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

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

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

Change Problem: Example

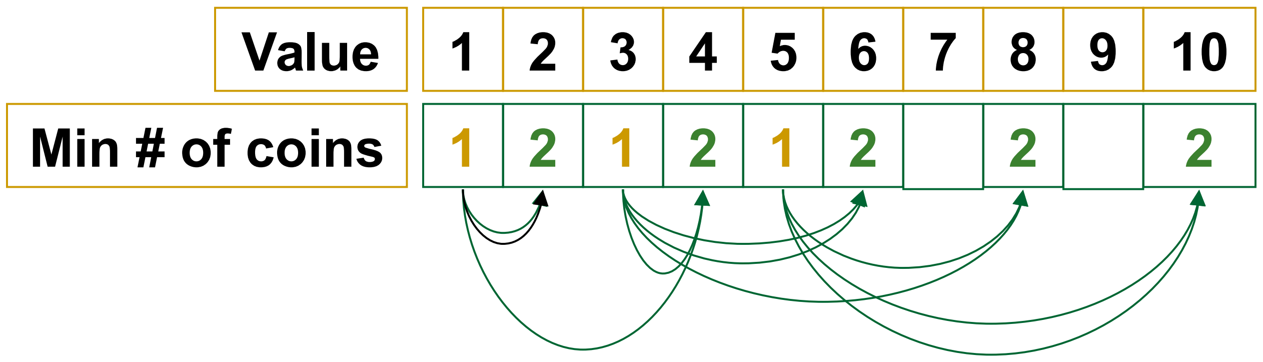
Given the denominations 1, 3, and 5, what is the minimum number of coins needed to make change for a given value?

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?



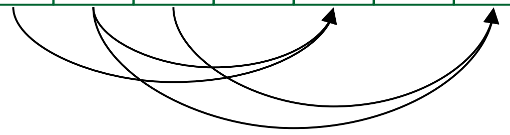
Value	1	2	3	4	5	6	7	8	9	10
Min # of coins	1	2	1	2	1	2		2		2

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?

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



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:

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

Change Problem: Recurrence (cont'd)

Given the denominations c: c_1, c_2, …, c_d, the recurrence relation is:

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

Change Problem: A Recursive Algorithm

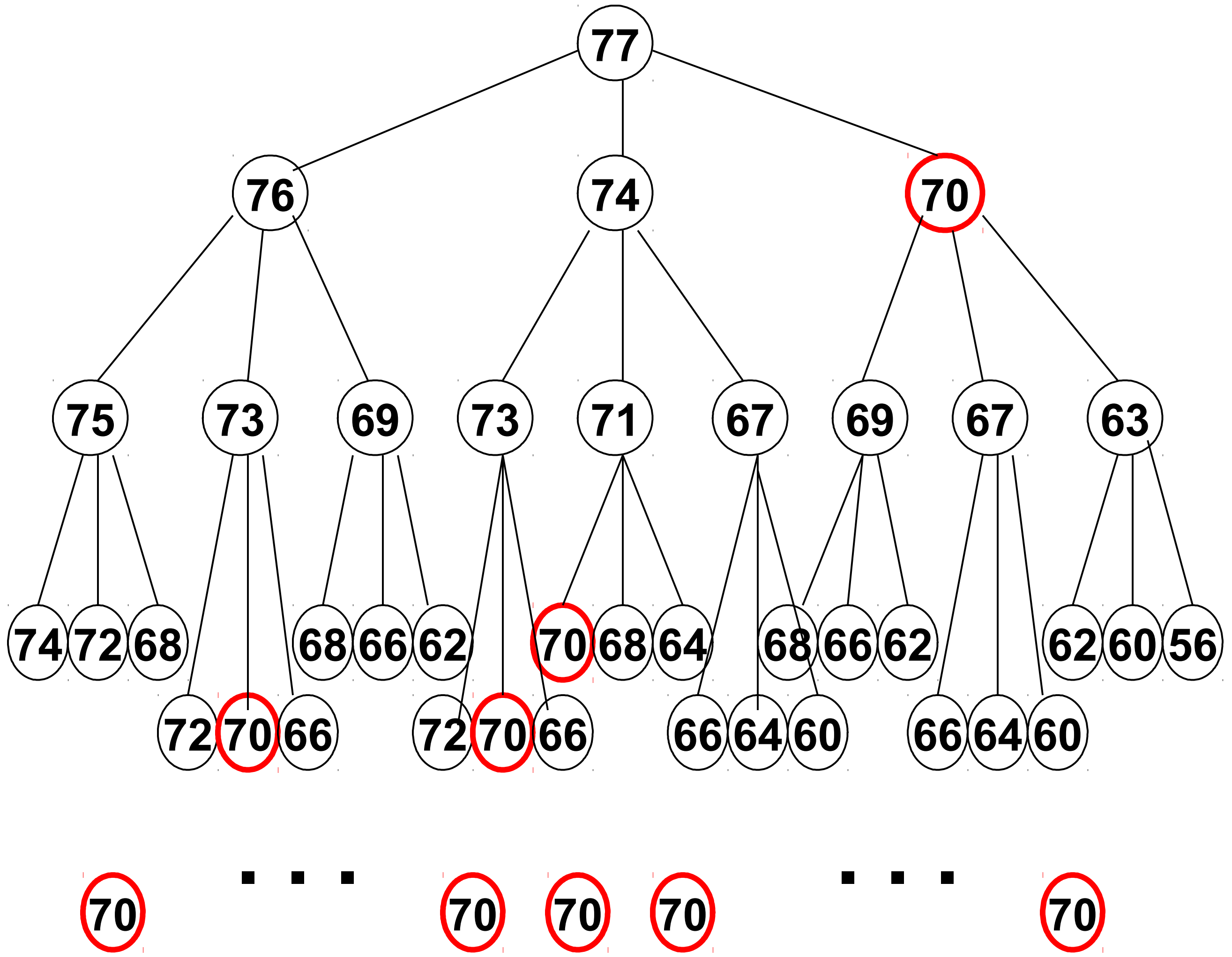
```
1.  RecursiveChange(M,c,d)
2.    if $M = 0$
3.      return 0
4.    bestNumCoins $\leftarrow$ infinity
5.    for $i \leftarrow 1$ to $d$
6.      if $M \geq c_i$
7.        numCoins $\leftarrow$ RecursiveChange($M - c_i$, c, d)
8.        if numCoins + 1 < bestNumCoins
9.          bestNumCoins $\leftarrow$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

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



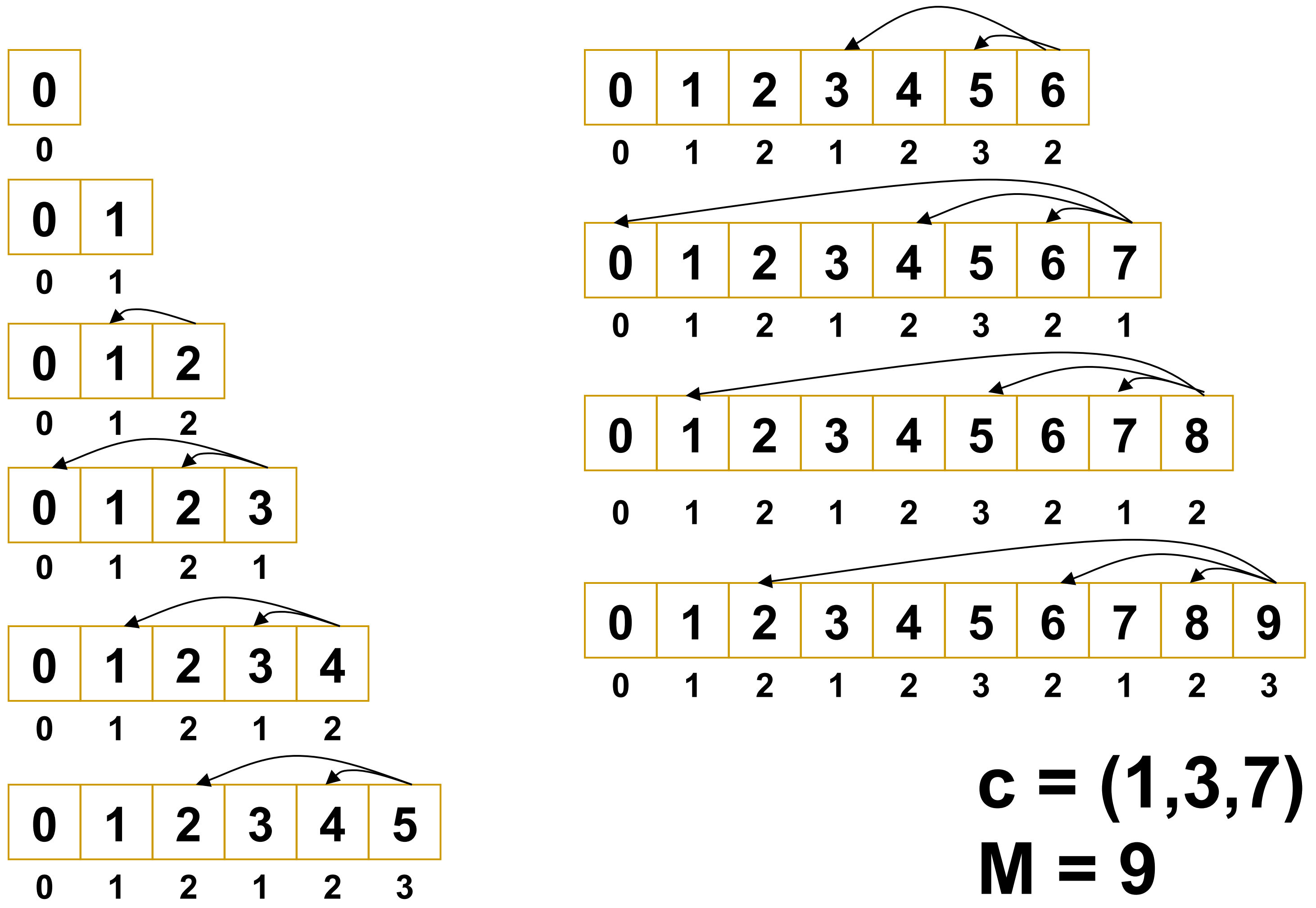
We Can Do Better

- 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

