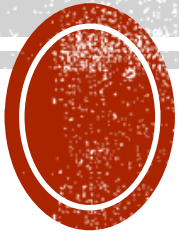
Lect 03. Dynamic Programming

Spring, 2020

School of Computer Science & Engineering

Pusan National University



Dynamic Programming

- Divide and conquer is suitable to solve divided problems which are independent.
- In Fibonacci algorithm, divided problems are related each other. To apply it to divide-and-conquer is not efficient because same problem is computed more than once time. Therefore divie-and-conquer algorithm is not suitable in this case.
- **Dynamic programming** is a **bottom-up** approach since the solution is constructed from the bottom up in the array. This method also solves **divided problem** first after dividing the problem like as divide-and-conquer. However result of solved problem is **stored** and **used it** in case needed to solve the problem again instead of recalculating it

Dynamic Programming

- There are two steps in the development of this approach.
  - First, *establish* the recursive property that gives the solution to an instance of the problem.
  - Second solve an instance of the problem in a *bottom-up* fashion by solving smaller instances first.

The Binomial Coefficient

- Formula to solve binomial coefficient

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \text{ for } 0 \le k \le n$$

- We cannot compute the binomial coefficient directly from this definition because $n!$ and $k!$ is very large, even for moderate values of n.

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k}, & 0 < k < n \\ 1, & k = 0 \text{ or } k = n. \end{cases}$$

Algorithm: Binomial Coefficient using Divide-and-Conquer

- Problem: Compute the binomial coefficient
- Inputs : nonnegative integer n and k, where $k \leq n$
- Outputs: bin, the binomial coefficient $\binom{n}{k}$
- Algorithm:

```
int bin(int n, int k) {
    if (k == 0 || n == k)
      return 1;
    else
      return bin(n-1,k-1) + bin(n-1,k)
}
```

Algorithm: Binomial Coefficient using Divide-and-Conquer

- Analysis of time complexity :
  - Divide-and-Conquer is simple to write but not efficient
  - Reason : same calculation is repeated when recursive call
  - Fore example, `bin(n-1,k-1)` and `bin(n-1,k)` are needed the result of `bin(n-2,k-1)`, so this term is calculated twice separately.
  - To determine $\binom{n}{k}$, the number of terms is $2\binom{n}{k} - 1$.

- **Proof**: (Mathematical induction for n)
- Induction base: # of term n=1, show that # of term is $2\binom{n}{k} - 1 = 2 \times 1 - 1 = 1$. The # of terms to calculate $\binom{1}{k}$ is always 1 when k=0 or 1.
- Induction hypothesis: Assume that the # of terms $\binom{n}{k}$ is $2\binom{n}{k} - 1$.
- Induction step: show that # of term $\binom{n+1}{k}$ is $2\binom{n+1}{k} - 1$.
- By the algorithm $\binom{n+1}{k} = \binom{n}{k-1} + \binom{n}{k}$, the total # of terms for determining $\binom{n+1}{k}$ is [# of terms for $\binom{n}{k-1}$]+[# of terms for $\binom{n}{k}$] +1. $\binom{n}{k-1} = 2\binom{n}{k-1} - 1$ and $\binom{n}{k} = 2\binom{n}{k} - 1$ by induction hypothesis.
- Therefore, the total # of terms is

$$
\begin{aligned}
&2\begin{bmatrix} n \\ k-1 \end{bmatrix} - 1 + 2\begin{bmatrix} n \\ k \end{bmatrix} - 1 + 1 \\
&= 2\left(\frac{n!}{(k-1)!(n-k+1)!} + \frac{n!}{k!(n-k)!}\right) - 1 \\
&= 2\left(\frac{n!(k+n+1-k)}{k!(n+1-k)!}\right) - 1 \\
&= 2\left(\frac{n!(n+1)}{k!(n+1-k)!}\right) - 1 \\
&= 2\left(\frac{(n+1)!}{k!(n+1-k)!}\right) - 1 \\
&= 2\begin{bmatrix} n+1 \\ k \end{bmatrix} - 1
\end{aligned}
$$

Dynamic Programming Approach

1. ***Establish*** a recursive property:

To construct array B, value of $\binom{i}{j}$ is stored in $B[i][j]$ and the value can be computed by the recurrence;

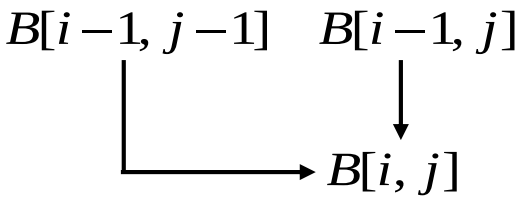
$$B[i][j] = \begin{cases} B[i-1][j-1] + B[i-1][j] & \text{if } 0 < j < j \\ 1 & \text{if } j = 0 \text{ or } j = i \end{cases}$$

Dynamic Programming Approach

2. ***Solve*** an instance of the problem in a bottom-up fashion by computing the rows in B in sequence starting with the first row. Each successive row is computed from the row preceding it using the recursive property established in step 1. The final value computed, $B[n][k]$, is $\binom{n}{k}$.

$\binom{n}{k}$

	0	1	2	3	4		j	k
0	1							
1	1	1						
2	1	2	1					
3	1	3	3	1				
4	1	4	6	4	1			
i								
n								



Binomial Coefficient using Dynamic Programming

- Problem: Compute the binomial coefficient.
- Inputs: nonnegative integers n and k, where $k \leq n$.
- Outputs: bin2, the binomial coefficient $\binom{n}{k}$.
- Algorithm:

```
int bin2(int n, int k) {
  index i, j;
  int B[0..n][0..k];
  for(i=0; i <= n; i++)
      for(j=0; j <= minimum(i,k); j++)
       if (j==0 || j == i)
         B[i][j] = 1;
       else B[i][j] = B[i-1][j-1] + B[i-1][j];
   return B[n][k];
}
```

Analysis of Binomial Coefficient

- Basic operation: statement in for-*j* loop
- Input size: *n*, *k*

i = 0, # of passes j-loop : 1
i = 1, # of passes j-loop : 2
i = 2, # of passes j-loop : 3
……..

i = *k*-1, # of passes j-loop : *k*
i = *k*, # of passes j-loop : *k*+1
i = *k*+1, # of passes j-loop : *k*+1
……..

i = *n*, # of passes j-loop : *k*+1

Therefore the total # of passes:

$$1+2+3+\Lambda+k+\underbrace{(k+1)+\Lambda+(k+1)}_{n-k+1 \text{ times}} = \frac{k(k+1)}{2}+(n-k+1)(k+1)$$

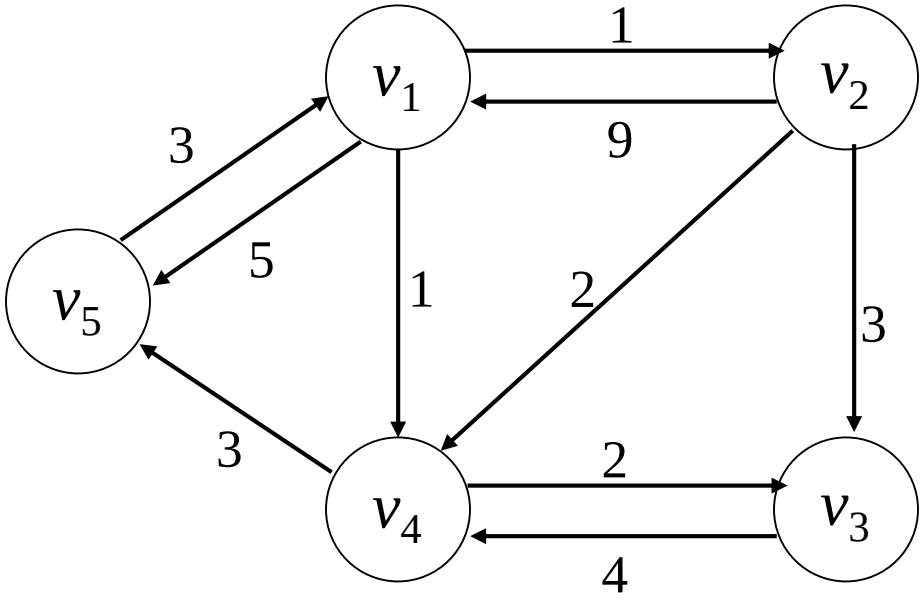
$$= \frac{(2n-k+2)(k+1)}{2} \in \Theta(nk)$$

Shortest Path Problem

Terminology of Graph

- Vertex, node
- Edge, arc
- Directed graph
- Weight, weighted graph
- Path, simple path – a seq. of vertex s.t. there is an edge from each vertex to its successor.
- Cycle – a path from a vertex to itself
- Cyclic graph vs acyclic graph
- length (of a path in a weighted graph) : the sum of the weights on the path

Example for weighted directed graph(digraph)



Shortest Path

- Example : Finding the fastest way to fly from one city to another when a direct flight does not exist.
- Problem: Finding the shortest path in directed graph with weight edge
- Optimization problem
  - There can be more than one candidate solution to an instance of an (optimization) problem. Each candidate solution has a value associated with it, and a solution to the instance is any candidate solution that has an **optimal value**. Depending on the problem, the optimal value is either the maximum or minimum of these lengths.
  - To determine the optimal solution is a **optimization problem**.
- The Shortest Paths problem is an **optimization problem**.

Brute-force algorithm for Finding Shortest Path

- Brute-force algorithm
  1. Finding a **length of all path** from a vertex to another.
  2. Determining a **minimum length** of them.
- Analysis:
  - Suppose there is an edge from every vertex to every other vertex.
  - Then a subset of all the paths from one vertex v_i to another vertex v_j is the set of all those paths that start at the first vertex, end at the other vertex, and pass through all the other vertices.
  - Because the second vertex on such a path can be any of *n-2* vertices, the third vertex on such a path can be any of *n-3* vertices, …, and the second-to-last vertex on such a path can be only one vertex, the total number of paths from one vertex to another vertex that pass through all the other vertices is
  - $(n\text{-}2)(n\text{-}3)\ldots 1 = (n\text{-}2)!$,
  - which is worse than exponential. Therefore this algorithm is definitely inefficient.

Data structure : Dynamic programming approach

	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0

W

	1	2	3	4	5
1	0	1	3	1	4
2	8	0	3	2	5
3	10	11	0	4	7
4	6	7	2	0	3
5	3	4	6	4	0

D

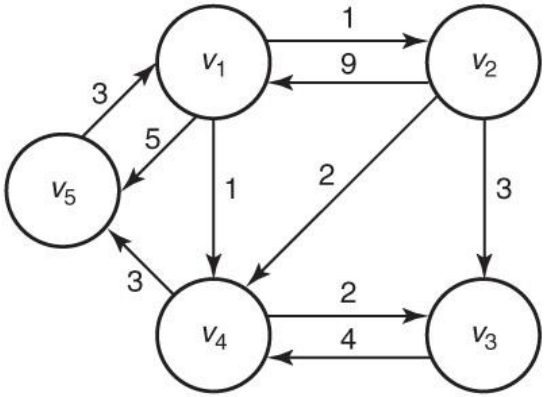
Figure 3.3: W represents the graph in figure 3.2 and D contains the lengths of the shortest paths. Our algorithm for the Shortest Paths problem computes the values in D from those in W.

Data structure : Dynamic programming approach

- adjacent matrix: W
- $$W[i][j] = \begin{cases} \text{weight on edge} & \text{if there is an edge from } v_i \text{ to } v_j \\ \infty & \text{if there is no edge from } v_i \text{ to } v_j \\ 0 & \text{if } i = j. \end{cases}$$

- Represent the lengths of the shortest paths in the graph: D
  - Creating a sequence of n+1 arrays $D^{(k)}, 0 \le k \le n$ and
  - where
  - $D^{(k)}[i][j]$ = length of a shortest path from v_i to v_j using only vertices in the set $\{v_1, v_2, \ldots, v_k\}$ as intermediate vertices.

Data structure



- Example 3.2:
  - W : adjacent matrix of graph
  - D : the lengths of the shortest paths

$W[i][j]$	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0

$D[i][j]$	1	2	3	4	5
1	0	1	3	1	4
2	8	0	3	2	5
3	10	11	0	4	7
4	6	7	2	0	3
5	3	4	6	4	0

- Calculate $D^{(k)}[2][5], 0 \leq k \leq 5$.
- See page 99.

Design procedure for dynamic programming

- $D^{(n)}[i][j]$: the length of a shortest path from v_i to v_j that is allowed to pass through any of the other vertices, it is the length of a shortest path from v_i to v_j.
- $D^{(0)} = W$ and $D^{(n)} = D$.
- Therefore to determine *D,* we find the method to obtain $D^{(n)}$ from $D^{(0)}$.

1. Establish a recursive property (process) with which we can compute $D^{(k)}$ from $D^{(k-1)}$.
2. Solve an instance of problem in a *bottom-up* fashion by repeating the process(established in step 1) for *k=1* to *n*. This create the sequence

   $D^0\ D^1\ D^2, \ldots, D^n$

   W D

Design procedure for dynamic programming

- 1. Establish a recursive property (process) with which we can compute $D^{(k)}$ from $D^{(k-1)}$.

$$D^{(k)}[i][j] = minimum(D^{(k-1)}[i][j], D^{(k-1)}[i][k] + D^{(k-1)}[k][j])$$

$case\,1$ $case\,2$

case 1: At least one shortest path from v_i to v_j, using only vertices in $\{v_1, v_2, \ldots, v_k\}$ as intermediate vertices, does not use v_k. Then

$$D^{(k)}[i][j] = D^{(k-1)}[i][j].$$

ex: $D^{(5)}[1][3] = D^{(4)}[1][3] = 3$

case 2: All shortest paths from v_i to v_j, using only vertices $\{v_1, v_2, \ldots, v_k\}$ as intermediate vertices, do use v_k.

$$D^{(k)}[i][j] = D^{(k-1)}[i][k] + D^{(k-1)}[k][j].$$

ex: $D^{(2)}[5][3] = D^{(1)}[5][2] + D^{(1)}[2][3] = 4 + 3 = 7$

- 2. use the recursive property in step1 to create the sequence of arrays as the following:

$$D^{(0)}, D^{(1)}, \ldots, D^{(n)}$$

Shortest path

- Example 3.3
  - See P. 101

Floyd's algorithm for shortest path 1

- Developed by Robert Floyd(1936~2001(65))
  - 1962
  - Turing award(1978)
- Problem: Compute the shortest paths from each vertex in a weighted graph to each of the other vertices. The weights are nonnegative numbers.
- Inputs: A weighted, directed graph and n, the # of vertices in the graph. Two-dimensional array W[i][j].
- Outputs:A two-dimensionall array D[i][j] where is the length of a shortest path from the ith vertex to jth vertex.

