Introduction to Algorithms

Kiyoko F. Aoki-Kinoshita

Computational problems

- A computational problem specifies an input-output relationship
  - What does the input look like?
  - What should the output be for each input?
- Example:
  - Input: an integer number N
  - Output: Is the number prime?
- Example:
  - Input: A list of names of people
  - Output: The same list sorted alphabetically
- Example:
  - Input: A picture in digital format
  - Output: An English description of what the picture shows

Algorithms

- An algorithm is an exact specification of how to solve a computational problem
- An algorithm must specify every step completely, so a computer can implement it without any further "understanding"
- An algorithm must work for all possible inputs of the problem.
- Algorithms must be:
  - Correct: For each input, terminate and produce an appropriate output
  - Efficient: run as quickly as possible, and use as little memory as possible – more about this later
- There can be many different algorithms for each computational problem.

Describing Algorithms

- Algorithms can be implemented in any programming language
- Usually we use “pseudo-code” to describe algorithms
-

```
Testing whether input N is prime:

For j = 2 .. N-1
  If the remainder of j/N is 0
    Output “N is composite” and halt
Output “N is prime”
```

- In this course we will just describe algorithms in Perl and pseudocode

Greatest Common Divisor

- The first algorithm “invented” in history was Euclid’s algorithm for finding the greatest common divisor (GCD) of two natural numbers
- **Definition:** The GCD of two natural numbers *x, y* is the largest integer *j* that divides both evenly (with remainder 0).

- **The GCD Problem:**
  - Input: natural numbers *x, y*
  - Output: *GCD(x,y)* – their GCD

Euclid's GCD Algorithm

```
sub gcd {
    my ($x, $y) = @_; // retrieve input x and y
    while ($y != 0) {  // while y is not equal to 0
        $t = $x % $y; // get the modulus of x and y
        $x = $y;   // replace x by y
        $y = $t;   // replace y by t
    }
    return $x; // return the result (gcd of x and y)
}

print gcd(14,21),"\n";
```

Euclid's GCD Algorithm – sample

```
while ($y != 0) {  // while y is not equal to 0
       $t = $x % $y; // get the modulus of x and y
       $x = $y;   // replace x by y
       $y = $t;   // replace y by t
}
```

Example: Computing GCD(48,120)

	t	x	y
After 0 rounds	--	72	120
After 1 round	72	120	72
After 2 rounds	48	72	48
After 3 rounds	24	48	24
After 4 rounds	0	24	0

Output: 24

Termination of Euclid's Algorithm

- Why does this algorithm terminate?
  - After any iteration we have that *x* > *y* since the new value of *y* is the remainder of the division by the new value of *x*.
  - In further iterations, we replace *(x, y)* with *(y, x%y)*, and *x%y* < *x*, thus the numbers decrease in each iteration.
  - Formally, the value of *xy* decreases at each iteration (except, maybe, the first one). When it reaches 0, the algorithm must terminate.

```
sub gcd {
    my ($x, $y) = @_; // retrieve input x and y
    while ($y != 0) {  // while y is not equal to 0
        $t = $x % $y; // get the modulus of x and y
        $x = $y;    // replace x by y
        $y = $t;    // replace y by t
    }
    return $x; // return the result (gcd of x and y)
}
```

Introduction to Algorithms

Running Time Analysis

How fast will your program run?

- The running time of your program will depend upon:
  - The algorithm
  - The input
  - Your implementation of the algorithm in a programming language
  - The compiler you use
  - The operating system (OS) on your computer
  - Your computer hardware
  - Maybe other things: temperature outside; other programs on your computer; …
- Our Motivation: analyze the running time of an algorithm as a function of only simple parameters of the input.

Basic idea: counting operations

- Each algorithm performs a sequence of basic operations:
  - Arithmetic: (low + high)/2
  - Comparison: if (x > 0) …
  - Assignment: temp = x
  - Branching: while (y != 0) { … }
  - …
- Idea: count the number of basic operations performed on the input.
- Difficulties:
  - Which operations are basic?
  - Not all operations take the same amount of time.
  - Operations take different times with different hardware or compilers

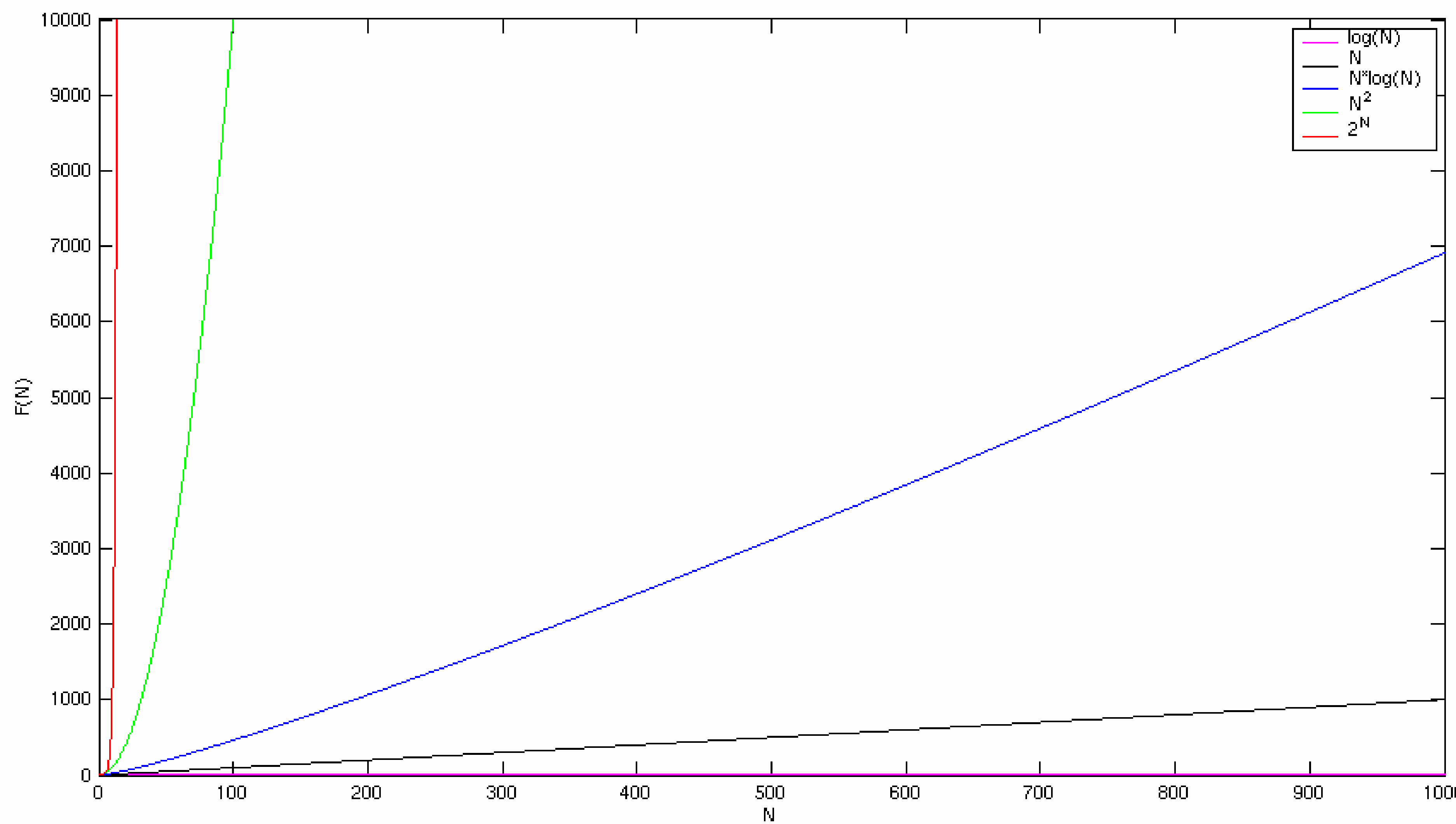
Asymptotic running times

- Operation counts are only problematic in terms of constant factors.
- The general form of the function describing the running time is invariant over hardware, languages or compilers!

```
sub myMethod{
  my $N = shift @_;
  my $sq = 0;
  for($j=0; $j<$N ; $j++)
    for($k=0; $k<$N ; $k++)
      $sq++;
  return $sq;
}
```

- Running time is “about” N^2.
- We use “Big-O” notation, and say that the running time is O(N^2)

Asymptotic behavior of functions



Mathematical Formalization

- Definition: Let f and g be functions from the natural numbers to the natural numbers. We write $f=O(g)$ if there exists a constant c such that for all n: $f(n) \leq cg(n)$.

  $$f=O(g) \quad \Leftrightarrow \quad \exists c \, \forall n: f(n) \leq cg(n)$$
- This is a mathematically formal way of ignoring constant factors, and looking only at the "shape" of the function.
- $f=O(g)$ should be considered as saying that "f is at most g, up to constant factors".
- We usually will have f be the running time of an algorithm and g a nicely written function. E.g. The running time of the previous algorithm was $O(N^2)$.

Asymptotic analysis of algorithms

- We usually embark on an *asymptotic worst case* analysis of the running time of the algorithm.
- Asymptotic:
  - Formal, exact, depends only on the algorithm
  - Ignores constants
  - Applicable mostly for large input sizes
- Worst Case:
  - Bounds on running time must hold for *all* inputs.
  - Thus the analysis considers the worst-case input.
  - Sometimes the “average” performance can be much better
  - Real-life inputs are rarely “average” in any formal sense

The running time of Euclid's GCD Algorithm

- How fast does Euclid's algorithm terminate?
  - After the first iteration we have that $x > y$. In each iteration, we replace (x, y) with $(y, x\%y)$.
  - In an iteration where $x>1.5y$ then $x\%y < y < 2x/3$.
  - In an iteration where $x \leq 1.5y$ then $x\%y \leq y/2 < 2x/3$.
  - Thus, the value of xy decreases by a factor of at least $2/3$ each iteration (except, maybe, the first one).

```
sub gcd {
    my ($x, $y) = @_; // retrieve input x and y
    while ($y != 0) {  // while y is not equal to 0
        $t = $x % $y; // get the modulus of x and y
        $x = $y;   // replace x by y
        $y = $t;   // replace y by t
    }
    return $x; // return the result (gcd of x and y)
}
```

The running time of Euclid's Algorithm

- <u>Theorem:</u> Euclid's GCD algorithm runs it time *O(N),* where N is the input length *(N=*$\log_2 x + \log_2 y$*).*
- <u>Proof:</u>
  - Every iteration of the loop (except maybe the first) the value of xy decreases by a factor of at least 2/3. Thus after *k+1* iterations the value of *xy* is at most $(2/3)^k$ the original value.
  - Thus the algorithm must terminate when k satisfies: $xy(2/3)^k < 1$ (for the original values of x, y).
  - Thus the algorithm runs for at most $1+\log_{3/2} xy$ iterations.
  - Each iteration has only a constant *L* number of operations, thus the total number of operations is at most $(1+\log_{3/2} xy)L$
  - Formally, $(1+\log_{3/2} xy)L \le L(1+2\log_2 x + 2\log_2 y) \le 3LN$
  - Thus the running time is *O(N).*

Introduction to Algorithms

Recursion

Designing Algorithms

- There is no single recipe for inventing algorithms
- There are basic rules:
  - Understand your problem well – may require much mathematical analysis!
  - Use existing algorithms (reduction) or algorithmic ideas
- There is a single basic algorithmic technique:

  Divide and Conquer
- In its simplest (and most useful) form it is simple induction
  - In order to solve a problem, solve a similar problem of smaller size
- The key conceptual idea:
  - Think only about how to use the smaller solution to get the larger one
  - Do not worry about how to solve the smaller problem (it will be solved using an even smaller one)

Recursion

- A recursive method is a method that contains a call to itself
- Technically:
  - All modern computing languages allow writing methods that call themselves
  - We will discuss how this is implemented later
- Conceptually:
  - This allows programming in a style that reflects divide-n-conquer algorithmic thinking
  - At the beginning recursive programs are confusing – after a while they become clearer than non-recursive variants

Factorial

```
sub factorial {
    my $n = shift @_; // retrieve input
    if ($n == 0) {
        return 1;  // if input is 0, return 1
    } else {
    // otherwise, compute the factorial of $n-1,
    // multiply it by $n and return the product
        return $n * factorial($n-1);
    }
}

print “5! = “,factorial(5),”\n”;
```

Elements of a recursive program

- Basis: a case that can be answered without using further recursive calls
  - In our case: if ($n==0) { return 1; }
- Creating the smaller problem, and invoking a recursive call on it
  - In our case: factorial($n-1)
- Finishing to solve the original problem
  - In our case: return $n; //solution of recursive call

Tracing the factorial method

```
        print "5! = ",factorial(5),"\n";

              5 * factorial(4)
                  4 * factorial(3)
                     3 * factorial(2)
                         2 * factorial(1)
                            1 * factorial(0)
                               return 1
                            return 1
                         return 2
                     return 6
                   return 24
               return 120
```

Correctness of factorial method

- Theorem: For every positive integer *n*, `factorial($n)` returns the value *n!*.
- Proof: By induction on n:
- Basis: for *n=0*, `factorial(0)` returns *1=0!*.
- Induction step: When called on *n>1*, factorial calls `factorial($n-1),` which by the induction hypothesis returns *(n-1)!*. The returned value is thus *n*(n-1)!=n!*.

