Design and Analysis of Algorithms (UE17CS251)

Unit IV - Dynamic Programming

Mr. Channa Bankapur
channabankapur@pes.edu

Eg: Coin-row problem

There is a row of n coins whose values are some positive integers c_1, c_2, …, c_n, not necessarily distinct. The goal is to pick up the maximum amount of money subject to the constraint that no two coins adjacent in the initial row can be picked up.

Eg: Coin-row problem

There is a row of n coins whose values are some positive integers $c_1, c_2, \ldots, c_n$, not necessarily distinct. The goal is to pick up the maximum amount of money subject to the constraint that no two coins adjacent in the initial row can be picked up.

$F(n) = \max\{ F(n-1), c_n + F(n-2) \}$ for $n>1$

where $F(0) = 0$, $F(1) = c_1$

Fibonacci numbers: 1, 1, 2, 3, 5, 8, 13, 21, …

F(n) = F(n-1) + F(n-2)

F(1) = F(2) = 1

Computing the nth Fibonacci number recursively (top-down):

F(n)

F(n-1) + F(n-2)

F(n-2) + F(n-3) F(n-3) + F(n-4)

...

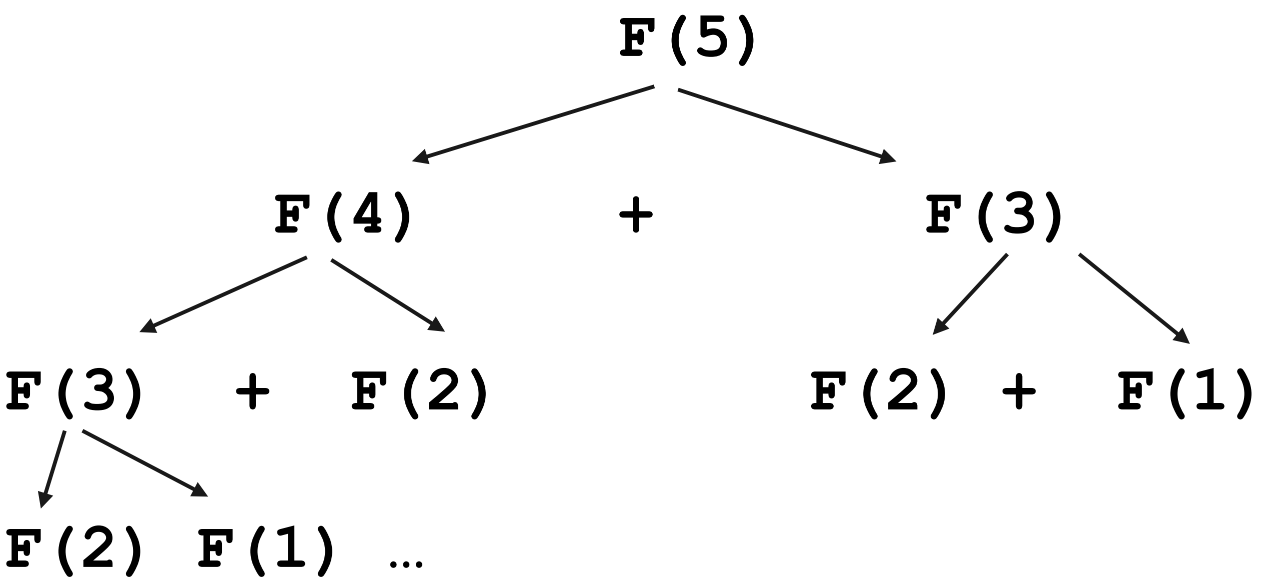
```
Algorithm Fib_TopDown(n)
//Computes nth Fibonacci Number recursively
//Input: positive integer n
//Output: nth Fibonacci Number

if(n = 1 OR n = 2) return 1
return Fib_TopDown(n-1) + Fib_TopDown(n-2)
```

Fibonacci numbers: 1, 1, 2, 3, 5, 8, 13, 21, …

F(n) = F(n-1) + F(n-2)

F(1) = F(2) = 1

Computing the 5th Fibonacci number recursively (top-down):



```
Algorithm Fibonacci_DP_TopDown(n, F)
//Computes nth Fibonacci Number using a table
// to avoid recomputing subproblems
//Input: positive integer n and array F where
// F[i] is either ith Fibonacci Number or -1
// indicating it's not yet computed.
//Output: nth Fibonacci Number

if(F[n] ≠ -1) return F[n]
F[n] ← Fibonacci_DP_TopDown(n-1, F) +
       Fibonacci_DP_TopDown(n-2, F)
return F[n]
```

Computing 6th Fibonacci number:

F(*6*) = *F*(*5*) + *F*(*4*)

F(5) = F(4) + F(3)

F(4) = *F(3)* + F(2)

F(3) = F(2) + F(1)

F(2) = 1

F(1) = 1

F(1)	**1**
F(2)	**1**
F(3)	-1
F(4)	-1
F(5)	-1
F(6)	-1

Computing 6th Fibonacci number:

F(*6*) = *F*(*5*) + *F*(*4*)

F(5) = F(4) + F(3)

F(4) = *F(3)* + F(2)

F(3) = 1 + 1 = 2

F(2) = 1

F(1) = 1

F(1)	1
F(2)	1
F(3)	**2**
F(4)	-1
F(5)	-1
F(6)	-1

Computing 6th Fibonacci number:

F(*6*) = *F*(*5*) + *F*(*4*)

F(5) = F(4) + F(3)

***F*(4) = *2* + 1 = 3**

F(3) = 1 + 1 = 2

F(2) = 1

F(1) = 1

F(1)	1
F(2)	1
F(3)	2
F(4)	**3**
F(5)	-1
F(6)	-1

Computing 6th Fibonacci number:

F(*6*) = *F*(*5*) + *F*(*4*)

***F*(5) = 3 + 2 = 5**

F(4) = *2* + 1 = 3

F(3) = 1 + 1 = 2

F(2) = 1

F(1) = 1

F(1)	1
F(2)	1
F(3)	2
F(4)	3
F(5)	**5**
F(6)	-1

Computing 6th Fibonacci number:

F*(*6*) = *5* + *3* = *8

F(5) = 3 + 2 = 5

F(4) = *2* + 1 = 3

F(3) = 1 + 1 = 2

F(2) = 1

F(1) = 1

F(1)	1
F(2)	1
F(3)	2
F(4)	3
F(5)	5
F(6)	**8**

```
Algorithm Fibonacci_DP_BottomUp(n)
//Computes nth Fibonacci Number using
// bottom-up approach of Dynamic Programming
//Input: positive integer n
//Output: nth Fibonacci Number

F[1] ← F[2] ← 1
for i ← 3 to n
  F[i] ← F[i-1] + F[i-2]
return F[n]
```

```
Algorithm Fibonacci_BottomUp(n)
//Computes nth Fibonacci Number
//Input: positive integer n
//Output: nth Fibonacci Number

F ← Fprev ← Fpp ← 1
for i ← 3 to n
   F ← Fprev + Fpp
      Fpp ← Fprev
      Fprev ← F
return F
```

Q: How many bit strings of length 8 does not have two consecutive zeros.
(10110110 is one of them, but 11100111 is not.)