Floyd's algorithm 1

- Algorithm:

```
void floyd(int n, const number W[][],
           number D[][]) {
  int i, j, k;
  D = W;
  for(k=1; k <= n; k++)
    for(i=1; i <= n; i++)
      for(j=1; j <= n; j++)
        D[i][j] = minimum(D[i][j],
                          D[i][k]+D[k][j]);
}
```

- Every-case Time Complexity :
  - ✓ Basic operation: the instance in the for-*j* loop
  - ✓ Input size : *n*, the # of vertices in the graph

$$T(n) = n \times n \times n = n^3 \in \Theta(n^3)$$

Floyd's algorithm for shortest path 2

- Problem: Compute the shortest paths from each vertex in a weighted(nonnegative) graph to each of the other vertices. And create the **its paths**
- Inputs: A weighted, directed graph and n, the # of vertices in the graph. Two-dimensional array W[i][j].
- Outputs: A two-dimensional array D[i][j] where is the length of a shortest path from the *i*th vertex to *j*th vertex. And **an array P**, which has both its rows and columns indexed from 1 to n, where

$$P[i][j] = \begin{cases} \text{hightest index of an intermediate vertex on the shortest path from } v_i \text{ to } v_j, & \text{if at least one intermediate vertex exists.} \\ 0, & \text{if no intermediate vertex exists.} \end{cases}$$

Floyd's algorithm for shortest path 2

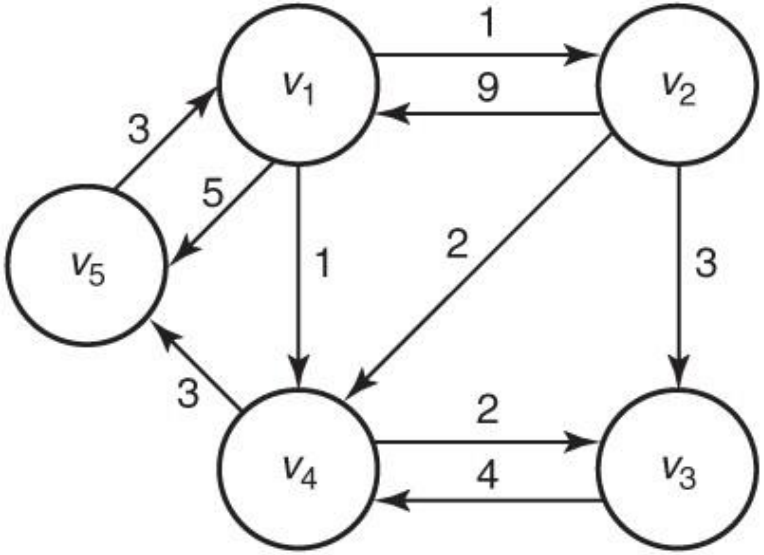
- Algorithm:

```
void floyd2(int n, const number W[][],
      number D[][], index P[][]) {
  index i, j, k;
   for(i=1; i <= n; i++)
     for(j=1; j <= n; j++)
      P[i][j] = 0;
  D = W;
   for(k=1; k<= n; k++)
     for(i=1; i <= n; i++)
      for(j=1; j<=n; j++)
        if (D[i][k] + D[k][j] < D[i][j]) {
         P[i][j] = k;
         D[i][j] = D[i][k] + D[k][j];
         }
}
```

Floyd's algorithm for shortest path 2

- Solve *D* and *P* applied to the graph.

$P[i][j]$	1	2	3	4	5
1	0	0	4	0	4
2	5	0	0	0	4
3	5	5	0	0	4
4	5	5	0	0	0
5	0	1	4	1	0



Print Shortest Path

- Problem : Print the intermediate vertices on a shortest path from one vertex to another vertex in a weighted graph
- Inputs the array P and two indices, q and r, of vertices in the graph
- Algorithm:

```
void path(index q,r) {
    if (P[q][r] != 0) {
     path(q,P[q][r]);
     count << " v" << P[q][r];
     path(P[q][r],r);
    }
}
```

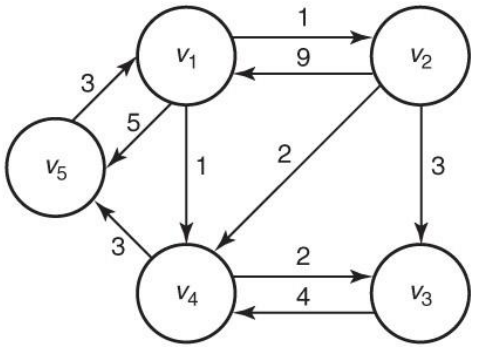
Print Shortest Path

- **Print** `path(5,3)` **given P**

```
path(5,3) = 4
    path(5,4) = 1
        path(5,1) = 0
        v1
        path(1,4) = 0
    v4
    path(4,3) = 0
```

Output: `v1 v4.`

	1	2	3	4	5
1	0	0	4	0	4
2	5	0	0	0	4
3	5	5	0	0	4
4	5	5	0	0	0
5	0	1	4	1	0



Therefore, the shortest path from v_5 to v_3 is a $\langle v_5, v_1, v_4, v_3 \rangle$.

Dynamic Programmng and Optimization Problems

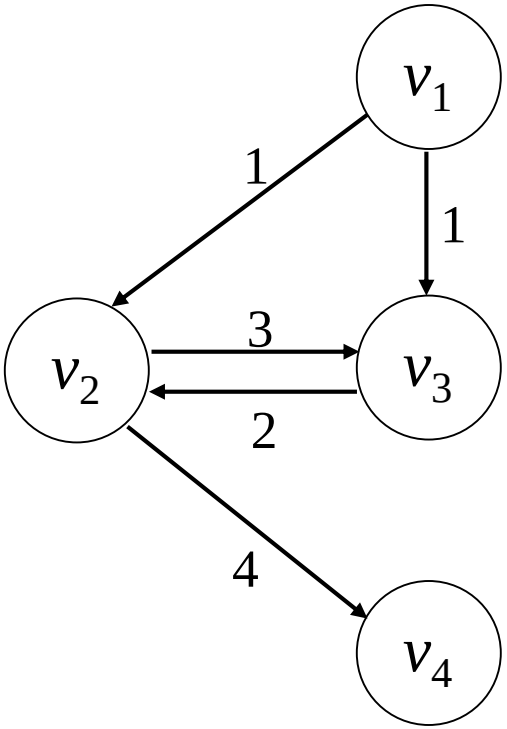
Third step in the development of a dynamic programming algorithm

1. ***Establish*** a recursive property that gives the optimal solution to an instance of the problem.
2. **Compute** the value of an **optimal solution** in a *bottom-up* fashion.
3. **Construct** an **optimal solution** in a *bottom-up* fashion.

Definition: the principle of optimality

- The ***principle of optimality*** is said to apply in a problem if an optimal solution to an instance of a problem always contains optimal solutions to all substances.
- Ex: In the case of the shortest paths problem we showed that if v_k is a vertex on an optimal path from v_i to v_j, the subpaths from v_i to v_k and from v_k to v_j must also be optimal.
- Therefore the problem is applied the **principle of optimality** and we can solve the problem using dynamic programming.

Not applied the principle: Longest Path Problem



- Longest path from v_1 to v_4: $[v_1, v_3, v_2, v_4]$
- However, the longest path from v_1 to v_3 is a $[v_1, v_2, v_3]$ not $[v_1, v_3]$.
- Therefore the principle of optimality does not apply.
- Notice : where simple path, i.e., without cycle, is considered.

Chained Matrix Multiplication

Chained Matrix Multiplication

- Suppose we want to multiply a 2x3 matrix times a 3 x 4 matrix

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 & 9 & 1 \\ 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 \end{bmatrix} = \begin{bmatrix} 29 & 35 & 41 & 38 \\ 74 & 89 & 104 & 83 \end{bmatrix}$$

- # of multiplications

  1x7+2x2+3x6 = 29 (3times)

  - 2 x 4 = 8 entries in the product, the total # of elementary multiplications is
  - 2 x 4 x 3 = 24

- In general, to multiply an ***ixj* matrix** times a ***jxk* matrix** using the standard method:

  ixjxk elementary multiplications

Chained Matrix Multiplication

- Consider the following matrix multiplication:

$$\begin{matrix} A & \times & B & \times & C & \times & D \\ 20 \times 2 & & 2 \times 30 & & 30 \times 12 & & 12 \times 8 \end{matrix}$$

- The order affect the # of mul.

A(B(CD)	30x12x8 + 2x30x8 + 20x2x8 =	3,680
(AB)(CD)	20x2x30 + 30x12x8 + 20x30x8 =	8,880
A((BC)D)	**2x30x12 + 2x12x8 + 20x2x8 =**	**1,232**
((AB)C)D	20x2x30 + 20x30x12 + 20x12x8 =	10,320
(A(BC))D	2x30x12 + 20x2x12 + 20x12x8 =	3,120

- Goal : to develop an algorithm that determines the **optimal order** for multiplying *n* matrices.

Brute-force:Chained Matrix Multiplication

- Algorithm: to consider all possible orders and take the minimum.
- Time complexity analysis : at least exponential-time
- Proof :
  - Let t_n : the # of different orders which can multiply n matrices :$A_1, A_2, \dots, A_n$.
  - A subset of all the orders is the set of orders for which $A_1$is the last matrix multiplied. The # of different orders in this subset is t_{n-1} , because it is the # of different orders with which we can multiply A_2 throgh A_n:

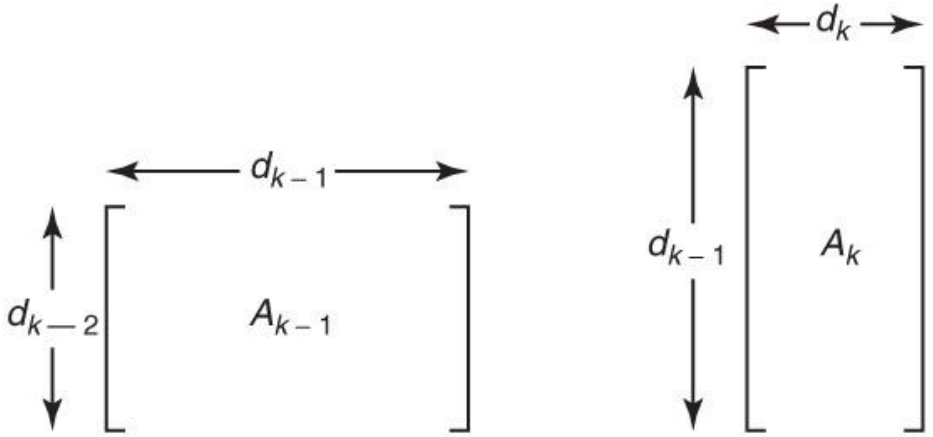
$$A_1(\quad A_2A_3 \ \dots A_n \quad)$$

t_{n-1} different orders

  - A second subset of all the orders is the set of orders for which A_n is the last matrix multiplied. Clearly, the # of different orders in this subset is also t_{n-1} .
  - Therefore $t_n \geq t_{n-1} + t_{n-1} = 2\ t_{n-1}$, where $t_1 = 1$.
  - $\therefore t_n \geq 2t_{n-1} \geq 2^2 t_{n-2} \geq \ \dots \geq 2^{n-2} = \Theta(2^n)$.

Chained Matrix Multiplications : dynamic programming approaches

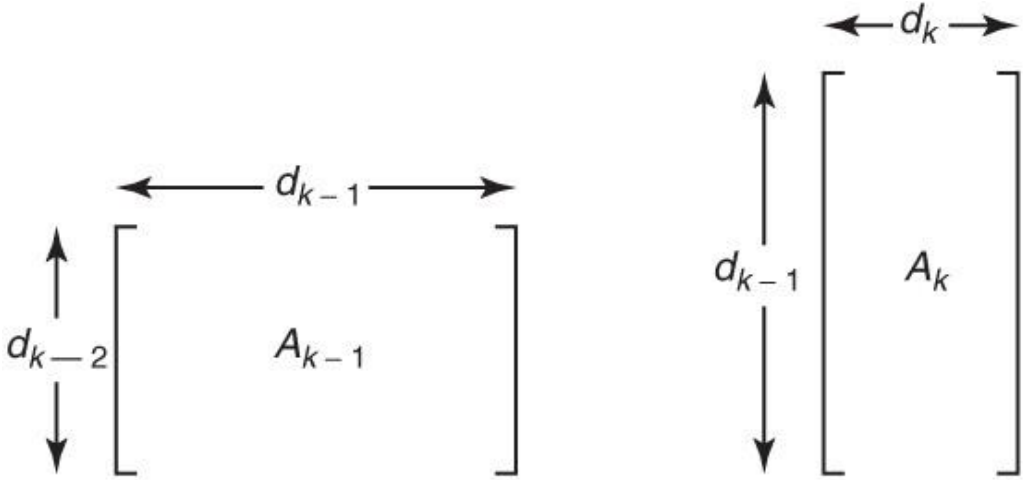
- **Ex.** $A_{k-1} \times A_k$.
  - The # of columns in A_{n-1} == the # of rows in A_n .
  - Let d_0: the # of rows in A_1 and d_k: the # of column of matrix A_k, for $1 \leq k \leq n$.
  - The dimension of A_k is $d_{k-1} \times d_k$.



Chained Matrix Multiplications : dynamic programming approaches

- Ex. $A_{k-1} \times A_k$.
  - For $1 \le i \le j \le n$. Let

$$\begin{cases} M[i][j] = \text{minimum } \# \text{ of multi. needed to multiply } A_i \text{ through } A_j, \text{ if } i < j, \\ M[i][i] = 0. \end{cases}$$



Example 3.5

- Example:

A_1	A_2	A_3	A_4	A_5	A_6
5×2	2×3	3×4	4×6	6×7	7×8
$d_0\ d_1$	$d_1\ d_2$	$d_2\ d_3$	$d_3\ d_4$	$d_4\ d_5$	$d_5\ d_6$

- To multiply $A_4, A_5,$ and A_6, we have following two orders and # of elementary multiplications:

$(A_4\ A_5)A_6$ # of multi. $= d_3 \times d_4 \times d_5 + d_3 \times d_5 \times d_6 = 4 \times 6 \times 7 + 4 \times 7 \times 8 = 392$

$A_4(\ A_5A_6)$ # of multi. $= d_4 \times d_5 \times d_6 + d_3 \times d_4 \times d_6 = 6 \times 7 \times 8 + 4 \times 6 \times 8 = 528$

Therefore, M[4][6]=$minimum$(392, 528)=392

Chained Matrix Multiplications : dynamic programming approaches(cont.)

- The optimal order for multiplying 6 matrices must have one of these factorizations:
  1. $A_1(A_2A_3A_4A_5A_6)$
  2. $(A_1\ A_2)(A_3A_4A_5A_6)$
  3. $(A_1A_2A_3)(A_4A_5A_6)$
  4. $(A_1A_2A_3A_4)(A_5A_6)$
  5. $(A_1A_2A_3A_4A_5)A_6$
- The # of multi. for the kth factorization is the minimum # needed to obtain each factor plus the # needed to multiply the two factors. This means that
  - $M[1][k] + M[k+1][6] + d_0d_kd_6$.

- $M[1][6] = \min_{1 \le k \le 5}(M[1][k] + M[k+1][6] + d_0d_kd_6)$.