```
1. DPChange(M,c,d)
2.   $bestNumCoins_0 \leftarrow 0$
3.   for m ← 1 to M
4.     $bestNumCoins_m \leftarrow$ infinity
5.     for i ← 1 to d
6.       if $m \geq c_i$
7.         if $bestNumCoins_{m - c_i} + 1 < bestNumCoins_m$
8.           $bestNumCoins_m \leftarrow bestNumCoins_{m - c_i} + 1$
9.   return $bestNumCoins_M$
```

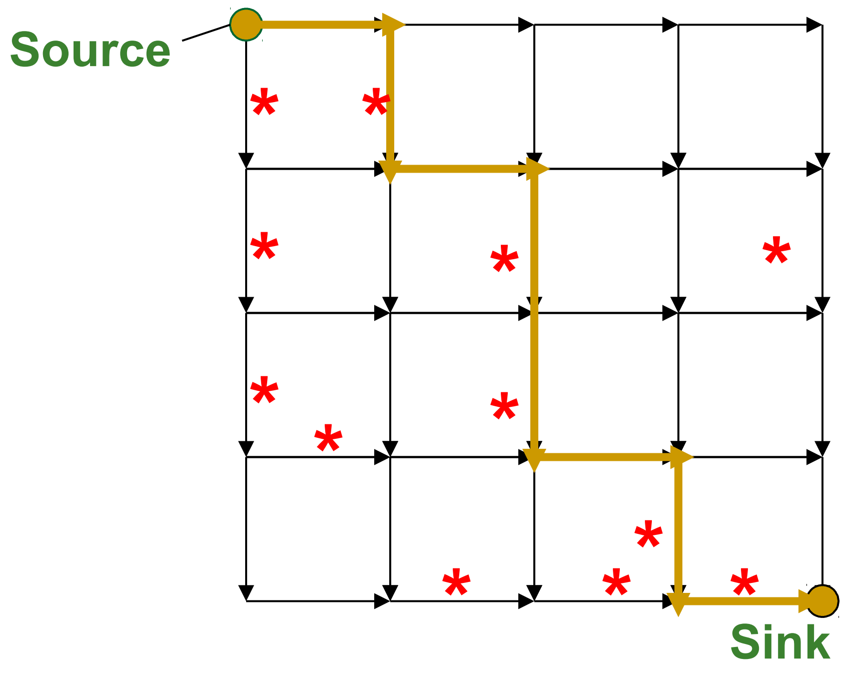
DPChange: Example



c = (1,3,7)
M = 9

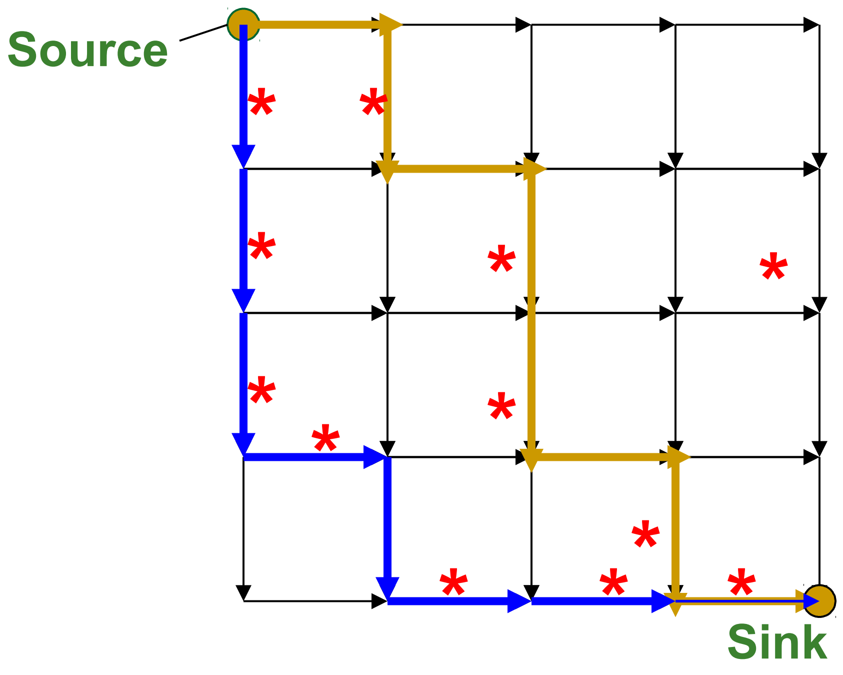
Manhattan Tourist Problem (MTP)

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



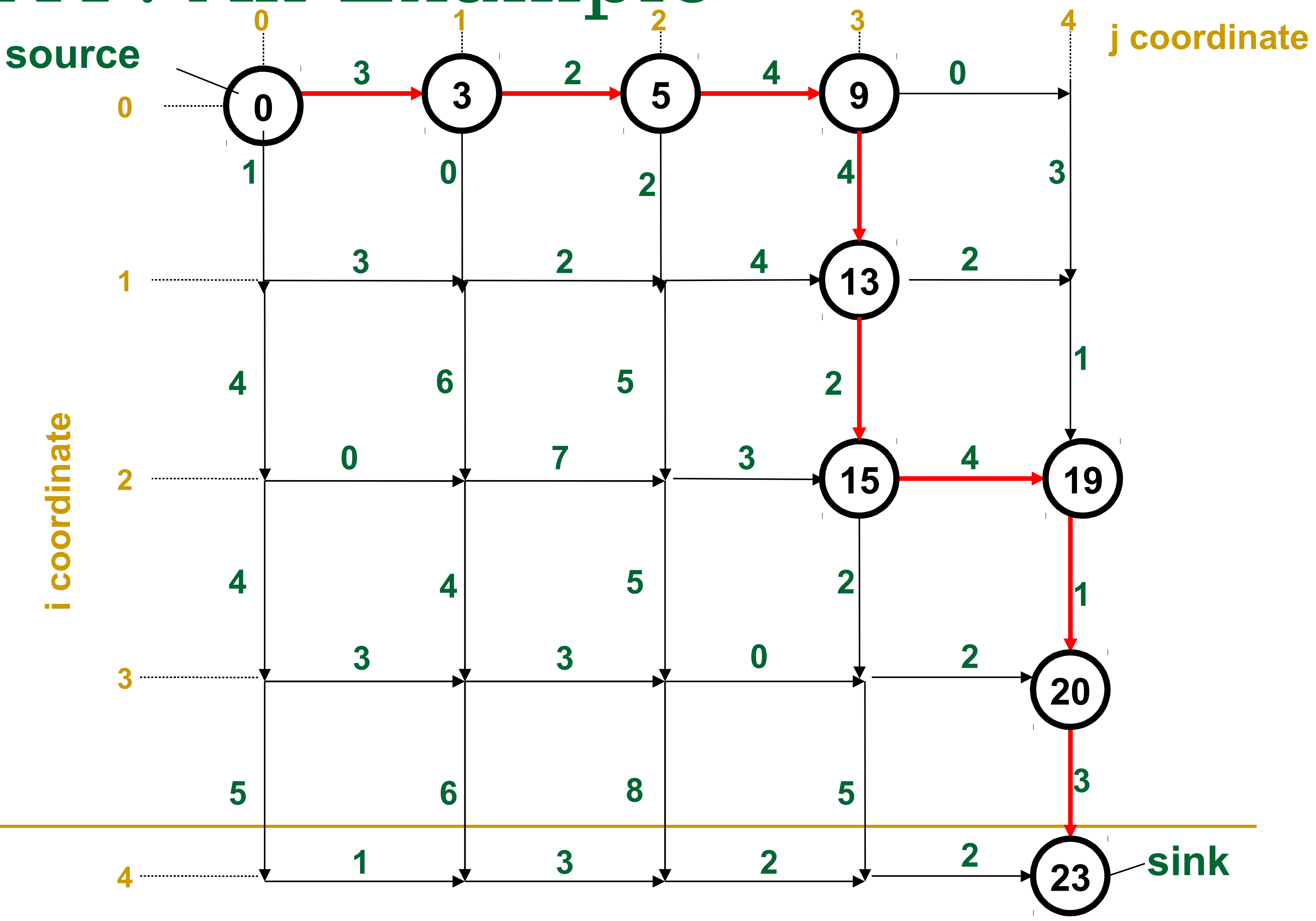
Manhattan Tourist Problem: Formulation

Goal: Find the longest path in a weighted grid.

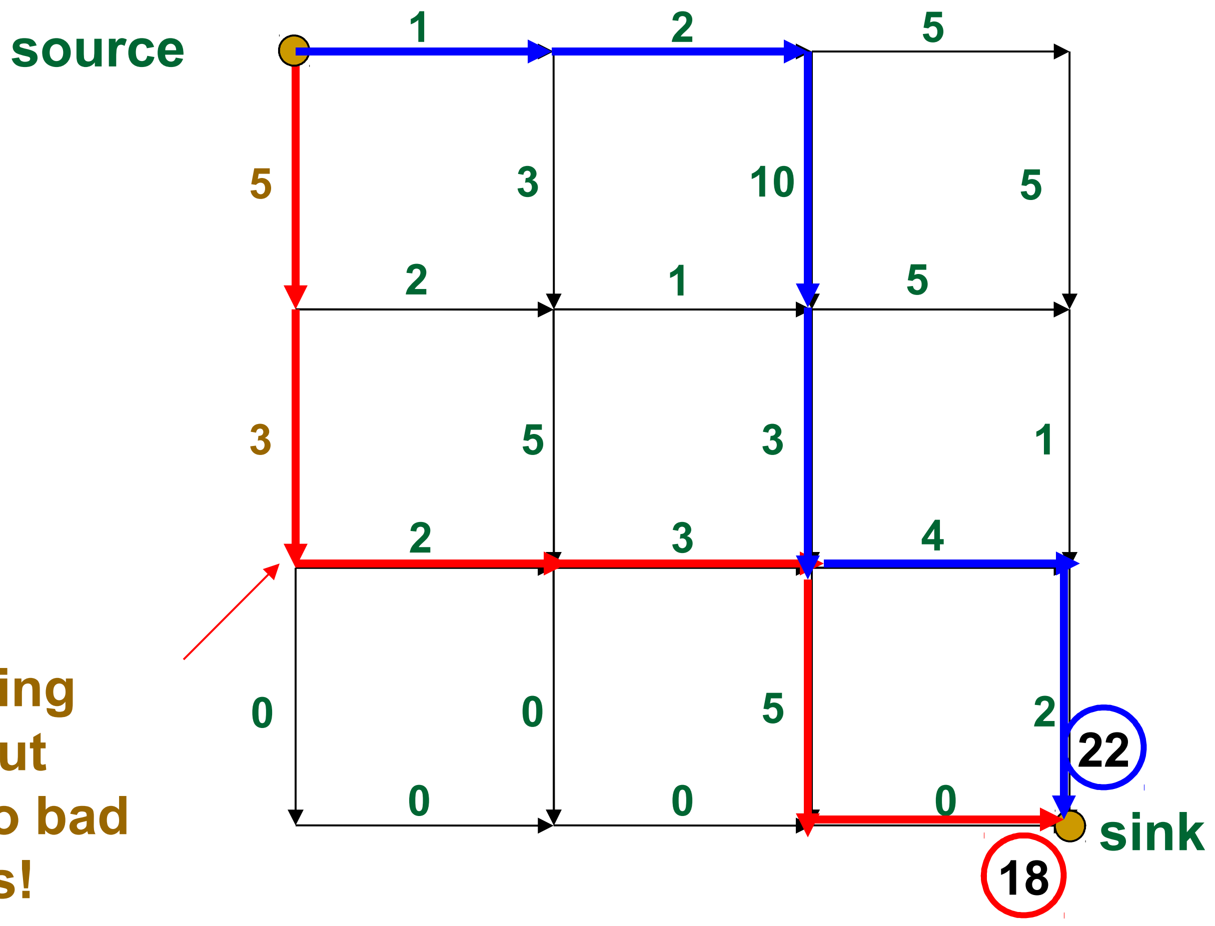
Input: A weighted grid G with two distinct vertices, one labeled “source” and the other labeled “sink”

Output: A longest path in G from “source” to “sink”

MTP: An Example



MTP: Greedy Algorithm Is Not Optimal

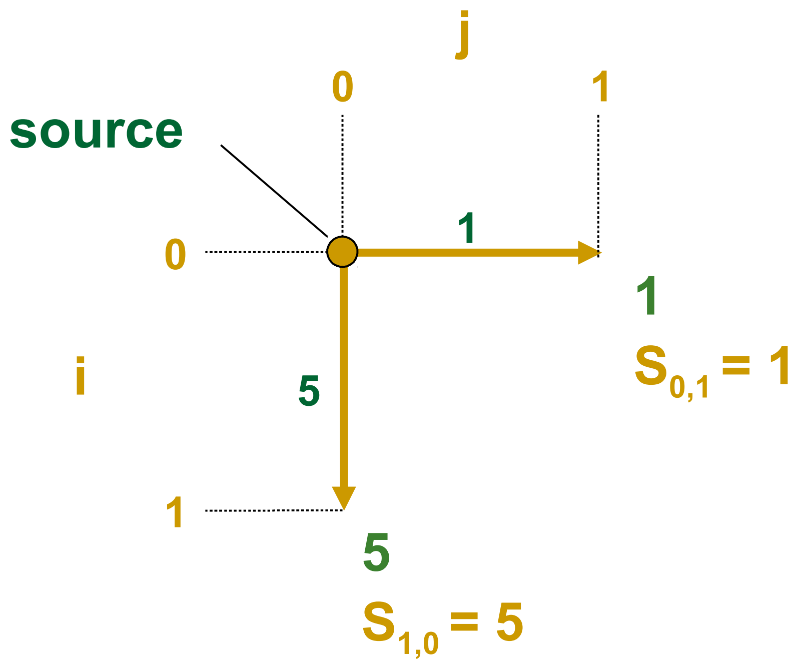


promising start, but leads to bad choices!

MTP: Simple Recursive Program

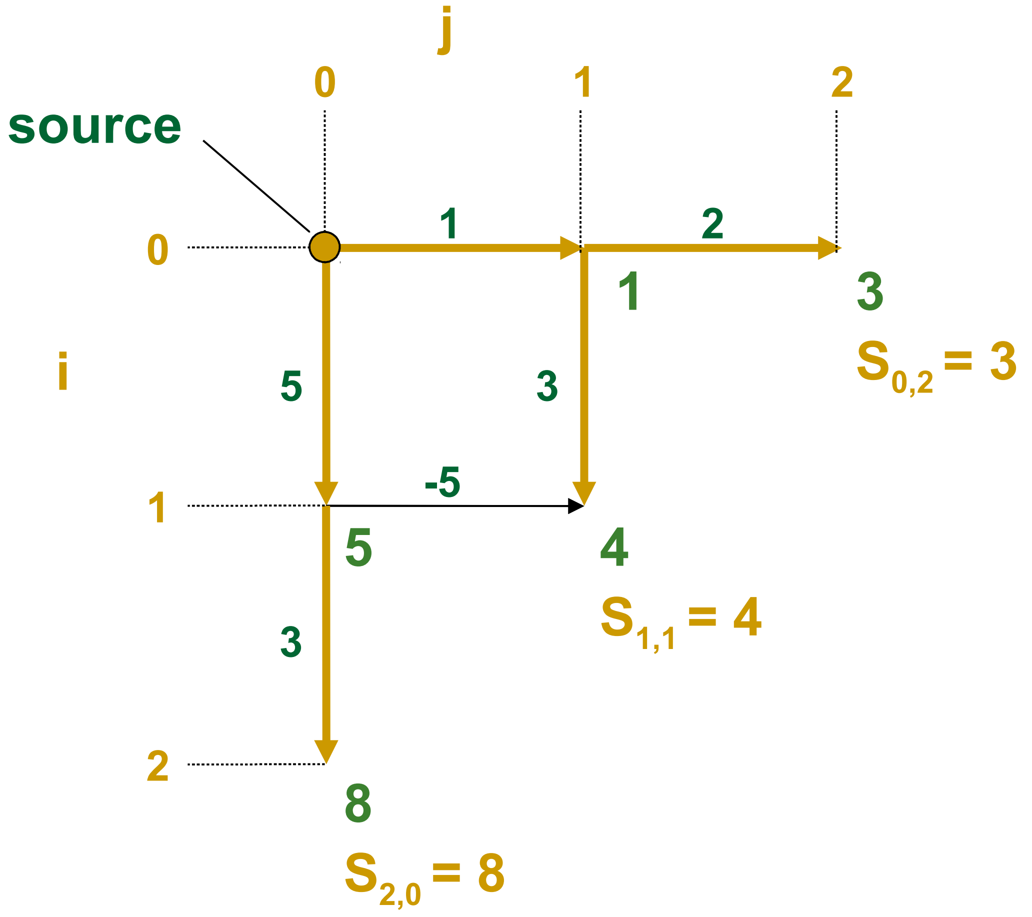
```
MT(n,m)
  if n=0  or m=0
    return MT(n,m)
  x ≋ MT(n-1,m)+
          length of the edge from (n- 1,m) to (n,m)
  y ≋ MT(n,m-1)+