Raising to power – take 1

```
sub power {
    my ($x, $n) = @_;  // retrieve the input
    if ($n == 0) {    // if $n is 0, return 1
        return 1.0;
    }
    // otherwise, return $x multiplied by the
    // result of power of x to the (n-1)th
    return $x * power($x, $n-1);
}

print “3^9 = “,power(3,9),”\n”;
```

Running time analysis

- Simplest way to calculate the running time of a recursive program is to add up the running times of the separate levels of recursion.
- In the case of the power method:
  - There are *n+1* levels of recursion
    - power(x,n), power(x,n-1), power(x, n-2), … power(x,0)
  - Each level takes *O(1)* steps
  - Total time = *O(n)*

Raising to power – take 2

```
sub power2 {
  my ($x, $n) = @_;
  if ($n == 0) {
    return 1.0;
  }
  if ($n%2 == 0) {
    my $t = power2($x, $n/2);
    return $t*$t;
  }
  return $x * power2($x, $n-1);
}
```

Analysis

- Theorem: For any *x* and positive integer *n*, the power method returns x^n.
- Proof: by complete induction on *n*.
  - Basis: For n=0, we return 1.
  - If n is even, we return power(x,n/2)*power(x,n/2). By the induction hypothesis power(x,n/2) returns $x^{n/2}$, so we return $(x^{n/2})^2 = x^n$
  - If n is odd, we return x*power(x,n-1). By the induction hypothesis power(x,n-1) returns x^{n-1}, so we return $x \cdot x^{n-1} = x^n$.
- The running time is now *O(log n):*
  - After 2 levels of recursion n has decreased by a factor of at least two (since either *n* or *n-1* is even, in which case the recursive call is with *n/2*)
  - Thus we reach n==0 after at most $2\log_2 n$ levels of recursion
  - Each level still takes O(1) time.

Introduction to Algorithms

Algorithms for bioinformatics

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 $\mathbf{c} = (c_1, c_2, \ldots, c_d)$, 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

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

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?

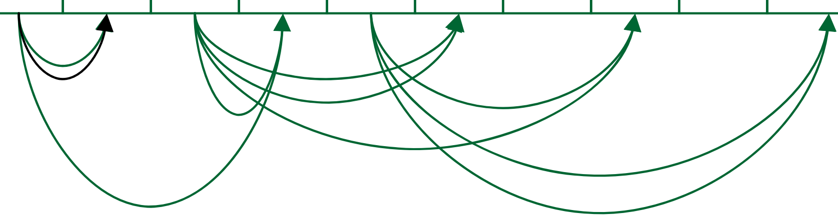
Value	1	2	3	4	5	6	7	8	9	10
Min # of coins	**1**		**1**		**1**					

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**

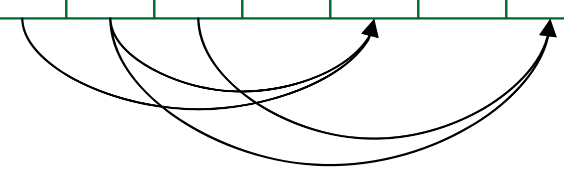


However, two coins are needed to make change for the values 2, 4, 6, 8, and 10.

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:

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

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

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

Change Problem: A Recursive Algorithm

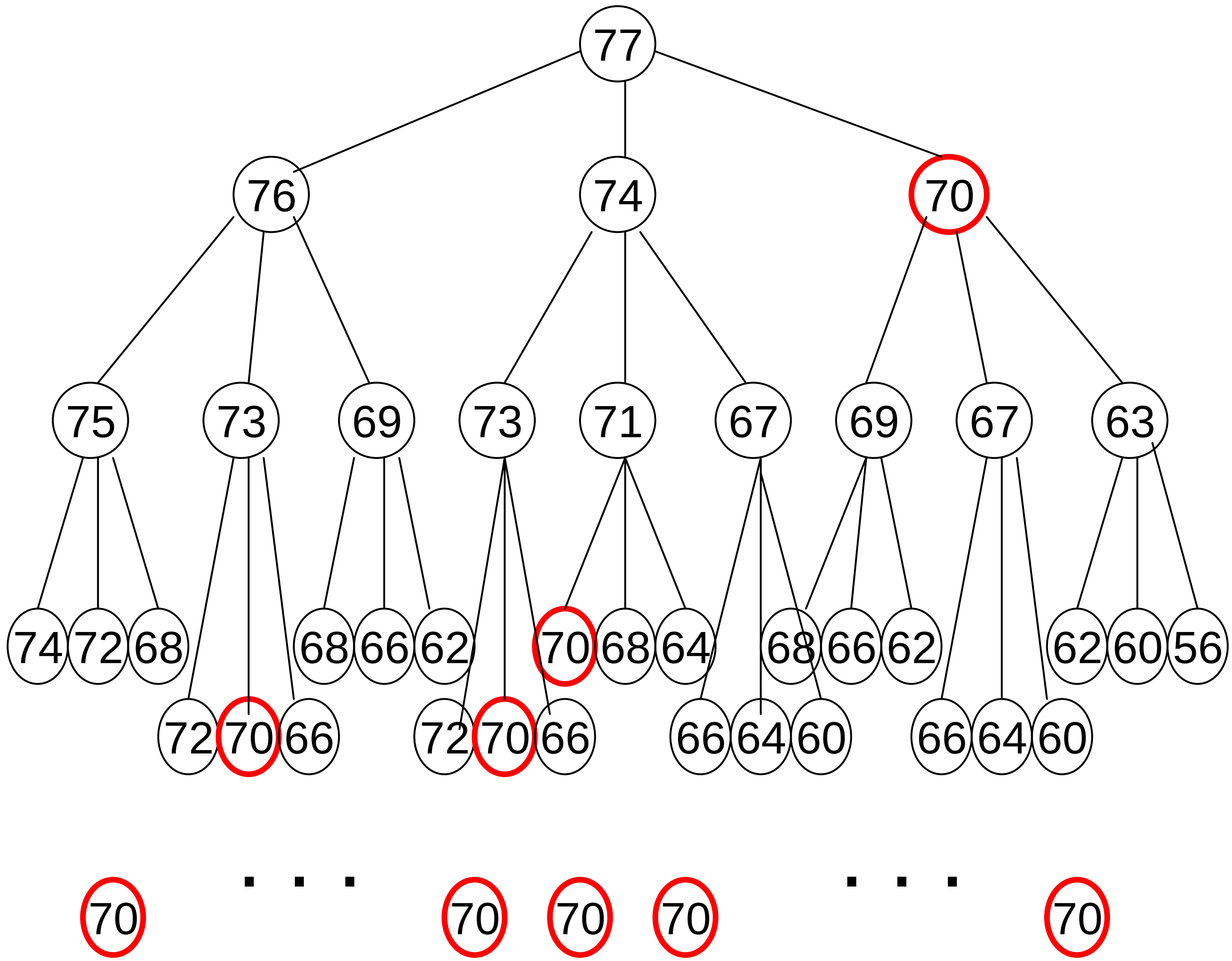
```
RecursiveChange(M,c,d)
  if M = 0
    return 0
  bestNumCoins = infinity
  for i = 1 to d
    if M ≥ c_i
      numCoins = RecursiveChange(M – c_i , c, d)
      if numCoins + 1 < bestNumCoins
        bestNumCoins = numCoins + 1
  return bestNumCoins
```

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

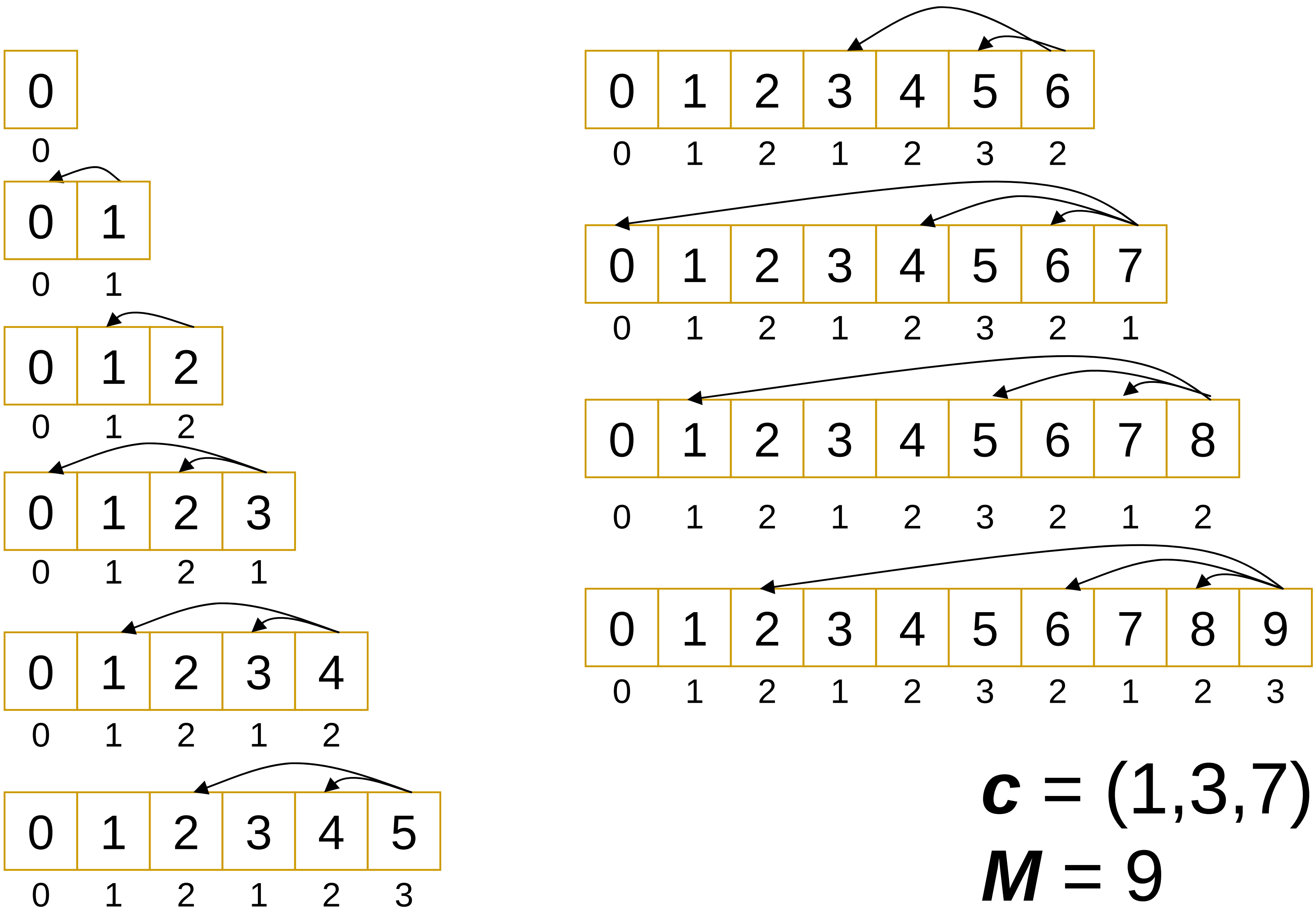
- We're re-computing values in our algorithm more than once
- Save results of each computation for 0 to ***M***
- This way, we can do a reference call to find an already computed value, instead of re-computing each time