Example 3.6

A_1	A_2	A_3	A_4	A_5	A_6
5×2	2×3	3×4	4×6	6×7	7×8
$d_0\ d_1$	$d_1\ d_2$	$d_2\ d_3$	$d_3\ d_4$	$d_4\ d_5$	$d_5\ d_6$

- Steps for dynamic programming
  - Compute diagonal 0 : $M[i][j] = 0, 1 \le i \le 6$
  - Compute diagonal 1 : $M[1][2] = \min_{1 \le k \le 1}(M[1][k] + M[k+1][2] + d_0 d_k d_2)$

    $= M[1][1] + M[2][2] + d_0 d_1 d_2$

    $= 0 + 0 + 5 \times 2 \times 3 = 30.$
  - The value of $M[2][3]$, $M[3][4]$, $M[4][5]$, and $M[5][6]$ are computed in the same way.
  - Compute diagonal 2:

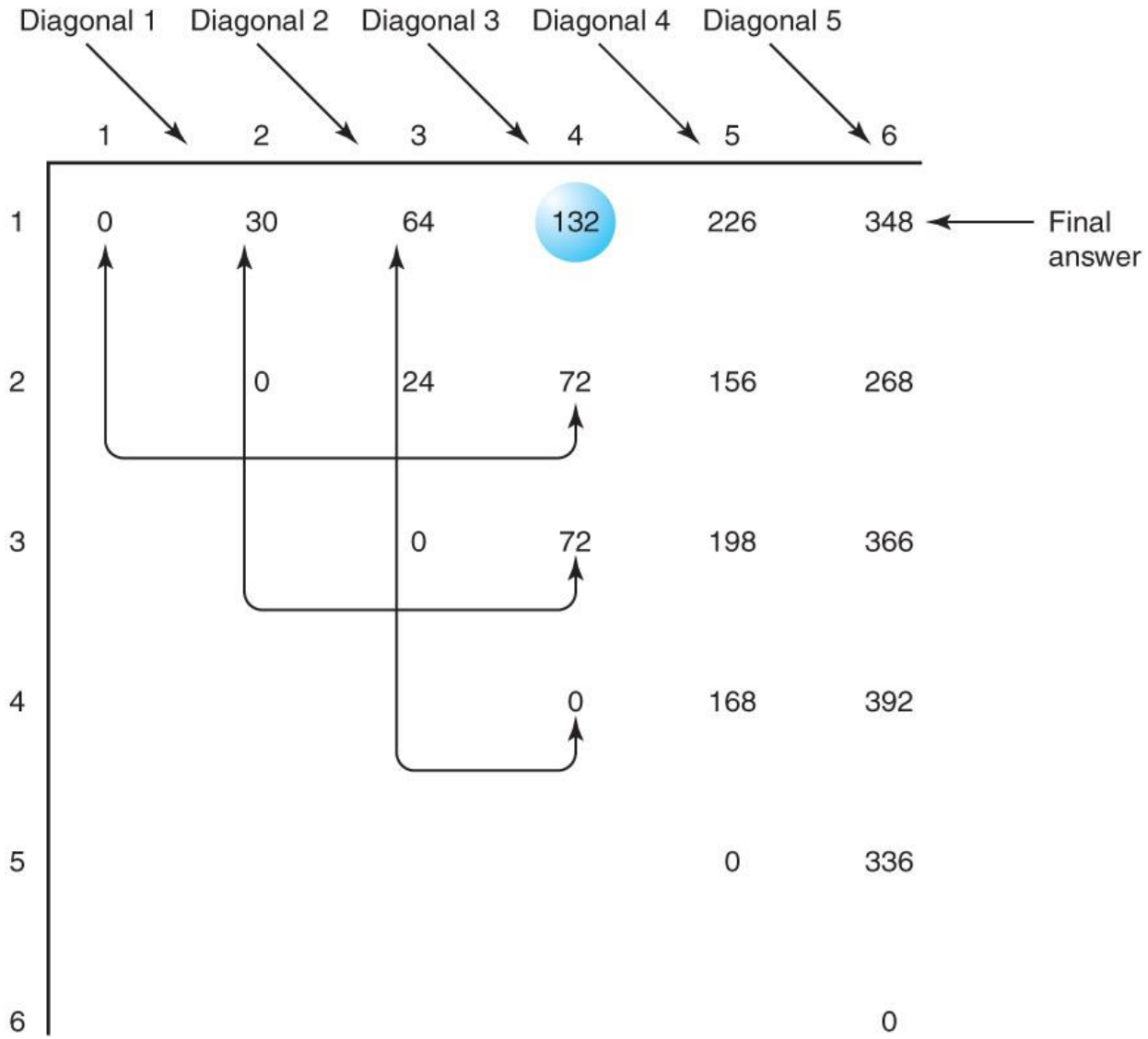
    $M[1][3] = \min_{1 \le k \le 2}(M[1][k] + M[k+1][3] + d_0 d_k d_3)$

    $= \min(M[1][1] + M[2][3] + d_0 d_1 d_3,$

    $M[1][2] + M[3][3] + d_0 d_2 d_3)$

    $= \min(0 + 72 + 5 \times 2 \times 6, 30 + 72 + 5 \times 3 \times 6, 64 + 0 + 5 \times 4 \times 6) = 132.$
  - …
  - Compute diagonal 4, 5.
  - Finally, the entry in diagonal 5 is computed in the same manner. This entry is the solution to our instance; it is the minimum # of elementary multi., and its value is given by
    - $M[1][6] = 348.$

Example


Diagonal 1
Diagonal 2
Diagonal 3
Diagonal 4
Diagonal 5
1 2 3 4 5 6
1 0 30 64 132 226 348
Final answer
2 0 24 72 156 268
3 0 72 198 366
4 0 168 392
5 0 336
6 0

Print Optimal Order

- To obtain optimal order, save k into $P[i][j]$ in computing $M[i][j]$.
- ex: $P[2][5] = 4$, optimal order: $(A_2 A_3 A_4)\ A_5$.
- *P:*

$P[i][j]$	1	2	3	4	5	6
1		1	1	1	1	1
2			2	3	4	5
3				3	4	5
4					4	5
5						5

Therefore, optimal factorization : $(A_1((((A_2 A_3)\ A_4)\ A_5)\ A_6))$.

Minimum Multiplications Algorithm

- Problem : Determining the minimum # of elementary multi. needed to multiply *n* matrices and an order that produces that minimum number.
- Inputs : the # of matrices *n*, and an array of integers *d*, indexed from 0 to *n*, where *d*[*i*-1]x*d*[*i*] is the dimension of the *i*th matrix.
- Outputs : *minmult*, the minimum # of elementary multi. needed to multiply the n matrices; a two-dimensional array *P* from which the optimal order can be obtained. *P* has its rows indexed from *1* to *n*-1 and its columns indexed from 1 to *n*. *P*[*i*][*j*] is the point where matrices *i* through *j* are split in an optimal order for multiplying the matrices.

- **Algorithm:**

```
int minmult(int n, const int d[], index P[][]) {
  index i, j, k, diagonal;
  int M[1..n, 1..n];
  for(i=1; i <= n; i++)
    M[i][j] = 0;
  for(diagonal = 1; diagonal <= n-1; diagonal++)
    for(i=1; i <= n-diagonal; i++) {
      j = i + diagonal;
      M[i][j] = minimum(M[i][k]+M[k+1][j]+
          d[i-1]*d[k]*d[j]);
                  where i <= k <= j-1
      P[i][j] = a value of k that gave the minimum
    }
  return M[1][n];
}
```

Every-Case Time Complexity(Minimum Multiplication)

- Basic operation: consider the instructions executed for each value of *k* to be the basic operation. Included is a comparison to test for the minimum.
- Input size : *n*, the # of matrices to be multiplied.
- Analysis : $j = i + diagonal$,
  - # of passes *k*-loop =
    $$(j-1)-i+1 = i+diagonal-1-i+1 = diagonal$$
  - # of passes for-*i*-loop $n - diagonal$ =

$$\sum_{diagonal=1}^{n-1} [(n-diagonal)\times diagonal] = \frac{n(n-1)(n+1)}{6} \in \Theta(n^3)$$

Print optimal order

- Problem : Print the optimal order for multiplying *n* matrices.
- Inputs : *n* and *P*.
- Outputs : the optimal order for multiplying the matrices.
- Algorithm:

```
void order(index i, index j) {
  if (i == j) cout << "A" << i;
   else {
     k = P[i][j];
     cout << "(";
     order(i,k);
     order(k+1,j);
     cout << ")";
  }
}
```

Print optimal order

- `order(i,j)`: Print an optimal multiplying order $A_i \times \ldots \times A_j$ with parenthesis.
- Analysis: $T(n) \in \Theta(n)$.

Other algorithms

- Yao(1982) - $\Theta(n^2)$
- Hu and Shing(1982, 1984) - $\Theta(n \lg n)$

The Traveling Salesperson Problem

The Traveling Salesperson Problem

- Suppose a salesperson is planning a sales trip that includes 20 cities. Each city is connected to some of the other cities by a road.
- To minimize travel time, we want to determine a shortest route that starts at the salesperson's home city, visits each of the cities once, and ends up at the home city.
- This problem of determining a shortest route is called the **Traveling Salesperson problem(TSP).**

History of the TSP

- The general form of TSP problem was considered by mathematicians in the 1930's
- During 1950's and 1960's it gained popularity in scientific circles
- Researchers(Dantzig, Fulkerson, Johnson) create a method to optimally solve TSP for 49 cities(1954)
- 1971 64 cities
- In 1972, Karp proves that TSP is NP complete
- 1975 100 cities
- 1977 120 cities
- 1980 318 cities
- 1987 666 cities
- 1987 2,392 cities(Electronic Wiring Example)
- 1994 7,397 cities
- 1998 13,509 cities(all towns in the USA with population >500)
- 2001 15,112 cities(towns in Germany)
- 2004 24,978 cities(places in Sweden)

TSP

- An instance of this **problem** can be represented by a **weighted graph**, in which each vertex represents a city.
- We generalize the problem to include the case in which the weight (distance) going in one direction is different from the weight going in another direction. We assume that the weights are nonnegative numbers.
- A **tour** in a directed graph is a path from a vertex to itself that passes through each of the other vertices only once.
- An **optimal tour** in a weighted, directed graph is such a path of minimum length.
- The **Traveling Salesperson problem** is **to find an optimal tour** in a weighted directed graph when at least **one tour exists**. Dynamic programming can be applied when the principle of optimality applies.

Brute Force

- Seems to be the obvious solution
- Computationally expensive - turns out to be O(n!)
- For n total cities, start from any city, there are n-1 choices for a second city, n-2 choices for a third…

$$(n-1)(n-2)\ldots 2\ 1 = (n-1)!$$

- As a point of reference,
  $20! = 2.43290201 \times 10^{18}$

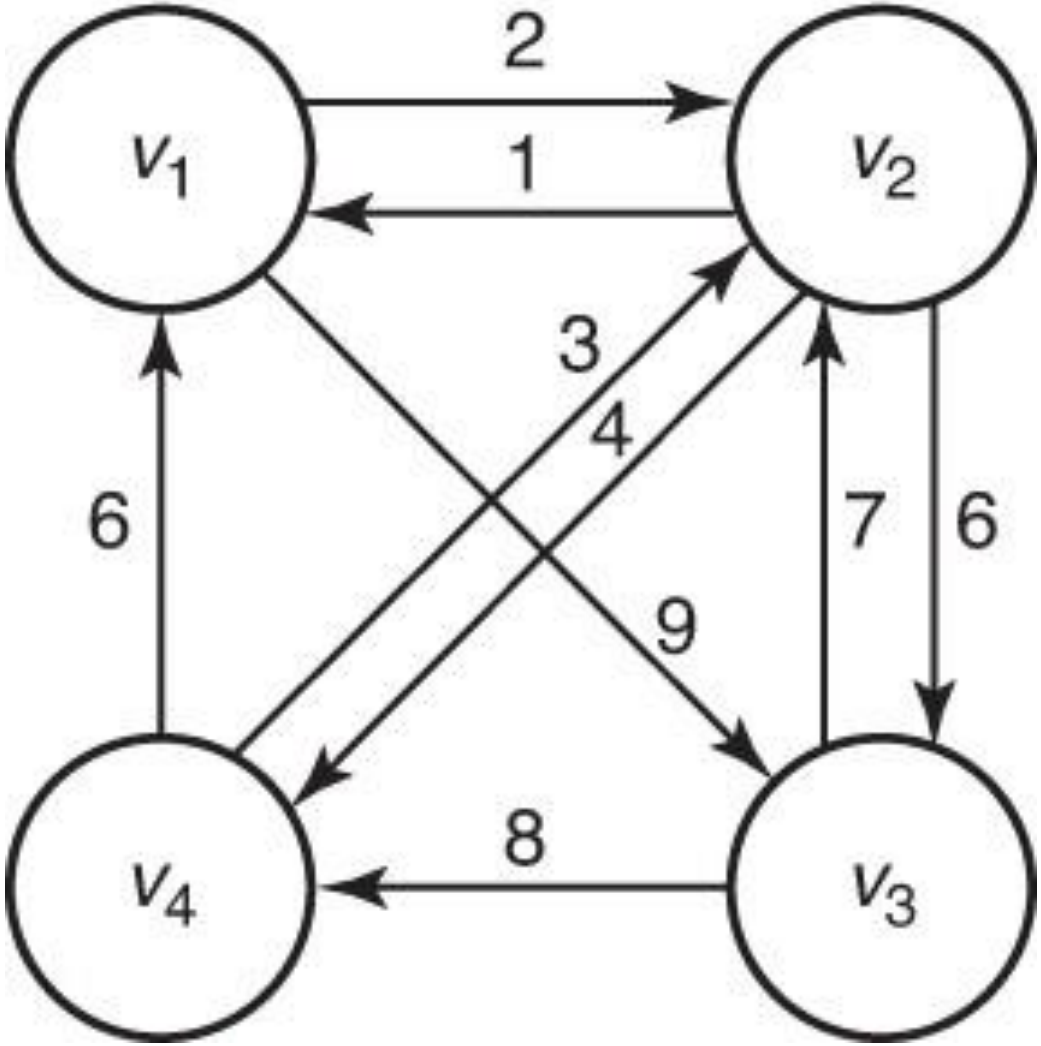
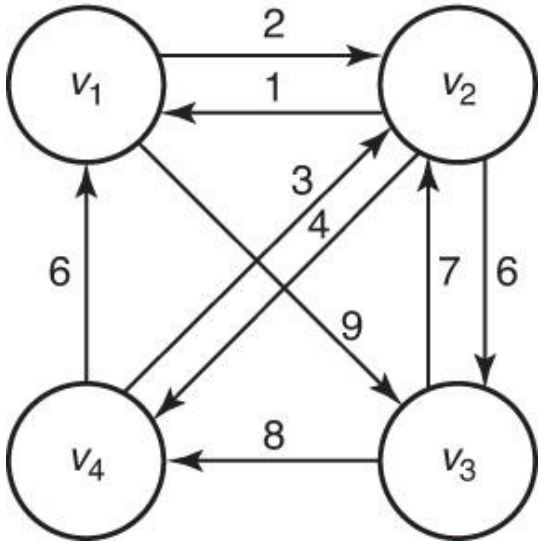


Figure 3.16: The optimal tour is [v1, v3, v4, v2, v1]

TSP

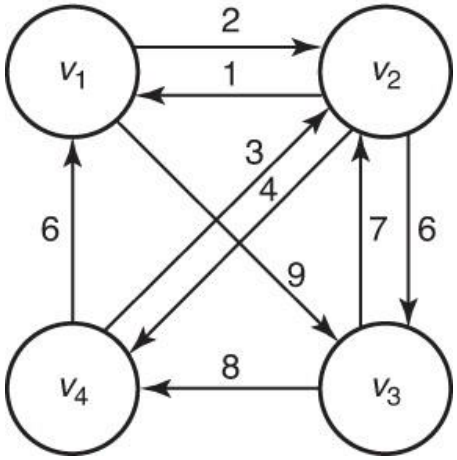
- Adjacency matrix
  - V = set of all the vertices
  - A = a subset of V
  - $D[v_i][A] =$ length of shortest path from v_i to v_1 passing through each vertex in A exactrly once.



	1	2	3	4
1	0	2	9	∞
2	1	0	6	4
3	∞	7	0	8
4	6	3	∞	0

Figure 3.17: The adjacency matrix representation W of the graph in Figure 3.16

Example



- $V = \{v_1, v_2, v_3, v_4\}$
- $[v_1, v_2, v_3, v_4]$: path
- If $A = \{v_3\}$, then $D[v_2][A] = length[v_2, v_3, v_1]$
  $= \infty$.
- If $A = \{v_3, v_4\}$, then
  $D[v_2][A] = \min(length[v_2, v_3, v_4, v_1], length[v_2, v_4, v_3, v_1])$
  $= \min(20, \infty) = 20$.