Soln: …

Q: How many bit strings of length 8 does not have consecutive two zeros.

(10110110 is one of them, but 11100111 is not.)

Soln: $f(n) = f(n-1) + f(n-2)$,

where $f(1) = 2$, $f(2) = 3$

Q: How many bit strings of length 8 does not have consecutive three zeros.

(10010110 is one of them, but 11000111 is not.)

Soln: $f(n) = f(n-1) + f(n-2) + f(n-3)$,

where $f(1) = 2$, $f(2) = 4$, $f(3) = 7$

Dynamic Programming: is a general algorithm design technique for solving problems defined by recurrences with overlapping subproblems.

Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems and later assimilated by Computer Science.

“Programming” here means “planning”.

Main idea:

- set up a recurrence of a solution, which happens to solve overlapping subproblems.
- solve subproblems once and record solutions in a table.
- extract solution from the table whenever required to solve the subproblem.

Example sequence: 1, 2, …, 20, 21, …, 30, 32, 34, …

$F(n) = F(n - 10) + F(n - 20)$

$F(i) = i$ for $1 \leq i \leq 20$

Computing the 107th number recursively (top-down):

```
                        F(107)

            F(97)         +          F(87)

F(87)  +   F(77)               F(77)  +   F(67)

F(77) F(67) …
```

Binomial coefficients are coefficients of the binomial formula:

$(a + b)^n = C(n,0)a^nb^0 + \ldots + C(n,k)a^{n-k}b^k + \ldots + C(n,n)a^0b^n$

Recurrence from the Pascal's identity:

C(n,k) = C(n-1,k-1) + C(n-1,k) for $n > k > 0$

C(n,0) = 1, C(n,n) = 1 for $n \geq 0$

Pascal's Triangle:

	0	**1**	**2**	**3**	**4**	**5**
0	1					
1	1	1				
2	1	2	1			
3	1	3	3	1		
4	1	4	6	4	1	
5	1	5	10	10	5	1

Value of C(n,k) can be computed by filling a table:

	0	**1**	**2**	**. . .**	**k-1**			**k**
0		1						
1		1	1					
2		1	2	1				
.								
.								
k		1	k					
		1						
k+1	1	k+1						
	k+1							
.								
.								
n-1	1					C(n-1,k-1)	C(n-	
1,k)								
n		1						

Bottom-up algorithm to compute Binomial Coefficient using Dynamic Programming technique

ALGORITHM *Binomial*(n, k)

//Computes $C(n, k)$ by the dynamic programming algorithm
//Input: A pair of nonnegative integers $n \geq k \geq 0$
//Output: The value of $C(n, k)$

for $i \leftarrow 0$ **to** n **do**
    for $j \leftarrow 0$ **to** $\min(i, k)$ **do**
        if $j = 0$ **or** $j = i$
            $C[i, j] \leftarrow 1$
        else $C[i, j] \leftarrow C[i-1, j-1] + C[i-1, j]$
return $C[n, k]$

Time complexity:

$$\sum_{i=1}^{k}\sum_{j=1}^{i-1} 1 + \sum_{i=k+1}^{n}\sum_{j=1}^{k} 1$$

$$= \sum_{i=1}^{k}(i - 1) + \sum_{i=k+1}^{n}(k)$$

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

Exercise for students:
Write a top-down algorithm to compute Binomial Coefficient using Dynamic Programming strategy.

Knapsack Problem:

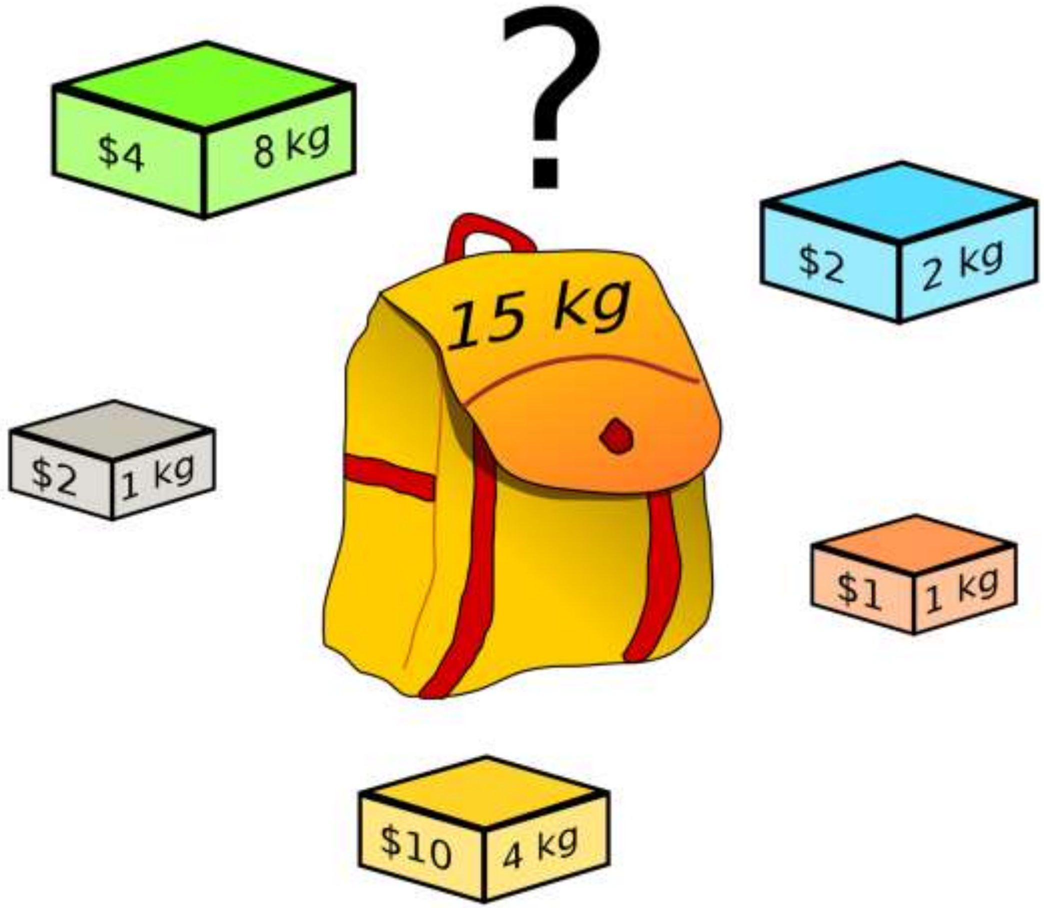
Given n items:

weights: w_1 w_2 … w_n

values: v_1 v_2 … v_n

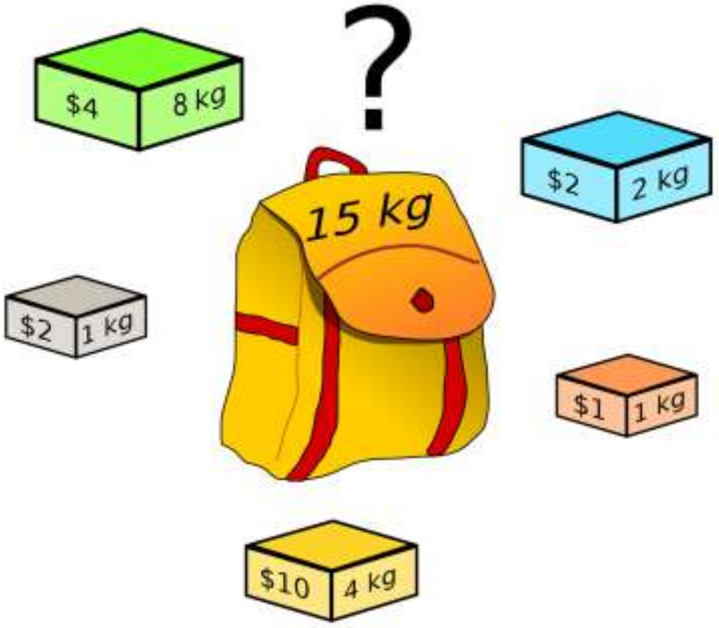
a knapsack of capacity W

Find most valuable subset of the items that fit into the knapsack.



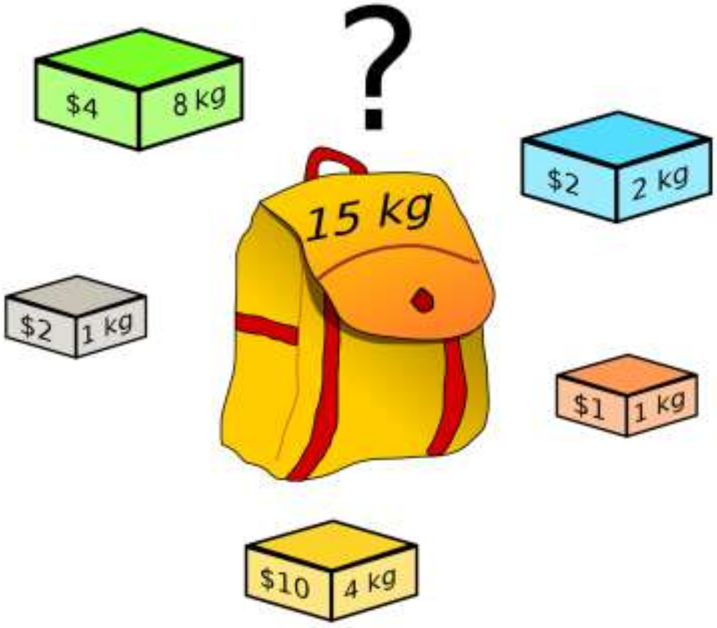
Knapsack Problem:

```
f(v[1..n], w[1..n], W)

if(w[n] > W) return
   f(v[1..n-1],w[1..n-1],W)

return
   max(v[n] +  f(v[1..n-1],
          w[1..n-1], W-w[n]),
   f(v[1..n-1],w[1..n-1],W))
```



Knapsack Problem:

```
f(v[1..n], w[1..n], W)

if(n=0 OR W=0) return 0

if(w[n] > W) return
   f(v[1..n-1],w[1..n-1],W)

return
   max(v[n] +  f(v[1..n-1],
         w[1..n-1], W-w[n]),
   f(v[1..n-1],w[1..n-1],W))
```



Knapsack Problem and Memory Functions

$$F(i, j) = \begin{cases} \max\{F(i-1, j),\ v_i + F(i-1, j-w_i)\} & \text{if } j - w_i \geq 0, \\ F(i-1, j) & \text{if } j - w_i < 0. \end{cases}$$

$$F(0, j) = 0 \text{ for } j \geq 0 \quad \text{and} \quad F(i, 0) = 0 \text{ for } i \geq 0.$$

		0	$j-w_i$	j	W
	0	0	0	0	0
	$i-1$	0	$F(i-1, j-w_i)$	$F(i-1, j)$	
w_i, v_i	i	0		$F(i, j)$	
	n	0			goal

Knapsack Problem

item	weight	value
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

capacity $W = 5$.

	i	capacity j 0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2, v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1, v_2 = 10$	2	0	10	12	22	22	22
$w_3 = 3, v_3 = 20$	3	0	10	12	22	30	32
$w_4 = 2, v_4 = 15$	4	0	10	15	25	30	**37**

Knapsack Problem and Memory Functions

item	weight	value
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

capacity $W = 5$.

		capacity j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2, v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1, v_2 = 10$	2	0	—	12	22	—	22
$w_3 = 3, v_3 = 20$	3	0	—	—	22	—	32
$w_4 = 2, v_4 = 15$	4	0	—	—	—	—	**37**