- Running time becomes ***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)
    $bestNumCoins_0 = 0$
    for $m = 1$ to M
        $bestNumCoins_m$ = infinity
        for $i = 1$ to d
            if $m \geq c_i$
                if $bestNumCoins_{m - c_i} + 1 < bestNumCoins_m$
                    $bestNumCoins_m = bestNumCoins_{m - c_i} + 1$
    return $bestNumCoins_M$

DPChange: Example

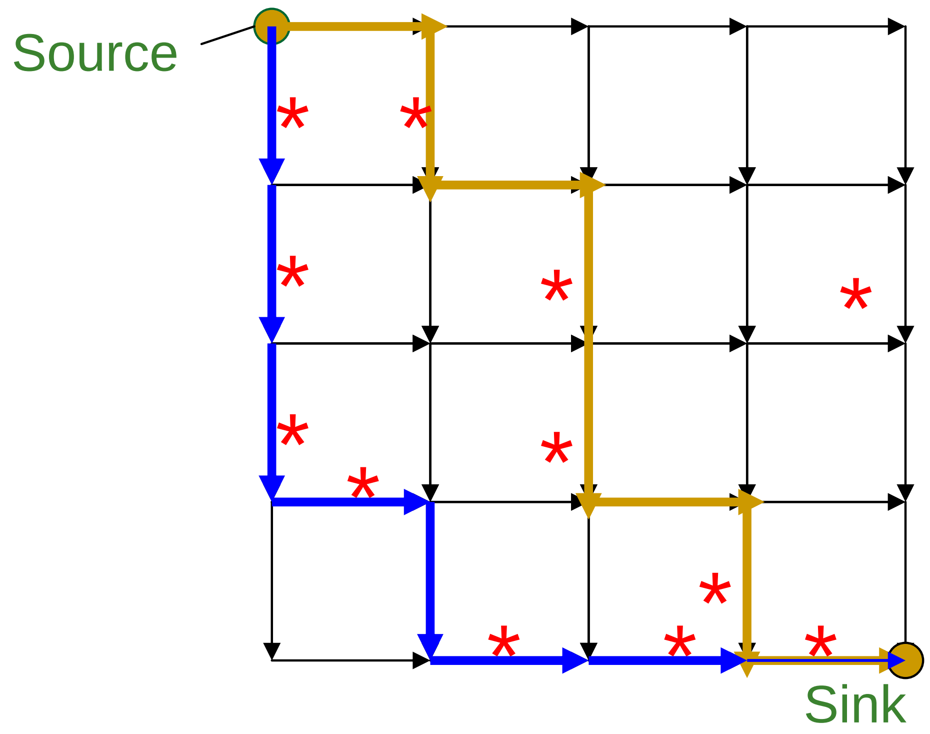


$\boldsymbol{c} = (1,3,7)$

$\boldsymbol{M} = 9$

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



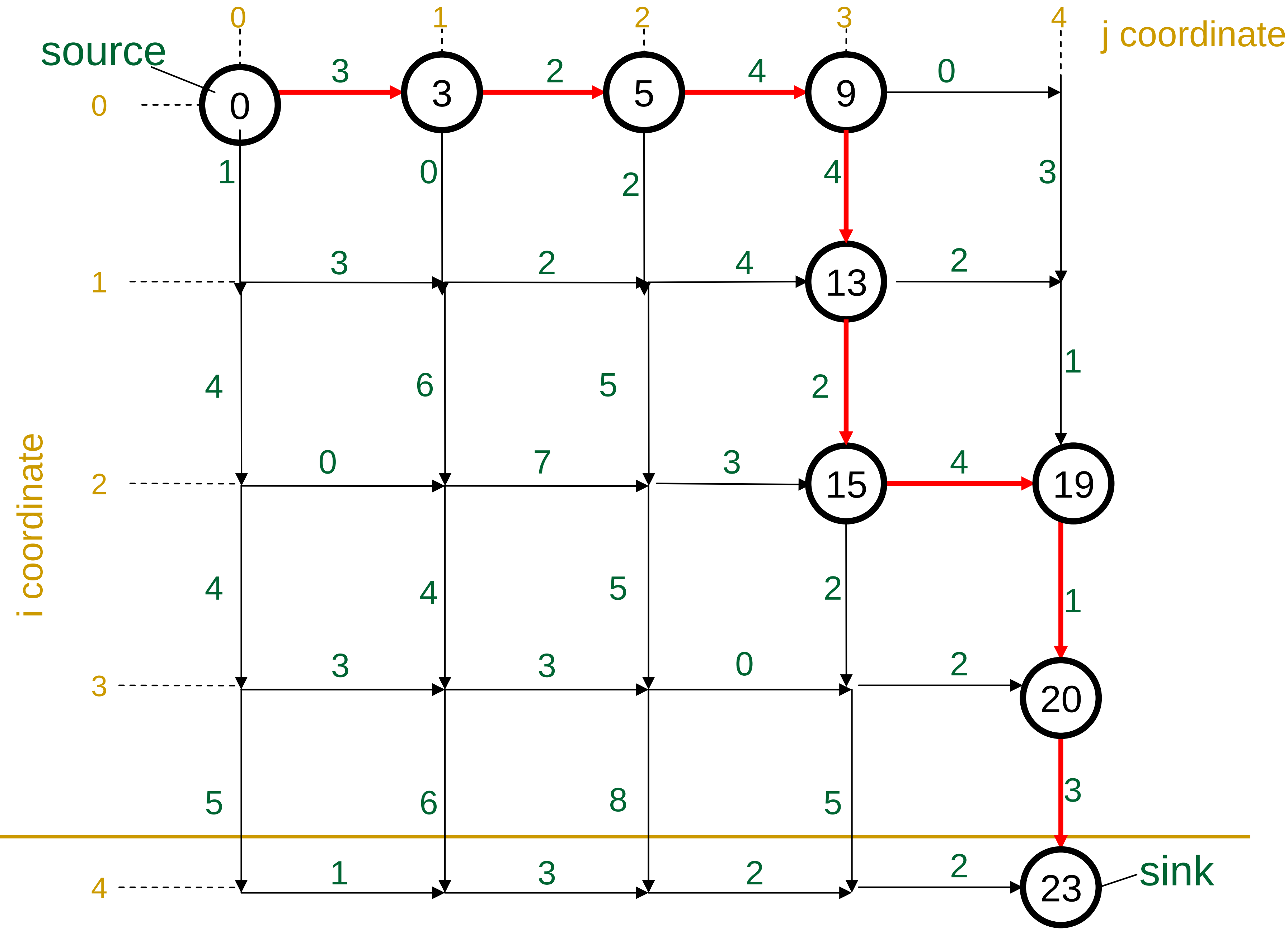
Manhattan Tourist Problem: Formulation

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*"

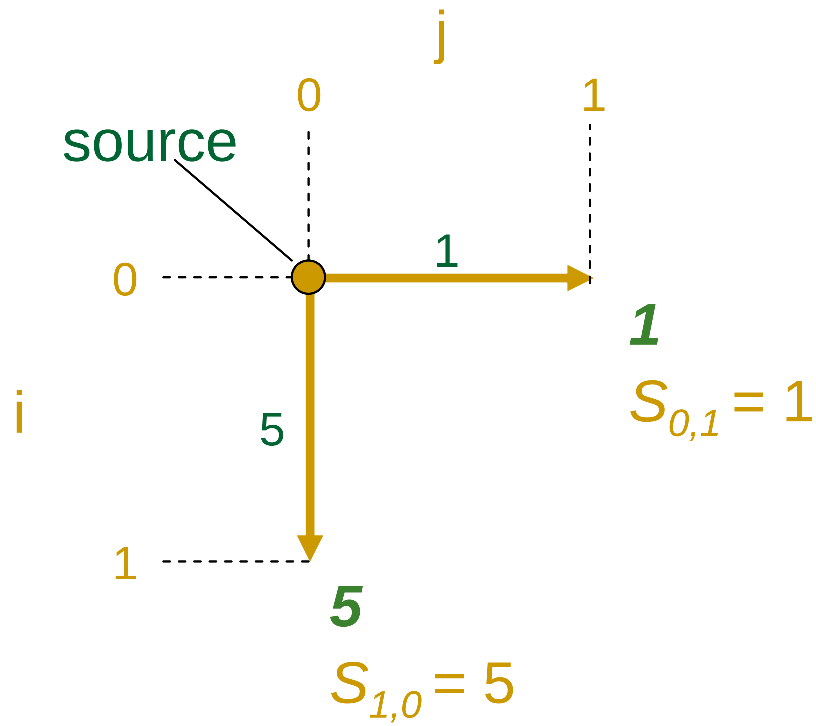
MTP: An Example



MTP: Simple Recursive Program

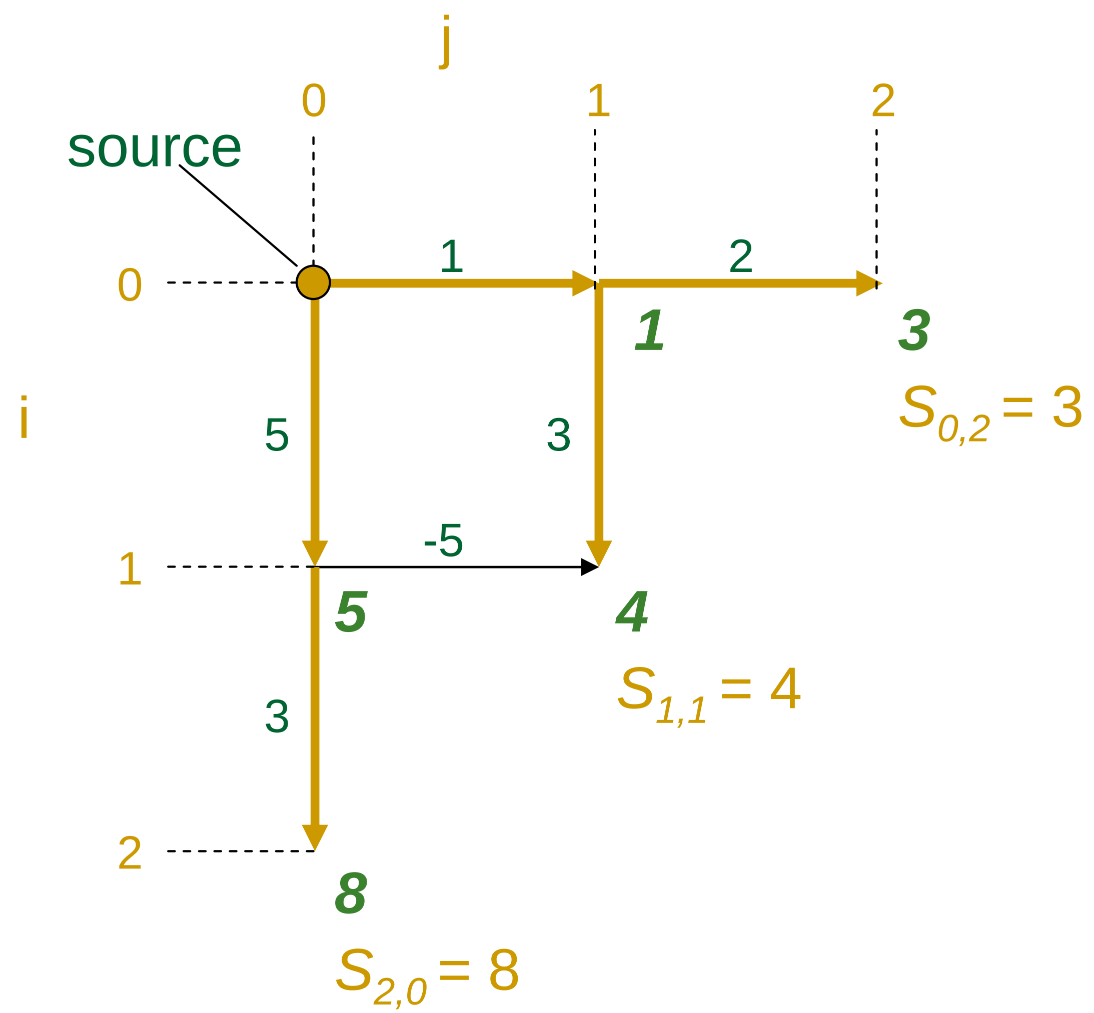
```
MT(n,m)
  if n=0 or m=0
    return MT(n,m)
  x = MT(n-1,m)+
              length of the edge from (n- 1,m) to (n,m)
  y = MT(n,m-1)+
              length of the edge from (n,m-1) to (n,m)
  return max{x,y}
```