          length of the edge from (n,m-1) to (n,m)
  return max{x,y}
```

MTP: Dynamic Programming

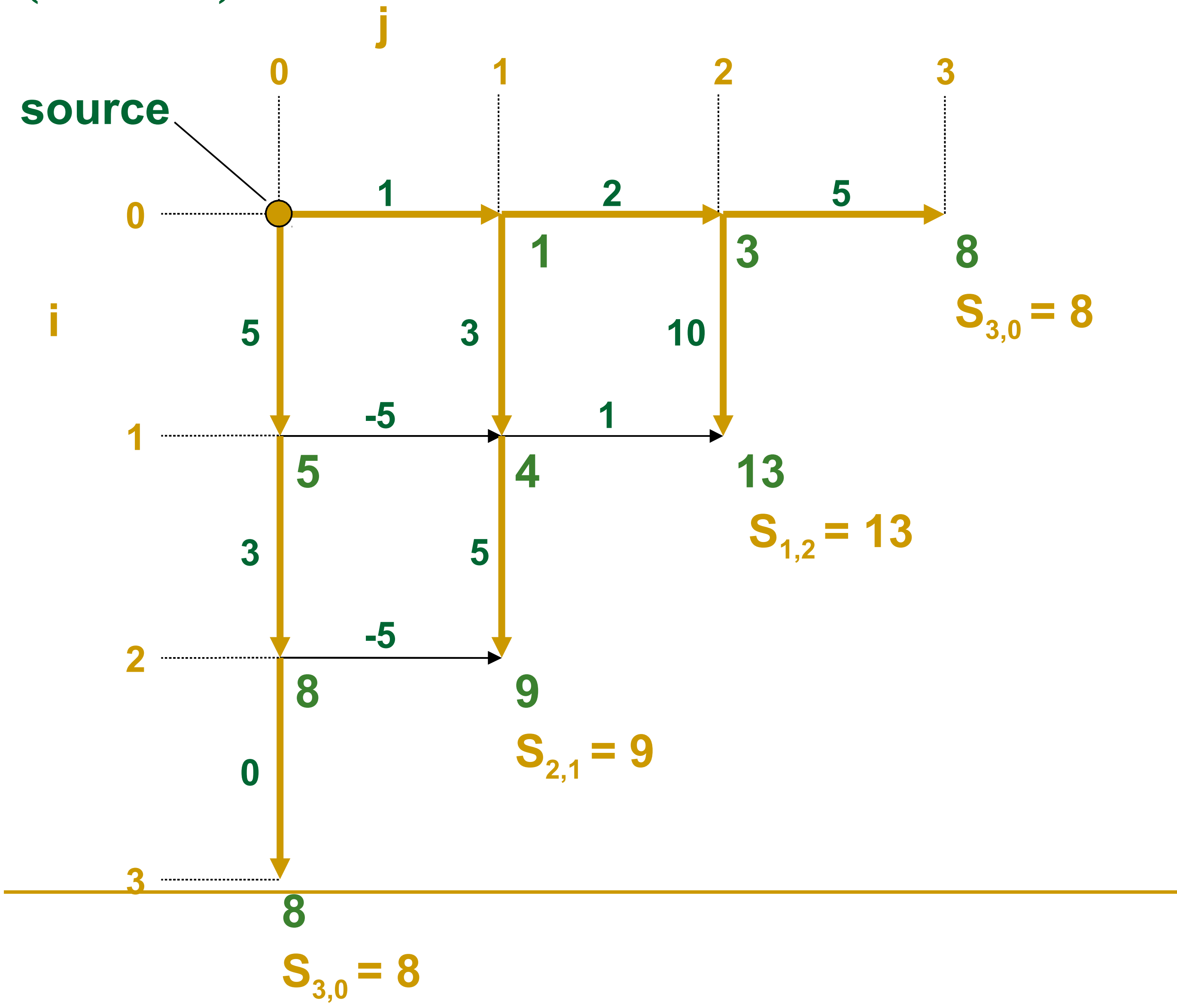


- Calculate optimal path score for each vertex in the graph
- Each vertex's score is the maximum of the prior vertices score plus the weight of the respective edge in between

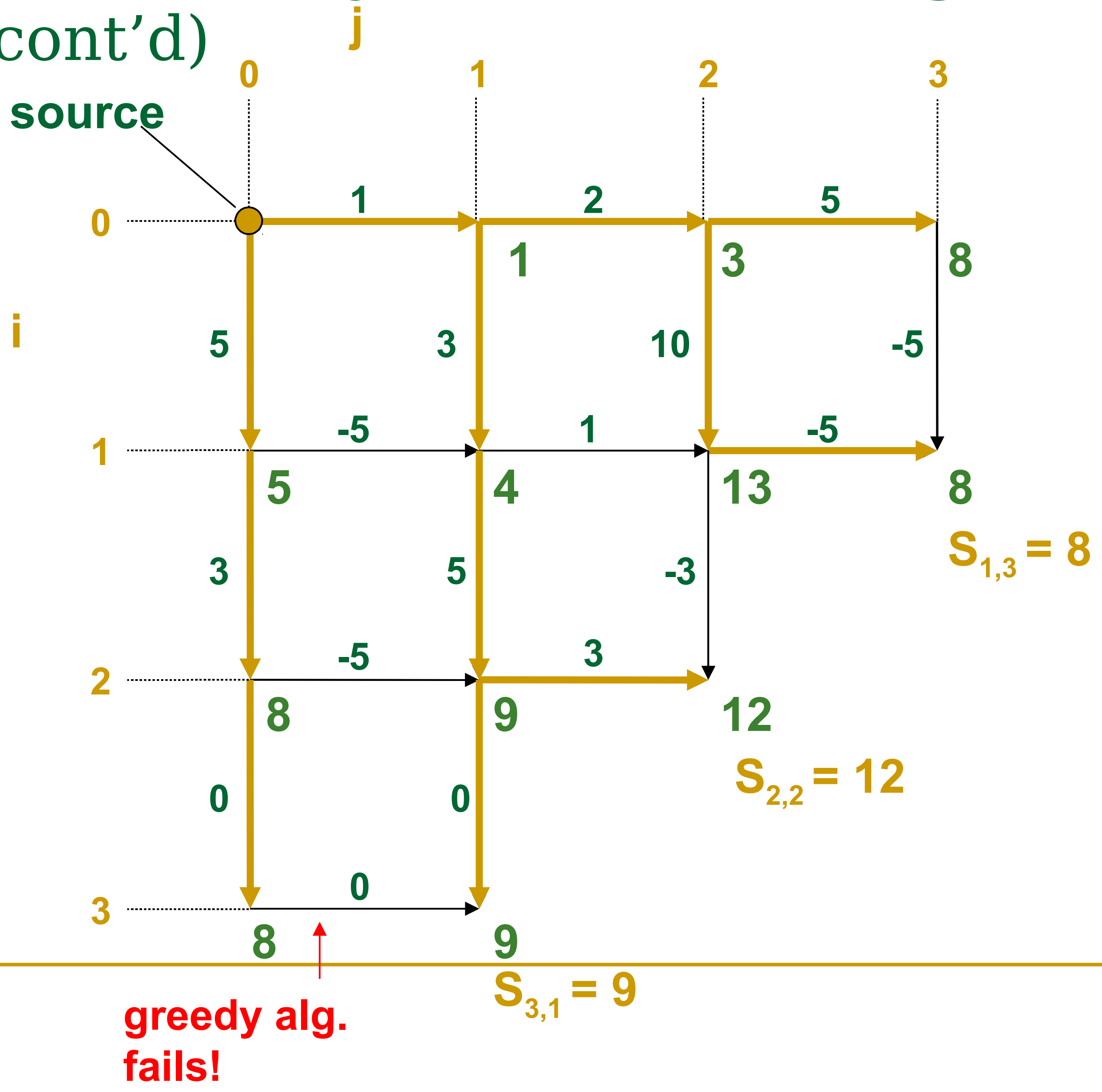
MTP: Dynamic Programming (cont'd)



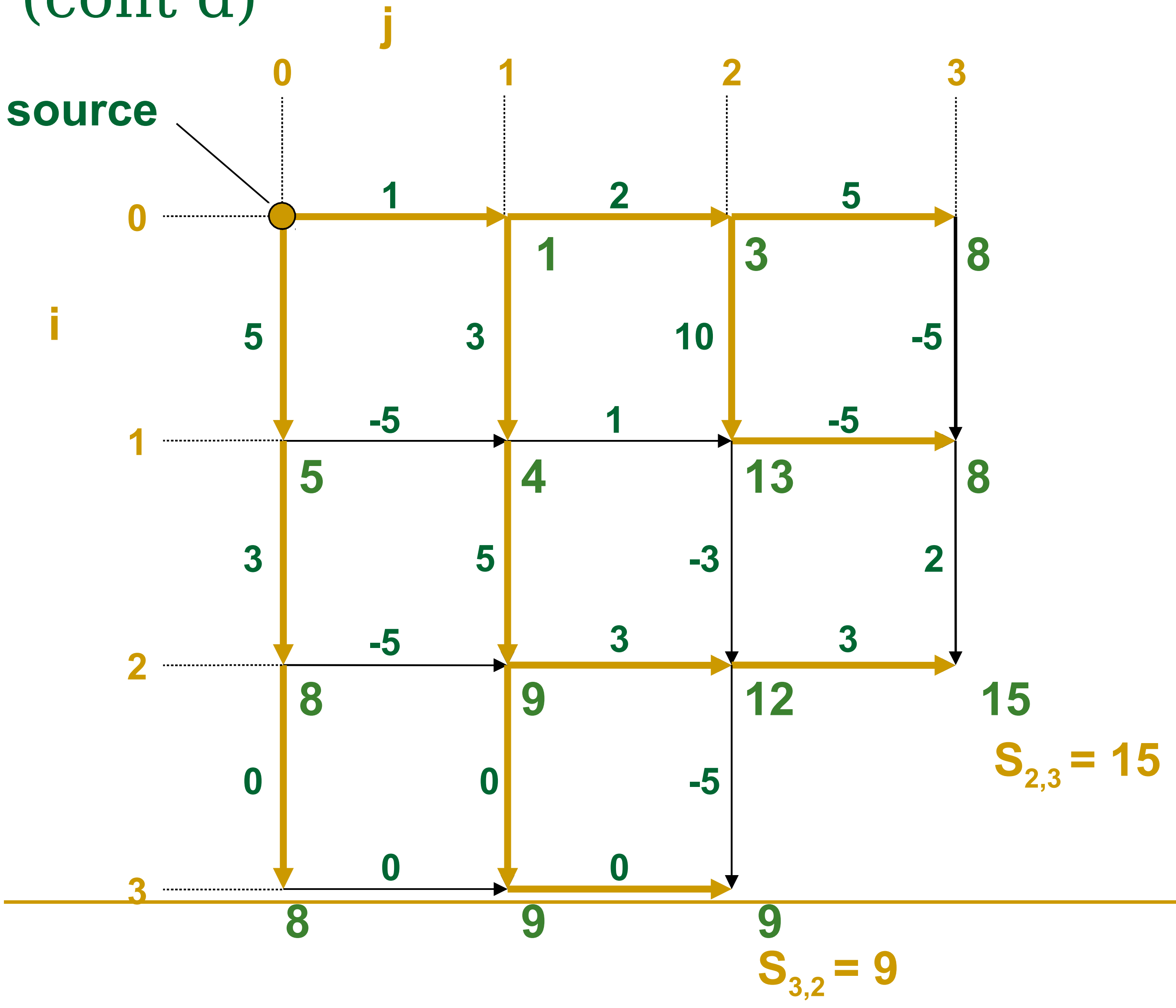
MTP: Dynamic Programming (cont'd)



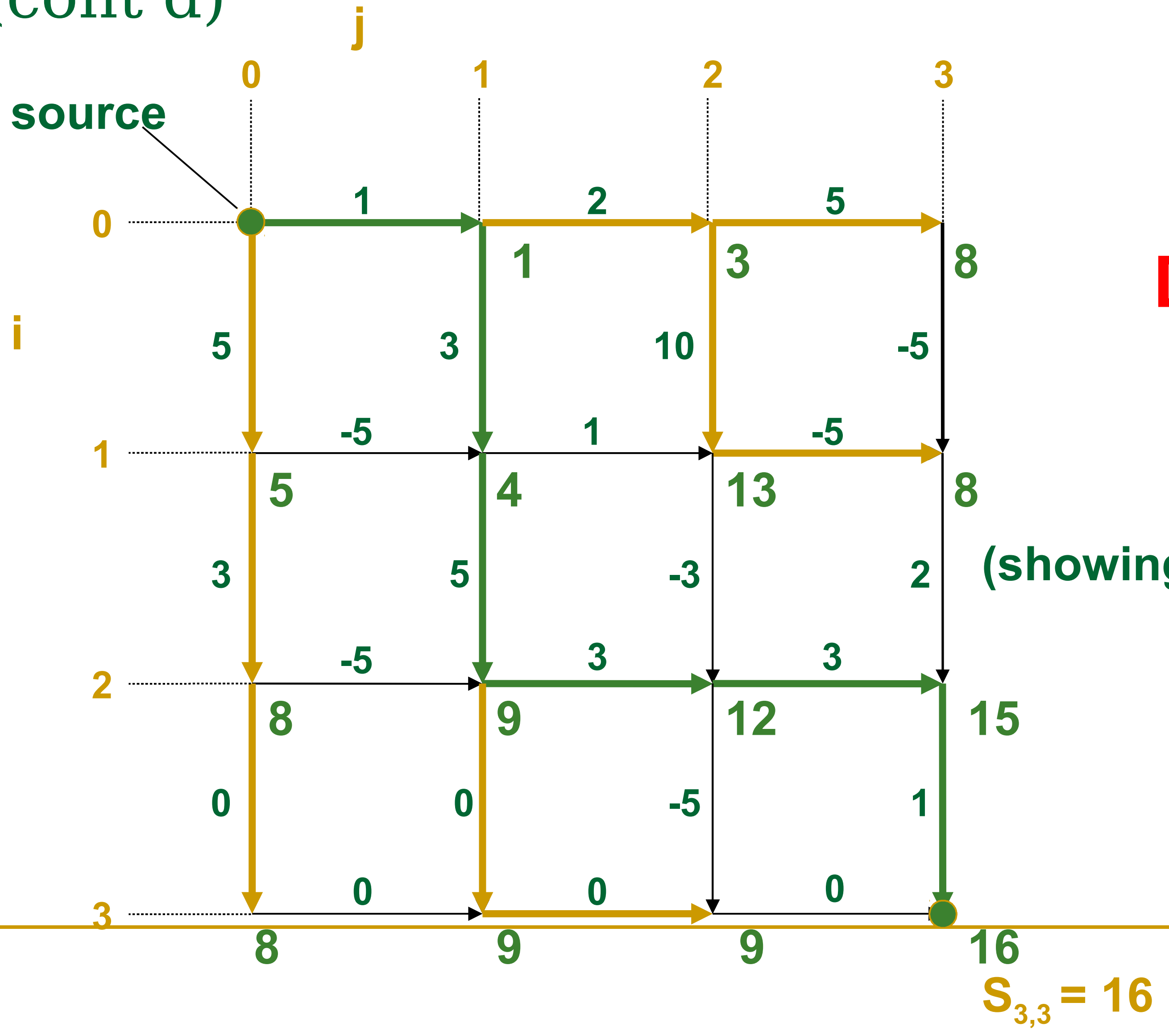
MTP: Dynamic Programming (cont'd)



MTP: Dynamic Programming (cont'd)



MTP: Dynamic Programming (cont'd)



Done!

(showing all back-traces)

MTP: Recurrence

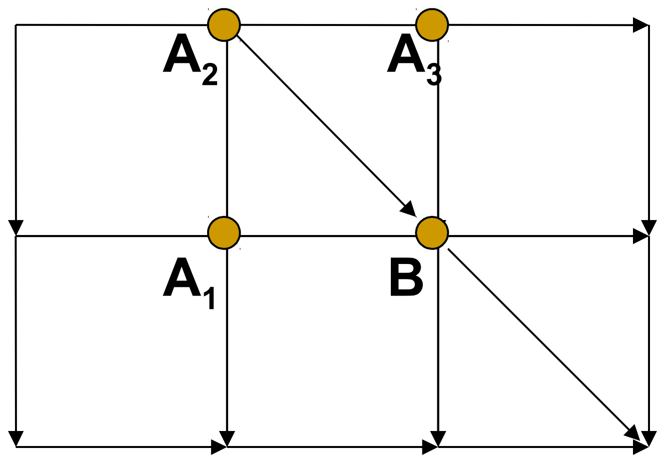
Computing the score for a point (i,j) by the recurrence relation:

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

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:

$$s_x = \max_{\text{of}} \left\{ \begin{array}{l} s_y + \text{weight of vertex } (y, x) \text{ where} \\ \qquad\qquad y \in \text{Predecessors}(x) \end{array} \right.$$

- 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

DAG: Directed Acyclic Graph

- 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:

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