ALGORITHM *MFKnapsack*(i, j)

```
//Implements the memory function method for the knapsack problem
//Input: A nonnegative integer i indicating the number of the first
//       items being considered and a nonnegative integer j indicating
//       the knapsack capacity
//Output: The value of an optimal feasible subset of the first i items
//Note: Uses as global variables input arrays Weights[1..n], Values[1..n],
//and table F[0..n, 0..W] whose entries are initialized with −1's except for
//row 0 and column 0 initialized with 0's
if F[i, j] < 0
    if j < Weights[i]
        value ← MFKnapsack(i − 1, j)
    else
        value ← max(MFKnapsack(i − 1, j),
                    Values[i] + MFKnapsack(i − 1, j − Weights[i]))
    F[i, j] ← value
return F[i, j]
```

capacity $\longrightarrow$

w_i v_i	item	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
(1, 4)	1	0						
(2, 5)	2	0						
(3, 10)	3	0						
(4, 13)	4	0						

capacity →

w_i v_i	item	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
(1, 4)	1	0	4	4	4	4	4	4
(2, 5)	2	0	4					

w_i v_i	item	capacity → 0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
(1, 4)	1	0	4	4	4	4	4	4
(2, 5)	2	0	4	5	9	9	9	9

capacity →

w_i v_i	item	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
(1, 4)	1	0	4	4	4	4	4	4
(2, 5)	2	0	4	5	9	9	9	9
(3, 10)	3	0	4	5	10	14	15	19
(4, 13)	4	0						

capacity →

w_i v_i	item	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
(1, 4)	1	0	4	4	4	4	4	4
(2, 5)	2	0	4	5	9	9	9	9
(3, 10)	3	0	4	5	10	14	15	19
(4, 13)	4	0	4	5	10	14	17	19

capacity →

w_i v_i	item	0	1	2	3	4	5	6
	0	$\boxed{0}$	0	0	0	0	0	0
(1, 4)	1	0	$\boxed{4}$ ✓	4	4	4	4	4
(2, 5)	2	0	4	5	$\boxed{9}$ ✓	9	9	9
(3, 10)	3	0	4	5	10	14	15	$\boxed{19}$ ✓
(4, 13)	4	0	4	5	10	14	17	$\boxed{19}$

Items in the knapsack = {1, 2, 3}, Value = 4+5+10 = 19

w_i v_i		0	1	2	3	4	5
	0	0	0	0	0	0	0
(2, 12)	1	0					
(1, 10)	2	0					
(3, 20)	3	0					
(2, 15)	4	0					

w_i v_i		0	1	2	3	4	5
	0						
(1, 1)	1						
(1, 2)	2						
(4, 10)	3						
(2, 2)	4						

(w_i, v_i)		0	1	2	3	4	5	6	7	8
	0									
(1, 10)	1									
(2, 5)	2									
(4, 41)	3									
(3, 30)	4									
(5, 52)	5									

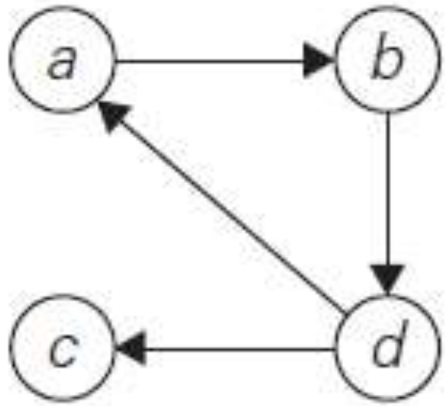
(w_i, v_i)		0	1	2	3	4	5
	0	0	0	0	0	0	0
(2, 12)	1	0	0	12	12	12	12
(1, 10)	2	0	10	12	22	22	22
(3, 20)	3	0	10	12	22	30	32
(2, 15)	4	0	10	15	25	30	37

(w_i, v_i)		0	1	2	3	4	5
	0	0	0	0	0	0	0
(1, 1)	1	0	1	1	1	1	1
(1, 2)	2	0	2	3	3	3	3
(4, 10)	3	0	2	3	3	10	12
(2, 2)	4	0	2	3	4	10	12

(w_i, v_i)		0	1	2	3	4	5	6	7	8
	0	0	0	0	0	0	0	0	0	0
(1, 10)	1	0	10	10	10	10	10	10	10	10
(2, 5)	2	0	10	10	15	15	15	15	15	15
(4, 41)	3	0	10	10	15	41	51	51	56	56
(3, 30)	4	0	10	10	30	41	51	51	71	81
(5, 52)	5	0	10	10	30	41	52	62	71	82

Transitive closure of a digraph

- to find all-pairs-reachability in a digraph



Digraph

$$A = \begin{array}{c c} & \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

Its adjacency matrix

Its transitive closure $$T = \begin{array}{c c} & \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

Finding Transitive Closure of a digraph

```
for i …
   for j …
      for k …
         if( A[i,j] AND A[j,k])
            then A[i,k] = 1
```

$O(n^3)$

Finding Transitive Closure of a digraph

```
for i ...
   for j ...
      for k ...
         if( A[i,j] AND A[j,k]) then A[i,k]=1
```

O(n^3) But, it's not correct

```
Flag ← TRUE
while(flag)
   flag ← FALSE
   for i ...
      for j ...
         for k ...
            if( A[i,j] AND A[j,k] AND A[i,k]=0)
               then A[i,k] = 1, flag = TRUE
```

O(n^4)

Finding Transitive Closure of a digraph

```
for each vertex v
      dfs(v)
      if i is reachable from v
         then A[v,i] ← 1
```

$O(n^3)$

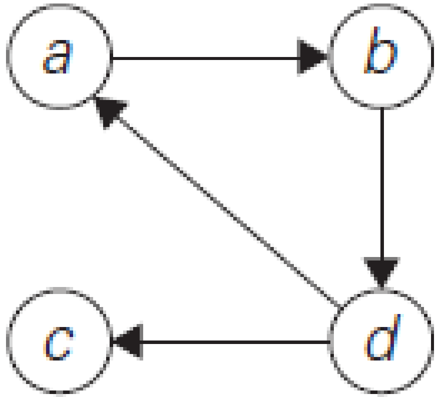
```
for each vertex v
      bfs(v)
      if i is reachable from v
         then A[v,i] ← 1
```

