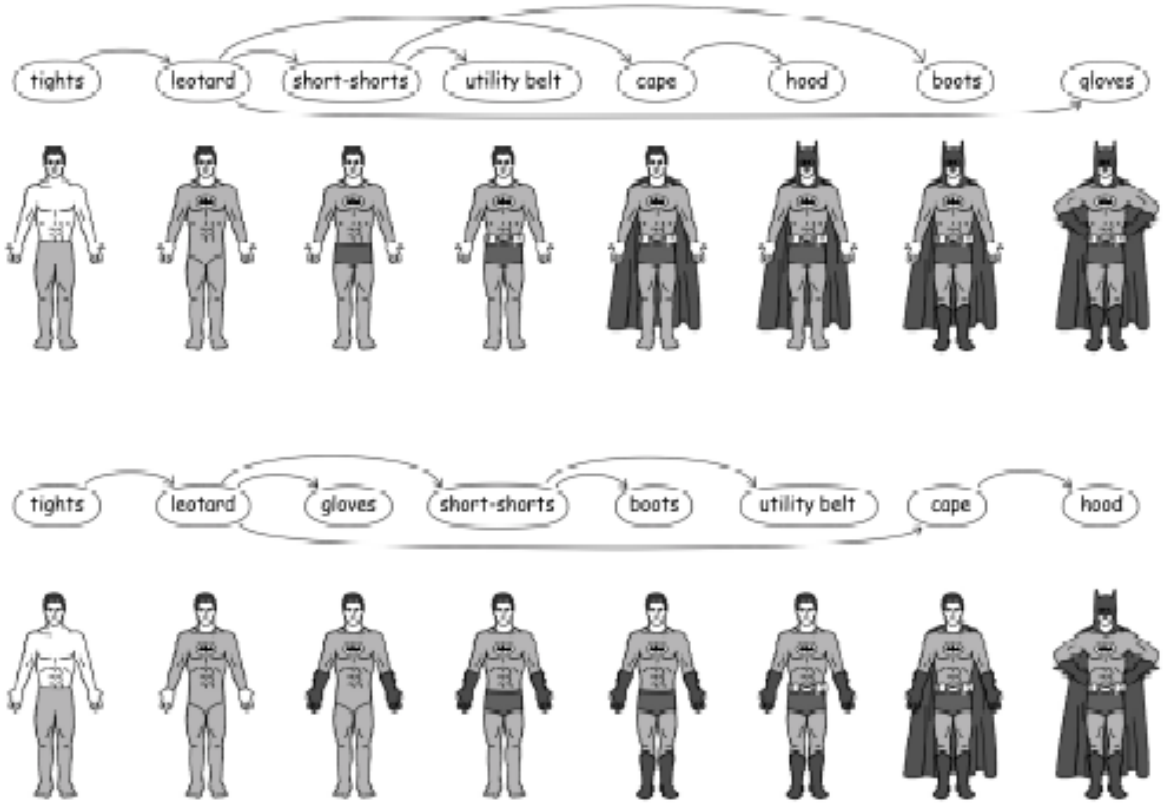


Topological Ordering

- A numbering of vertices of the graph is called ***topological ordering*** of the DAG if every edge of the DAG connects a vertex with a smaller label to a vertex with a larger label
- In other words, if vertices are positioned on a line in an increasing order of labels then all edges go from left to right.

Topological ordering

- 2 different topological orderings of the 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:

$$s_v = \max \text{ of } \begin{cases} s_{u_1} + \text{weight of edge from } u_1 \text{ to } v \\ s_{u_2} + \text{weight of edge from } u_2 \text{ to } v \\ s_{u_3} + \text{weight of edge from } u_3 \text{ to } v \end{cases}$$

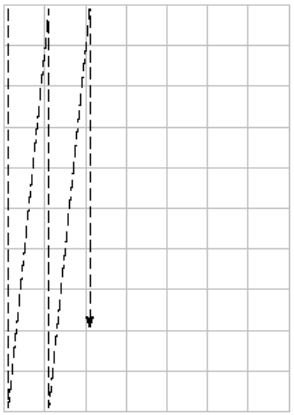
In General:

$s_V = \max_U$ (s_U + weight of edge from ***u*** to ***v***)

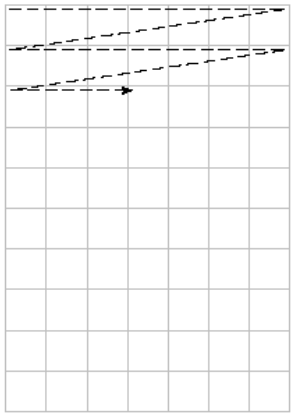
Traversing the Manhattan Grid

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

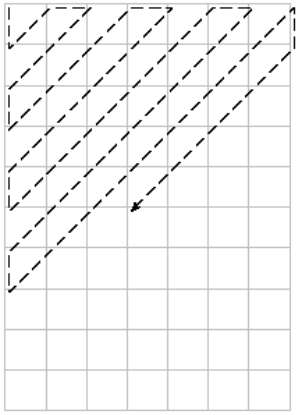
a)



b)



c)



Alignment: 2 row representation

Given 2 DNA sequences **v** and **w**:

v : A T C T G A T ***m*** = 7
w : T G C A T A ***n*** = 6

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

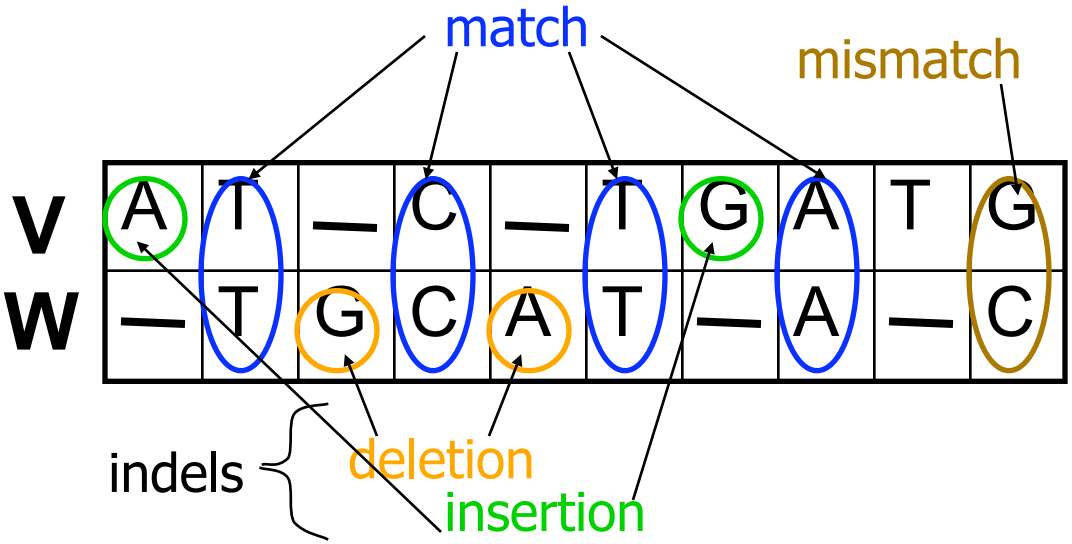
letters of ***v***	A	T	--	G	T	T	A	T	--
letters of ***w***	A	T	C	G	T	--	A	--	C