MTP: Dynamic Programming

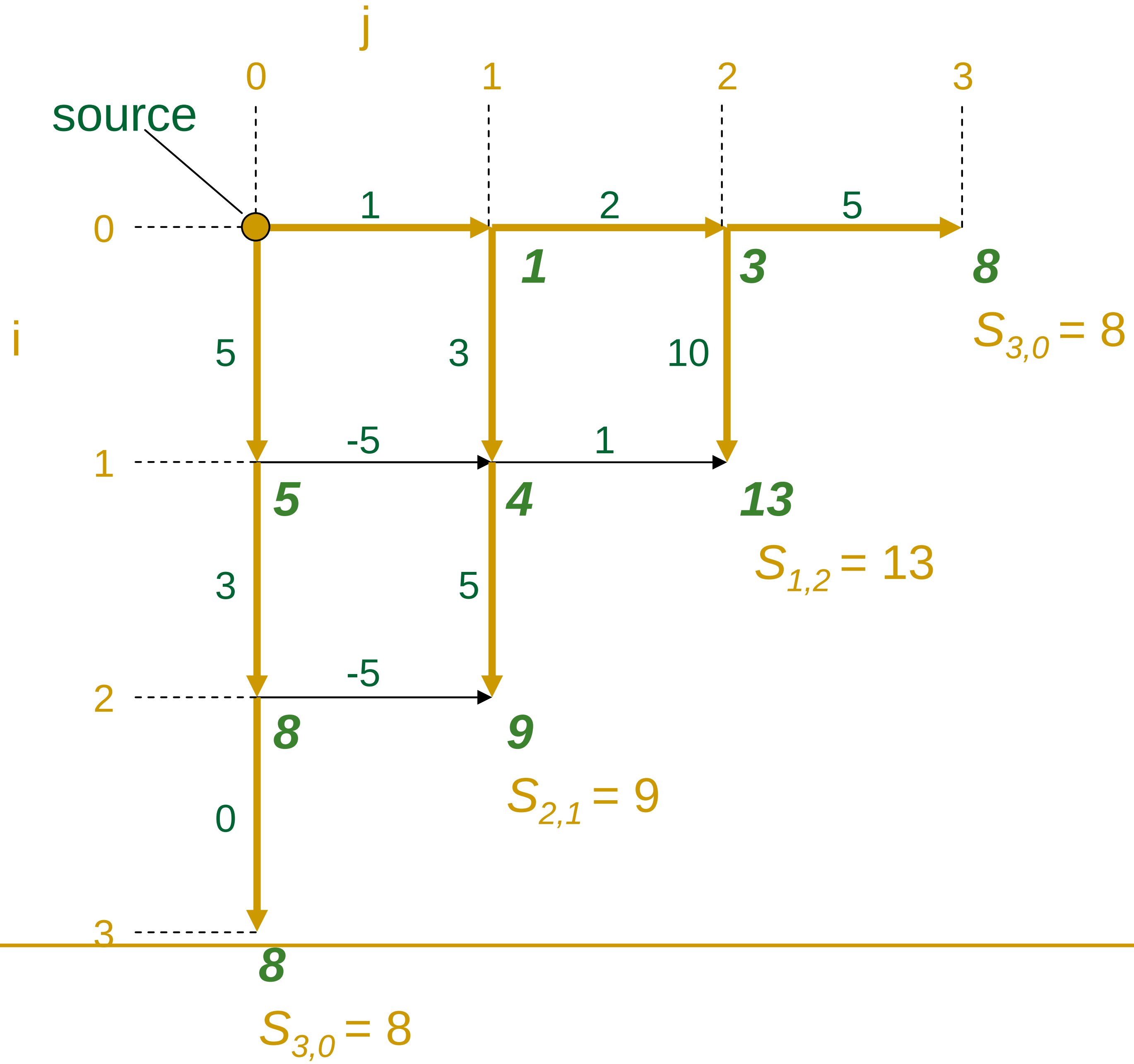


- Calculate optimal path score for each vertex in the graph
- Each vertex's score is the maximum of the prior vertices score plus the weight of the respective edge in between

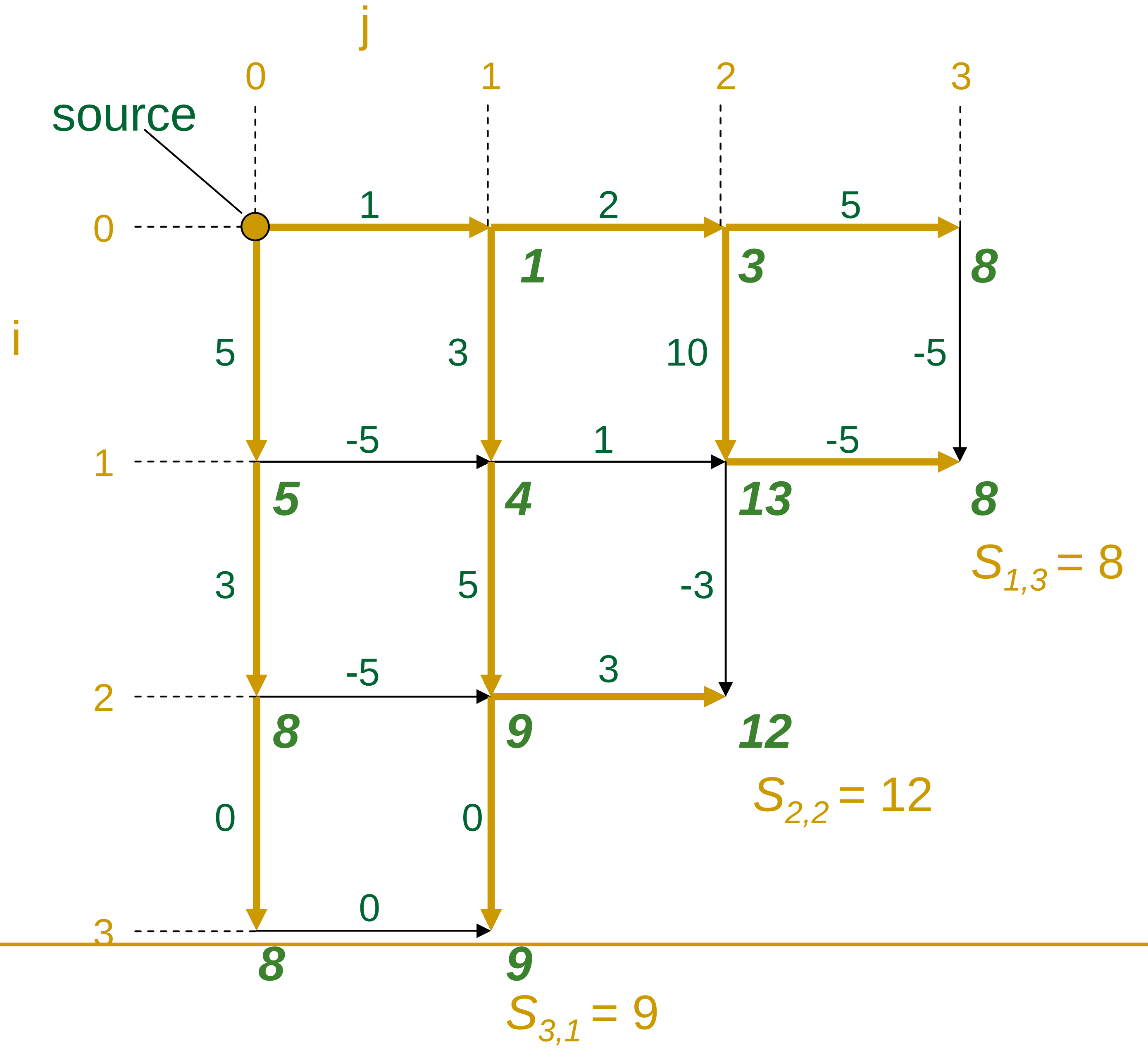
MTP: Dynamic Programming (cont'd)



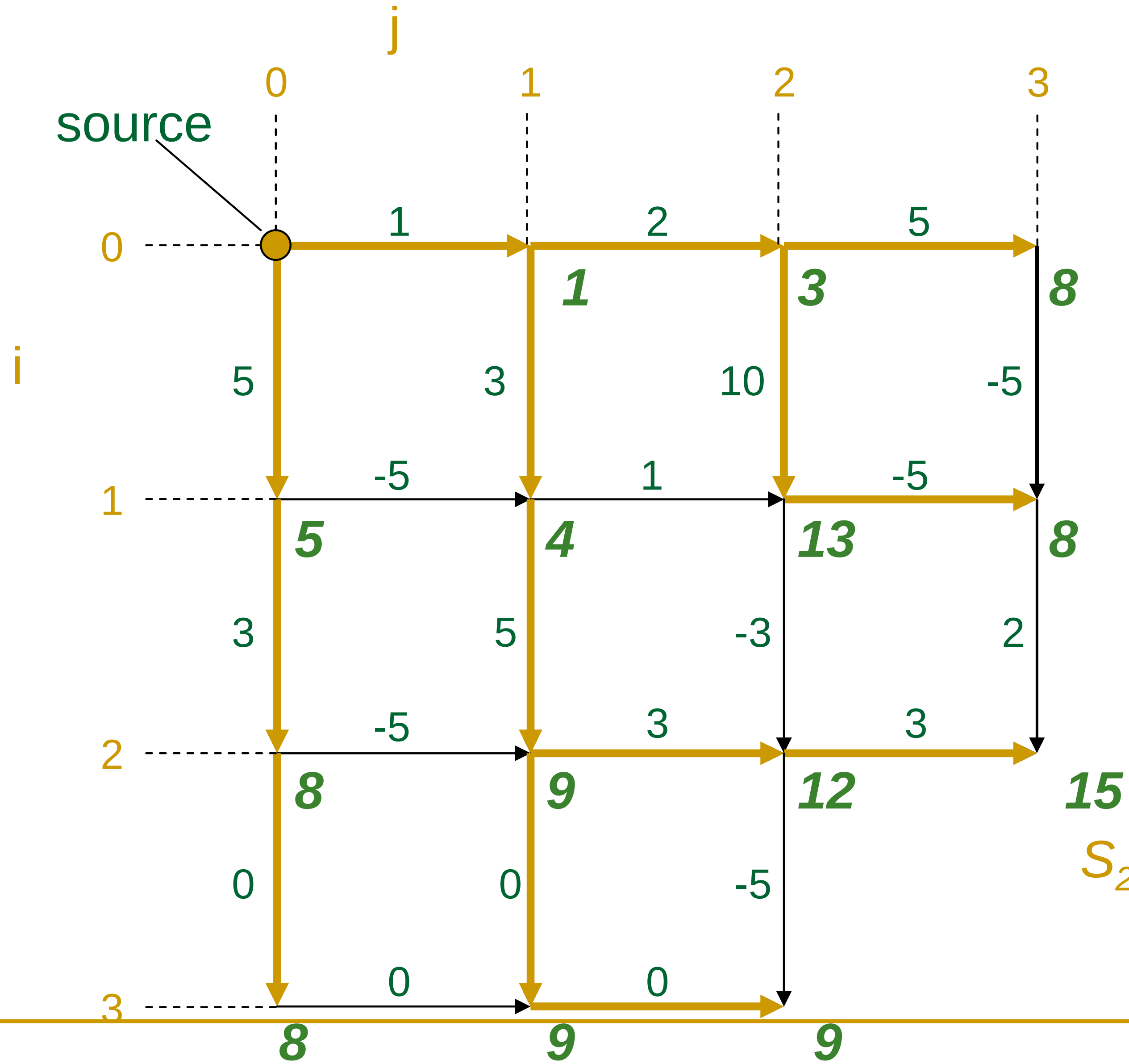
MTP: Dynamic Programming (cont'd)