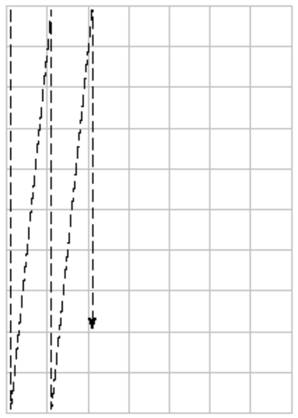
<u>In General</u>:

$$s_v = \max_u \left(s_u + \text{weight of edge from } u \text{ to } v\right)$$

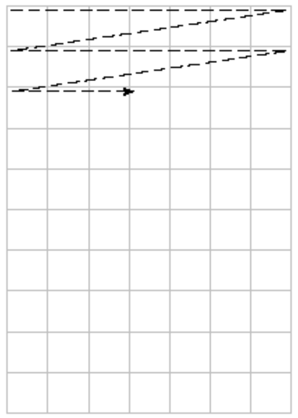
Traversing the Manhattan Grid

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

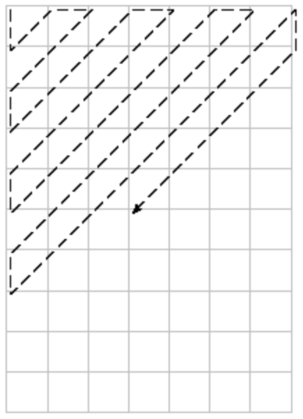
a)



b)



c)



ALIGNMENT

Alignment: 2 row representation

Given 2 DNA sequences v and w:

V : ATGTTAT
W : ATCGTAC

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

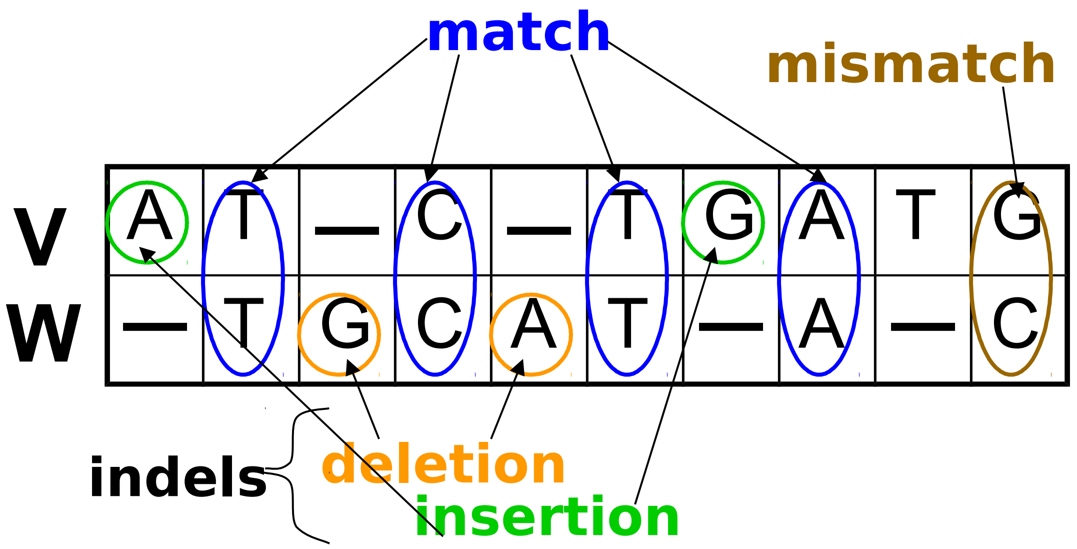
letters of v	A	T	--	G	T	T	A	T	--
letters of w	A	T	C	G	T	--	A	--	C

5 matches **2 insertions** **2 deletions**

Aligning DNA Sequences

V = ATCTGATG **n = 8**

W = TGCATAC **m = 7**



4 matches
1 mismatch
2 insertions
3 deletions

Longest Common Subsequence (LCS) – Alignment without Mismatches

- Given two sequences

  $v = v_1\ v_2 \ldots v_m$ 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

i coords:	0	1	2	2	3	3	4	5	6	7	8
elements of v		A	T	--	C	--	T	G	A	T	C
elements of w		--	T	G	C	A	T	--	A	--	C
j coords:	0	0	1	2	3	4	5	5	6	6	7

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

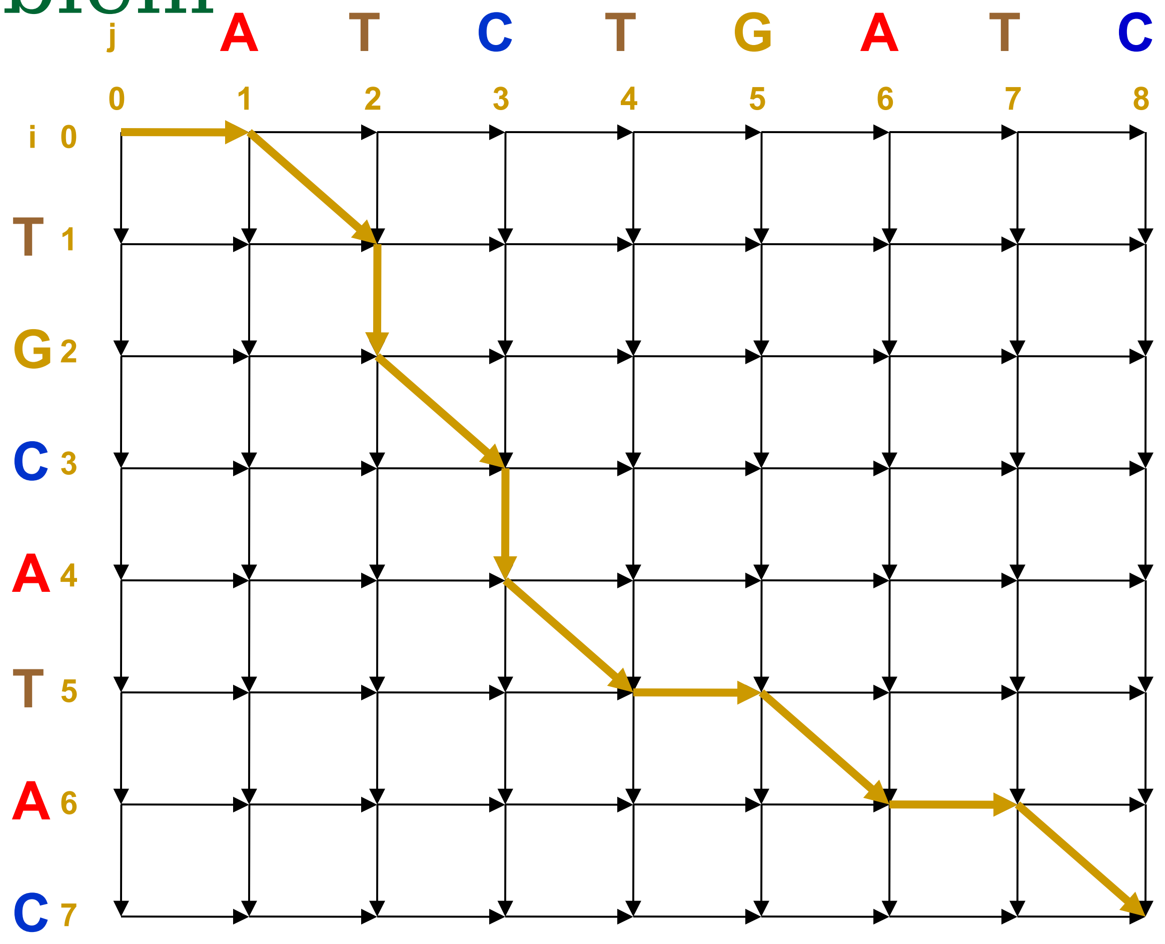
Matches shown in red

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