4 matches 2 insertions 2 deletions

Aligning DNA Sequences

v = ATCTGATG n = 8

w = TGCATAC m = 7



4 matches
1 mismatches
2 insertions
2 deletions

Longest Common Subsequence (LCS) – Alignment without Mismatches

- Given two sequences

$$\boldsymbol{v} = v_1\, v_2 \ldots v_m \text{ and } \boldsymbol{w} = w_1\, w_2 \ldots w_n$$

- The LCS of ***v*** and ***w*** is a sequence of positions in

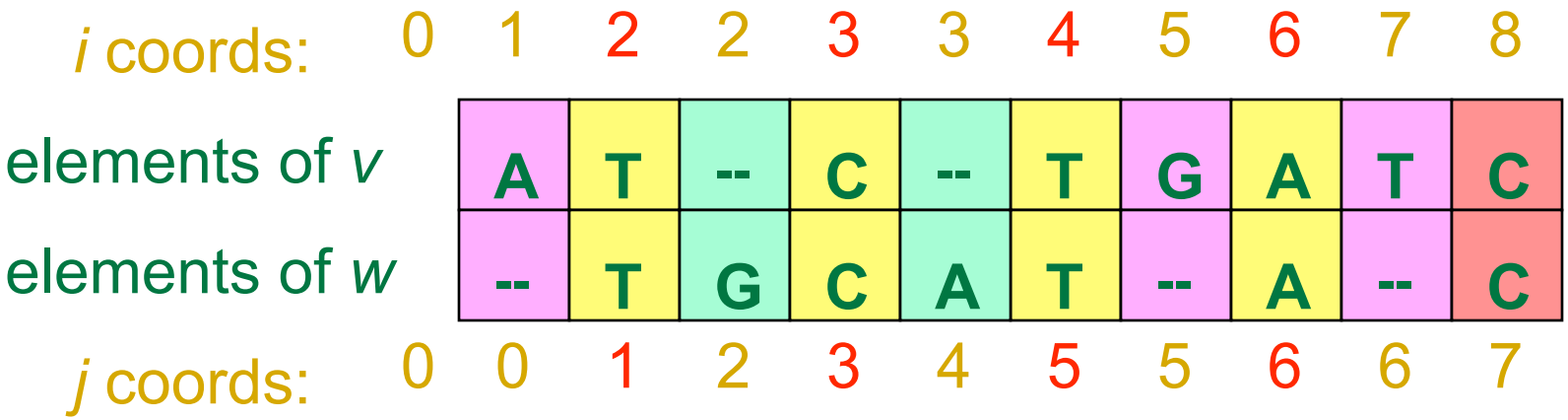
$$\boldsymbol{v}\text{: } 1 \le i_1 < i_2 < \ldots < i_t \le m$$

and a sequence of positions in

$$\boldsymbol{w}\text{: } 1 \le j_1 < j_2 < \ldots < j_t \le n$$

such that i_t *-th* letter *of* ***v*** *equals to* j_t*-letter of* ***w*** and ***t*** is maximal