MTP: Dynamic Programming (cont'd)



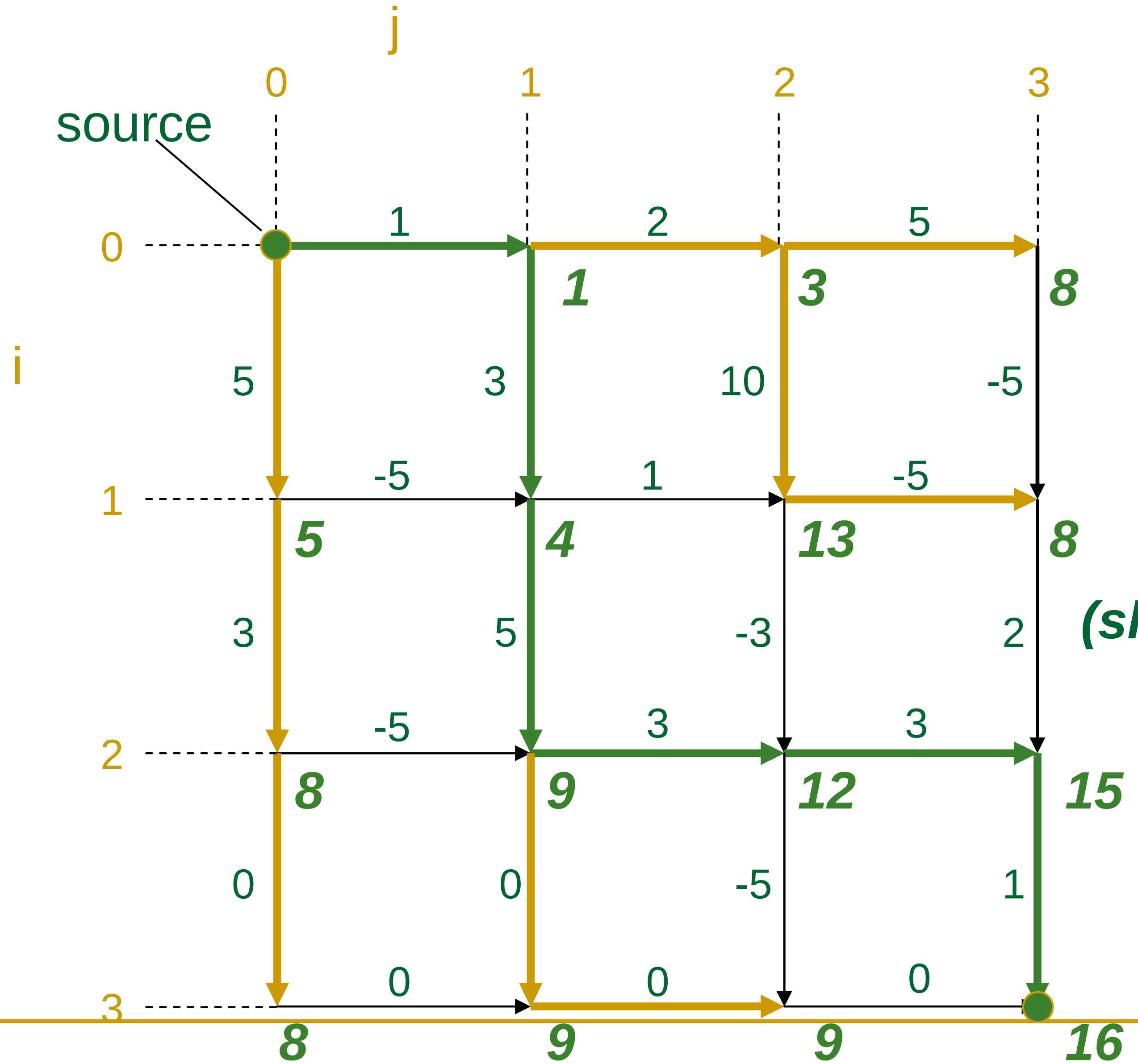
MTP: Dynamic Programming (cont'd)



$S_{2,3} = 15$

$S_{3,2} = 9$

MTP: Dynamic Programming (cont'd)



Done!

(showing all back-traces)

$S_{3,3} = 16$

MTP: Recurrence

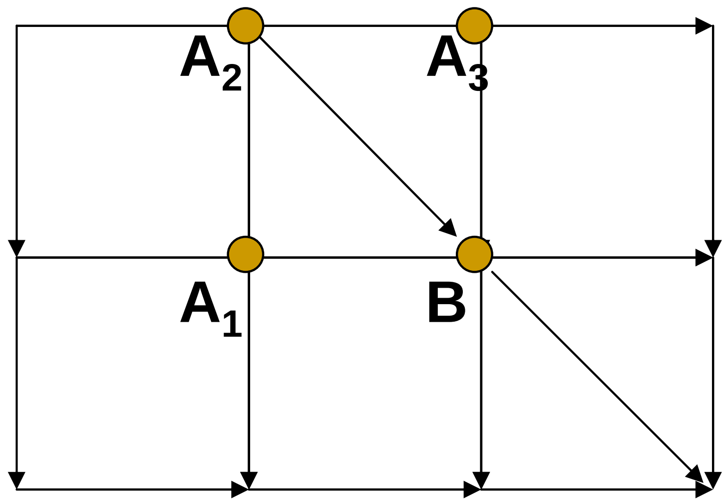
Computing the score for a point *(i,j)* by the recurrence relation:

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

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 then 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:

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

- Predecessors (*x*) = set of vertices that have edges leading to *x*

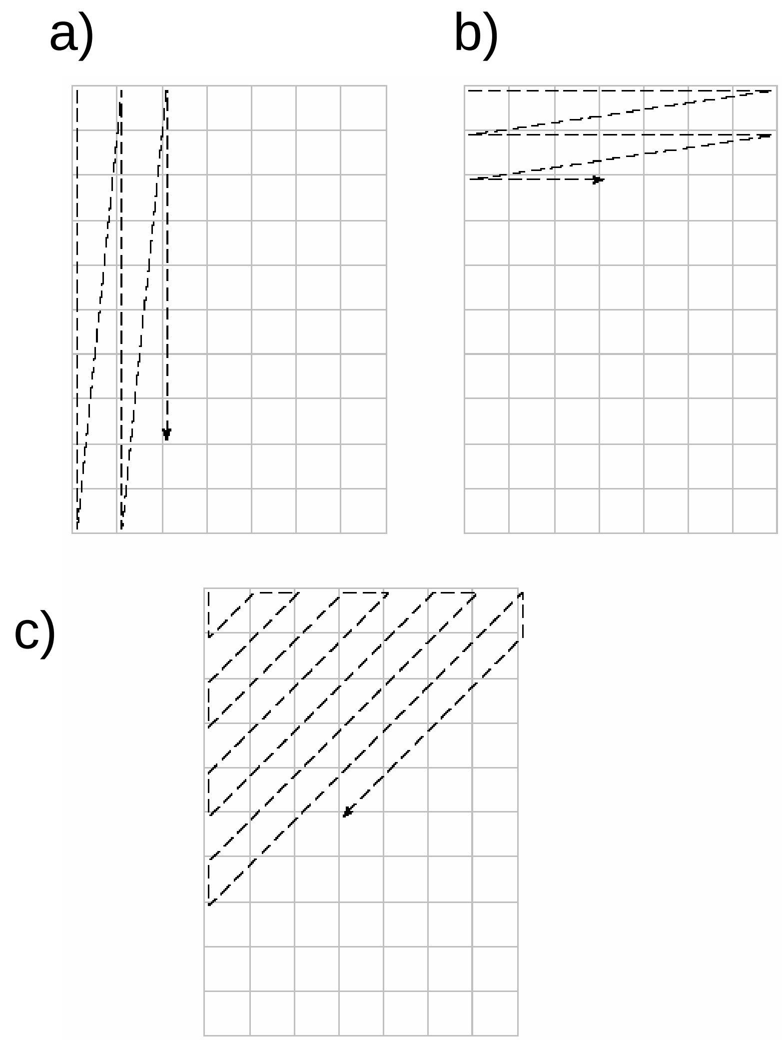
- The running time for a graph G(***V***, ***E***) (***V*** is the set of all vertices and ***E*** is the set of all edges) is O(***E***) since each edge is evaluated once

Traveling in the Grid

- The only hitch is that one must decide on the order in which to 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

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**:

v : ATCTGAT $\quad m = 7$

w : TGCATA $\quad n = 6$

Alignment : 2 * ***k*** matrix (***k*** ≥ max(***m***, ***n*** ***))***

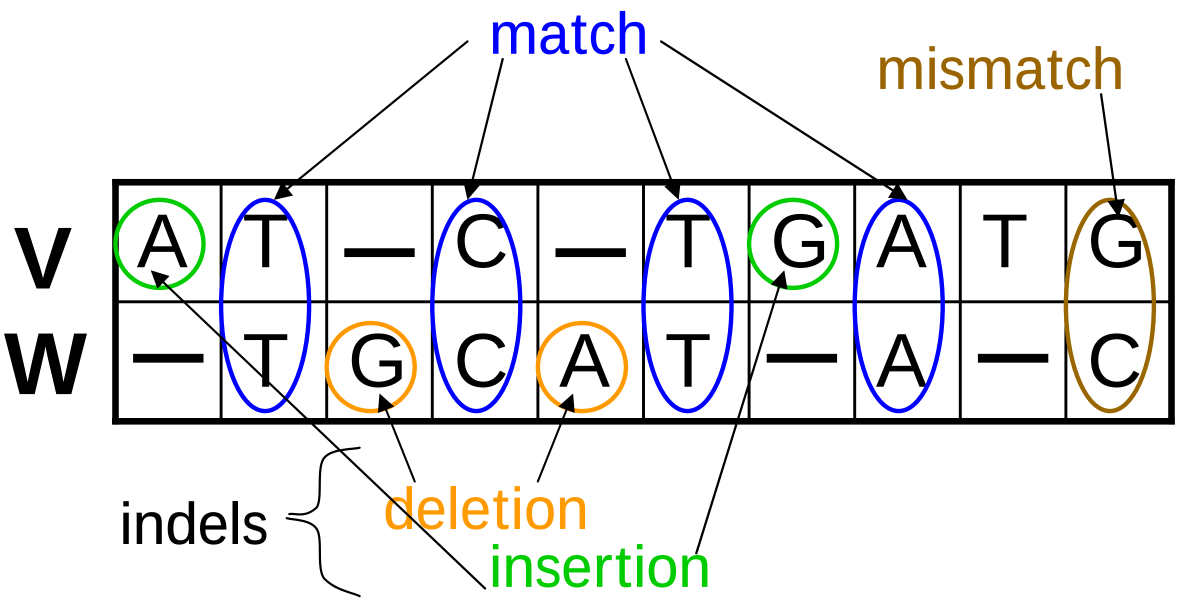
letters of ***v***	A	T	--	G	T	T	A	T	--
letters of ***w***	A	T	C	G	T	--	A	--	C