Optimal tour

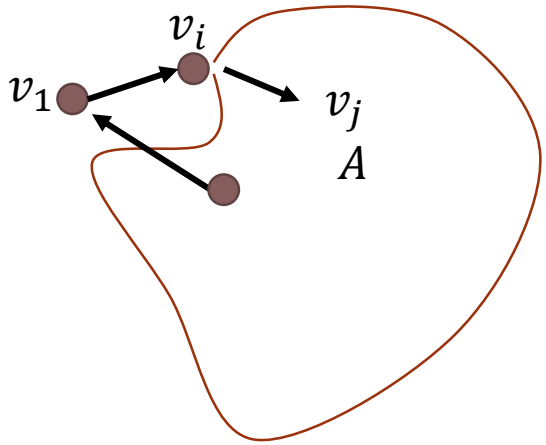
- V- $\{v_1, v_j\}$ contains all the vertices except v_1 and v_j and the principle of optimality applies, we have

$$\text{length of an optimal tour} = \min_{2 \le j \le n} \left(W[1][j] + D\left[v_j\right]\left[V - \left\{v_1, v_j\right\}\right] \right),$$

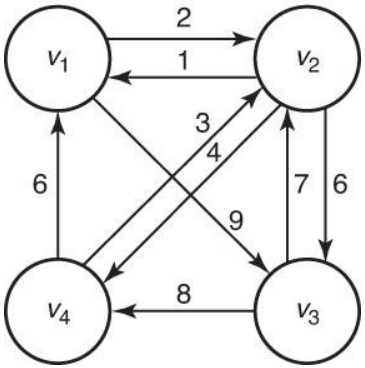
and, in general for $i \neq 1$ and v_i not in A,

$$D[v_i][A] = \min_{j : v_j \in A} \left(W[i][j] + D\left[v_j\right]\left[A - \left\{v_j\right\}\right] \right) \text{ if } A \neq \emptyset$$

$D[v_i][\emptyset] = W[i][1]$.



Example 3.11



- Determine an optimal tour
  - Empty set:

$$D[v_2][\emptyset] = W[2][1] = 1$$
$$D[v_3][\emptyset] = W[3][1] = \infty$$
$$D[v_4][\emptyset] = W[4][1] = 6$$

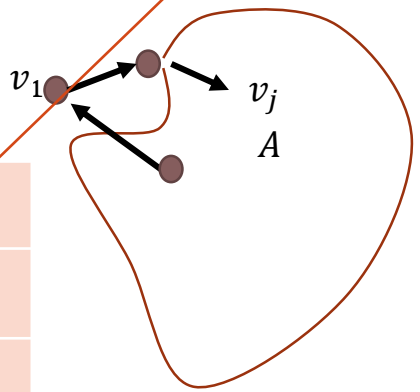
Next consider all sets containing one element:

$$D[v_3][\{v_2\}] = \min_{j:v_j \in \{v_2\}} (W[3][j] + D[v_j][\{v_2\} - \{v_j\}])$$
$$= W[3][2] + D[v_2][\emptyset] = 7 + 1 = 8$$

Similarly, $D[v_4][\{v_2\}] = (W[4][2] +) = 3 + 1 = 4$

$$D[v_2][\{v_3\}] = 6 + \infty = \infty$$
$$D[v_4][\{v_3\}] = \infty + \infty = \infty$$
$$D[v_2][\{v_4\}] = 4 + 6 = 10$$
$$D[v_3][\{v_4\}] = 8 + 6 = 14$$

	1	2	3	4
1	0	2	9	∞
2	1	0	6	4
3	∞	7	0	8
4	6	3	∞	0



1	0			
2	1	0	6+∞ =∞	4+6 =10
3	∞	7+1 =8	0	8+6 =14
4	6	3+1 =4	∞+∞ =∞	0

Example 3.11(2)

- Next consider all sets containing two elements:

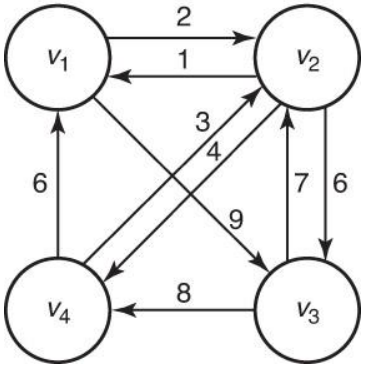
$$D[v_4][\{v_2, v_3\}] = \min_{j:v_j \in \{v_2, v_3\}} (W[4][j] + D[v_j][\{v_2, v_3\} - \{v_2\}])$$

$$= \min(W[4][2] + D[v_2][\{v_3\}], W[4][3] + D[v_3][v_2\}]) = \min(3 + \infty, \infty + 8) = \infty$$

Similarly,

$$D[v_3][\{v_2, v_4\}] = \min(7 + 10, 8 + 4) = 12$$

$$D[v_2][\{v_3, v_4\}] = \min(6 + 14, 4 + \infty) = 20$$

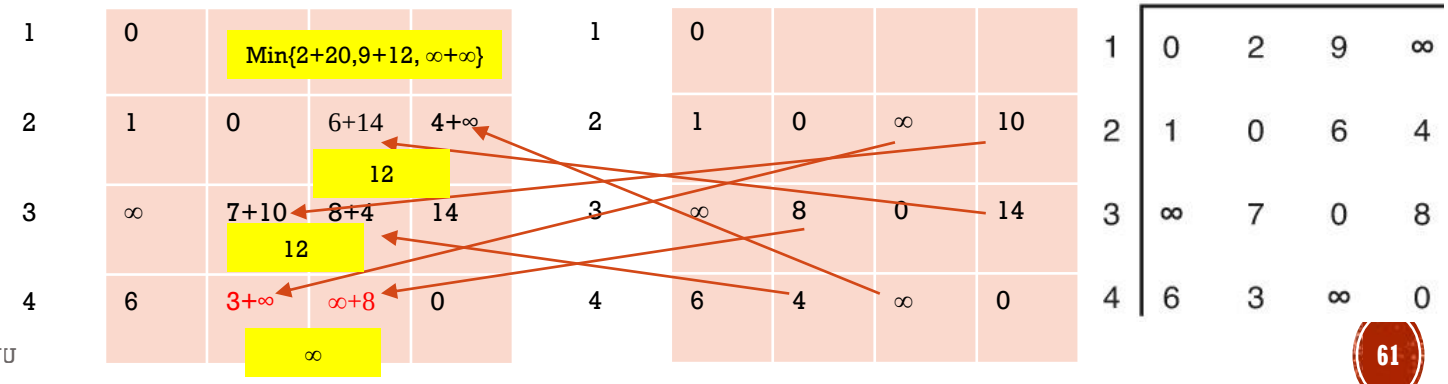


Finally, compute the length of an optimal tour:

$$D[v_1][\{v_2, v_3, v_4\}] = \min_{j:v_j \in \{v_2, v_3, v_4\}} (W[1][j] + D[v_j][\{v_2, v_3, v_4\} - \{v_j\}])$$

$$= \min \begin{matrix} (W[1][2] + D[v_2][\{v_3, v_4\}], \\ W[1][3] + D[v_3][v_2, v_4\}], \\ W[1][4] + D[v_4][\{v_2, v_3\}]) \end{matrix}$$

$$= \min(2 + 20, 9 + 12, \infty + \infty) = 21$$



Dynamic Programming for TSP

- Problem : Determine an optimal tour in a weighted, directed graph. The weights are nonnegative numbers.

- Inputs : a weighted, directed graph, and n, the # of vertices in the graph. The graph is represented by a two-dimensional array W, which has both its rows and columns indexed from 1 to n, where $W[i][j]$ is the weight on the edge from ith vertex to the jth vertex.

- Outputs : a variable *minlength*, whose value is the length of an optimal tour, and a two-dimensional array $P[1..n]$[subset of $V - \{v_1\}$] from which an optimal tour can be constructed. $P[i][A]$ is the index of the first vertex after v_i on a shortest path from v_i to v_1 that passes through all the vertices in A exactly once.

Dynamic Programming for TSP

void travel (int n, const number W[][],
index P[][], number& minlength)
{
index i, j, k;
number D[1..n][subset of $V - \{v_1\}$] ;

for (i=2; i<=n; i++)
$D[i][\emptyset] = W[i][1]$;
for (k=1;k<=n-2;k++)
for(all subsets $A \subseteq V - \{v_1\}$ containing k vertices)
for(i such that $i \neq 1$ and v_i is not in A){
$D[i][A] = \min_{j:v_j \in A}(W[i][j] + D[j][A - \{v_j\}])$;
$P[i][A] =$ value of j that gave the minimum;
}
$D[1][V - \{v_1\}] = \min_{2 \le j \le n}(W[1][j] + D[j][V - \{v_1, v_j\}])$;
$P[1][V - \{v_1\}] =$ value of j that gave the minimum;
minlength = $D[1][V - \{v_1\}]$;
}

TSP-analysis

- For all $n \geq 1$

$$\sum_{k=1}^{n} k \binom{n}{k} = n2^{n-1}.$$

- Proof : It is left as an exercise to show that

$$k \binom{n}{k} = n \binom{n-1}{k-1}.$$

$$\because k \binom{n}{k} = k \frac{n!}{k!(n-k)!} = \frac{k}{k} \frac{n(n-1)!}{(k-1)!(n-k)!} = n \binom{n-1}{k-1}$$

Therefore,

$$\sum_{k=1}^{n} k \binom{n}{k} = \sum_{k=1}^{n} n \binom{n-1}{k-1} = n \sum_{k=0}^{n-1} \binom{n-1}{k} = n2^{n-1}.$$

The last equality is obtained by applying the result : $\sum_{k=0}^{n} \binom{n}{k} = \sum_{k=0}^{n} \binom{n}{k} 1^k 1^{n-k} = (1+1)^n = 2^n$.

Every-case time and space complexity

- Basic operation: the instructions executed for each value of v_j to be the basic operation.
- Input size : n, the # of vertices in the graph.

For each set A containing k vertices, we must consider n-1-k vertices, and for each of these vertices, the basic operation is done k times. Because the # of subsets A of $V-\{v_1\}$ containing k vertices is equal to $\binom{n-1}{k}$, the total # of times the basic operation is done is given by

$$T(n) = \sum_{k=1}^{n-2}(n-1-k)k\binom{n-1}{k}. \text{ ------- } (*)$$

It is not hard to show that

$$(n-1-k)\binom{n-1}{k} = (n-1)\binom{n-2}{k}$$

$$\because (n-1-k)\binom{n-1}{k} = (n-1-k)\frac{(n-1)!}{k!\,(n-1-k)!} = \frac{(n-1-k)(n-1)(n-2)!}{k!\;(n-1-k)(n-2-k)!}$$

Substituting this equality into Eq (*), we have

$$T(n) = (n-1)\sum_{k=1}^{n-2} k\binom{n-2}{k}.$$

Finally, applying previous Th, we have

$$T(n) = (n-1)(n-2)2^{n-3} \in \Theta(n^2 2^n).$$

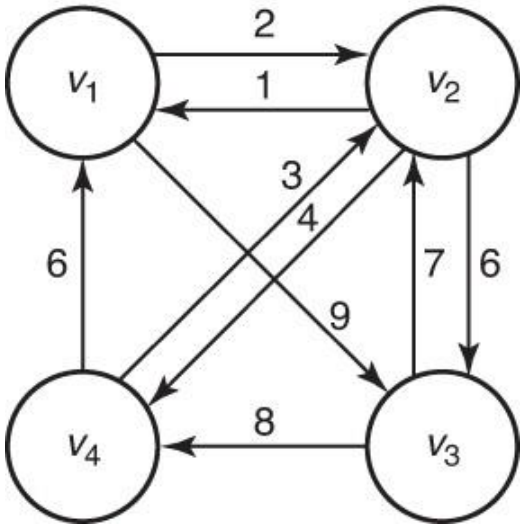
Example 3.12

- 20 cities
  - Brute-force algorithm : 19! μs=3857 years
  - Dynamic Programming algorithm: $(20-1)(20-2)2^{20-3}\mu s = 45$ seconds.

Retrieve an optimal tour P

- How to retrieve an optimal tour from the array *P*.
- P : for representing tour

3	4	2
$P[1,\{v_2,v_3,v_4\}]$	$P[3,\{v_2,v_4\}]$	$P[4,\{v_2\}]$



- We obtain an optimal tour as follows:

Index of first node = $P[1][\{v_2,v_3,v_4\}] = 3$ (see slide 61)

Index of second node = $P[3][\{v_2,v_4\}] = 4$ (see slide 60)

Index of third node = $P[4][\{v_2\}] = 2$

The optimal tour is, therefore,

$$[v_1, v_3, v_4, v_2, v_1].$$

- No one has ever found an algorithm for the TSP whose worst-case time complexity is better exponential. Yet no one has ever proved that such an algorithm is not possible.

Longest Common Sequence

Longest Common Subsequence(LCS) problem

- A **sequence** is an ordered list. Like a set, it contains members(also called elements, or terms).
- A **subsequence** of a string *S*, is a set of characters that appear in left-to-right order, but not necessarily consecutively.
- A **subsequence** is a sequence that can be derived from another sequence by deleting some elements without changing the order of the remaining elements.
  - Example.

    ACTTGCG

    ACT, ATTC, T, ACTTGC : subsequences

    TTA : not a subsequence
- A **common subsequence** of two strings is a subsequence that appears in both strings. A **longest common subsequence** is a common subsequence of maximal length.
  - Example.

    S1 = AAACCGTGAGTTATTCGTTCTAGAA

    S2 = CACCCCTAAGGTACCTTTGGTTC

    LCS is **ACCTAGTACTTTG**
- The **longest common subsequence** (**LCS**) **problem** is to find the longest subsequence common to all sequences in a set of sequences (often just two)

Longest Common Subsequence (LCS)

Application: comparison of two DNA strings

Ex: X= {A B C B D A B }, Y= {B D C A B A}

Longest Common Subsequence:

X = A **B** **C** **B** D **A** B

Y = **B** D **C** A **B** **A**

Brute force algorithm would compare each subsequence of X with the symbols in Y

Brute force solution

- **Solution: For every subsequence of x, check if it is a subsequence of y.**
- **Analysis :**
  - |x| = *m*, |y| = *n*,
  - **Then there are 2^m subsequences of x.**
  - **Each check takes O(n) time, since we scan y for first element, and then scan for second element, etc. (**we must compare each with y (n comparisons)
  - **The worst case running time is $O(n2^m)$.**

LCS Algorithm

- Notice that the LCS problem has *optimal substructure*: solutions of subproblems are parts of the final solution.
- Subproblems: “find LCS of pairs of *prefixes* of X and Y”

LCS Algorithm

- First we'll find the length of LCS. Later we'll modify the algorithm to find LCS itself.
- Define X_i, Y_j to be the prefixes of X and Y of length i and j respectively
- Define $c[i,j]$ to be the length of LCS of X_i and Y_j
- Then the length of LCS of X and Y will be $c[m,n]$

$$c[i,j] = \begin{cases} c[i-1, j-1]+1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

LCS recursive solution

$$c[i, j] = \begin{cases} c[i-1, j-1]+1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

- We start with $i = j = 0$ (empty substrings of x and y)
- Since X_0 and Y_0 are empty strings, their LCS is always empty (i.e. $c[0,0] = 0$)
- LCS of empty string and any other string is empty, so for every i and j: $c[0, j] = c[i,0] = 0$

LCS recursive solution

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x[i] = y[j], \\ \max(c[i,j-1], c[i-1,j]) & \text{otherwise} \end{cases}$$

- When we calculate $c[i,j]$, we consider two cases:
- **First case:** $x[i]=y[j]$: one more symbol in strings X and Y matches, so the length of LCS X_i and Y_j equals to the length of LCS of smaller strings X_{i-1} and Y_{i-1} , plus 1

LCS recursive solution

$$c[i, j] = \begin{cases} c[i-1, j-1]+1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

- **Second case:** $x[i] != y[j]$
- As symbols don't match, our solution is not improved, and the length of LCS(X_i , Y_j) is the same as before (i.e. maximum of LCS(X_i, Y_{j-1}) and LCS(X_{i-1},Y_j)

Why not just take the length of LCS(X_{i-1}, Y_{j-1}) ?