LCS: Example



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

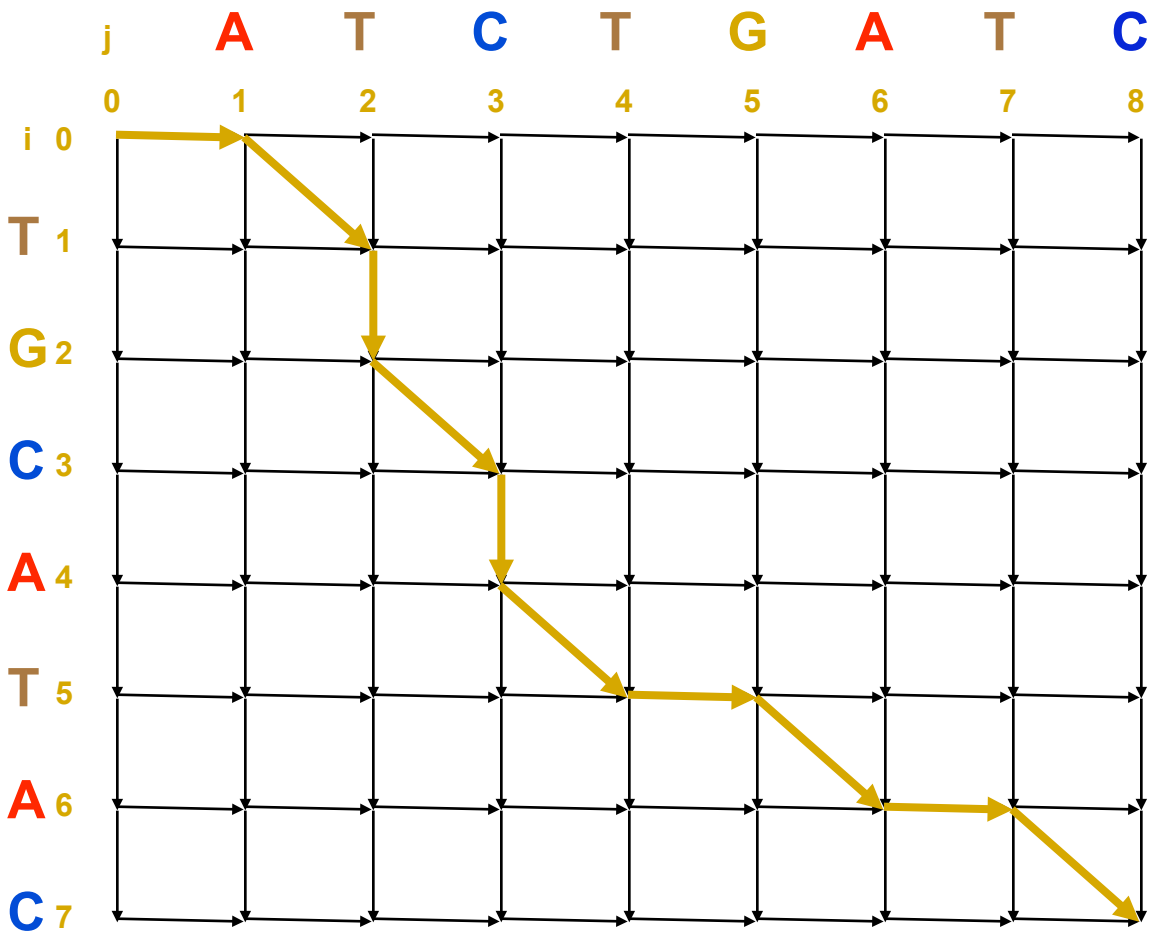
Matches shown in red

positions in *v*: 2 < 3 < 4 < 6 < 8

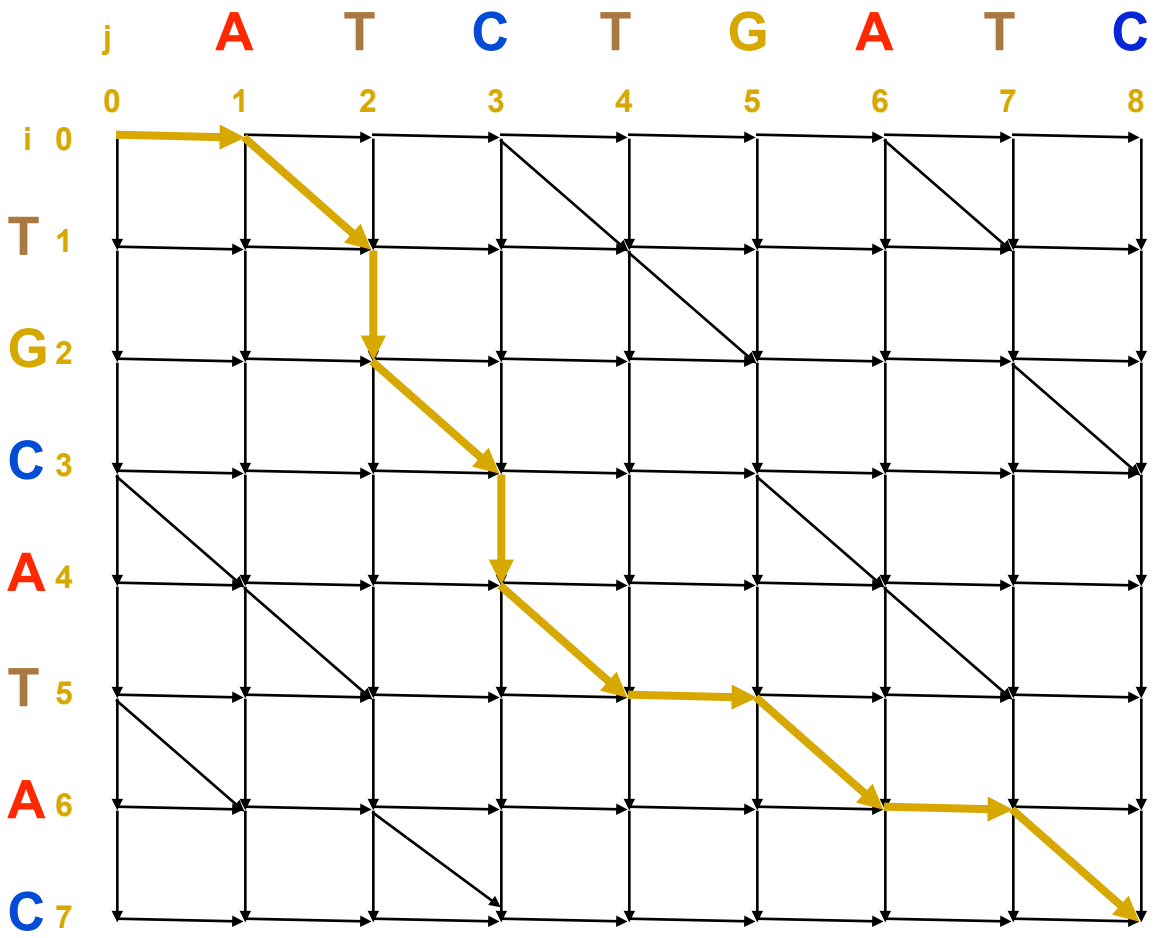
positions in *w*: 1 < 3 < 5 < 6 < 7

Every common subsequence is a path in 2-D grid

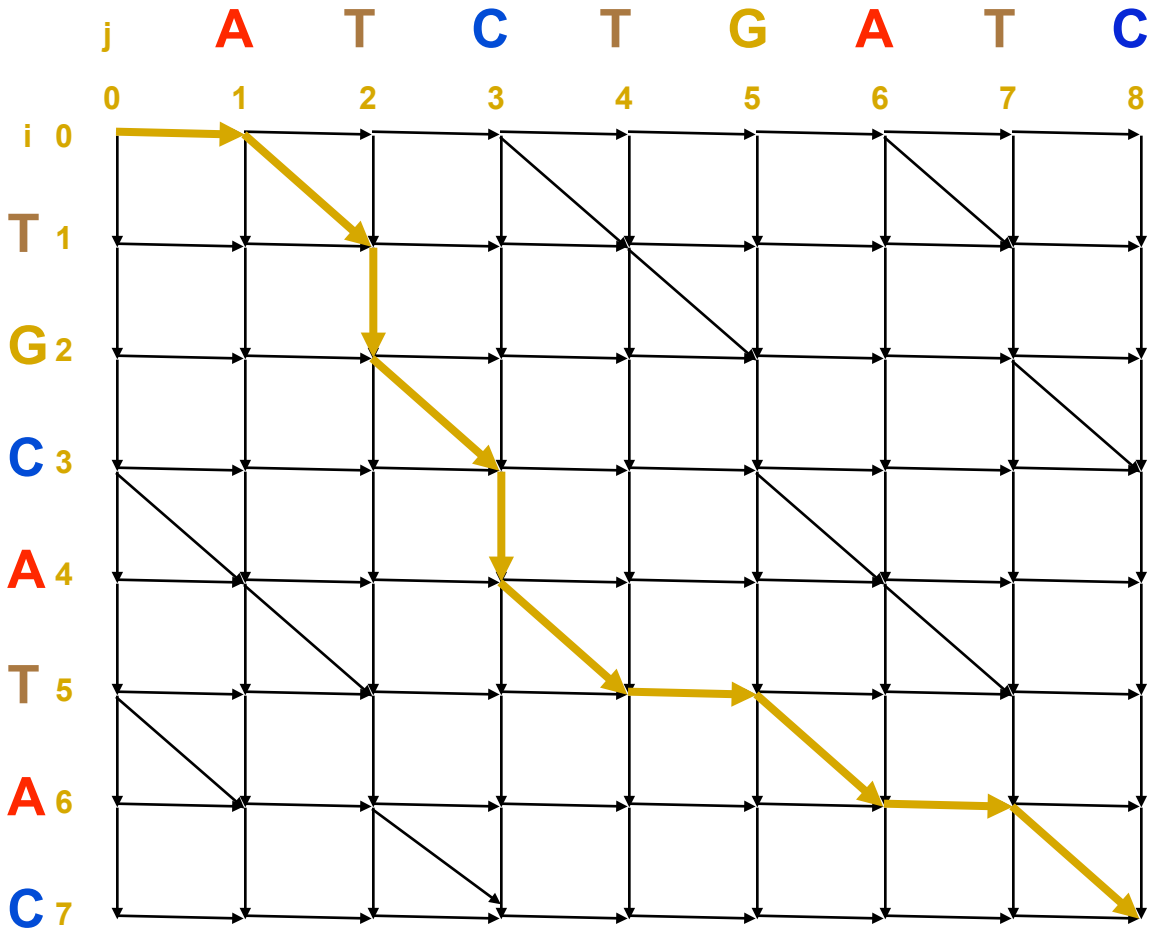
LCS Problem as Manhattan Tourist Problem



Edit Graph for LCS Problem



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 $\boldsymbol{v}_i$ = prefix of $\boldsymbol{v}$ of length i: $v_1 \ldots v_i$

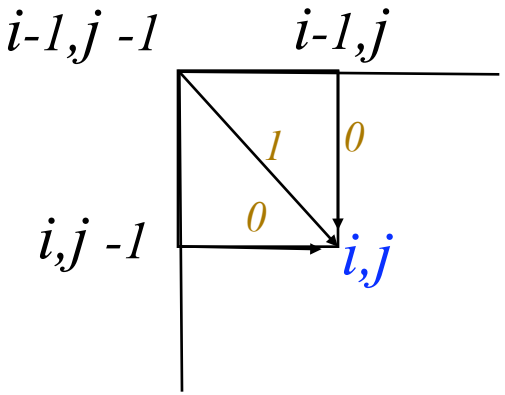
and $\boldsymbol{w}_j$ = prefix of $\boldsymbol{w}$ of length j: $w_1 \ldots w_j$