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

Note: insertions and deletions are together called indels

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

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

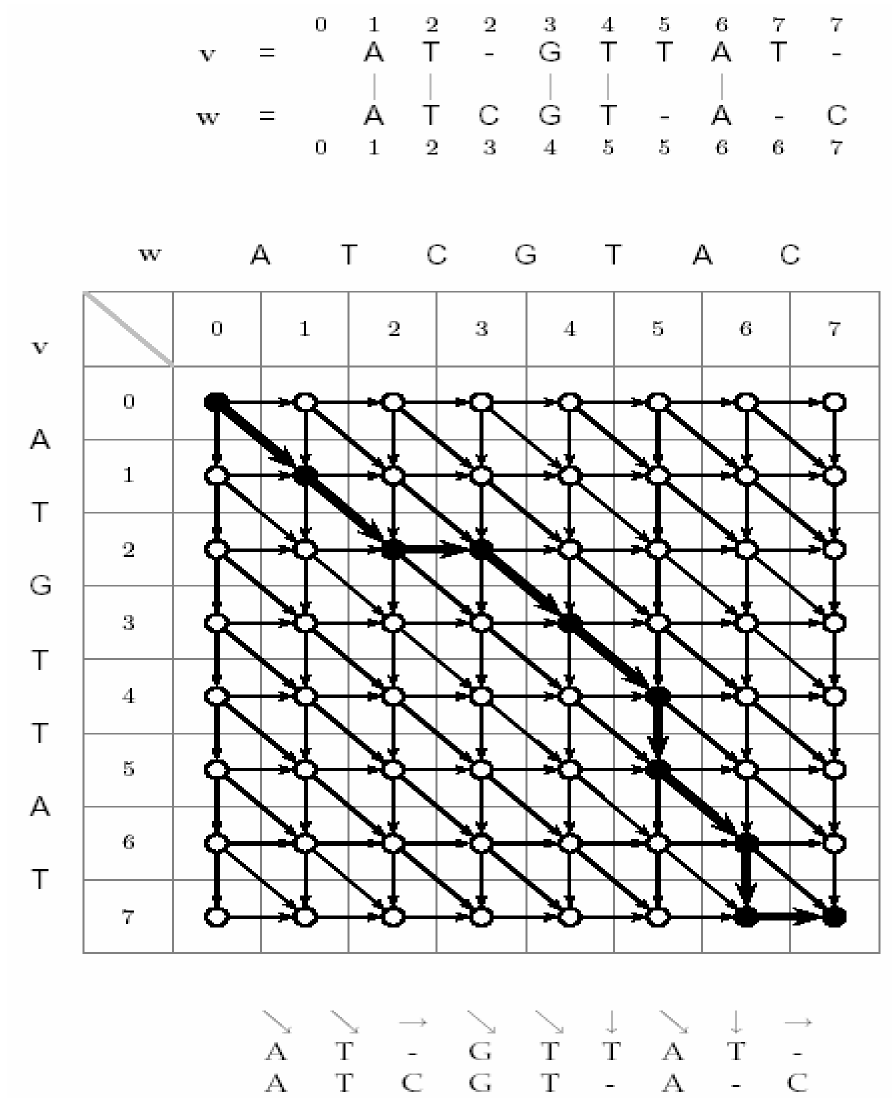
Every common subsequence is a path in 2-D grid

LCS: Dynamic Programming

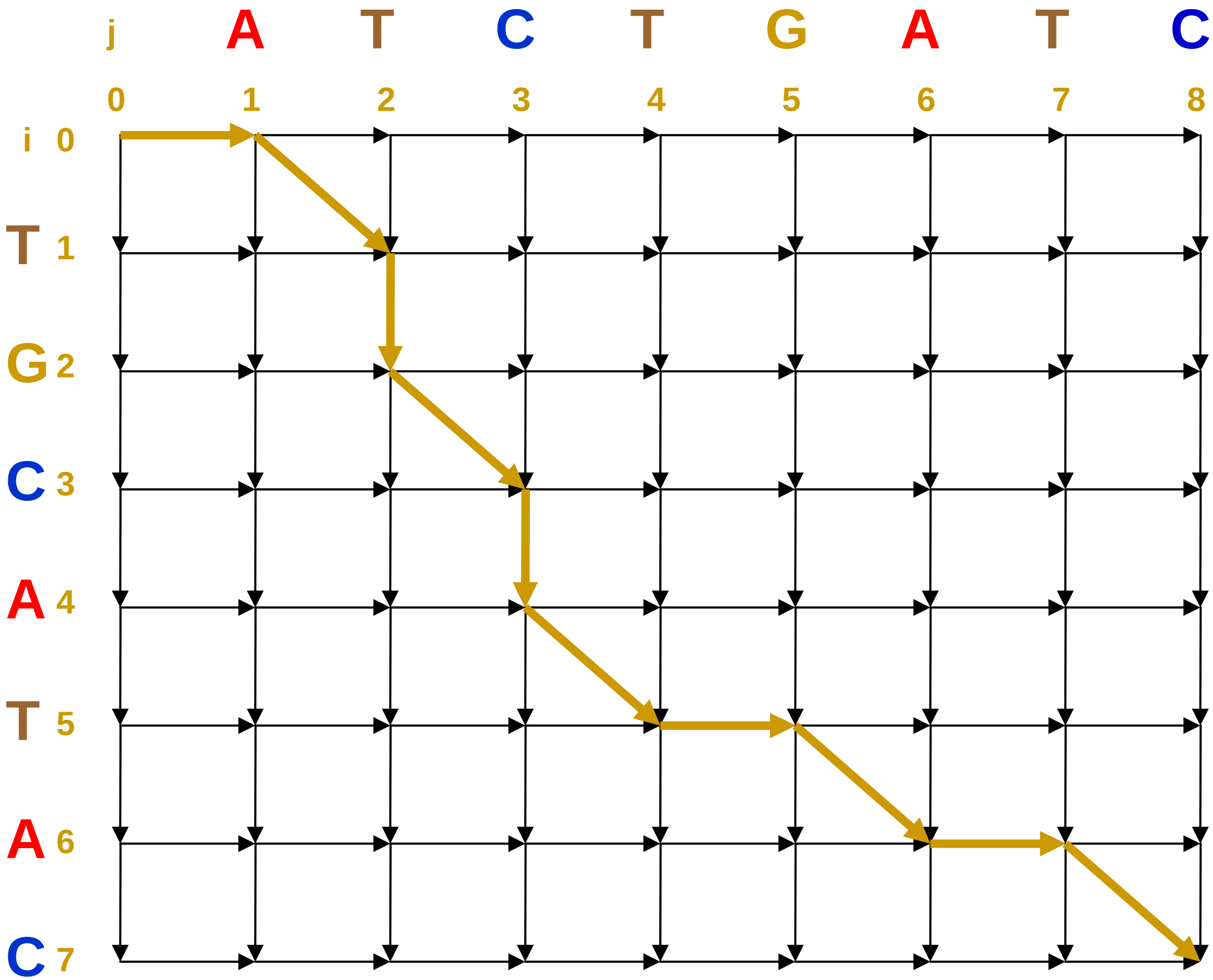
- Find the LCS of two strings

  Input: A weighted graph *G* with two distinct vertices, one labeled "*source*" one labeled "*sink*"

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



LCS Problem as Manhattan Tourist Problem



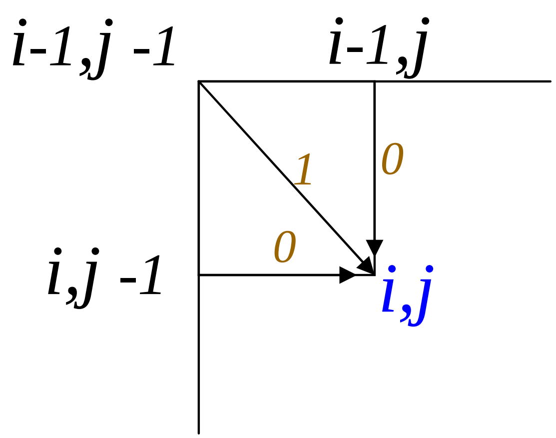
Computing LCS

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

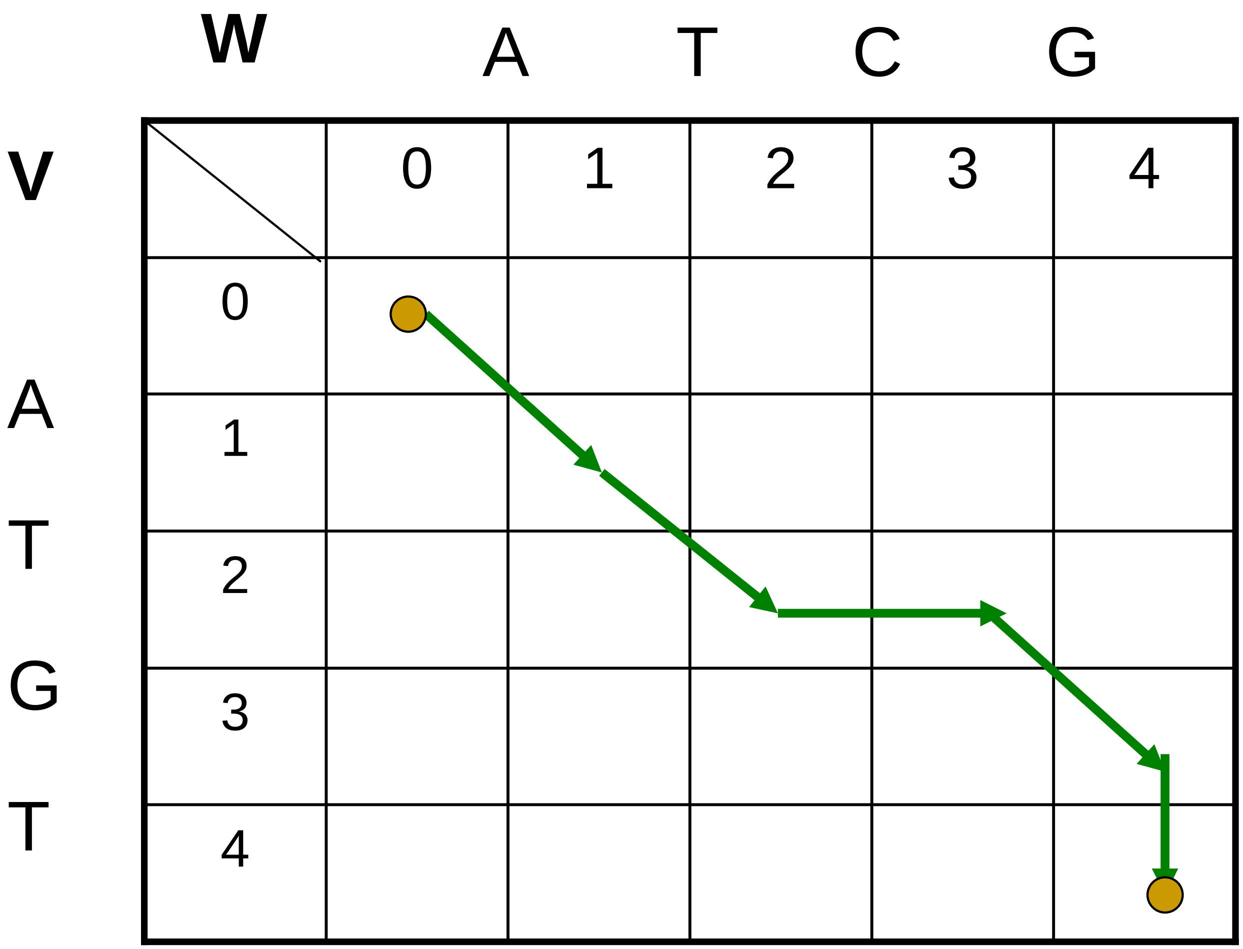
and $\boldsymbol{w}_j$ = prefix of $\boldsymbol{w}$ of length j: $w_1 \ldots w_j$