LCS Length Algorithm

LCS-Length(X, Y)

1. m = length(X) // get the # of symbols in X
2. n = length(Y) // get the # of symbols in Y
3. for i = 1 to m c[i,0] = 0 // special case: Y_0
4. for j = 1 to n c[0,j] = 0 // special case: X_0
5. for i = 1 to m // for all X_i
6. for j = 1 to n // for all Y_j
7. if (X_i == Y_j)
8. c[i,j] = c[i-1,j-1] + 1
9. else c[i,j] = max(c[i-1,j], c[i,j-1])
10. return c

LCS Example

We'll see how LCS algorithm works on the following example:

- X = ABCB
- Y = BDCAB

What is the Longest Common Subsequence of X and Y?

LCS(X, Y) = BCB

X = A **B** **C** **B**

Y = **B** D **C** A **B**

LCS Example (0)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi						
1	**A**						
2	**B**						
3	**C**						
4	**B**						

X = ABCB; m = |X| = 4
Y = BDCAB; n = |Y| = 5
Allocate array c[5,4]

ABCB
BDCAB

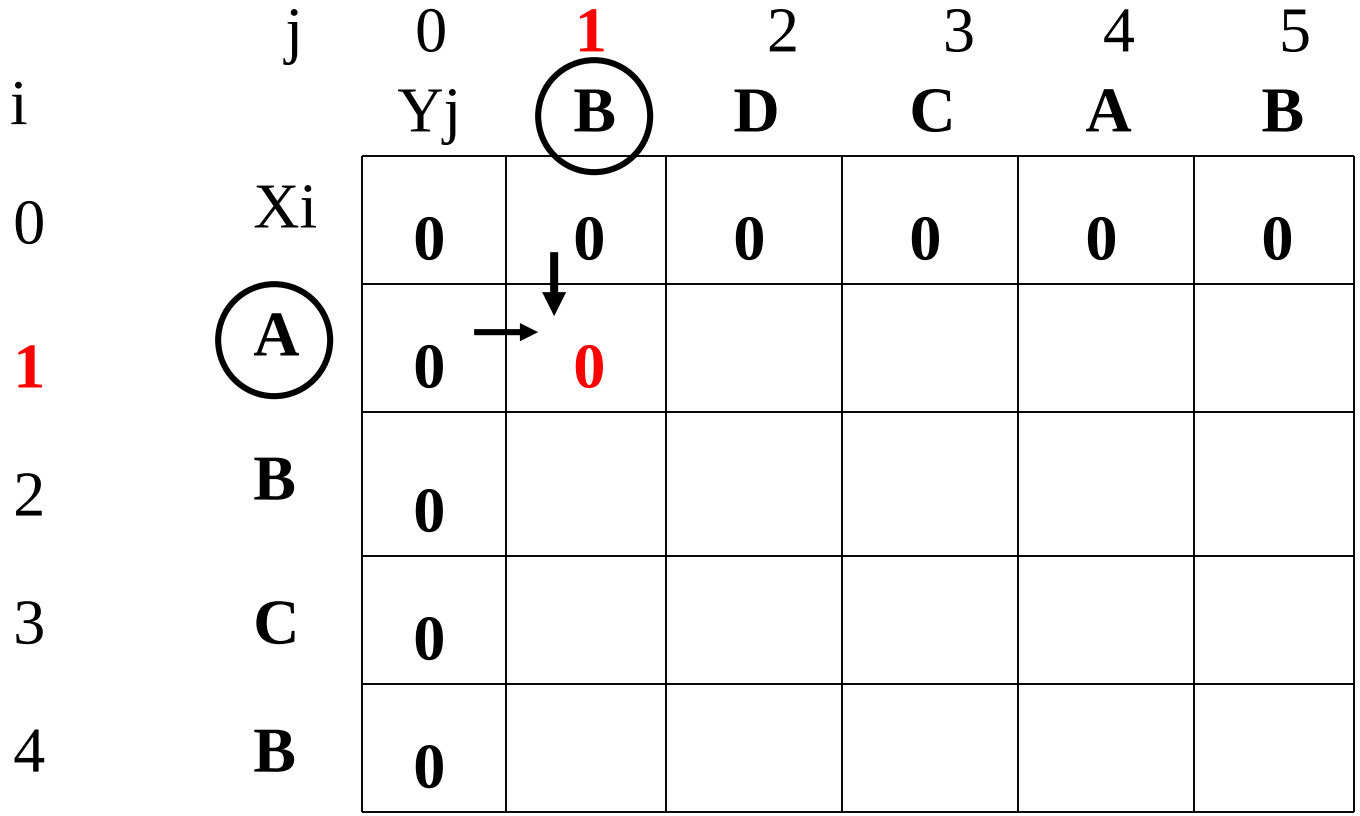
LCS Example (1)

i	j	0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**					
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

for i = 1 to m c[i,0] = 0
for j = 1 to n c[0,j] = 0

ABCB
BDCAB

LCS Example (2)



if ($X_i == Y_j$)

c[i,j] = c[i-1,j-1] + 1

else c[i,j] = max(c[i-1,j], c[i,j-1])

ABCB
BDCAB

LCS Example (3)

i	j	0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**		
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

if ($X_i == Y_j$)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

ABCB
BDCAB

LCS Example (4)

i	j	0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

if (X_i == Y_j)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (5)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1** →	**1**
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

if (X_i == Y_j)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

ABCB
BDCAB

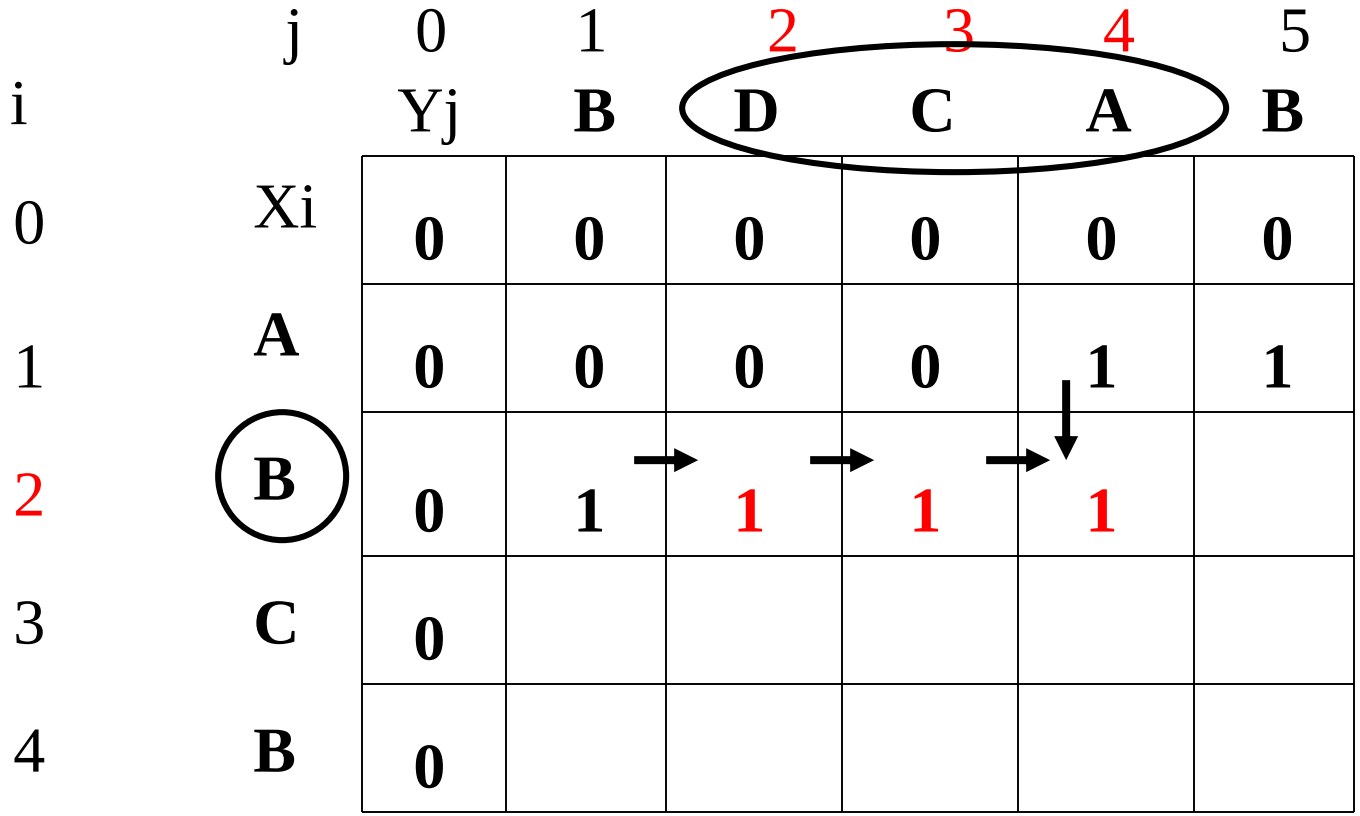
LCS Example (6)

i	j	0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**				
3	**C**	**0**					
4	**B**	**0**					

if (X_i == Y_j)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

ABCB
BDCAB

LCS Example (7)



if (X_i == Y_j)

c[i,j] = c[i-1,j-1] + 1

else c[i,j] = max(c[i-1,j], c[i,j-1])

ABCB
BDCAB

LCS Example (8)

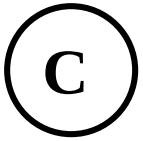
i	j	0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**					
4	**B**	**0**					

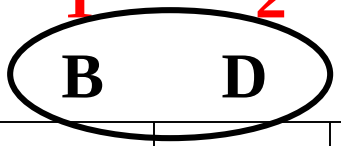
if (X_i == Y_j)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

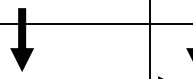
ABCB
BDCAB

LCS Example (10)

i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1			
4	B	0					







if ($X_i == Y_j$)

c[i,j] = c[i-1,j-1] + 1

else c[i,j] = max(c[i-1,j], c[i,j-1])

ABCB
BDCAB

LCS Example (11)

i	j	0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**		
4	**B**	**0**					

$$\text{if } (X_i == Y_j)$$
$$c[i,j] = c[i-1,j-1] + 1$$
$$\text{else } c[i,j] = \max(c[i-1,j], c[i,j-1])$$

LCS Example (12)

ABCB
BDCAB

i		j	0	1	2	3	4	5
			Yj	**B**	**D**	**C**	**A**	**B**
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**	**1**	**1**	**1**	**1**	**2**
3	**C**		**0**	**1**	**1**	**2**	**2**	**2**
4	**B**		**0**					

if ($X_i == Y_j$)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

ABCB
BDCAB

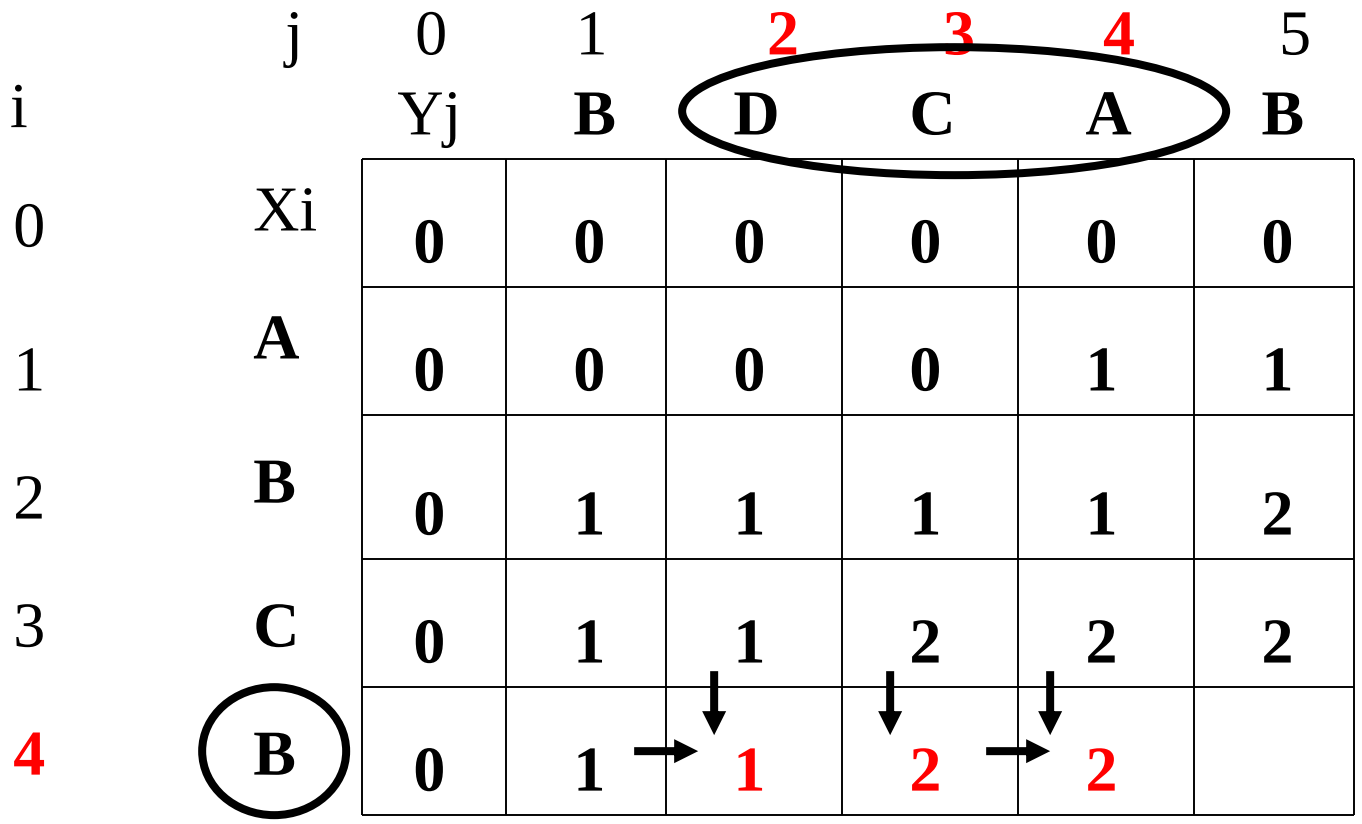
LCS Example (13)

i		j	0	1	2	3	4	5
			Yj	**B**	**D**	**C**	**A**	**B**
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**	**1**	**1**	**1**	**1**	**2**
3	**C**		**0**	**1**	**1**	**2**	**2**	**2**
4	**B**		**0**	**1**				

if ($X_i == Y_j$)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (14)

ABCB
BDCAB



i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	

if ($X_i == Y_j$)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

ABCB
BDCAB

LCS Example (15)

i	j	0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**	**1**	**1**	**2**	**2**	**3**

if (X_i == Y_j)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Algorithm Running Time

- LCS algorithm calculates the values of each entry of the array c[m,n]
- So what is the running time?