The length of LCS($\boldsymbol{v}_i$,$\boldsymbol{w}_j$) is computed by:

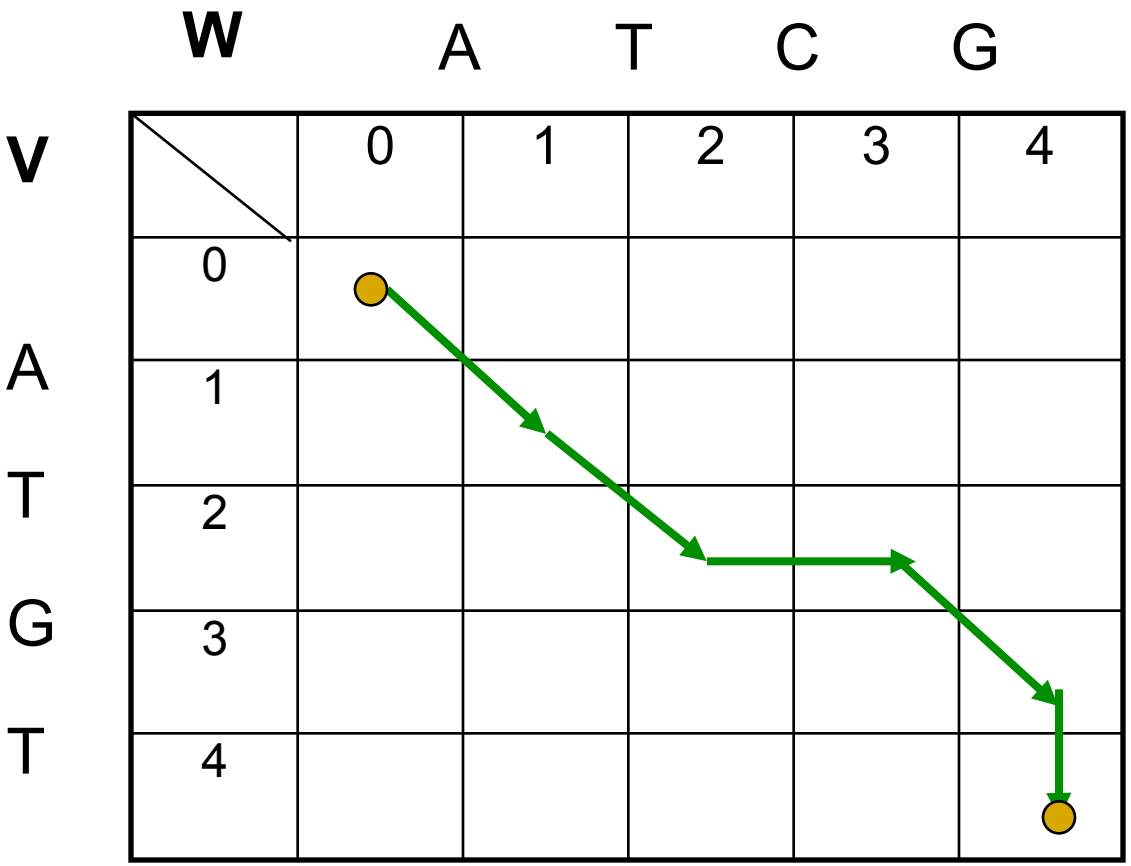
$$s_{i,j} = \max \begin{cases} s_{i-1,\,j} \\ s_{i,\,j-1} \\ s_{i-1,\,j-1} + 1 \text{ if } v_i = w_j \end{cases}$$

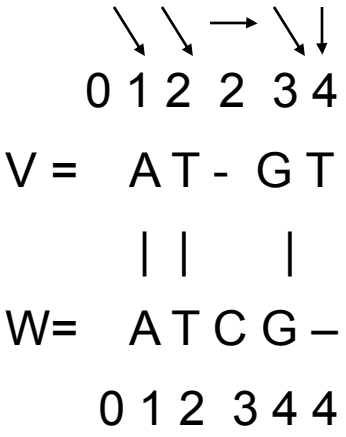
Computing LCS (cont'd)

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



Every Path in the Grid Corresponds to an Alignment





Aligning Sequences without Insertions and Deletions: Hamming Distance

Given two DNA sequences ***v*** and ***w*** :

v : **A T A T A T A T**
w : **T A T A T A T A**

- The Hamming distance: $d_H(\boldsymbol{v}, \boldsymbol{w}) = 8$ is large but the sequences are very similar

Aligning Sequences with Insertions and Deletions

By shifting one sequence over one position:

```
v : ATATATAT--
w : --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

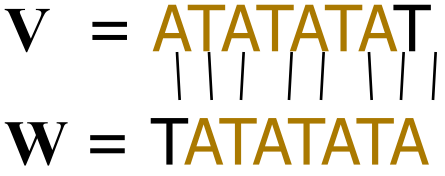
$d(\mathbf{v},\mathbf{w})$ = MIN number of elementary operations to transform **v** à **w**

Edit Distance vs Hamming Distance

Hamming distance always compares

i-th letter of **v** with

i-th letter of **w**



Hamming distance:

$d(\mathbf{v}, \mathbf{w})=8$

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 = ATATATAT

w = TATATATA

Just one shift

Make it all line up

Edit distance may compare

i-th letter of **v** with

j-th letter of **w**

v = - ATATATAT

w = TATATATA

Hamming distance:

$d(\mathbf{v}, \mathbf{w})=8$

Computing Hamming distance is a trivial task

Edit distance:

$d(\mathbf{v}, \mathbf{w})=2$

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 = ATATATAT
    | | | | | | |
w = TATATATA
```

Hamming distance:

$d(\mathbf{v}, \mathbf{w})=8$

Edit distance may compare

i-th letter of **v** with

j-th letter of **w**

```
v = - ATATATAT
      | | | | | | |
w = TATATATA
```

Edit distance:

$d(\mathbf{v}, \mathbf{w})=2$

(one insertion and one deletion)

How to find what *j* goes with what *i* ???

Edit Distance: Example

TGCATAT à ATCCGAT in 5 steps

TGCATAT à (delete last T)
TGCATA à (delete last A)
TGCAT à (insert A at front)
ATGCAT à (substitute C for 3rd G)
ATCCAT à (insert G before last A)
ATCCGAT (Done)

Edit Distance: Example

TGCATAT à ATCCGAT in 5 steps

TGCATAT à (delete last T)
TGCATA à (delete last A)
TGCAT à (insert A at front)
ATGCAT à (substitute C for 3rd G)
ATCCAT à (insert G before last A)
ATCCGAT (Done)

What is the edit distance? 5?