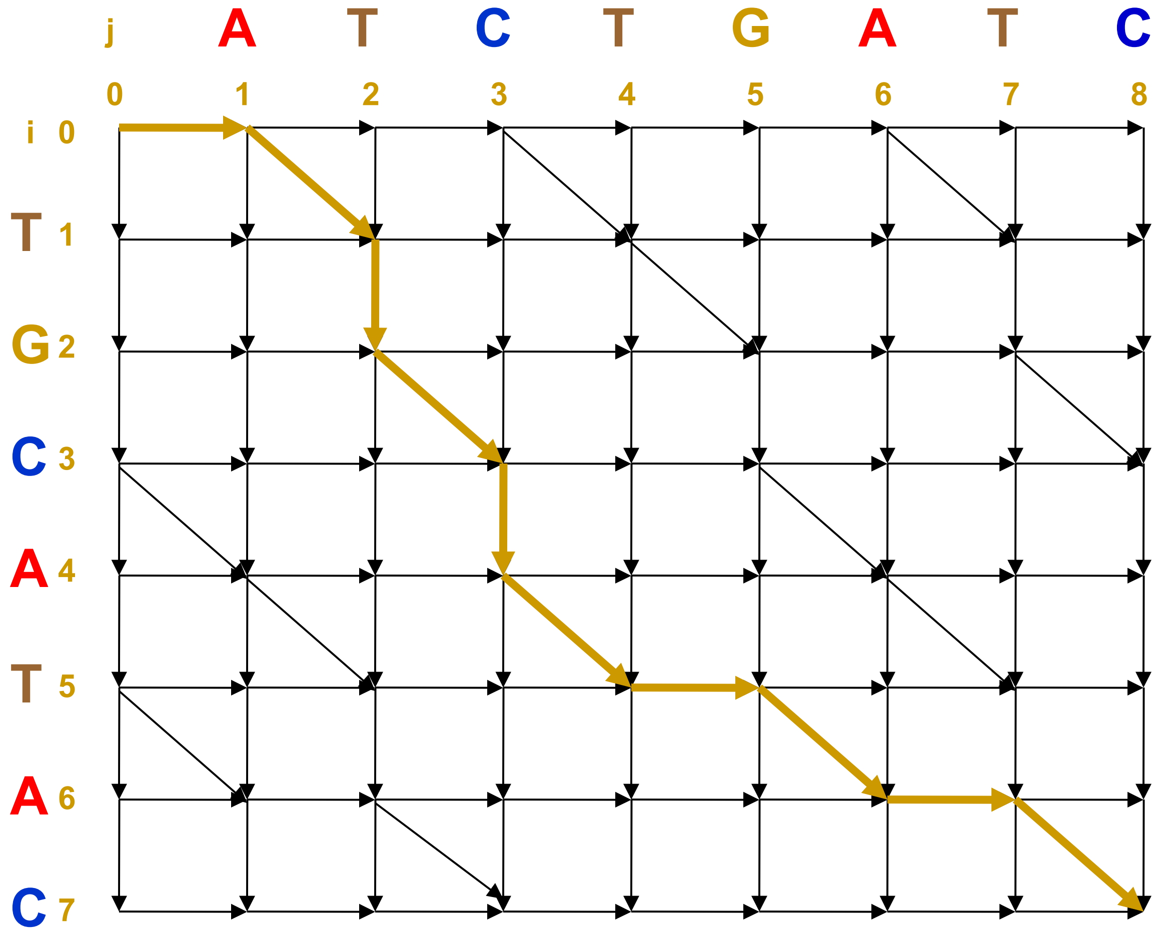
positions in w: 1 < 3 < 5 < 6 < 7

Every common subsequence is a path in 2-D grid

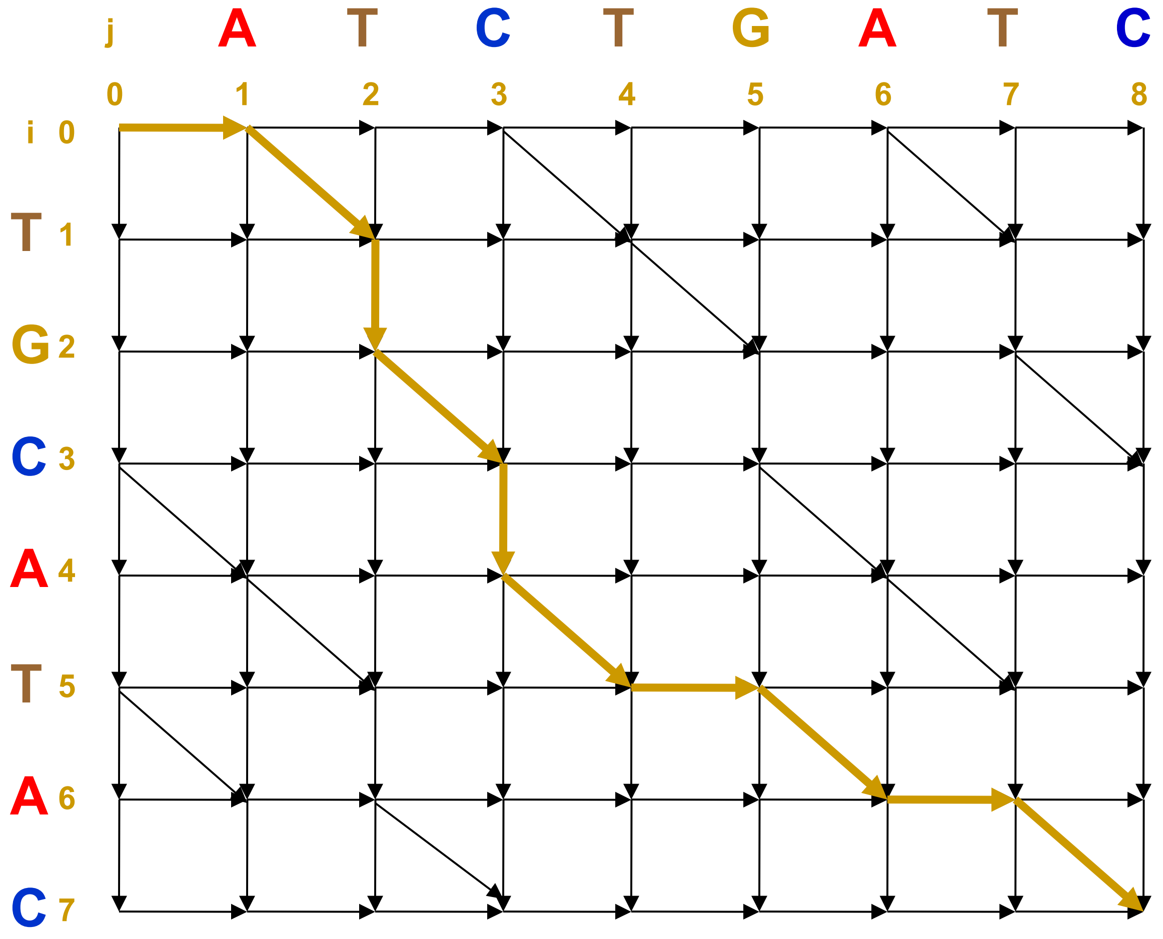
LCS Problem as Manhattan Tourist Problem



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 v_i = prefix of v of length i: $v_1 \ldots v_i$

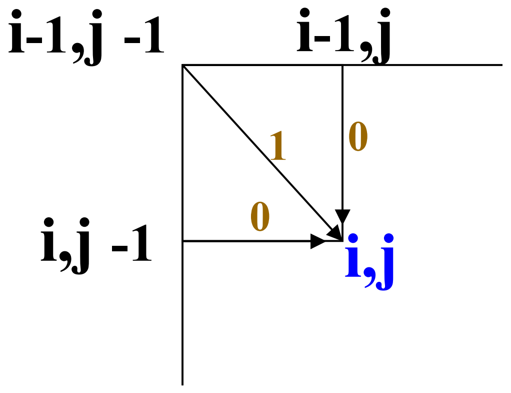
and w_j = prefix of w of length j: $w_1 \ldots w_j$

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

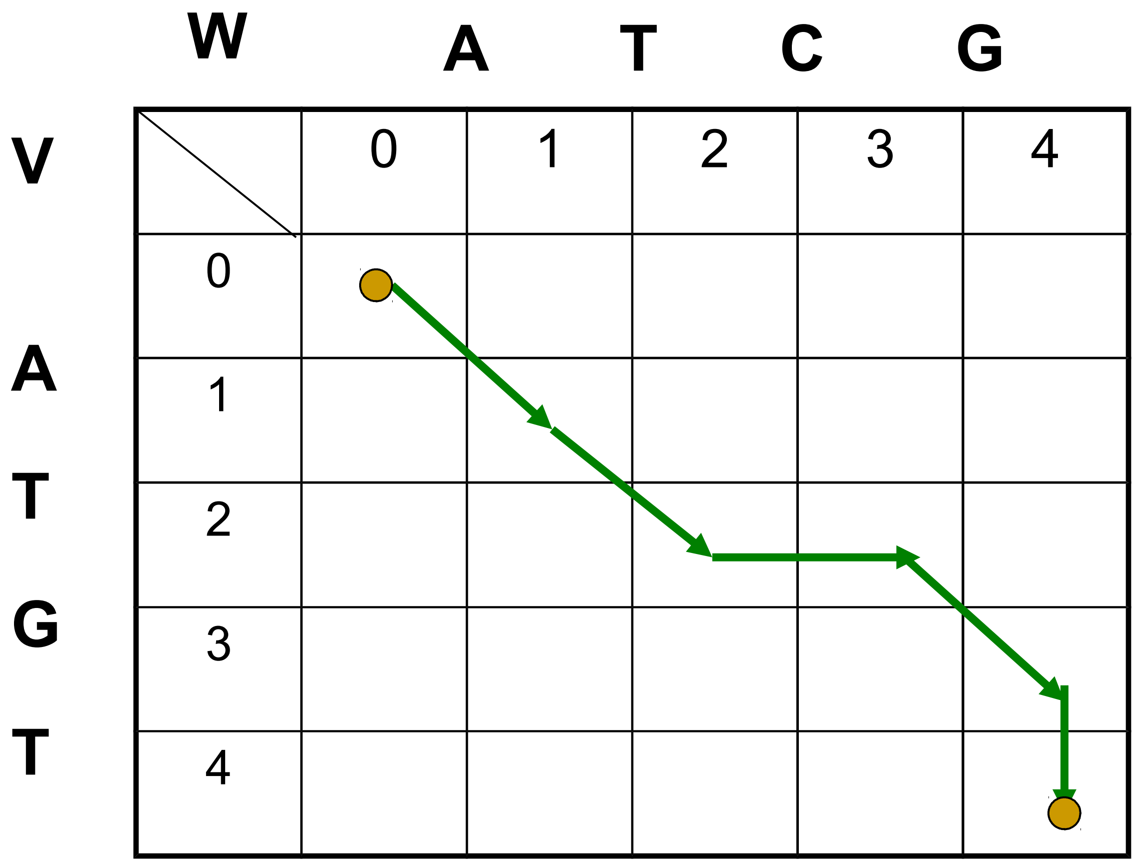
$$
s_{i,j} = \max \begin{cases} s_{i-1,j} \\ s_{i,j-1} \\ s_{i-1,j-1} + 1 \text{ if } v_i = w_j \end{cases}
$$

Computing LCS (cont'd)

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



Every Path in the Grid Corresponds to an Alignment



$$\begin{array}{lccccc}
 & \searrow & \searrow & \rightarrow & \searrow & \downarrow \\
 & 0 & 1 & 2 & 2 & 3\;4 \\
V = & A & T & - & G & T \\
 & | & | & & | & \\
W = & A & T & C & G & - \\
 & 0 & 1 & 2 & 3 & 4\;4
\end{array}$$

DISTANCE BETWEEN STRINGS

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: $d_H(v, w) = 8$ is large but the sequences are very similar**

Aligning Sequences with Insertions and Deletions

By shifting one sequence over one position:

```
v : 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(v,w) = MIN number of elementary operations to transform v ✉ w

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:

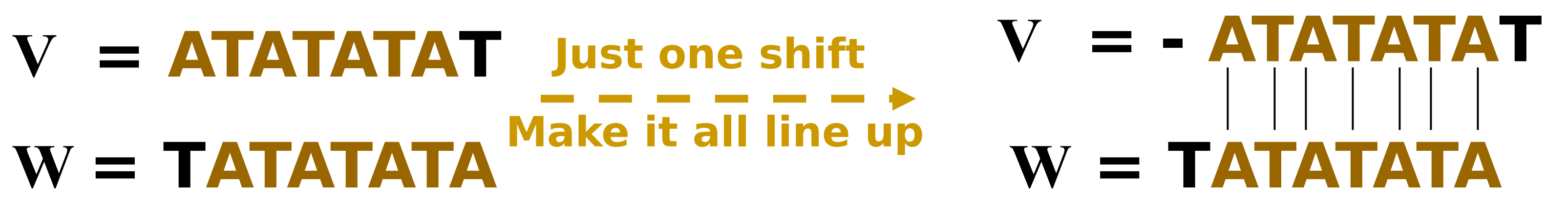
d(v, w)=8

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

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



Hamming distance:

$d(v, w)=8$

Computing Hamming distance is a trivial task

Edit distance:

$d(v, w)=2$

Computing edit distance is a non-trivial task

Edit Distance: Example

TGCATAT ✉ ATCCGAT in 5 steps

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

Edit Distance: Example

TGCATAT ✉ ATCCGAT in 5 steps

TGCATAT ✉ (delete last T)

TGCATA ✉ (delete last A)

TGCAT ✉ (insert A at front)

ATGCAT ✉ (substitute C for 3rd G)

ATCCAT ✉ (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 6th T)

ATGCATA ✉ (substitute G for 5th A)

ATGCGTA ✉ (substitute C for 3rd G)

ATCCGAT (Done)

Edit Distance: Example (cont’d)

TGCATAT ✉ ATCCGAT in 4 steps

TGCATAT ✉ (insert A at front)

ATGCATAT ✉ (delete 6th T)

ATGCATA ✉ (substitute G for 5th A)

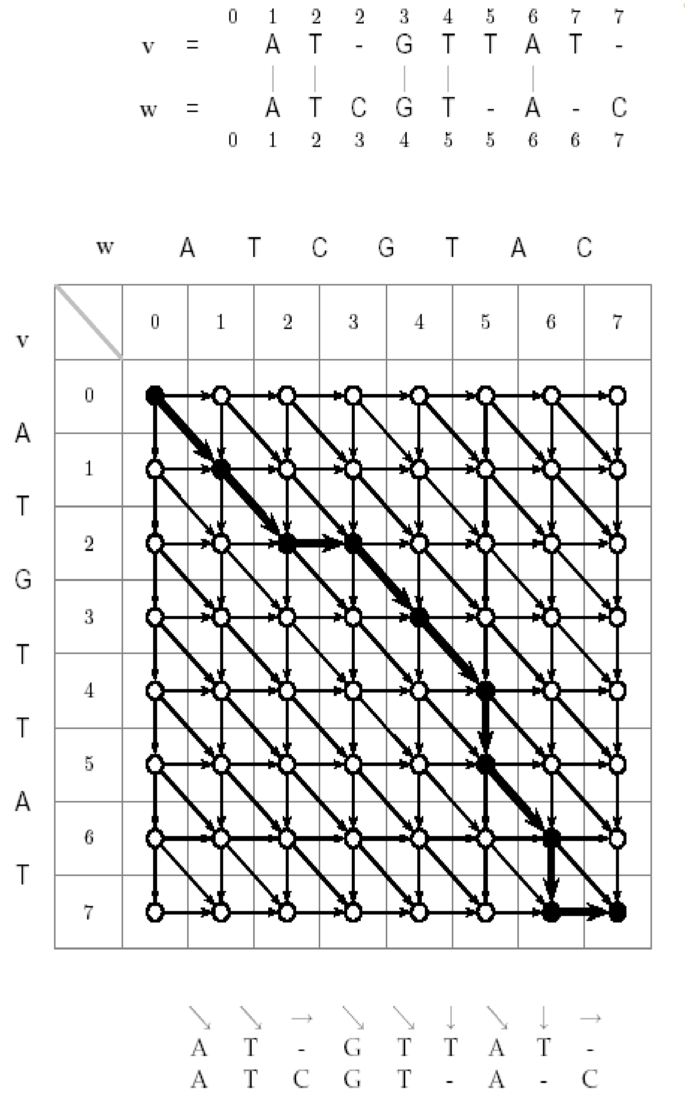
ATGCGTA ✉ (substitute C for 3rd G)

ATCCGAT (Done)

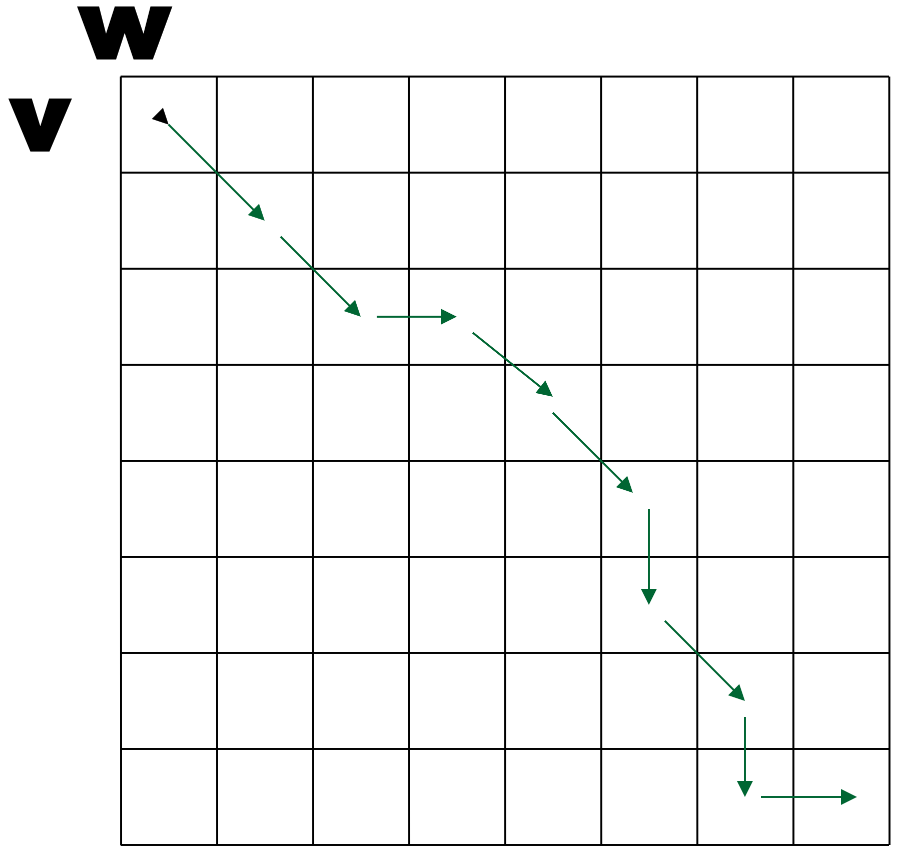
Can it be done in 3 steps???