Warshall's Algorithm

$$R^{(0)}, \ldots, R^{(k-1)}, R^{(k)}, \ldots R^{(n)}$$

$$r_{ij}^{(k)} = r_{ij}^{(k-1)} \quad \text{or} \quad \left(r_{ik}^{(k-1)} \text{ and } r_{kj}^{(k-1)} \right)$$

$$R^{(k-1)} = \begin{array}{cc} & \begin{array}{ccc} j & & k \end{array} \\ \begin{array}{c} k \\ \\ i \end{array} & \left[\begin{array}{ccc} 1 & & \\ \uparrow & & \\ 0 & \rightarrow & 1 \end{array} \right] \end{array} \implies R^{(k)} = \begin{array}{cc} & \begin{array}{cc} j & k \end{array} \\ \begin{array}{c} k \\ i \end{array} & \left[\begin{array}{cc} 1 & \\ 1 & 1 \end{array} \right] \end{array}$$



$$R^{(0)} = \begin{array}{cc} & \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

$$R^{(1)} = \begin{array}{cc} & \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & \mathbf{1} & 1 & 0 \end{bmatrix} \end{array}$$

$$R^{(2)} = \begin{array}{cc} & \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \begin{bmatrix} 0 & 1 & 0 & \mathbf{1} \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & \mathbf{1} \end{bmatrix} \end{array}$$

$$R^{(3)} = \begin{array}{cc} & \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

$$R^{(4)} = \begin{array}{cc} & \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \begin{bmatrix} \mathbf{1} & 1 & \mathbf{1} & 1 \\ \mathbf{1} & \mathbf{1} & \mathbf{1} & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

Warshall's Algorithm

Adjacency Matrix A :

0 1 0 0 0 0

0 0 1 0 0 0

1 0 0 1 0 0

0 0 0 0 1 0

0 0 0 1 0 0

0 0 0 0 1 0

Find the transitive closure of A.

Warshall's Algorithm

Adjacency Matrix A :

0 1 0 0 0 0
0 0 1 0 0 0
1 0 0 1 0 0
0 0 0 0 1 0
0 0 0 1 0 0
0 0 0 0 1 0

Transitive closure of A :

1 1 **1** **1** **1** 0
1 **1** 1 **1** **1** 0
1 **1** **1** 1 **1** 0
0 0 0 **1** 1 0
0 0 0 1 **1** 0
0 0 0 **1** 1 0

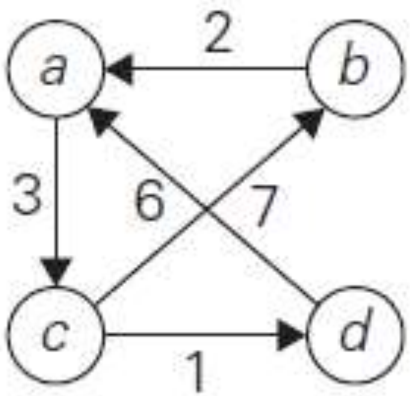
ALGORITHM *Warshall*$(A[1..n, 1..n])$

//Implements Warshall's algorithm for computing the transitive closure
//Input: The adjacency matrix A of a digraph with n vertices
//Output: The transitive closure of the digraph

$R^{(0)} \leftarrow A$
for $k \leftarrow 1$ **to** n **do**
    for $i \leftarrow 1$ **to** n **do**
        for $j \leftarrow 1$ **to** n **do**
            $R^{(k)}[i, j] \leftarrow R^{(k-1)}[i, j]$ **or** $(R^{(k-1)}[i, k]$ **and** $R^{(k-1)}[k, j])$
return $R^{(n)}$

Floyd's Algorithm

To find all-pairs shortest-paths in a weighted connected graph (undirected or directed) which does not contain a cycle of negative length.



$$W = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{array}$$ weight matrix

$$D = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 7 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{array}$$ distance matrix

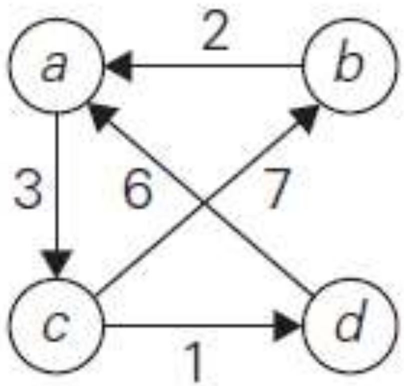
Floyd's Algorithm
To find all-pairs shortest-paths in a weighted connected graph (undirected or directed) which does not contain a cycle of negative length.

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

the element $d_{ij}^{(k)}$ in the ith row and the jth column of matrix $D^{(k)}$ ($i, j = 1, 2, \ldots, n$, $k = 0, 1, \ldots, n$) is equal to the length of the shortest path among all paths from the ith vertex to the jth vertex with each intermediate vertex, if any, numbered not higher than k.

ALGORITHM $Floyd(W[1..n, 1..n])$

```
//Implements Floyd's algorithm for the all-pairs shortest-paths problem
//Input: The weight matrix W of a graph with no negative-length cycle
//Output: The distance matrix of the shortest paths' lengths
D ← W //is not necessary if W can be overwritten
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            D[i, j] ← min{D[i, j], D[i, k] + D[k, j]}
return D
```



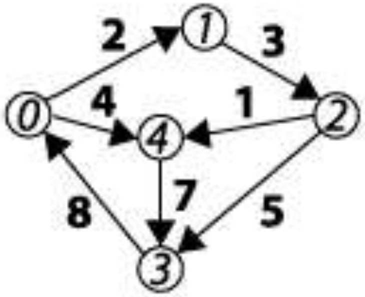
$$D^{(0)} = \begin{array}{cc} & \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

$$D^{(1)} = \begin{array}{cc} & \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \mathbf{5} & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \mathbf{9} & 0 \end{bmatrix} \end{array}$$

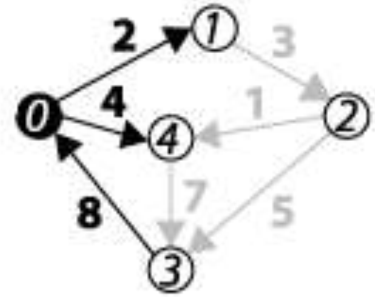
$$D^{(2)} = \begin{array}{cc} & \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & 5 & \infty \\ \mathbf{9} & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{bmatrix} \end{array}$$

$$D^{(3)} = \begin{array}{cc} & \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \begin{bmatrix} 0 & \mathbf{10} & 3 & \mathbf{4} \\ 2 & 0 & 5 & \mathbf{6} \\ 9 & 7 & 0 & 1 \\ 6 & \mathbf{16} & 9 & 0 \end{bmatrix} \end{array}$$

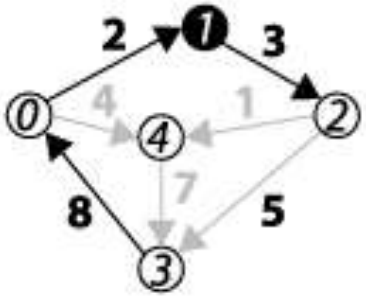
$$D^{(4)} = \begin{array}{cc} & \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ \mathbf{7} & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{array}$$