O(m*n)

since each c[i,j] is calculated in constant time, and there are m*n elements in the array

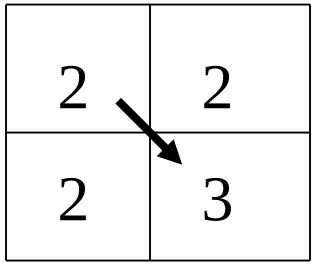
How to find actual LCS

- So far, we have just found the *length* of LCS, but not LCS itself.
- We want to modify this algorithm to make it output Longest Common Subsequence of X and Y

Each *c*[*i*,*j*] depends on *c*[*i-1*,*j*] and *c*[*i*,*j-1*]

or *c*[*i-1*, *j-1*]

For each c[i,j] we can say how it was acquired:



For example, here
c[i,j] = c[i-1,j-1] +1 = 2+1=3

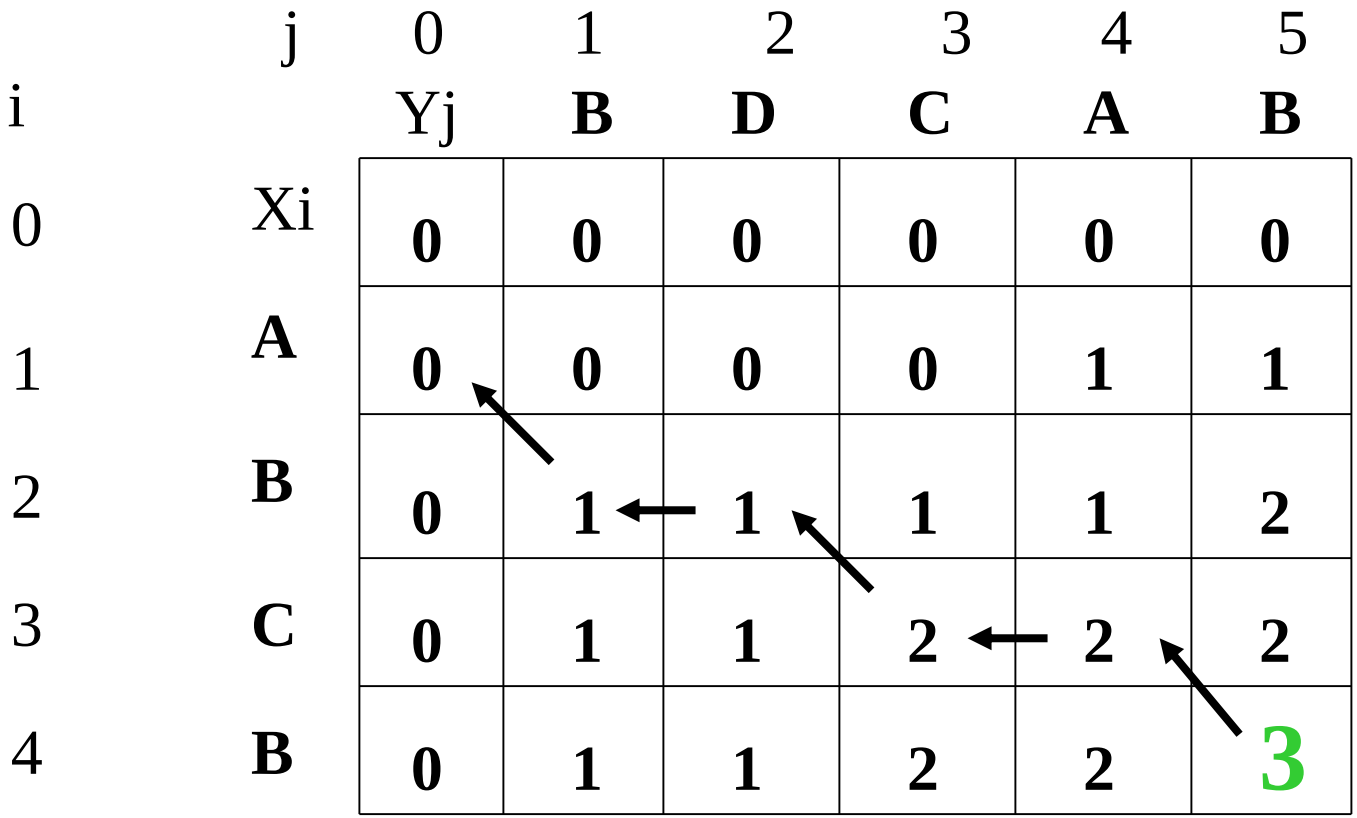
How to find actual LCS - continued

- Remember that

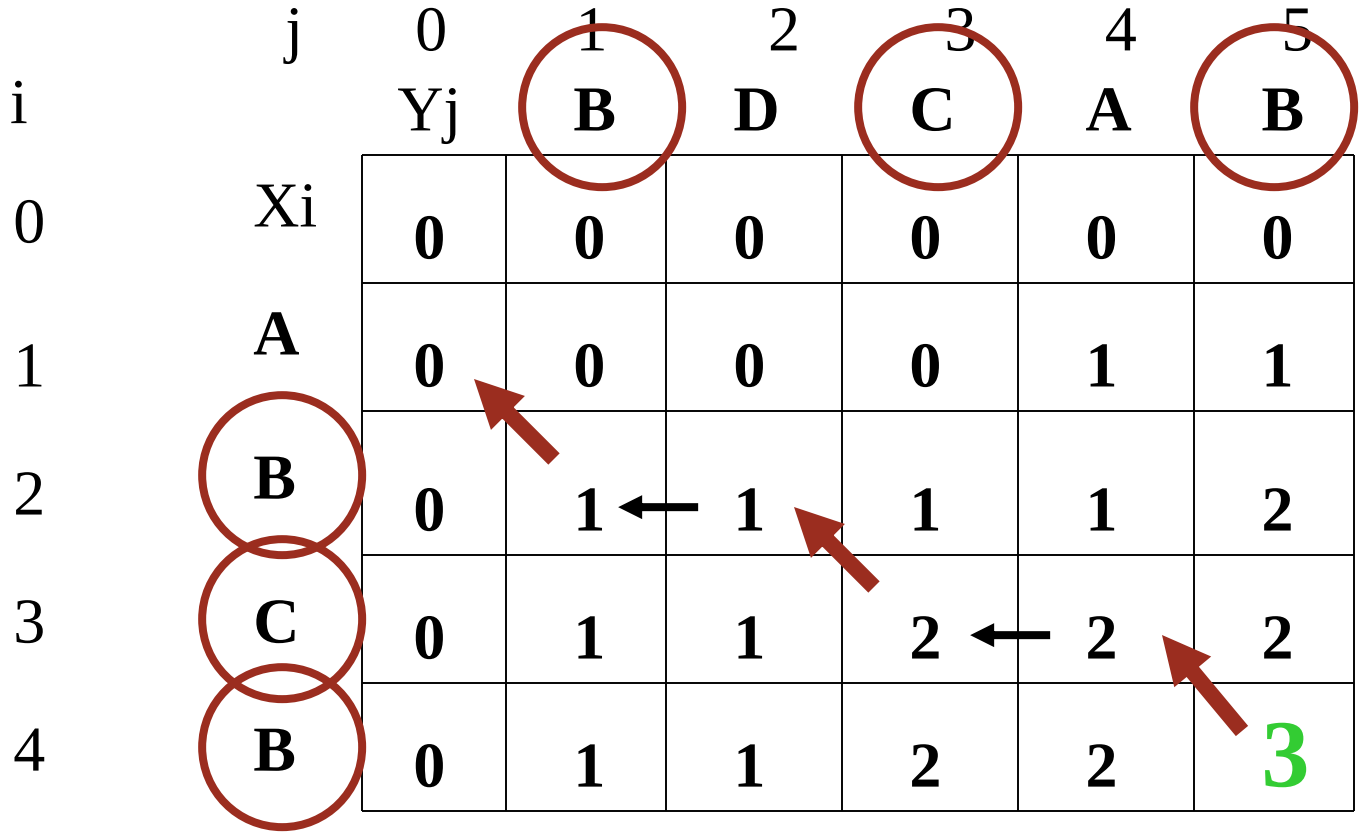
$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x[i] = y[j], \\ \max(c[i,j-1], c[i-1,j]) & \text{otherwise} \end{cases}$$

- So we can start from $c[m,n]$ and go backwards
- Whenever $c[i,j] = c[i-1,j-1]+1$, remember $x[i]$ (because $x[i]$ is a part of LCS)
- When i=0 or j=0 (i.e. we reached the beginning), output remembered letters in reverse order

Finding LCS


j 0 1 2 3 4 5
i
Yj B D C A B
0 Xi 0 0 0 0 0 0
1 A 0 0 0 0 1 1
2 B 0 1 1 1 1 2
3 C 0 1 1 2 2 2
4 B 0 1 1 2 2 3

Finding LCS (2)



LCS (reversed order): **B C B**

LCS (straight order): **B C B**

(this string turned out to be a palindrome)

Sequence Alignment

Sequence Alignment

- DNA(deoxyribonucleic acid)
  - Purines or pyrimidines
  - Purines : A(adenine), G(guanine)
  - Pyrimidines : C(cytosine), T(thymine)
- A DNA sequence is a section of DNA.

Figure 3.18: A section of DNA

Sequence Alignment

- a **sequence alignment** is a way of arranging the sequences of DNA, RNA, or protein to identify regions of similarity that may be a consequence of functional, structural, or evolutionary relationships between the sequences

```
AAB24882        TYHMCQFHCRYVNNHSGEKLYECNERSKAFSCPSHLQCHKRRQIGEKTHEHNQCGKAFPT 60
AAB24881        --------------------YECNQCGKAFAQHSSLKCHYRTHIGEKPYECNQCGKAFSK 40
                                     ****: .***:  * *:** * :****.:* *******..

AAB24882        PSHLQYHERTHTGEKPYECHQCGQAFKKCSLLQRHKRTHTGEKPYE-CNQCGKAFAQ- 116
AAB24881        HSHLQCHKRTHTGEKPYECNQCGKAFSQHGLLQRHKRTHTGEKPYMNVINMVKPLHNS 98
                 **** *:***********:***:**.: .****************    :  *.: :
```

A sequence alignment, produced by ClustalW, of two human zinc finger proteins, identified on the left by GenBankaccession number.
Key: Single letters: amino acids. Red: small, hydrophobic, aromatic, not Y. Blue: acidic. Magenta: basic. Green: hydroxyl, amine, amide, basic. Gray: others. "*": identical. ":": conserved substitutions (same colour group). ".": semi-conserved substitution (similar shapes).

Example

- Homologous sequence

  AACAGTTACC

  TAAGGTCA

- We could align them in many ways. The following shows two possible alignments:

```
-AACAGTTACC
 ^||^*||^^|*
TAA-GGT--CA

AACAGTTACC
  *|^||*|^|*
TA-AGGT-CA
```

-(^) : inserting a gap * : mismatching

From LCS to Alignment

- Simplest scoring schema: For some positive numbers μ and σ:
  - Match Premium: +1
  - Mismatch Penalty: $-\mu$
  - Indel Penalty: $-\sigma$

- Under these assumptions, the alignment score becomes as follows:

$$\text{Score} = \#\text{matches} - \mu(\#\text{mismatches}) - \sigma(\#\text{indels})$$

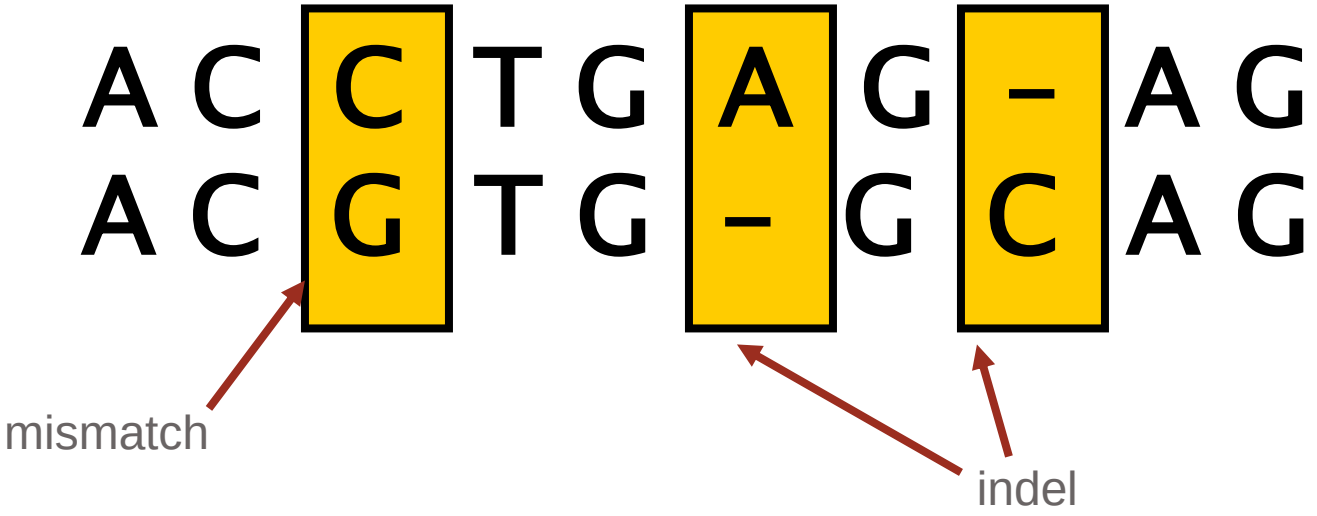
- Our specific choice of μ and σ depends on how we wish to penalize mismatches and indels.

Global Alignment Problem

- Input : Strings **v** and **w** and a scoring schema
- Output : An alignment with maximum score
- We can use dynamic programming to solve the Global Alignment Problem:

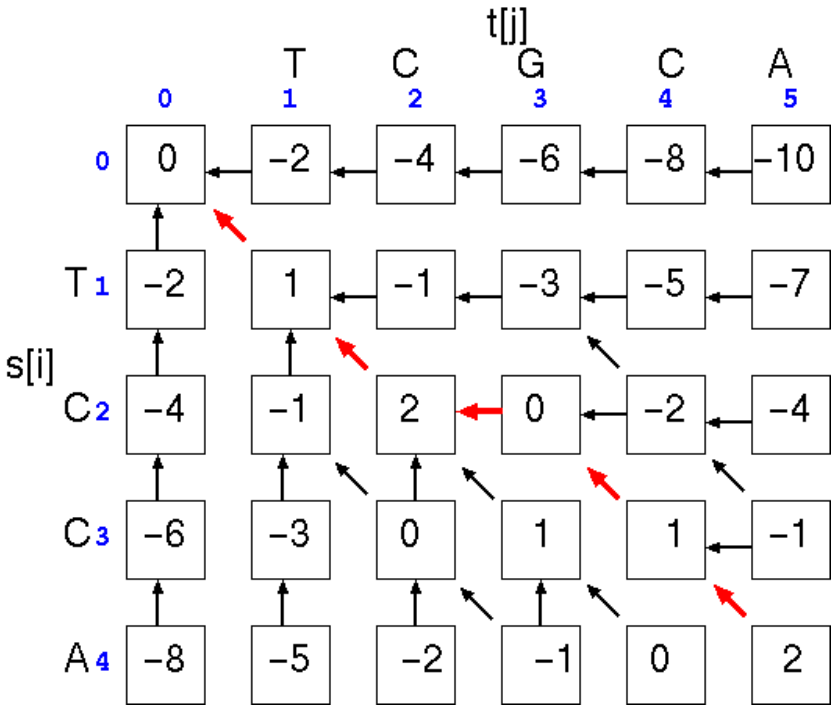
$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + 1 & \text{if } v_i = w_j \\ s_{i-1,j-1} - \mu & \text{if } v_i \neq w_j \\ s_{i-1,j} - \sigma \\ s_{i,j-1} - \sigma \end{cases}$$