Edit Distance: Example (cont'd)

TGCATAT à ATCCGAT in 4 steps

TGCATAT à (insert A at front)

ATGCATAT à (delete 6th T)

ATGCATA à (substitute G for 5th A)

ATGCGTA à (substitute C for 3rd G)

ATCCGAT (Done)

Edit Distance: Example (cont'd)

TGCATAT à ATCCGAT in 4 steps

TGCATAT à (insert A at front)

ATGCATAT à (delete 6th T)

ATGCATA à (substitute G for 5th A)

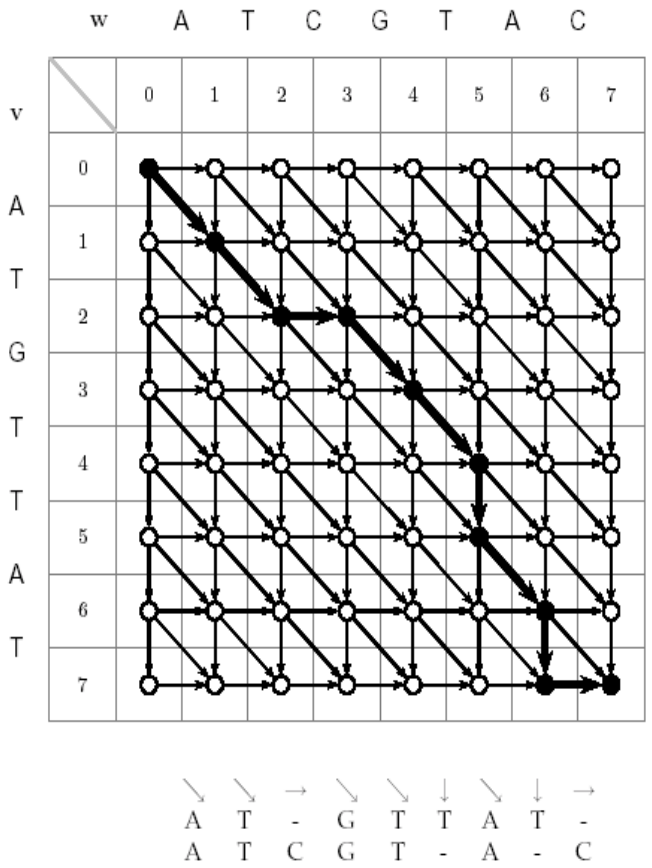
ATGCGTA à (substitute C for 3rd G)

ATCCGAT (Done)

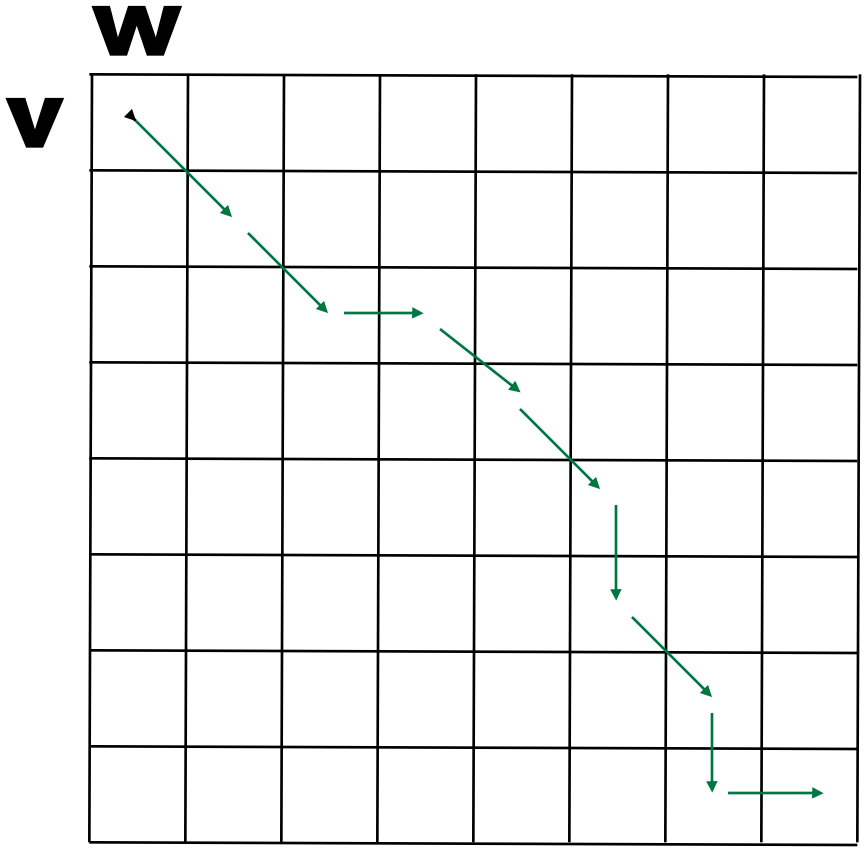
Can it be done in 3 steps???

The Alignment Grid

- Every alignment path is from source to sink



Alignment as a Path in the Edit Graph

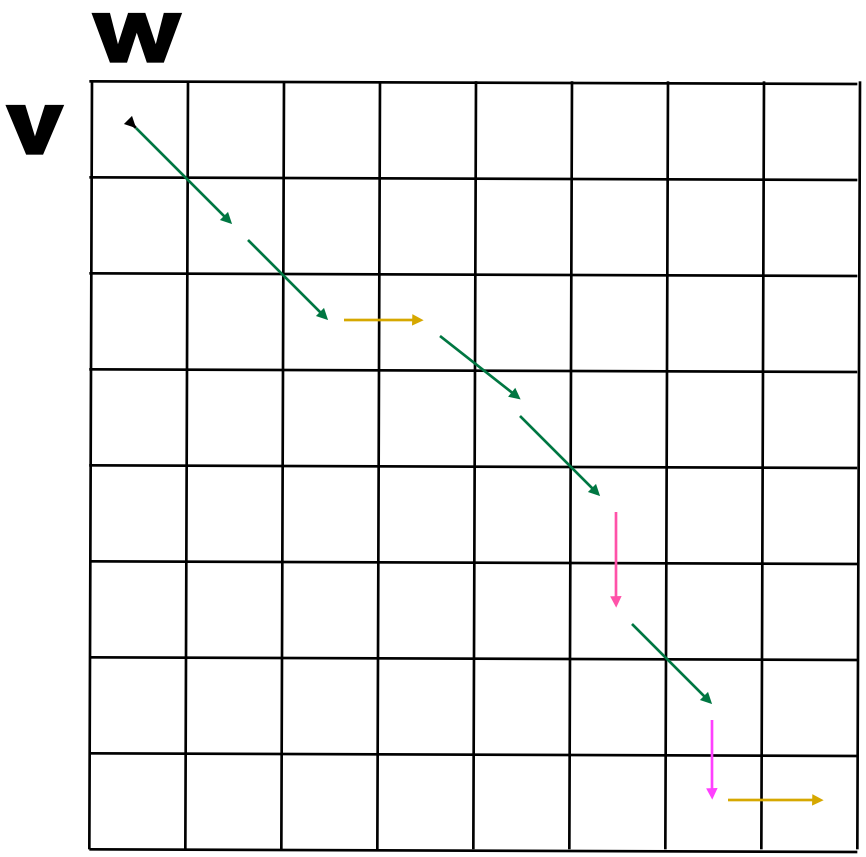


0 1 2 2 3 4 5 6 7 7
A T _ G T T A T _
A T C G T _ A _ C
0 1 2 3 4 5 5 6 6 7

- 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)