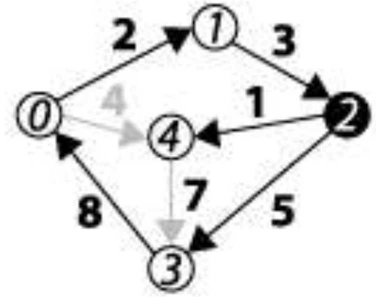
	0	1	2	3	4
0	0	2	∞	∞	4
1	∞	0	3	∞	∞
2	∞	∞	0	5	1
3	8	∞	∞	0	∞
4	∞	∞	∞	7	0



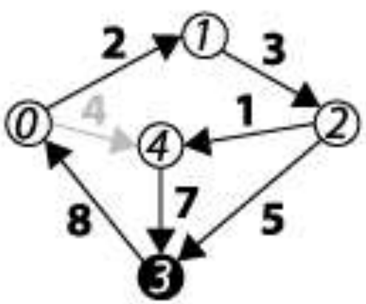
	0	1	2	3	4
0	0	2	∞	∞	4
1	∞	0	3	∞	∞
2	∞	∞	0	5	1
3	8	10	∞	0	12
4	∞	∞	∞	7	0



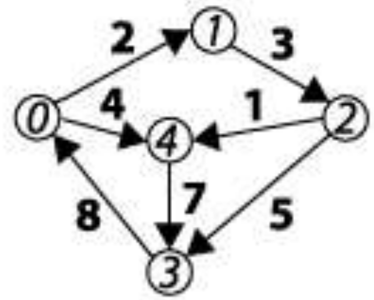
	0	1	2	3	4
0	0	2	5	∞	4
1	∞	0	3	∞	∞
2	∞	∞	0	5	1
3	8	10	13	0	12
4	∞	∞	∞	7	0



	0	1	2	3	4
0	0	2	5	10	4
1	∞	0	3	8	4
2	∞	∞	0	5	1
3	8	10	13	0	12
4	∞	∞	∞	7	0

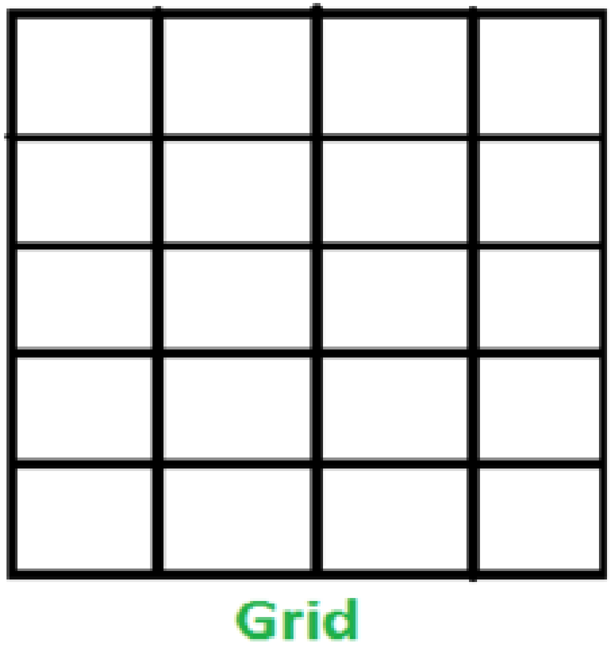


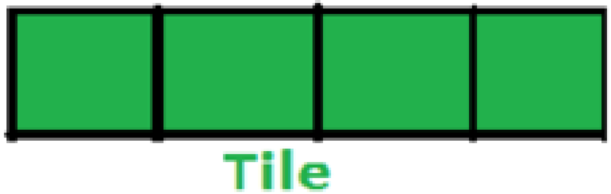
	0	1	2	3	4
0	0	2	5	10	4
1	16	0	3	8	4
2	13	15	0	5	1
3	8	10	13	0	12
4	15	17	20	7	0



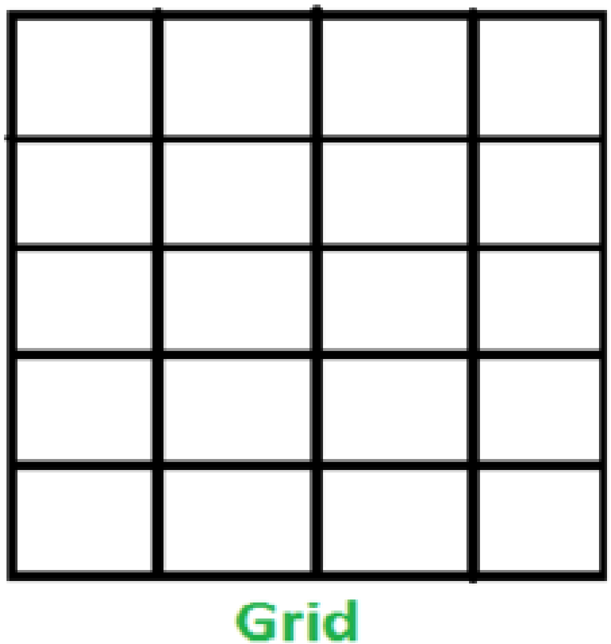
	0	1	2	3	4
0	0	2	5	10	4
1	16	0	3	8	4
2	13	15	0	5	1
3	8	10	13	0	12
4	15	17	20	7	0

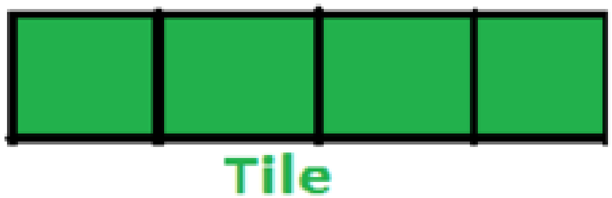
Count number of ways to fill a
"n x 4" grid using "1 x 4" tiles.





Count number of ways to fill a "n x 4" grid using "1 x 4" tiles.





F(n) = **F(n-1)** + **F(n-4)** for n>4
where F(1) = F(2) = F(3)= 1, F(4)= 2

Eg: Coin-collecting problem

Several coins are placed in cells of an **n × m** board, no more than one coin per cell. A robot, located in the upper left cell of the board, needs to collect as many of the coins as possible and bring them to the bottom right cell. On each step, the robot can move either one cell to the **right** or one cell **down** from its current location. When the robot visits a cell with a coin, it always picks up that coin. Design an algorithm to find the **maximum number of coins** the robot can collect and a path it needs to follow to do this.