μ : mismatch penalty
σ : indel penalty

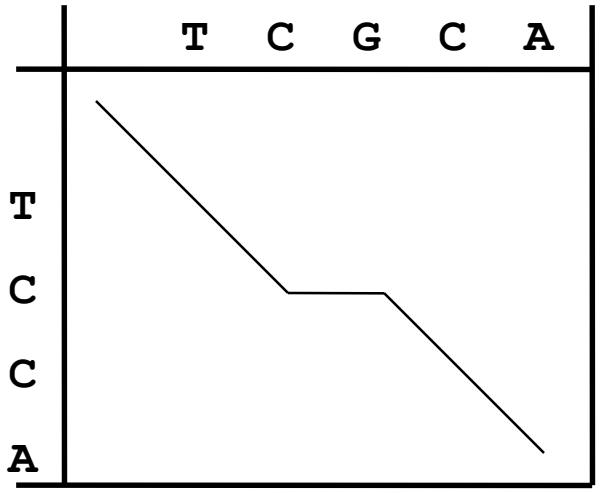

A C C T G A G – A G
A C G T G – G C A G
mismatch
indel

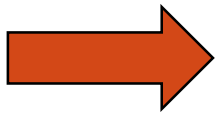
Pairwise alignment: the solution

"**Dynamic programming**"
(the Needleman-Wunsch algorithm)

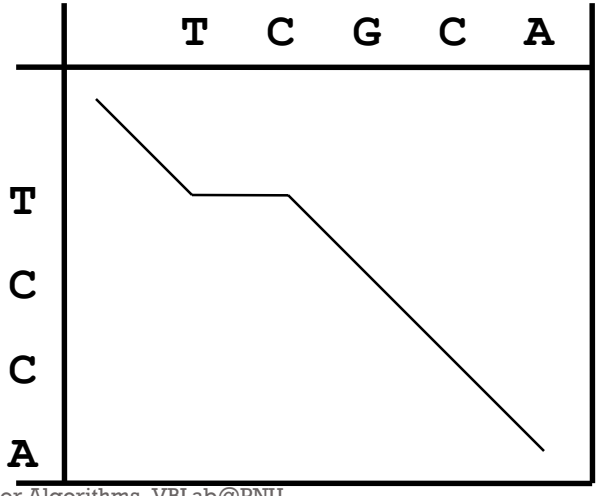


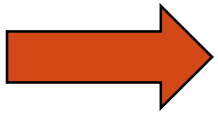
Alignment depicted as path in matrix





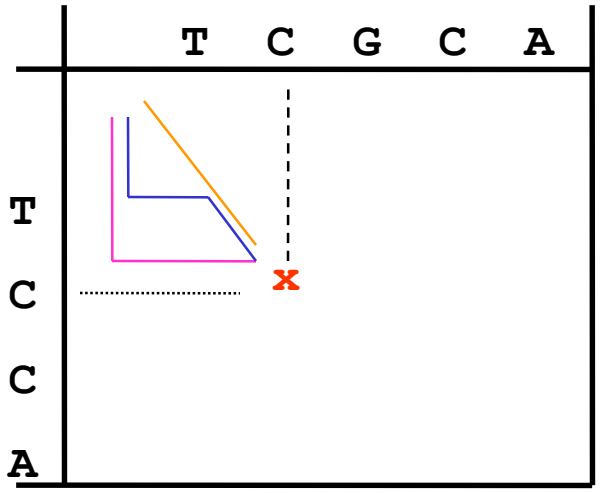
TCGCA
TC-CA





TCGCA
T-CCA

Alignment depicted as path in matrix

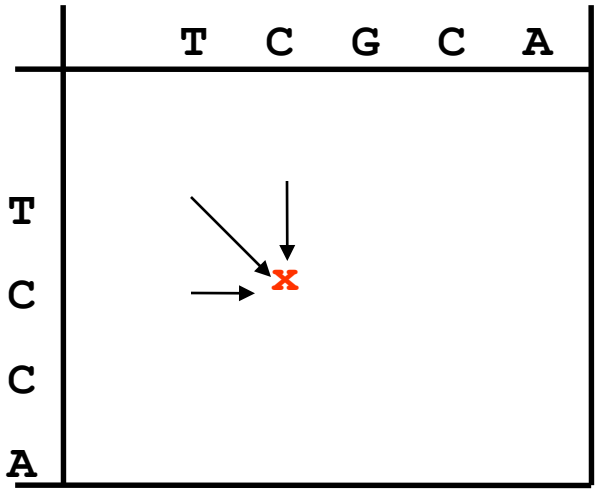


Meaning of point in matrix: all residues up to this point have been aligned (but there are many different possible paths).

Position labeled "x": **TC** aligned with **TC**

--TC TC--	-TC T-C	TC TC

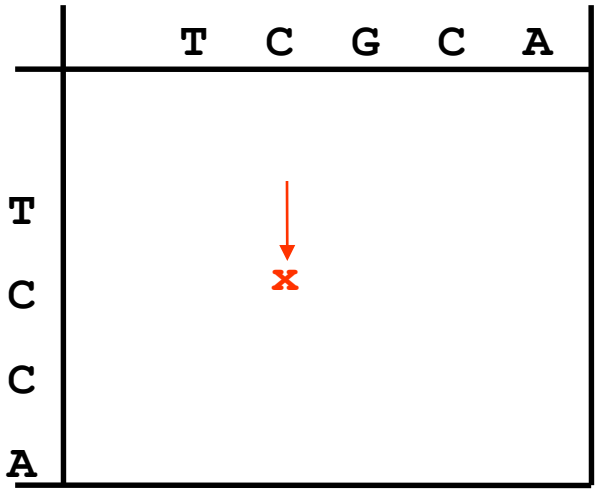
Dynamic programming: computation of scores



Any given point in matrix can only be reached from three possible previous positions (you cannot “align backwards”).

=> Best scoring alignment ending in any given point in the matrix can be found by choosing the highest scoring of the three possibilities.

Dynamic programming: computation of scores

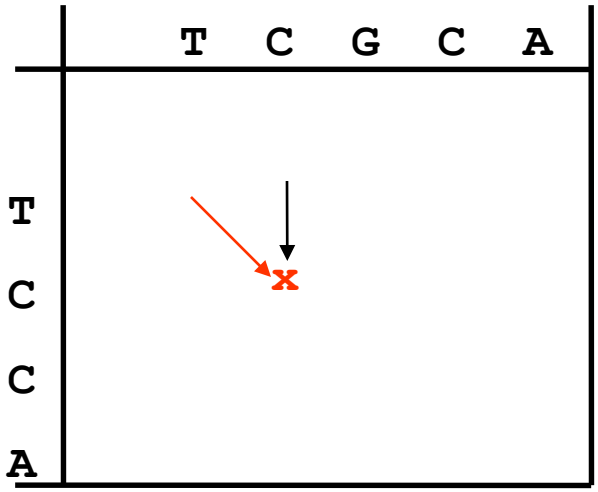


Any given point in matrix can only be reached from three possible positions (you cannot “align backwards”).

=> Best scoring alignment ending in any given point in the matrix can be found by choosing the highest scoring of the three possibilities.

$$\text{score}(x,y) = \max \begin{cases} \text{score}(x,y-1) - \text{gap-penalty} \\ \\ \\ \end{cases}$$

Dynamic programming: computation of scores

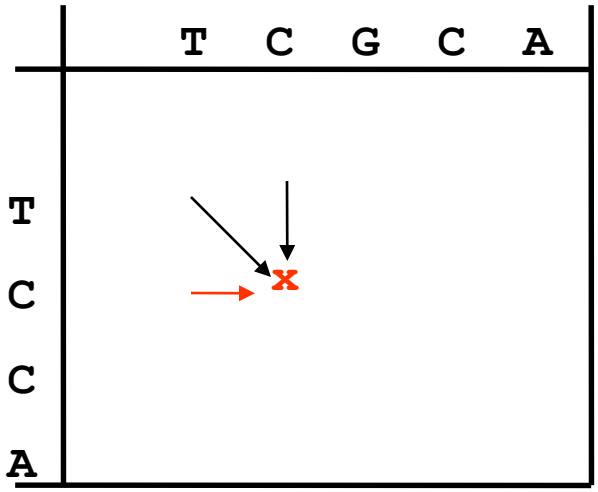


Any given point in matrix can only be reached from three possible positions (you cannot “align backwards”).

=> Best scoring alignment ending in any given point in the matrix can be found by choosing the highest scoring of the three possibilities.

$$\text{score}(x,y) = \max \begin{cases} \text{score}(x,y-1) - \text{gap-penalty} \\ \text{score}(x-1,y-1) + \text{substitution-score}(x,y) \end{cases}$$

Dynamic programming: computation of scores

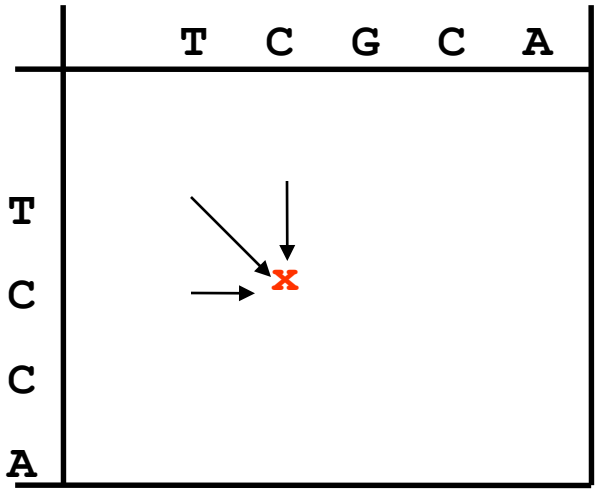


Any given point in matrix can only be reached from three possible positions (you cannot “align backwards”).

=> Best scoring alignment ending in any given point in the matrix can be found by choosing the highest scoring of the three possibilities.

$$\text{score}(x,y) = \max \begin{cases} \text{score}(x,y-1) - \text{gap-penalty} \\ \text{score}(x-1,y-1) + \text{substitution-score}(x,y) \\ \text{score}(x-1,y) - \text{gap-penalty} \end{cases}$$

Dynamic programming: computation of scores



Any given point in matrix can only be reached from three possible positions (you cannot “align backwards”).

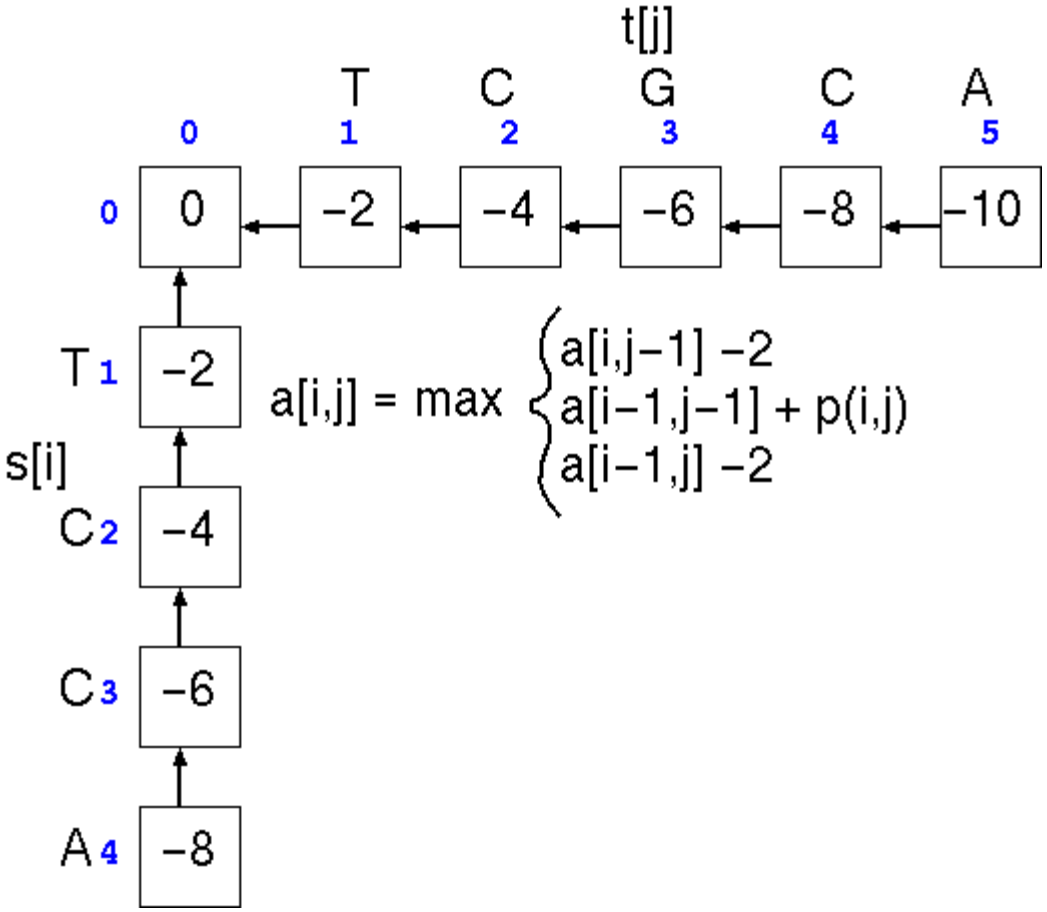
=> Best scoring alignment ending in any given point in the matrix can be found by choosing the highest scoring of the three possibilities.

Each new score is found by choosing the maximum of three possibilities. For each square in matrix: keep track of where best score came from.

Fill in scores one row at a time, starting in upper left corner of matrix, ending in lower right corner.

$$\text{score}(x,y) = \max \begin{cases} \text{score}(x,y-1) - \text{gap-penalty} \\ \text{score}(x-1,y-1) + \text{substitution-score}(x,y) \\ \text{score}(x-1,y) - \text{gap-penalty} \end{cases}$$