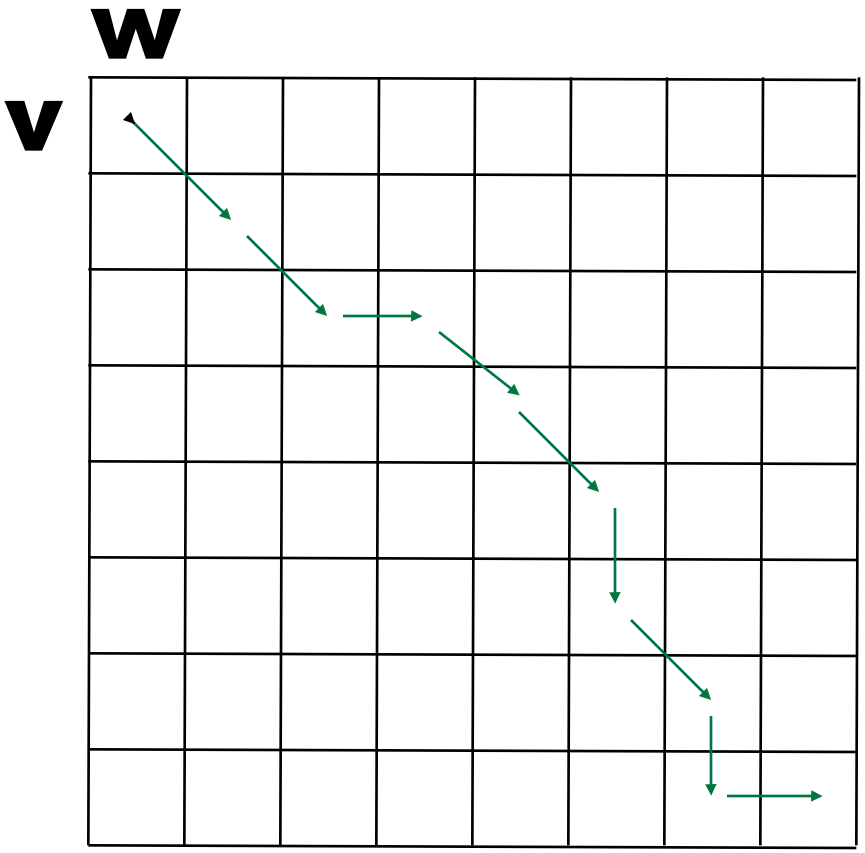


↓ and → represent indels in **v** and **w** with score 0.

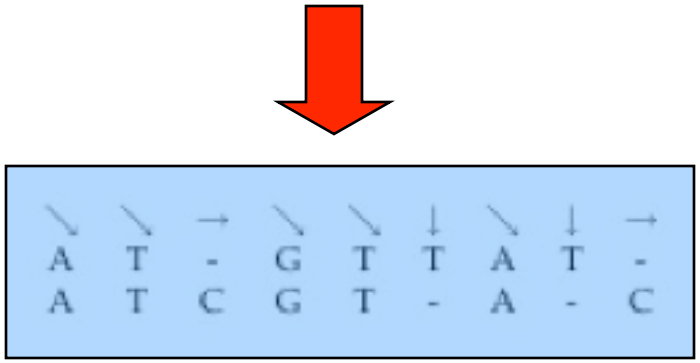
↘ represent matches with score 1.

- The score of the alignment path is 5.

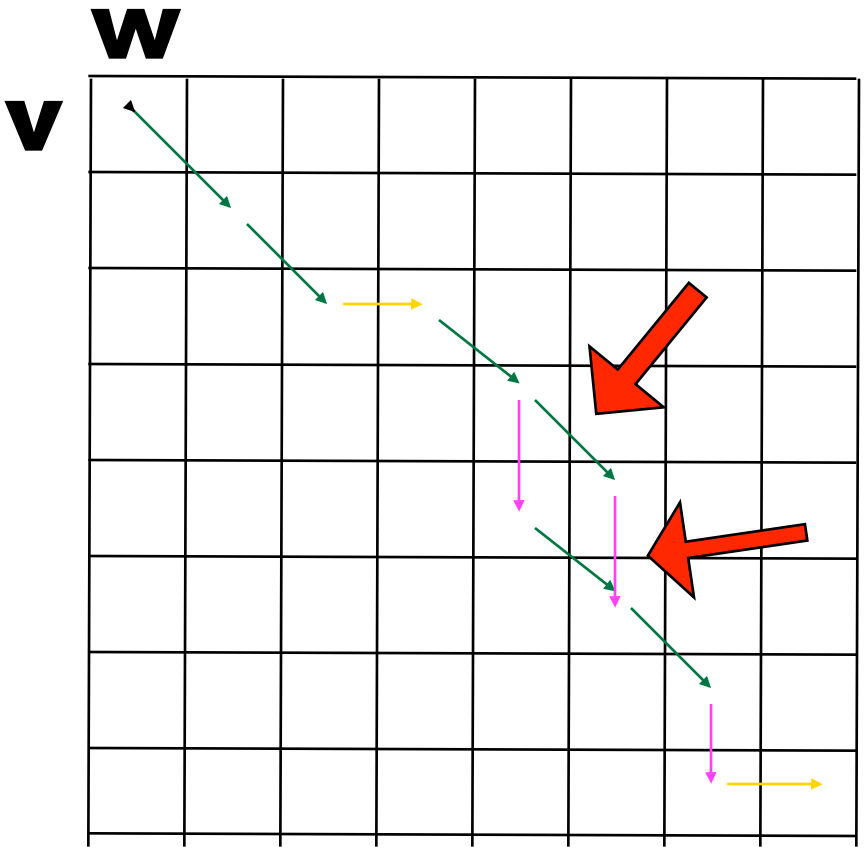
Alignment as a Path in the Edit Graph



Every path in the edit graph corresponds to an alignment:



Alignment as a Path in the Edit Graph



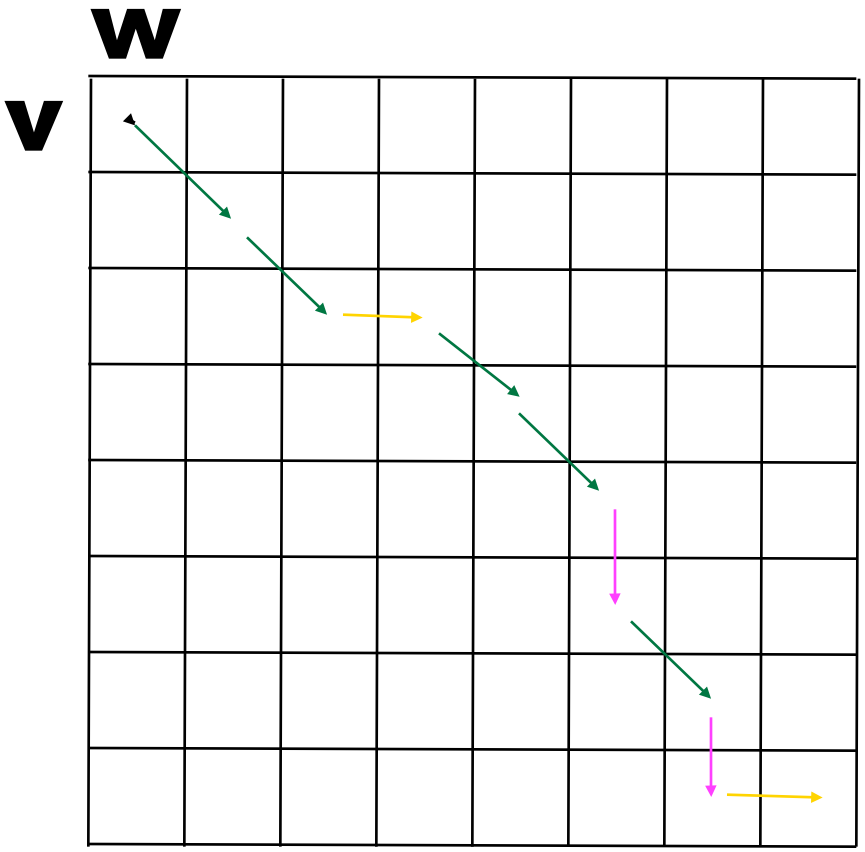
Old Alignment

0122345677
v= AT_GTTAT_
w= ATCGT_A_C
0123455667

New Alignment

0122345677
v= AT_GTTAT_
w= ATCG_TA_C
0123455667

Alignment as a Path in the Edit Graph



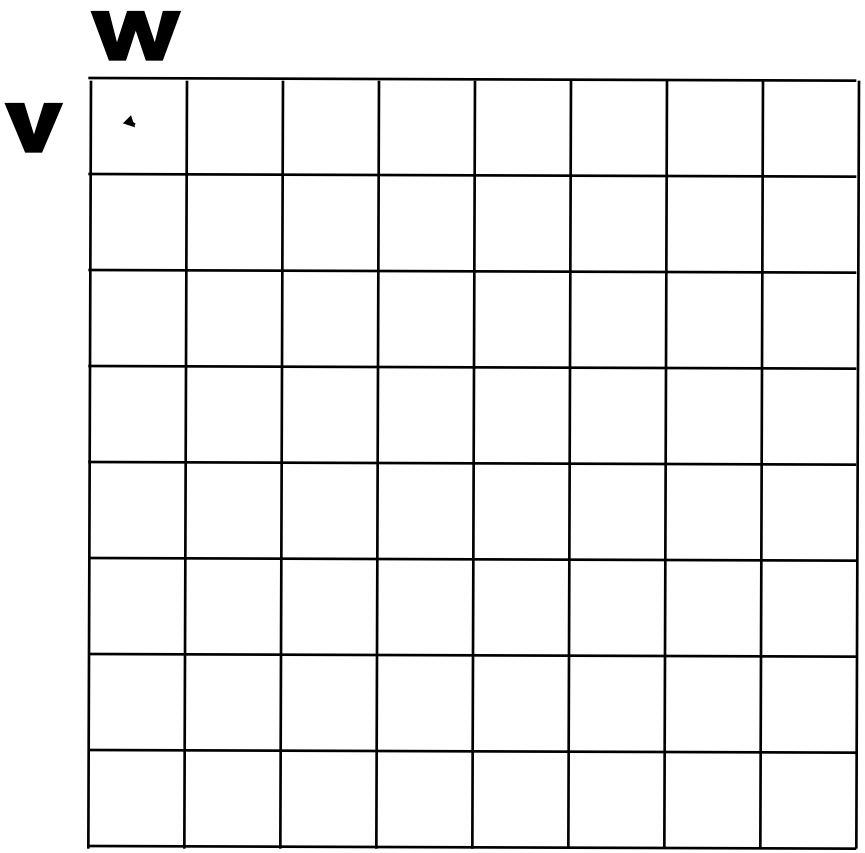
0122345677
v= AT_GTTAT_
w= ATCGT_A_C
0123455667

(0,0) , (1,1) , (2,2), (2,3), (3,4), (4,5), (5,5), (6,6), (7,6), (7,7)

Alignment: Dynamic Programming

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

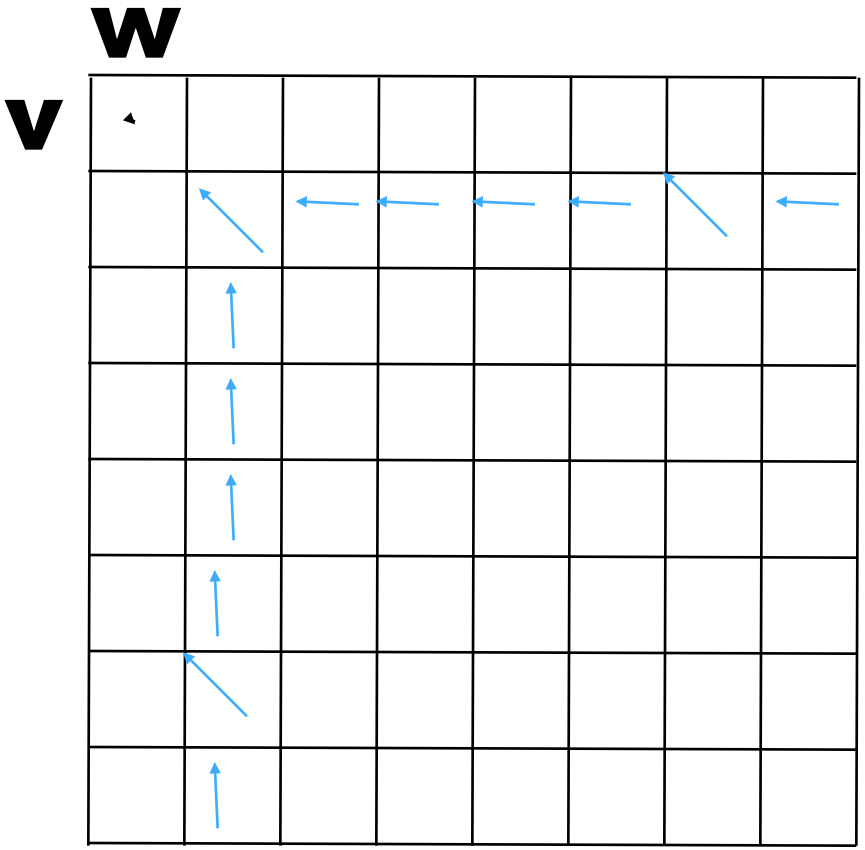
Dynamic Programming Example



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

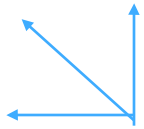
Or, to be more precise, initialize 0^{th} row and *0^{th} column to be all zeroes.*

Dynamic Programming Example



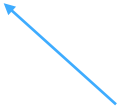
$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} & \text{value from NW +1, if } v_i = w_j \\ s_{i-1,j} & \text{value from North (top)} \\ s_{i,j-1} & \text{value from West (left)} \end{cases}$$

Alignment: Backtracking

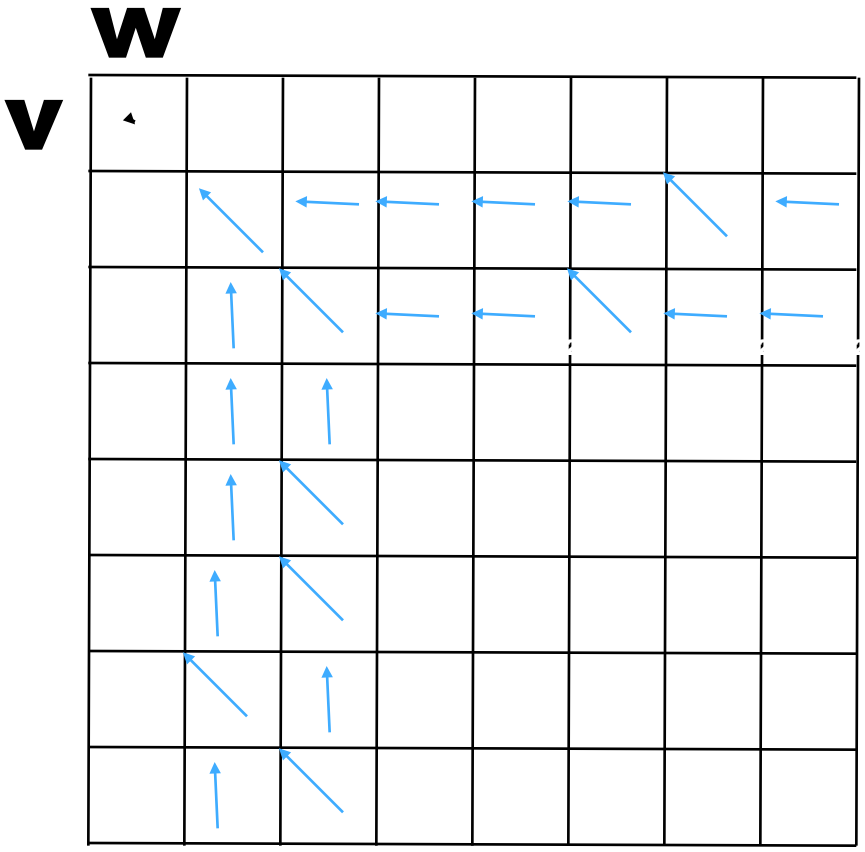
Arrows  show where the score originated from.

 if from the top

 if from the left

 if $v_i = w_j$

Backtracking Example



Find a match in row and column 2.