The Alignment Grid

- Every alignment path is from source to sink



Alignment as a Path in the Edit Graph

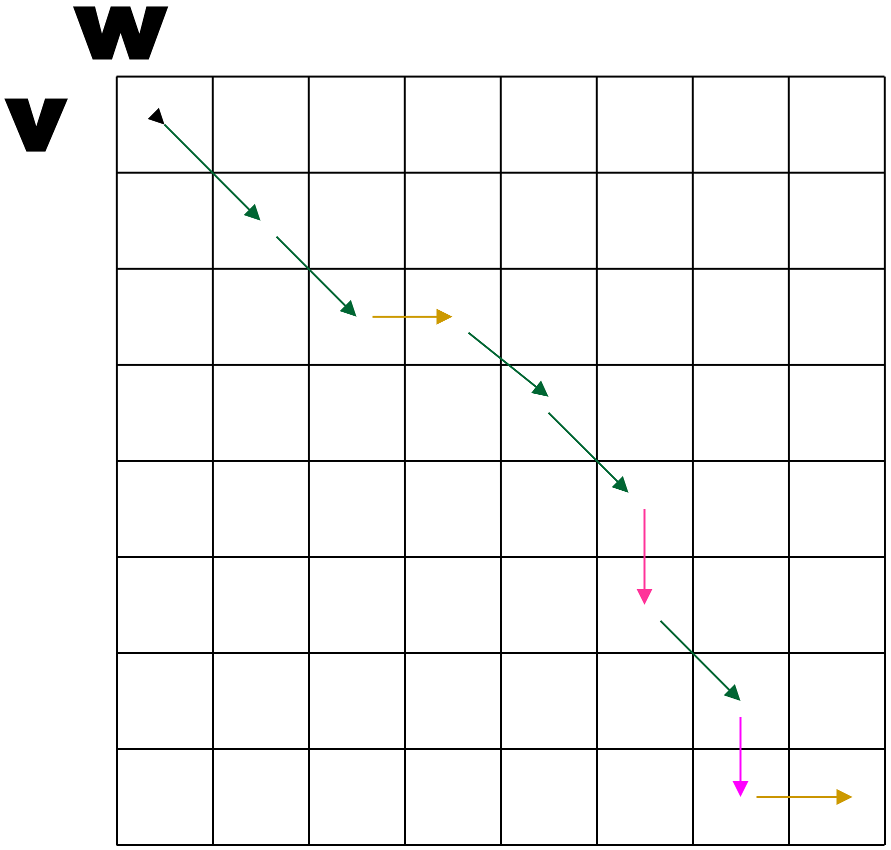


```
0 1 2 2 3 4 5 6 7 7
  A T _ G T T A T _
  A T C G T _ A _ C
0 1 2 3 4 5 5 6 6 7
```

- 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)

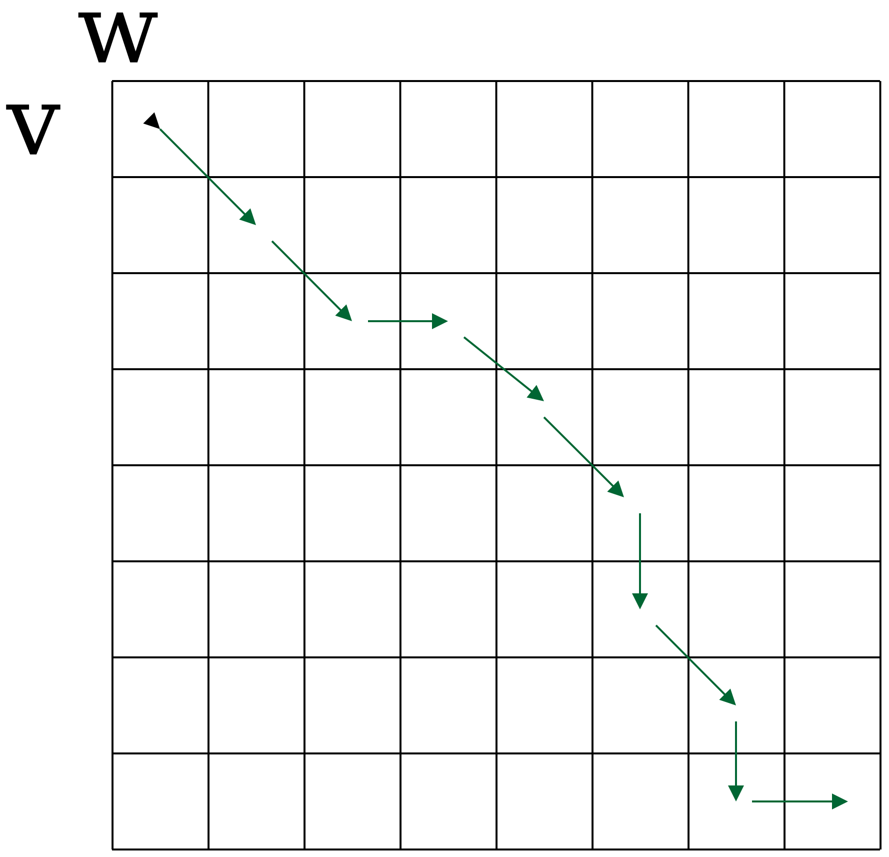


↓ and → represent indels in v and w with score 0.

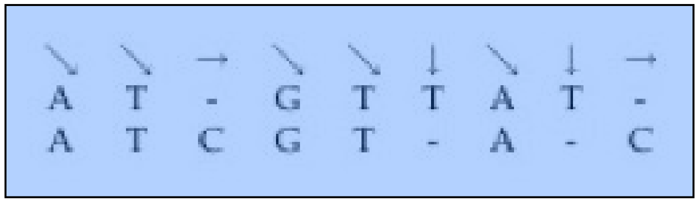
↘ represent matches with score 1.

- The score of the alignment path is 5.

Alignment as a Path in the Edit Graph

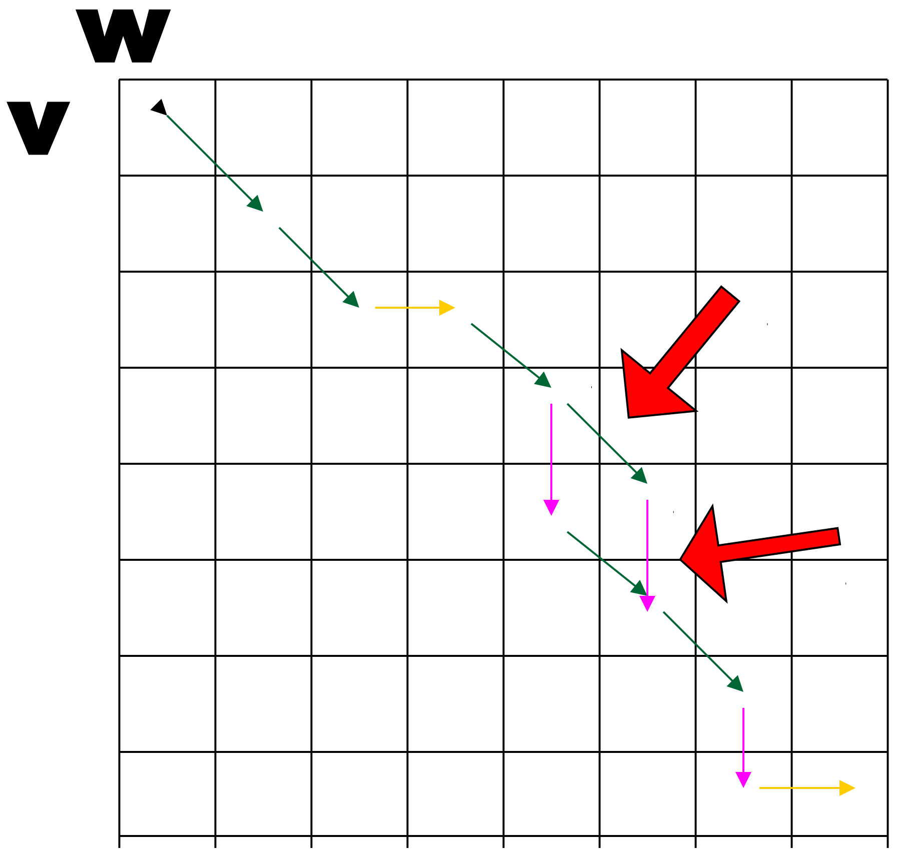


Every path in the edit graph corresponds to an alignment:



↘	↘	→	↘	↘	↓	↘	↓	→
A	T	-	G	T	T	A	T	-
A	T	C	G	T	-	A	-	C

Alignment as a Path in the Edit Graph



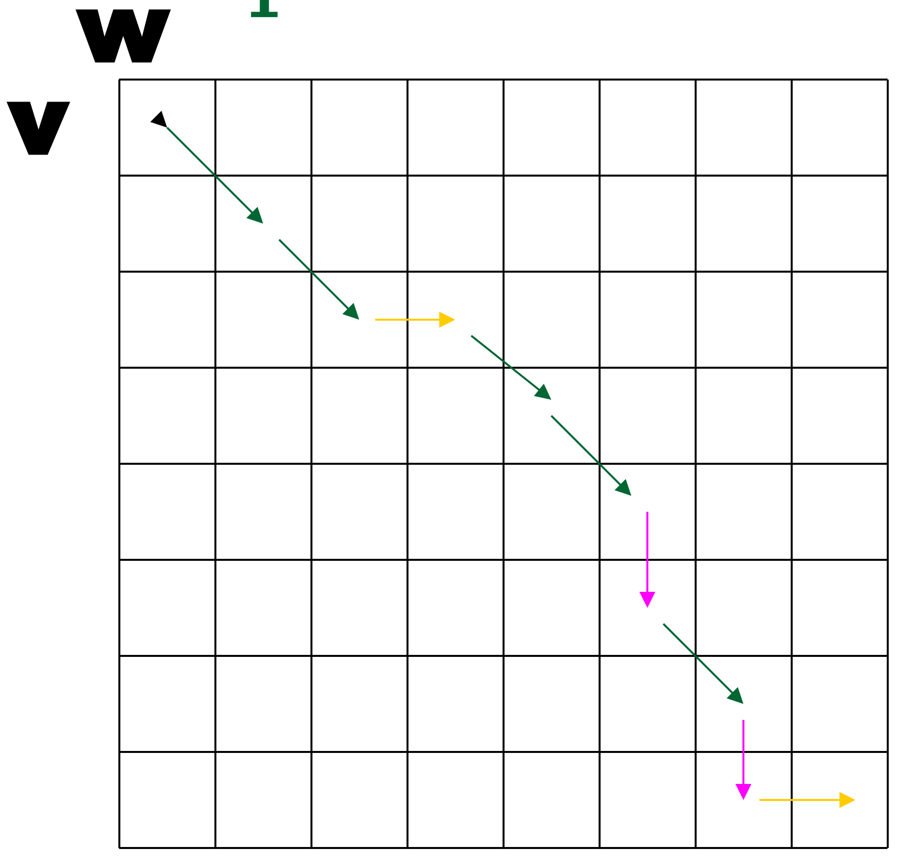
Old Alignment

```
     0122345677
v=   AT_GTTAT_
w=   ATCGT_A_C
     0123455667
```

New Alignment

```
     0122345677
v=   AT_GTTAT_
w=   ATCG_TA_C
     0123445667
```

Alignment as a Path in the Edit Graph



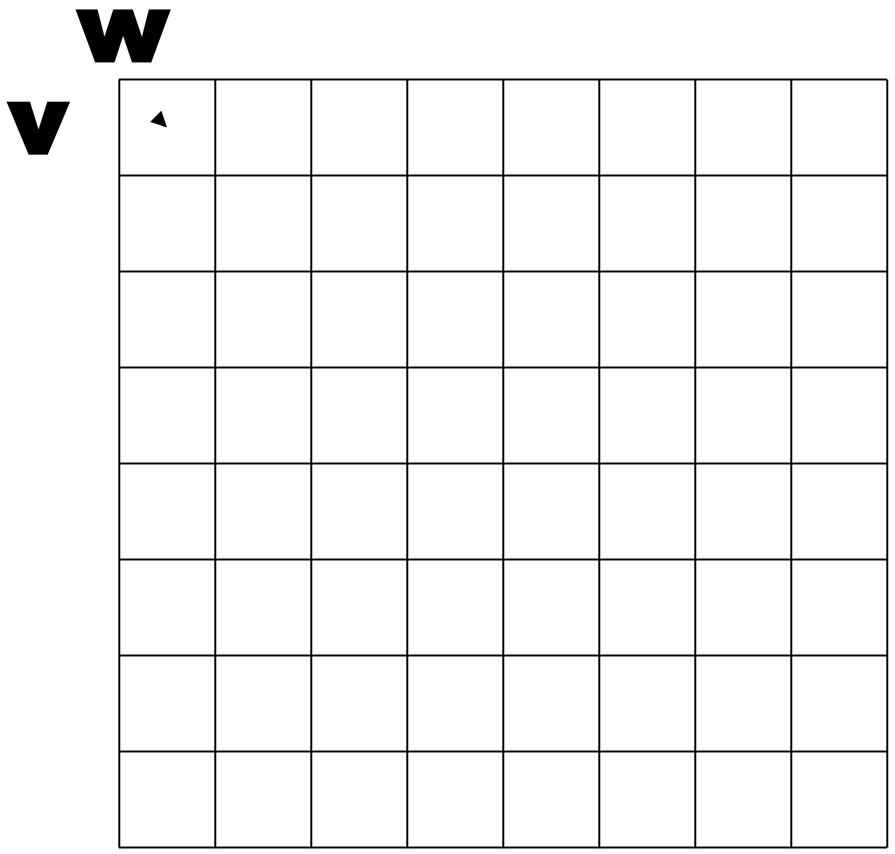
```
     012234567 7
v=    AT_GTTAT_
w=    ATCGT_A_C
     0123455667
```