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

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



Every Path in the Grid Corresponds to an Alignment



0 1 2 2 3 4

V = A T - G T

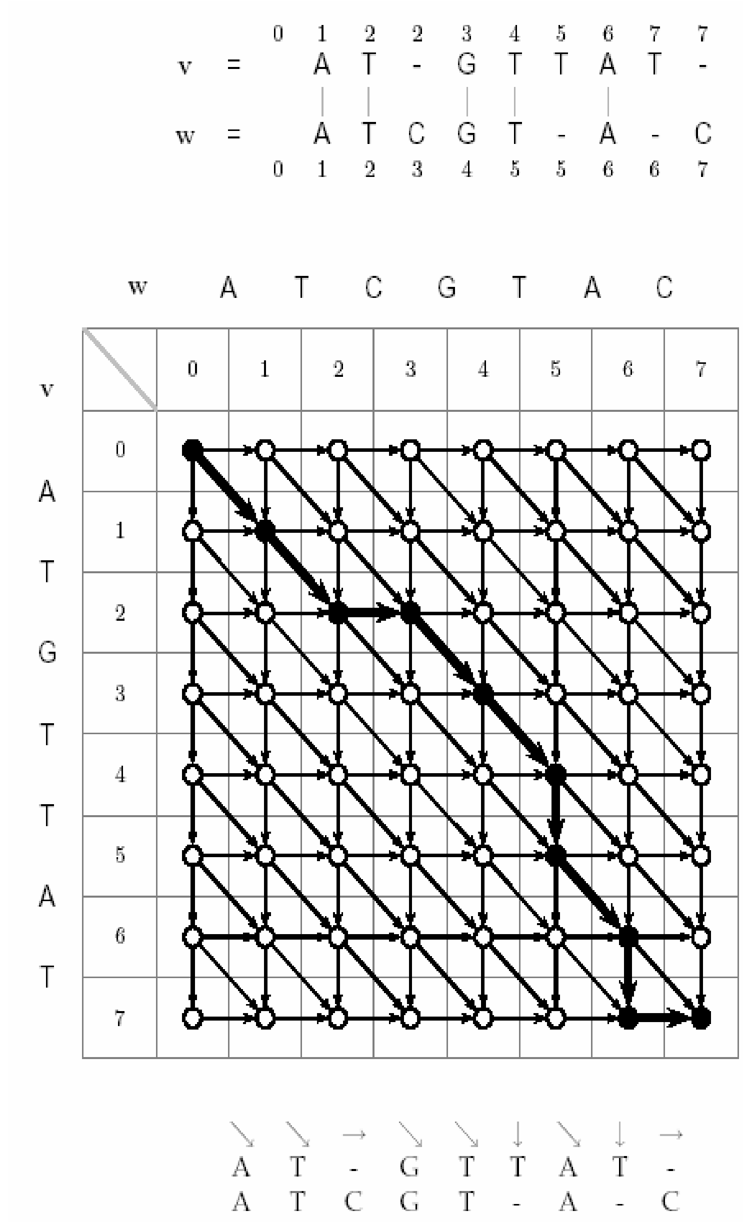
| | |

W= A T C G –

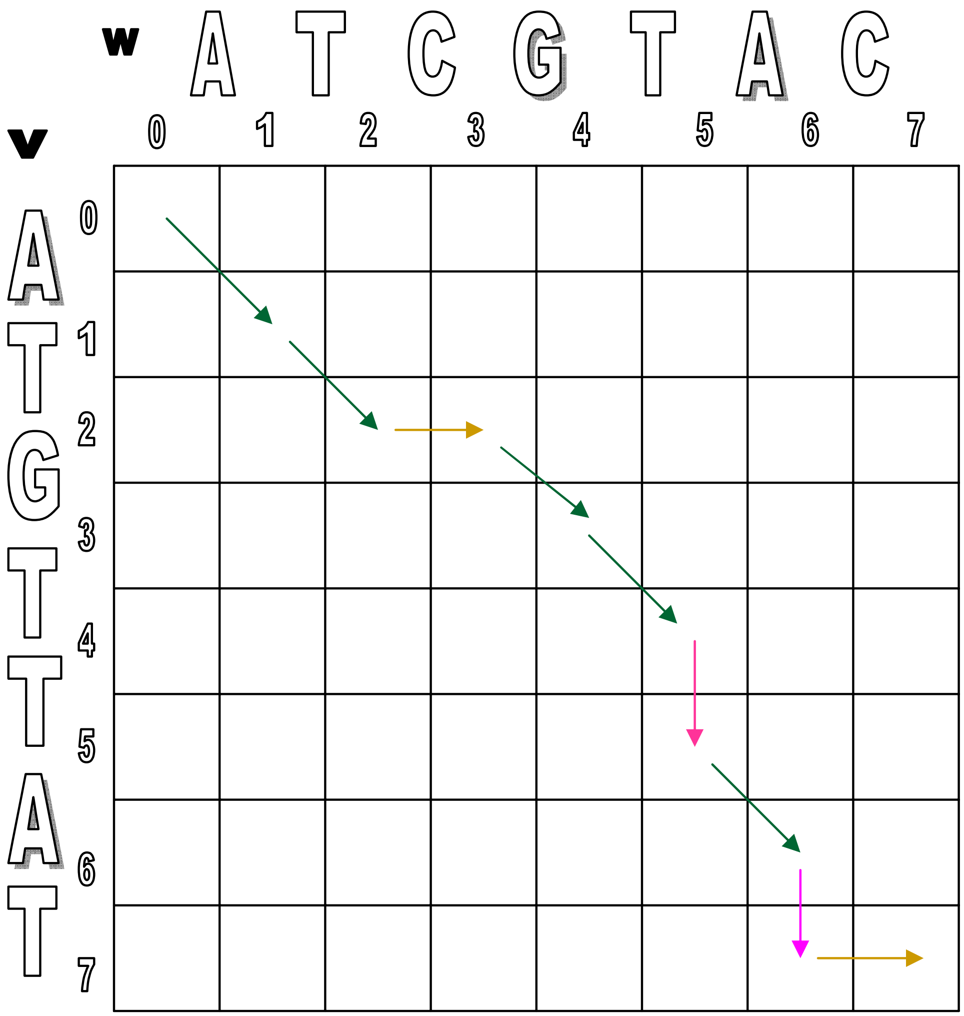
0 1 2 3 4 4

The Alignment Grid

- Every alignment path is from source to sink



Alignments in Edit Graph (cont'd)

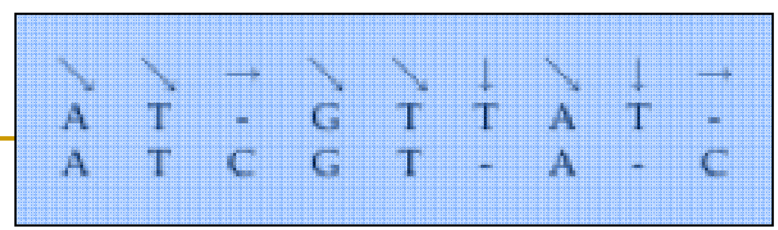


↓ and → represent indels in **v** and **w** with score 0.

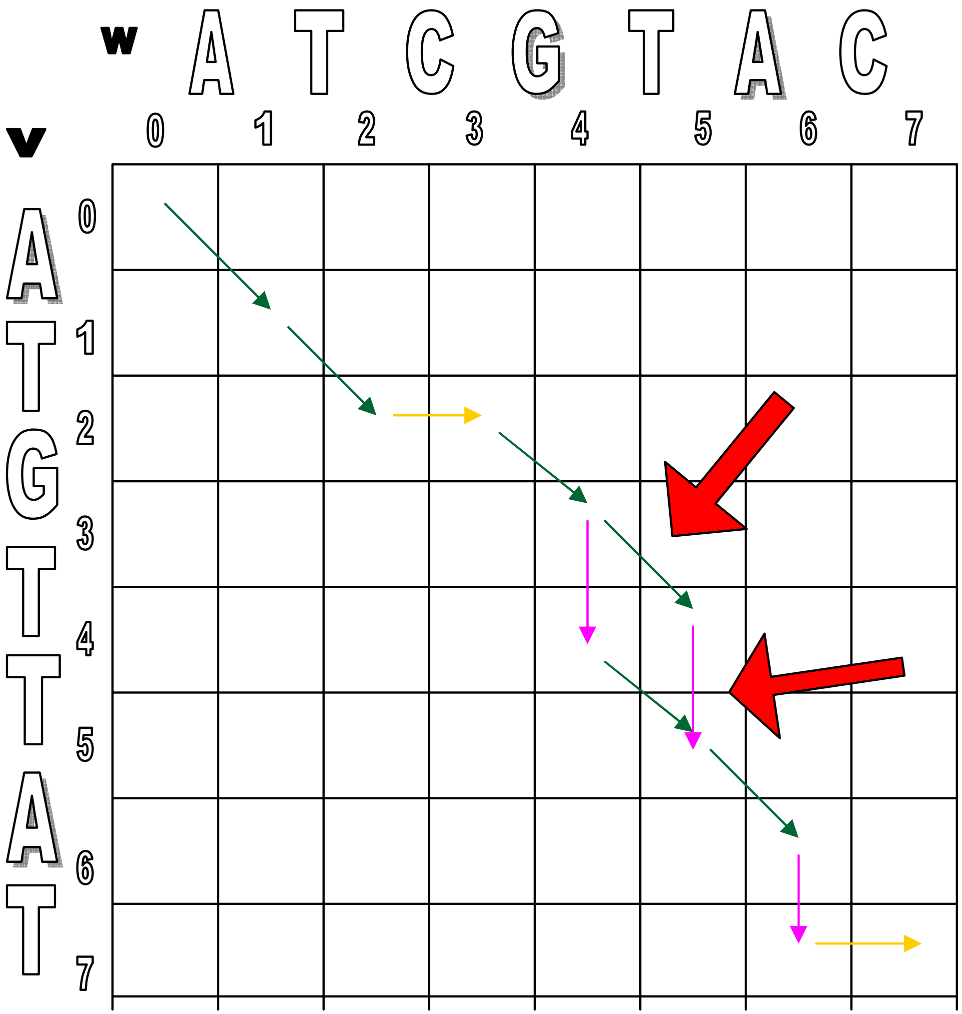
↘ represent matches with score 1.

- The score of the alignment path is 5.