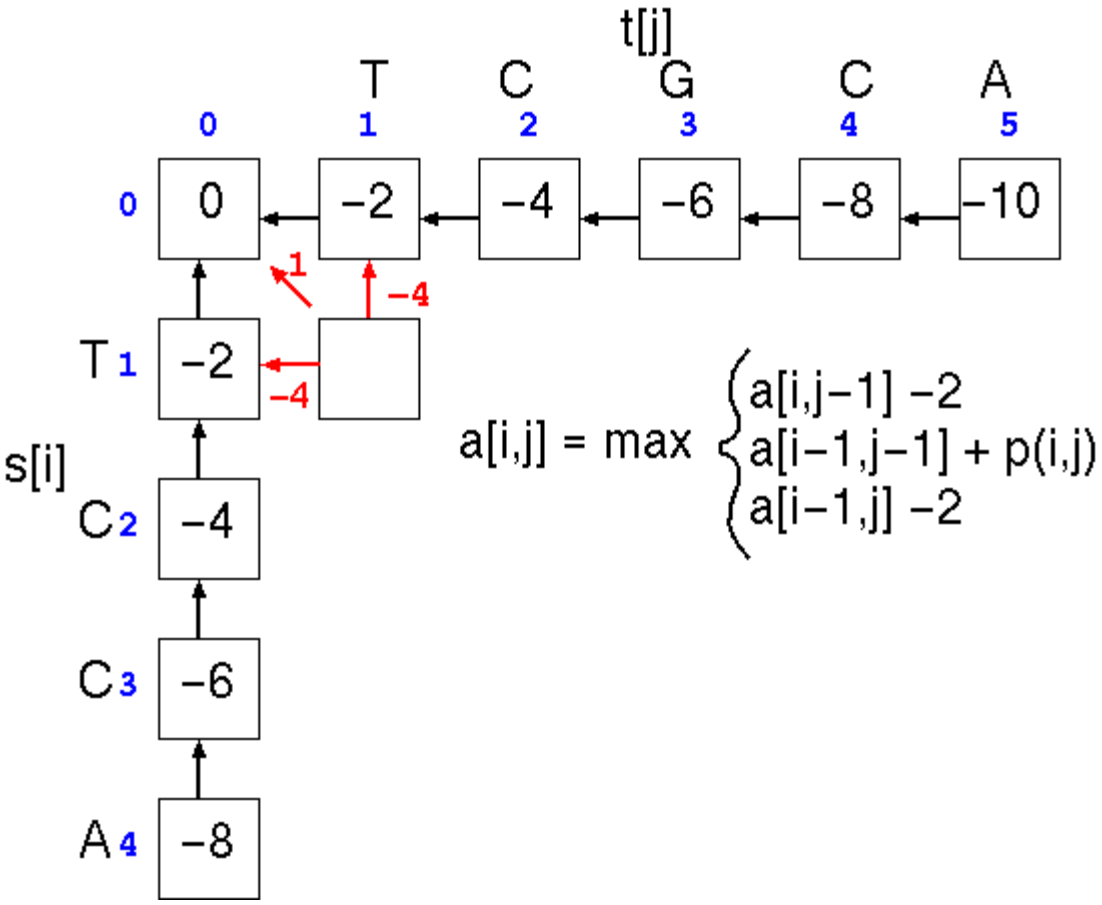
Dynamic programming: example



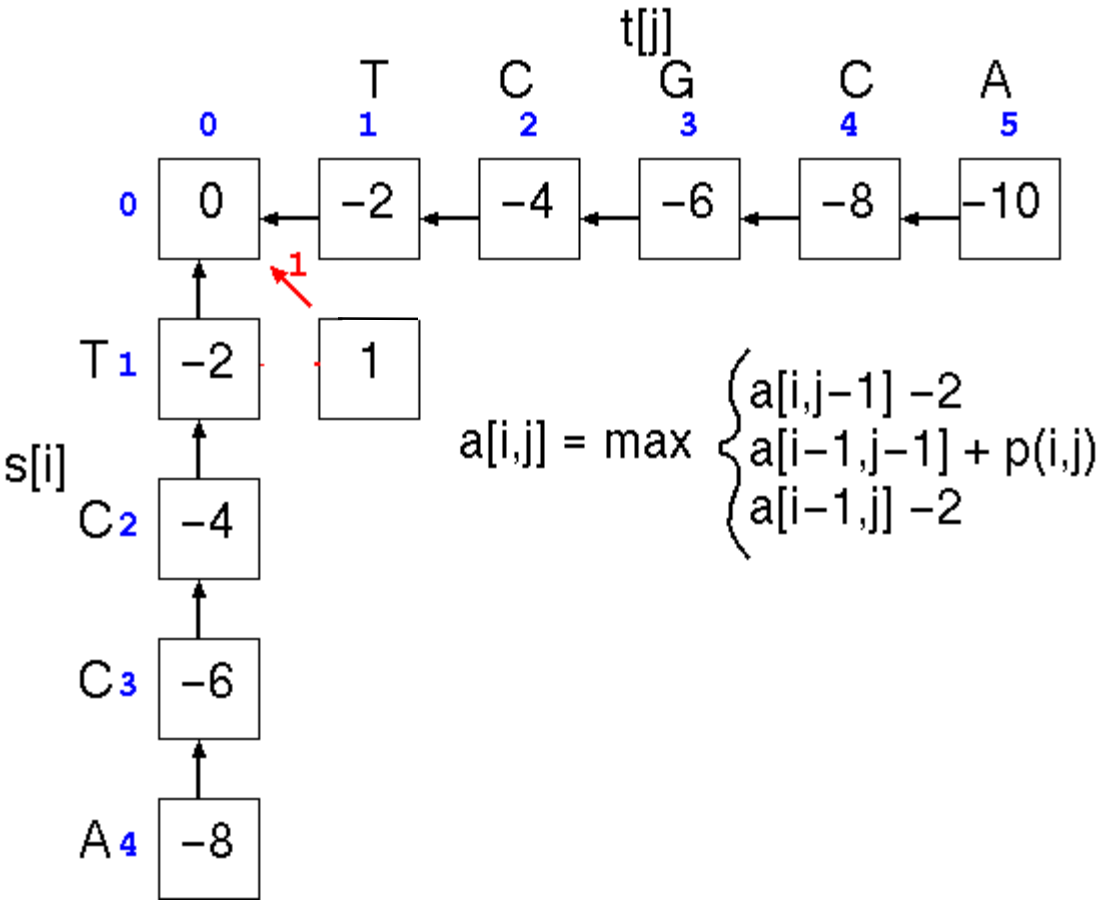
	A	C	G	T
A	1	-1	-1	-1
C	-1	1	-1	-1
G	-1	-1	1	-1
T	-1	-1	-1	1

Gaps: -2

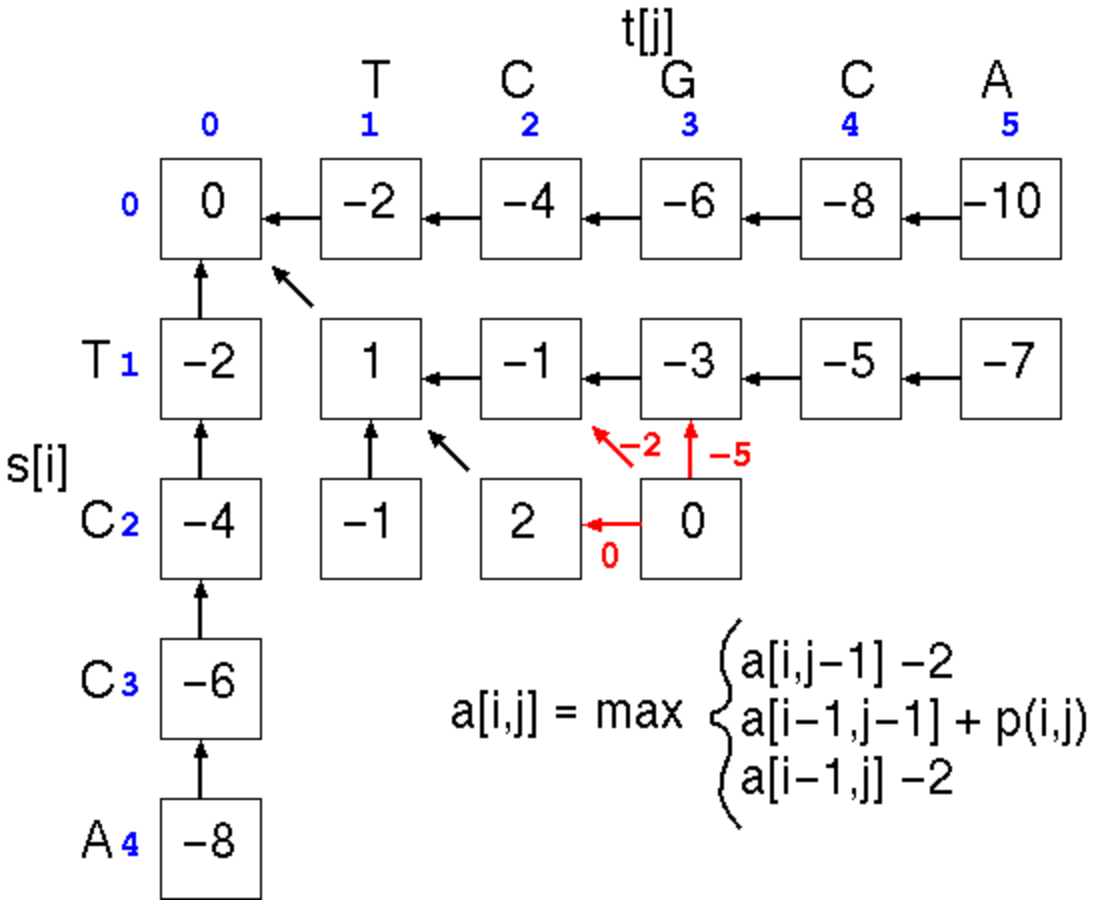
Dynamic programming: example



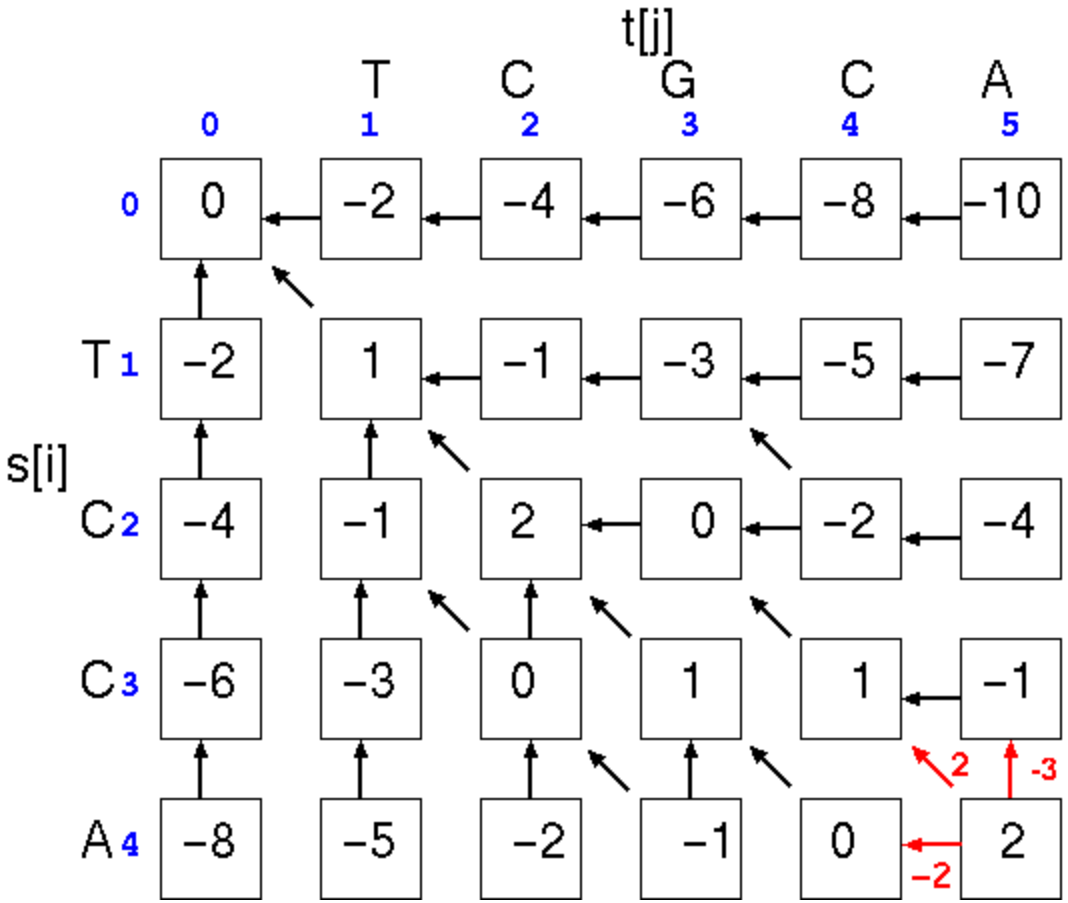
Dynamic programming: example



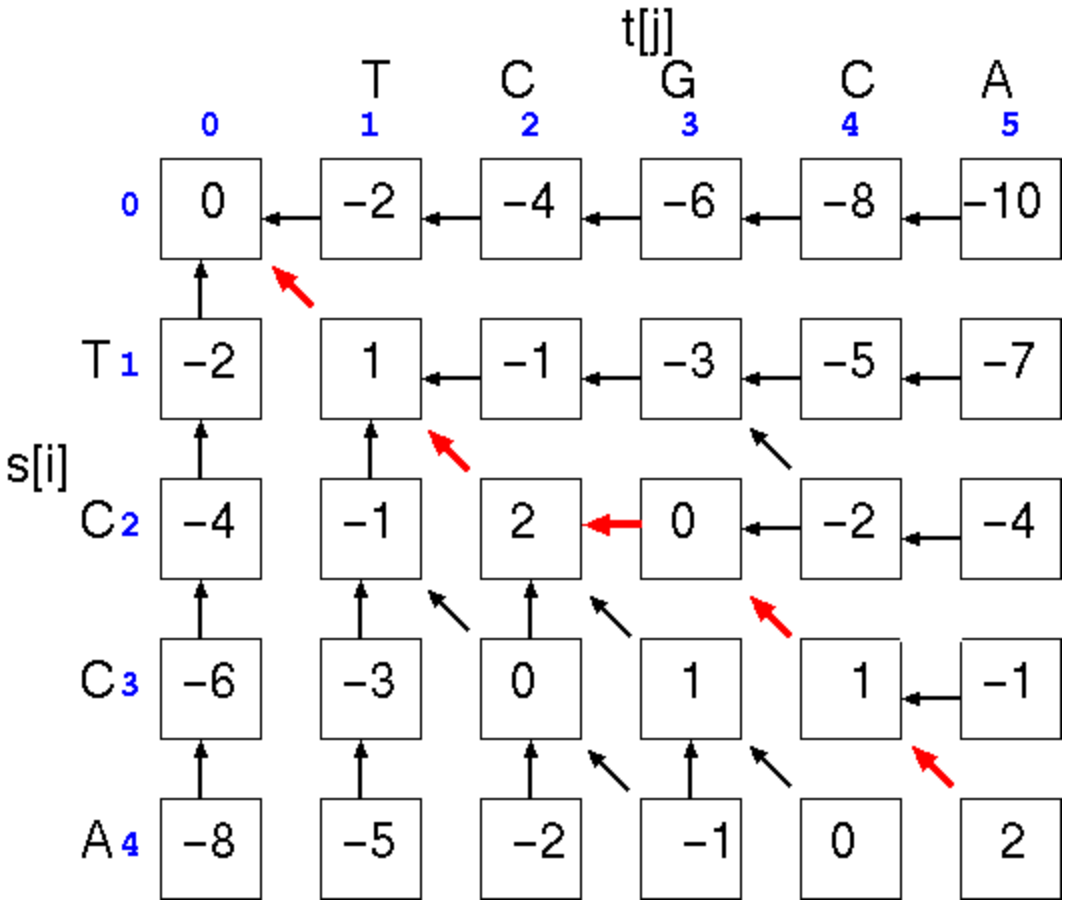
Dynamic programming: example



Dynamic programming: example



Dynamic programming: example

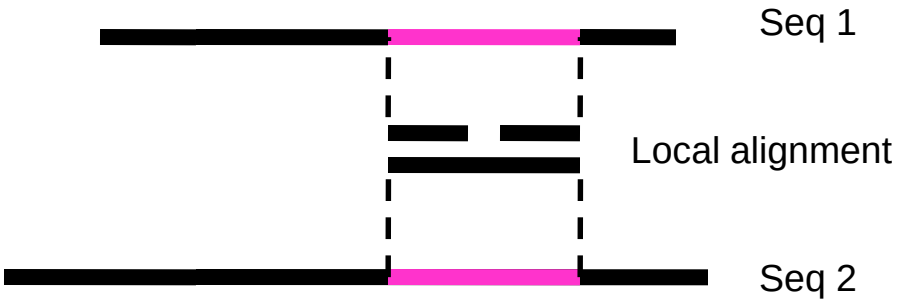



```
T C G C A
: :   : :
T C - C A
---------
1+1-2+1+1 = 2
```

Global versus local alignments

- Global alignment: align full length of both sequences. (The "Needleman-Wunsch" algorithm).

- Local alignment: find best partial alignment of two sequences (the "Smith-Waterman" algorithm).



Local alignment overview

- The recursive formula is changed by adding a fourth possibility: zero. This means local alignment scores are never negative.

$$\text{score}(x,y) = \max \begin{cases} \text{score}(x,y-1) - \text{gap-penalty} \\ \text{score}(x-1,y-1) + \text{substitution-score}(x,y) \\ \text{score}(x-1,y) - \text{gap-penalty} \\ 0 \end{cases}$$

- Trace-back is started at the highest value rather than in lower right corner
- Trace-back is stopped as soon as a zero is encountered

Local alignment: example