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

Alignment: Dynamic Programming

$$s_{i,j} = \max \begin{array}{ll} s_{i-1,\, j-1}+1 \text{ if } v_i = w_j & \searrow \\ s_{i-1,\, j} & \downarrow \\ s_{i,\, j-1} & \rightarrow \end{array}$$

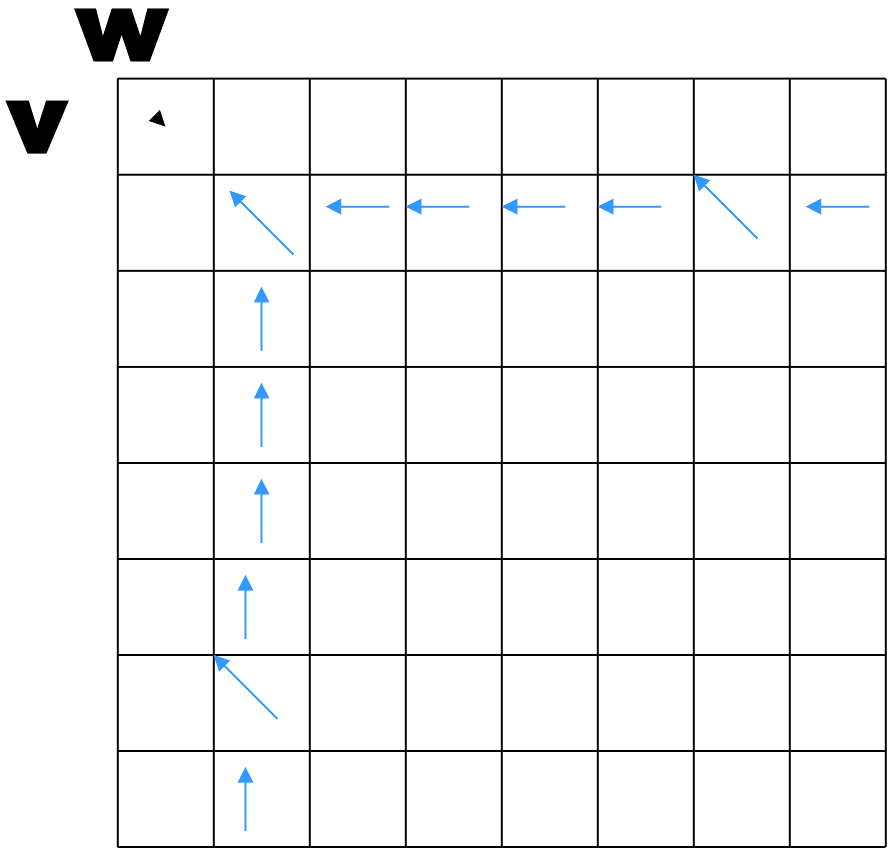
Dynamic Programming Example



Initialize 1st row and 1st column to be all zeroes.

Or, to be more precise, initialize 0th row and 0th column to be all zeroes.

Dynamic Programming Example



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

Alignment: Backtracking

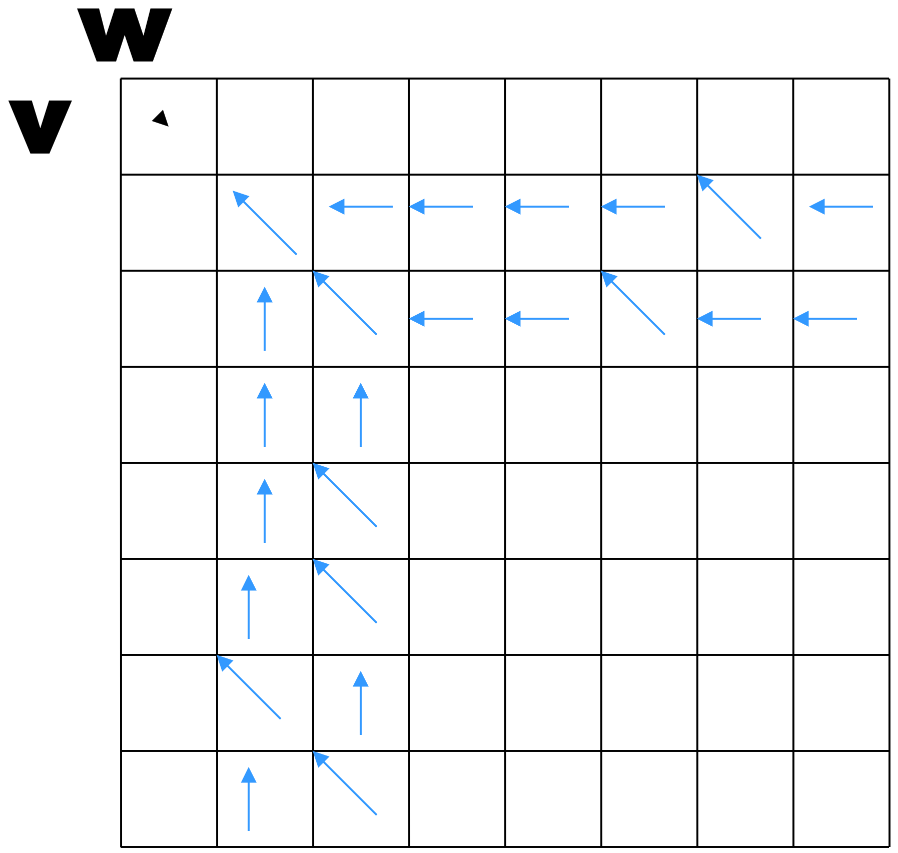
Arrows ↖ ↑ ← show where the score originated from.

↑ if from the top

← if from the left

↖ if $v_i = w_j$

Backtracking Example



Find a match in row and column 2.

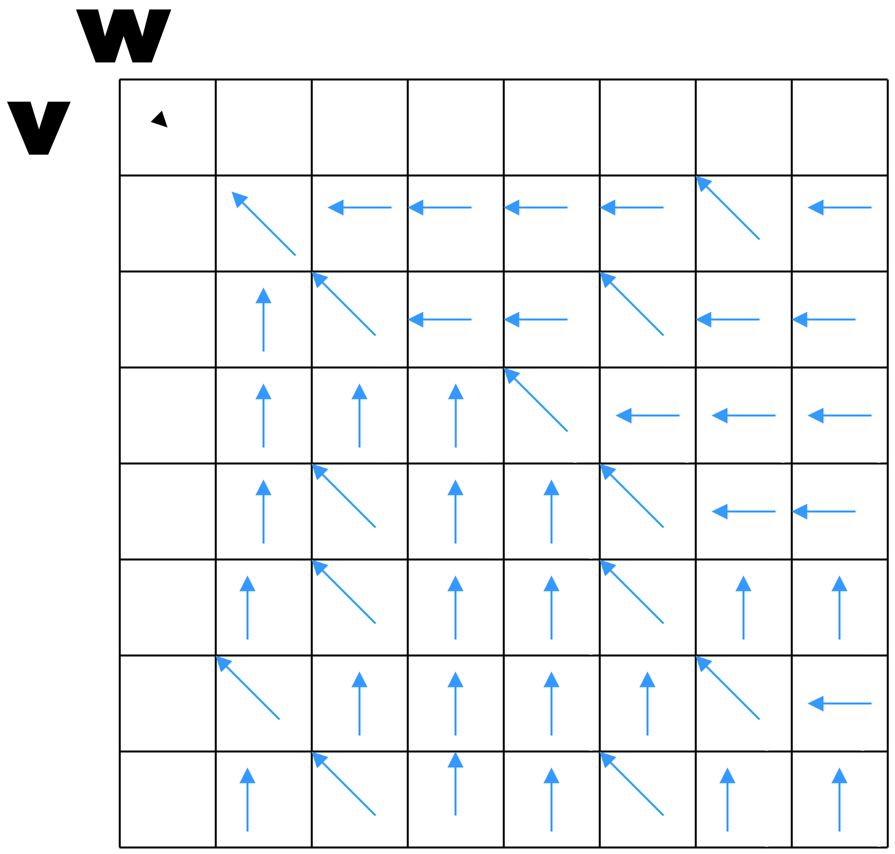
i=2, j=2,5 is a match (T).

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

$$s_{i,j} = \max \begin{array}{ll} s_{i-1,\, j-1} + 1 \text{ if } v_i = w_j & \searrow \\ s_{i-1,\, j} & \downarrow \\ s_{i,\, j-1} & \rightarrow \end{array}$$

Alignment: Dynamic Programming

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

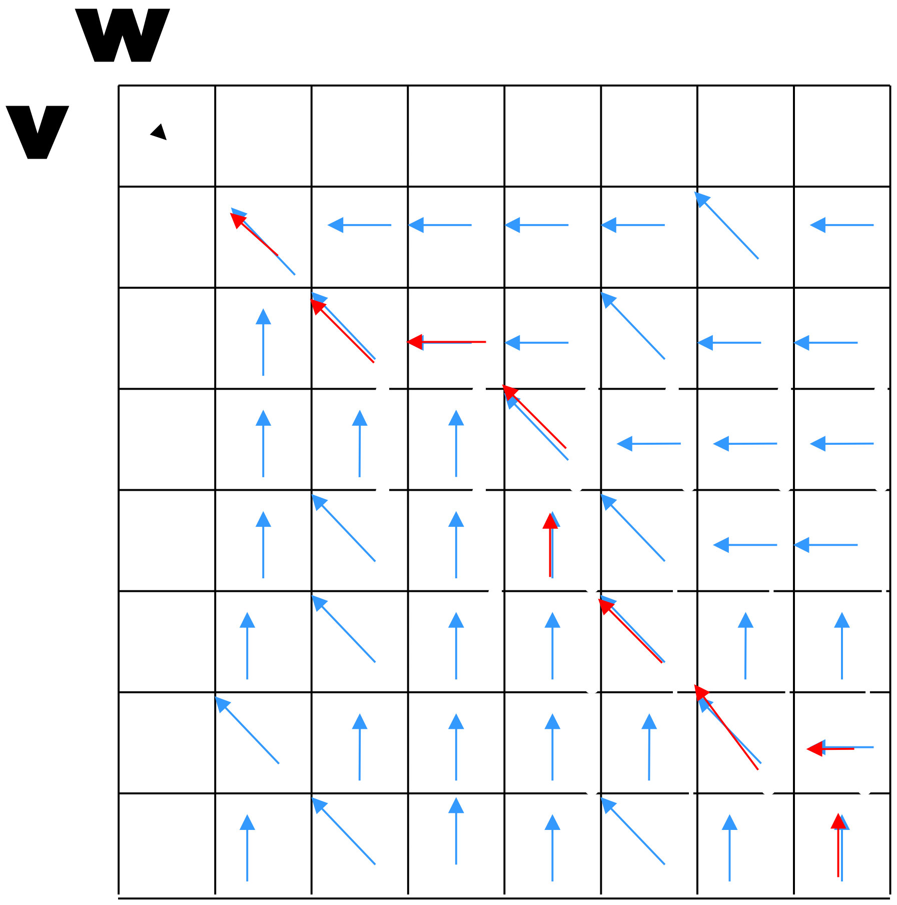
This recurrence corresponds to the Manhattan Tourist problem (three incoming edges into a vertex) with all horizontal and vertical edges weighted by zero.

LCS Algorithm

```
1.  LCS(v,w)
2.    for $i \gtrapprox 1$ to $n$
3.      $s_{i,0} \gtrapprox 0$
4.    for j $\gtrapprox$ 1 to $m$
5.      $s_{0,j} \gtrapprox 0$
6.    for $i \gtrapprox 1$ to $n$
7.      for $j \gtrapprox 1$ to $m$
8.                          $s_{i-1,j}$
9.          $s_{i,j} \gtrapprox$ max{ $s_{i,j-1}$
10.                         $s_{i-1,j-1} + 1$, if $v_i = w_j$
11.                    { “  ↑  “    if  $s_{i,j} = s_{i-1,j}$
                             ←
■          $b_{i,j} \gtrapprox$     “  ↖  “    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 $b_{i,j}$ = "$\nwarrow$"
5.          PrintLCS(b,v,$i-1$,$j-1$)
6.          print $v_i$
7.      else
8.          if $b_{i,j}$ = "$\uparrow$"
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.