Eg: Coin-collecting problem

$\mathbf{F(i, j)} = \mathbf{c_{i,j}} + \max\{ \mathbf{F(i\text{-}1, j)}, \mathbf{F(i, j\text{-}1)} \}$

for $1 \leq i \leq n$, $1 \leq j \leq n$

where $F(i, 0) = 0$ for $1 \leq i \leq n$

$F(0, j) = 0$ for $1 \leq j \leq n$

Longest Common Subsequence (LCS)

Given two sequences, find the length of longest subsequence present in both of them. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous. For example, “abc”, “abg”, “bdf”, “aeg”, ‘”acefg”, .. etc are subsequences of “abcdefg”. So a string of length **n** has **2^n** different possible subsequences.

Examples:

LCS for input sequences “**A**BC**D**G**H**” and “**A**E**D**F**H**R” is “**ADH**” of length **3**.

LCS for input sequences “AG**GTAB**” and “**G**X**T**X**A**Y**B**” is “**GTAB**” of length **4**.

LCS for input sequences “ABCD” and “ZYXPQRS” is “” of length **0**.

Longest Common Subsequence (LCS)

Step 1: The optimal substructure

Let **L[0..k-1]** be an LCS of two sequences **X[0..m-1]** and **Y[0..n-1])**.

- if(X[m-1] = Y[n-1]) then L[k-1] = X[m-1] = Y[n-1] and L[0..k-2] is an LCS of X[0..m-2] and Y[0..n-2].
- if(X[m-1] ≠ Y[n-1]) then
  - if(L[k-1] ≠ X[m-1]) then L[0..k-1] is an LCS of X[0..m-2] and Y[0..n-1].
  - if(L[k-1] ≠ Y[n-1]) then L[0..k-1] is an LCS of X[0..m-1] and Y[0..n-2].

This has an optimal substructure.

Longest Common Subsequence (LCS)

Step 2: A recursive solution

Let **L(X[0..m-1], Y[0..n-1])** be the length of the LCS of two sequences X[0..m-1] and Y[0..n-1]. Following is a recursive definition of L(X[0..m-1], Y[0..n-1]).

Case 1: If last characters of both sequences match (that is, X[m-1] == Y[n-1]) then

L(X[0..m-1], Y[0..n-1]) = 1 + L(X[0..m-2], Y[0..n-2])

Case 2: If last characters of both sequences do not match (that is, X[m-1] != Y[n-1]) then

L(X[0..m-1], Y[0..n-1]) =

MAX (**L(X[0..m-2], Y[0..n-1])**, **L(X[0..m-1], Y[0..n-2])**)

Base case: if n=0 or m=0, then L(X[0..m-1], Y[0..n-1]) = 0

LCS Recurrence

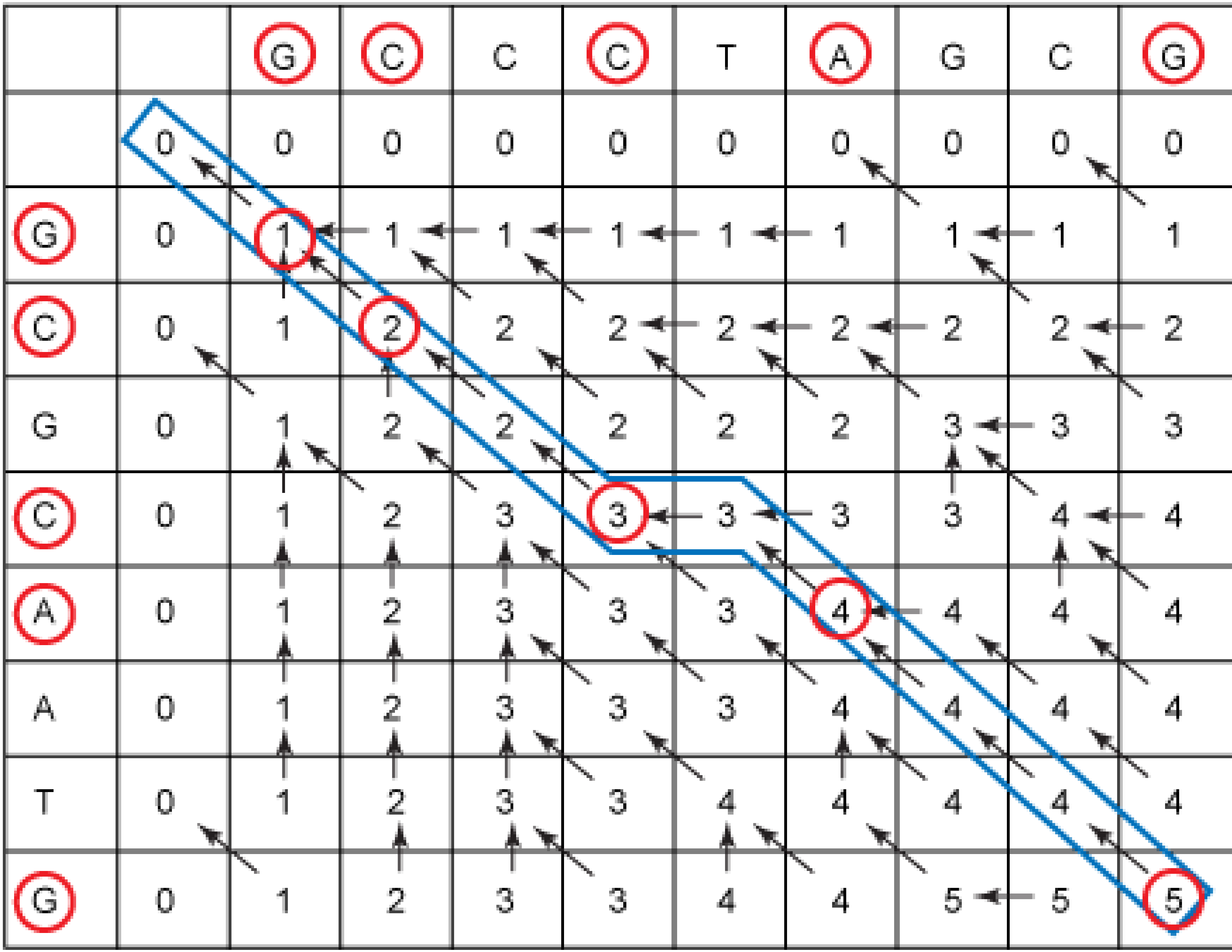
$$LCS(X_i, Y_j) = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ 1 + LCS(X_{i-1}, Y_{j-1}) & \text{if } x_i = y_j \\ \max(LCS(X_i, Y_{j-1}), LCS(X_{i-1}, Y_j)) & \text{if } x_i \neq y_j \end{cases}$$