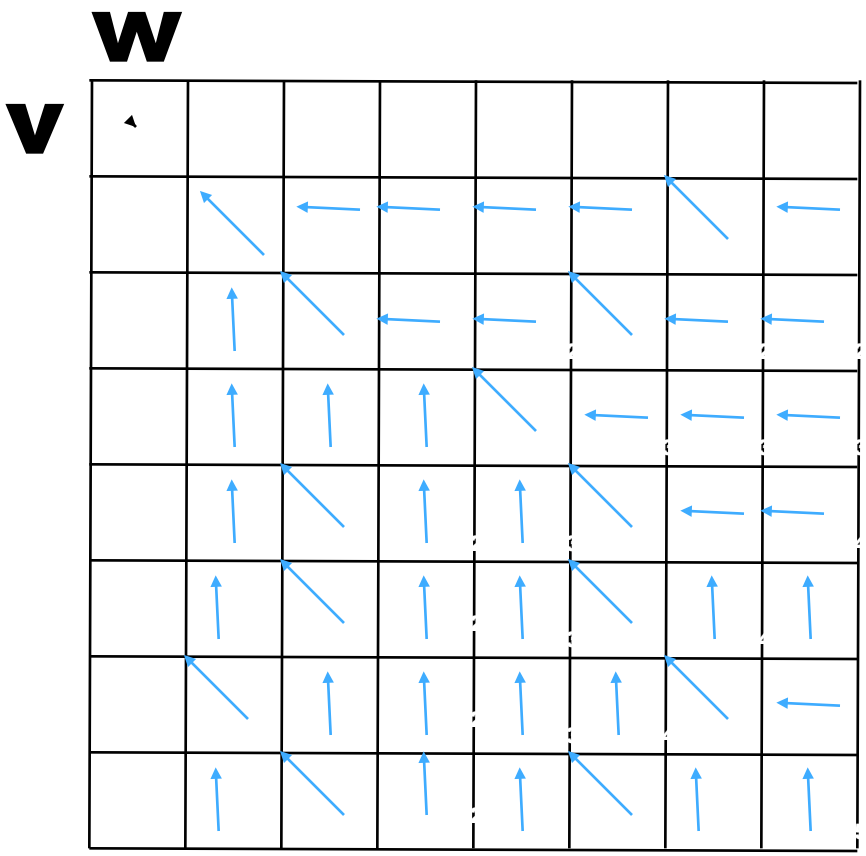
i=2, j=2,5 is a match (T).

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

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

Alignment: Dynamic Programming

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

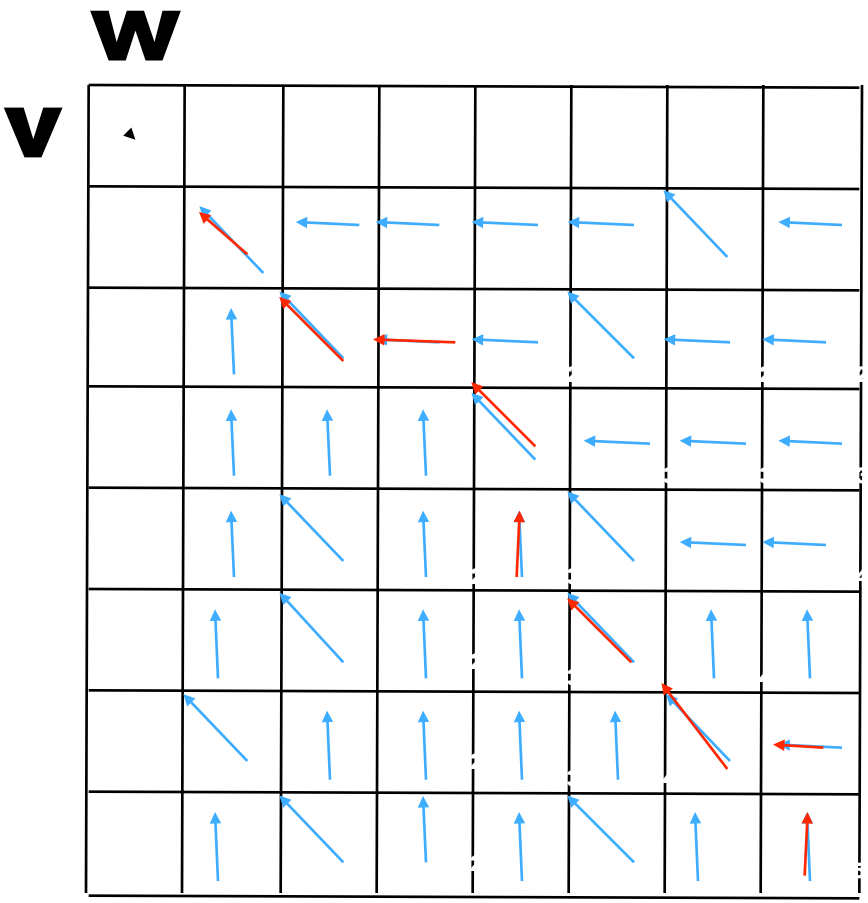
This recurrence corresponds to the Manhattan Tourist problem (three incoming edges into a vertex) with all horizontal and vertical edges weighted by zero.

LCS Algorithm

1. LCS(v,w)
2. for i ß 1 to n
3. $s_{i,0}$ ß 0
4. for j ß 1 to m
5. $s_{0,j}$ ß 0
6. for i ß 1 to n
7. for j ß 1 to m
8. $s_{i-1,j}$ {
9. $s_{i,j}$ ß max $s_{i,j-1}$
10. $s_{i-1,j-1} + 1$, if $v_i = w_j$
11. " " if $s_{i,j} = s_{i-1,j}$
- $b_{i,j}$ ß " " if $s_{i,j} = s_{i,j-1}$
- " " if $s_{i,j} = s_{i-1,j-1} + 1$
- 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



Printing LCS: Backtracking

```
1.  PrintLCS(b,v,i,j)
2.      if  i = 0 or j = 0
3.          return
4.      if b_{i,j} = “ ↖ “
5.          PrintLCS(b,v,i-1,j-1)
6.          print v_i
7.      else
8.          if b_{i,j} = “ ↑ “
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 *nxm* dynamic programming matrix.

- Why O(*nm*)? The pseudocode consists of a nested “for” loop inside of another “for” loop to set up a *nxm* matrix.