Every path in the edit graph corresponds to an alignment:



Alignment as a Path in the Edit Graph



<u>Old Alignment</u>

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

<u>New Alignment</u>

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

Dynamic Programming Example

v \ w			A	T	C	G	T	A	C
		0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0	0
A	1	0							
T	2	0							
G	3	0							
T	4	0							
T	5	0							
A	6	0							
T	7	0							

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

v \ w			A	T	C	G	T	A	C
		0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0	0
A	1	0	↖ 1	← 1	← 1	← 1	← 1	↖ 1	← 1
T	2	0	↑ 1						
G	3	0	↑ 1						
T	4	0	↑ 1						
T	5	0	↑ 1						
A	6	0	↖ 1						
T	7	0	↑ 1						

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

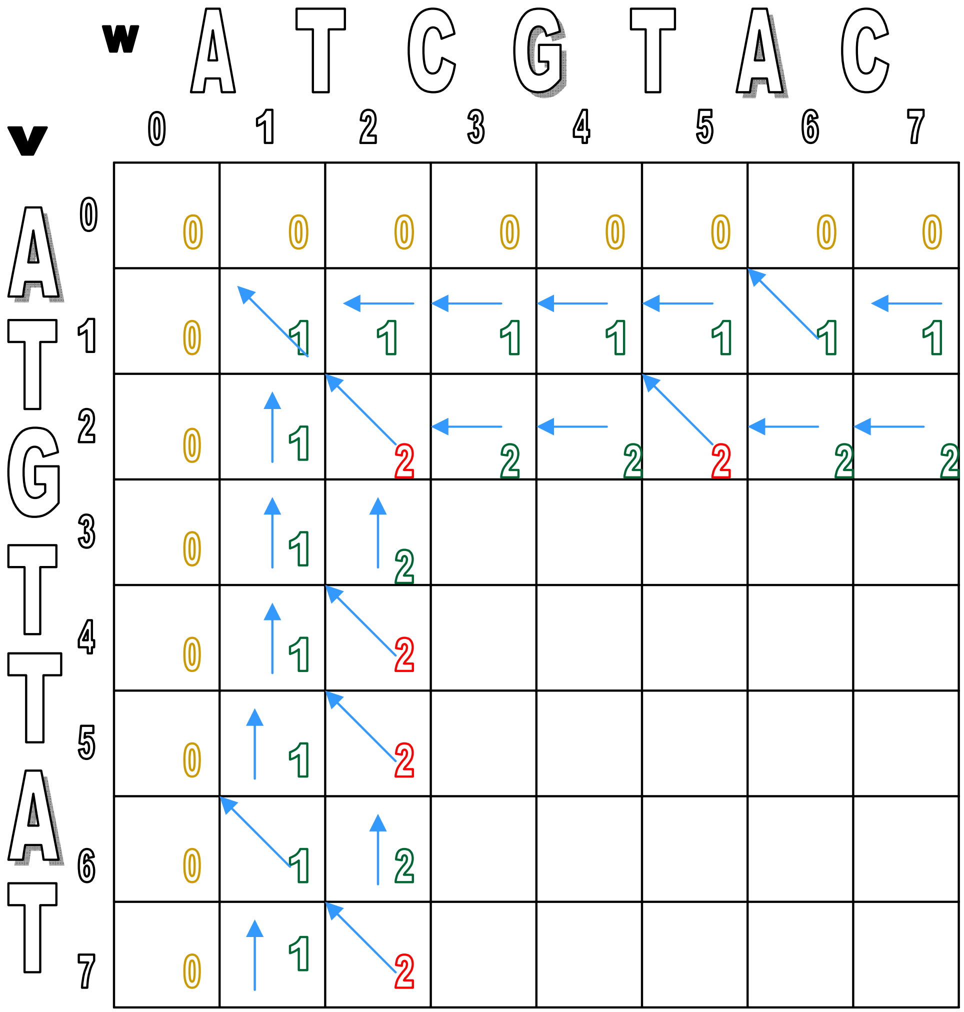
Arrows ↖ ↑ ← show where the score originated from.

↑ if from the top

← if from the left

↖ if $v_i = w_j$

Backtracking Example



Find a match in row and column 2.

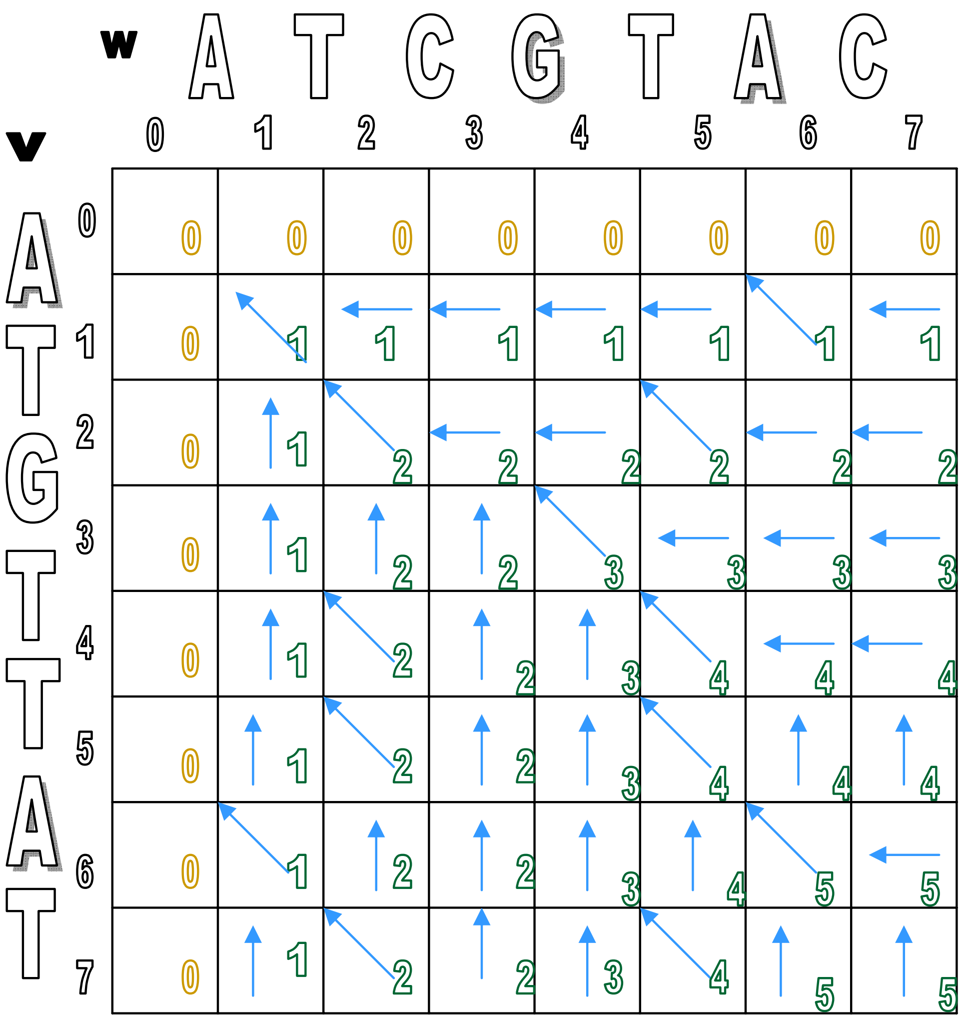
i=2, j=2,5 is a match (T).

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.

LCS Algorithm

<u>LCS(v,w)</u>

for $i = 1$ to n

$\quad s_{i,0} = 0$

for j = 1 to m

$\quad s_{0,j} = 0$

for $i = 1$ to n

$\quad$ for $j = 1$ to m

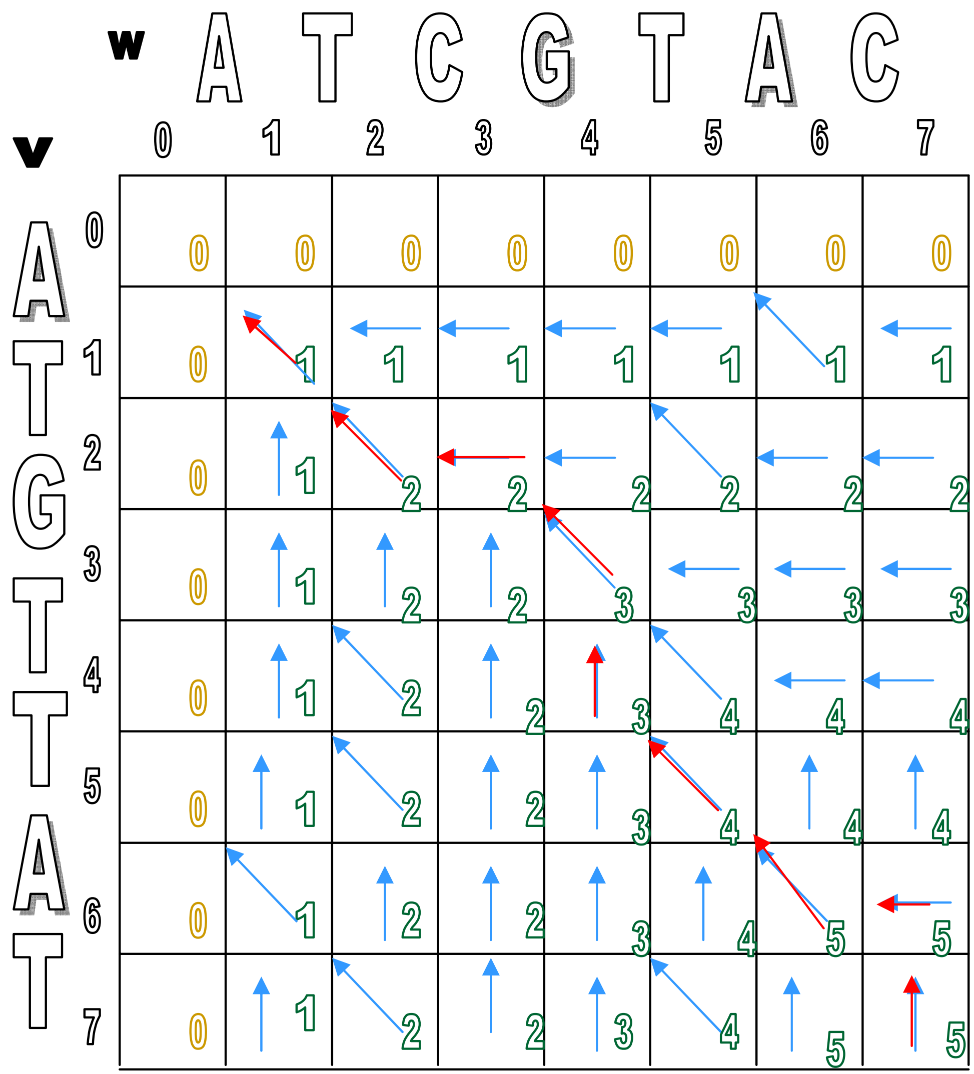
$$s_{i,j} = \max \begin{cases} s_{i-1,j} \\ s_{i,j-1} \\ s_{i-1,j-1} + 1, \text{ if } v_i = w_j \end{cases}$$

$$b_{i,j} = \begin{cases} \text{“}\uparrow\text{”} & \text{if } s_{i,j} = s_{i-1,j} \\ \text{“}\leftarrow\text{”} & \text{if } s_{i,j} = s_{i,j-1} \\ \text{“}\nwarrow\text{”} & \text{if } s_{i,j} = s_{i-1,j-1} + 1 \end{cases}$$

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

```
PrintLCS(b,v,i,j)
    if  i = 0 or j = 0
        return
    if b_{i,j} = "↖"
        PrintLCS(b,v,i-1,j-1)
        print v_i
    else
        if b_{i,j} = "↑"
            PrintLCS(b,v,i-1,j)
        else
            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.

Summary

- The running times of algorithms is important!
  - If it doesn't scale up, it won't be useful, especially in bioinformatics
- Recursion is a basic technique which is useful for breaking down problems into simpler ones
- Dynamic programming, which uses recursion, is often used in bioinformatics as well
  - Shown to be mathematically accurate
  - However, it can be inefficient for more than two sequences
    - BLAST and FASTA use heuristics (human-like techniques to speed up the computations)