		G	C	C	C	T	A	G	C	G
G										
C										
G										
C										
A										
A										
T										
G										

		G	C	C	C	T	A	G	C	G
	0	0	0	0	0	0	0	0	0	0
G	0									
C	0									
G	0									
C	0									
A	0									
A	0									
T	0									
G	0									

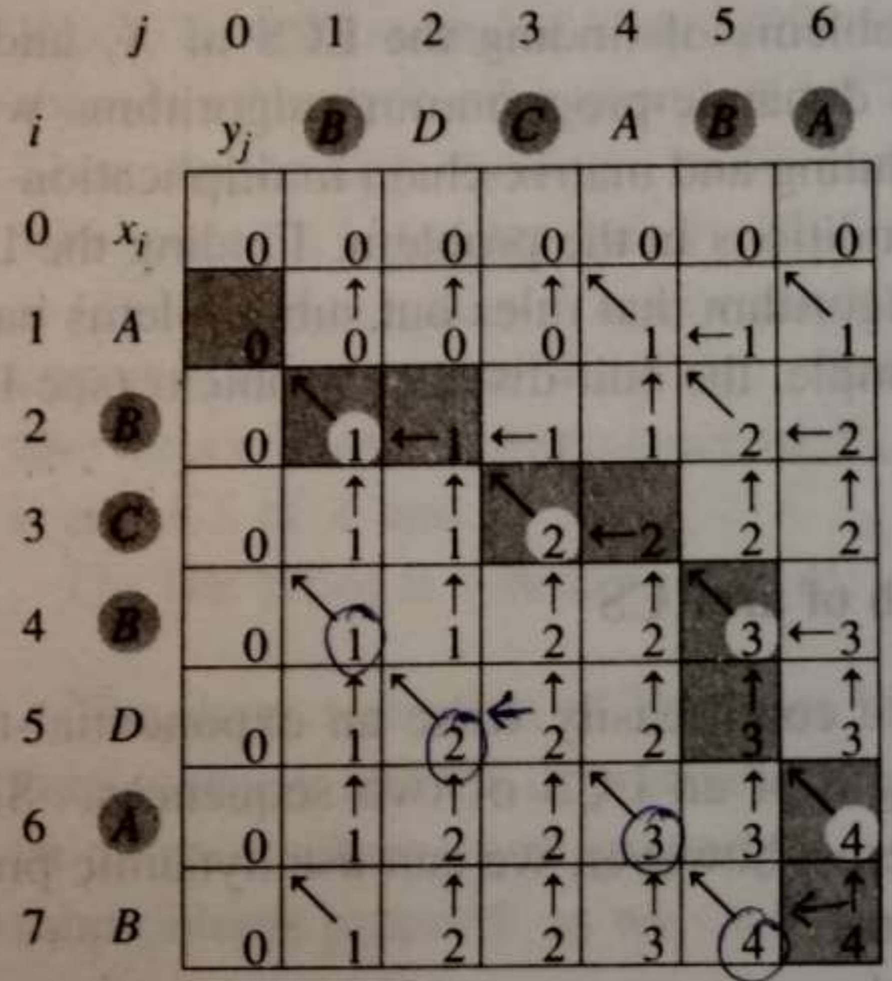
		G	C	C	C	T	A	G	C	G
	0	0	0	0	0	0	0	0	0	0
G	0	1	1	1	1	1	1	1	1	1
C	0	1	2	2	2	2	2	2	2	2
G	0	1	2	2	2	2	2	3	3	3
C	0	1	2	3						
A	0									
A	0									
T	0									
G	0									

		G	C	C	C	T	A	G	C	G
	0	0	0	0	0	0	0	0	0	0
G	0	1	1	1	1	1	1	1	1	1
C	0	1	2	2	2	2	2	2	2	2
G	0	1	2	2	2	2	2	3	3	3
C	0	1	2	3	3	3	3	3	4	4
A	0	1	2	3	3	3	4	4	4	4
A	0	1	2	3	3	3	4	4	4	4
T	0	1	2	3	3	4	4	4	4	4
G	0	1	2	3	3	4	4	5	5	5



G C C C T A G C G
0 0 0 0 0 0 0 0 0 0
G 0 1 1 1 1 1 1 1 1 1
C 0 1 2 2 2 2 2 2 2 2
G 0 1 2 2 2 2 2 3 3 3
C 0 1 2 3 3 3 3 3 4 4
A 0 1 2 3 3 3 4 4 4 4
A 0 1 2 3 3 3 4 4 4 4
T 0 1 2 3 3 4 4 4 4 4
G 0 1 2 3 3 4 4 5 5 5

```
LCS-LENGTH(X, Y)
m ← length[X]
n ← length[Y]
for i ← 1 to m
      do c[i, 0] ← 0
for j ← 0 to n
      do c[0, j] ← 0
for i ← 1 to m
      do for j ← 1 to n
             do if x_i = y_j
                   then c[i, j] ← c[i − 1, j − 1] + 1
                        b[i, j] ← "↖"
                   else if c[i − 1, j] ≥ c[i, j − 1]
                           then c[i, j] ← c[i − 1, j]
                                b[i, j] ← "↑"
                           else c[i, j] ← c[i, j − 1]
                                b[i, j] ← "←"
return c and b
```



BCBA ← Soln 1

BCAB ← Soln 2

BDAB ← Soln 3

Longest Common Subsequence (LCS)

LCS for input sequences “AG**GTAB**” and “**G**X**T**X**A**Y**B**” is “**GTAB**” of length **4**.

	A	G	G	T	A	B
G	-	-	4	-	-	-
X	-	-	-	-	-	-
T	-	-	-	3	-	-
X	-	-	-	-	-	-
A	-	-	-	-	2	-
Y	-	-	-	-	-	-
B	-	-	-	-	-	1

Dynamic Programming is a powerful technique that allows one to solve many different types of problems in $O(n^2)$ or $O(n^3)$ time for which a naive approach would take exponential time!

</ End of Dynamic Programming >