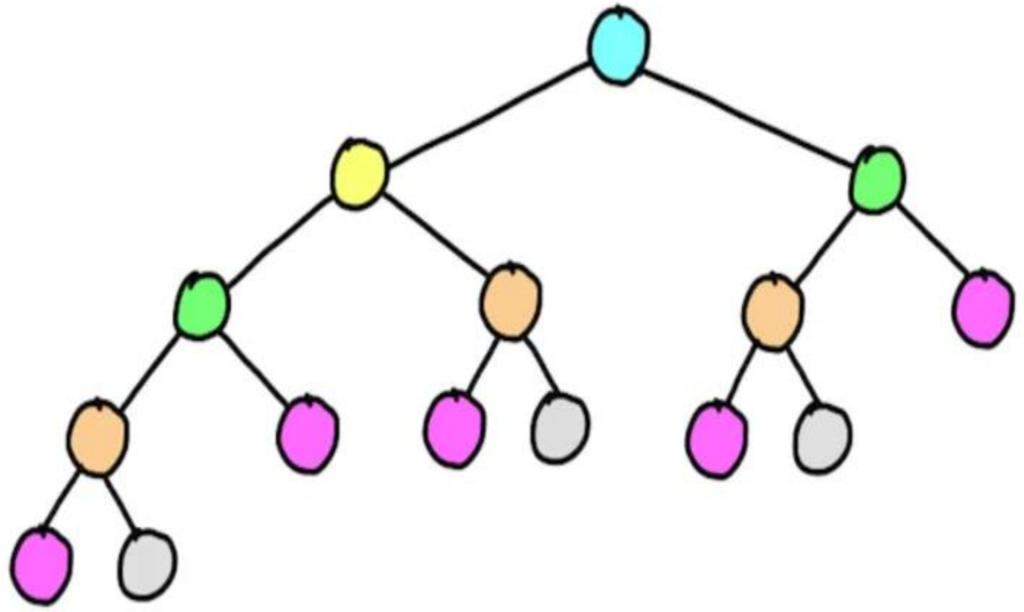
DSE 2256 DESIGN & ANALYSIS OF ALGORITHMS

Lecture 32, 33, 34, 35

Dynamic Programming

Introduction (Finding the n^{th} Fibonacci)
Computing the Binomial Coefficient
Warshall's Algorithm for Transitive Closure
Floyd's All-Pairs Shortest Paths Algorithm
Knapsack Problem using DP



"Those who cannot remember the past are condemned to repeat it."

– **George Santayana**

Instructors:

Dr. Savitha G,
Assistant Professor, DSCA, MIT, Manipal

Dr. Abhilash K. Pai,
Assistant Professor, DSCA, MIT, Manipal

Dynamic Programming

- Dynamic Programming is a general algorithm design technique for solving problems defined by or formulated as recurrences with overlapping sub-instances.

- Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems and "Programming" here means "planning".

Main idea:

1. Set up a recurrence, relating a solution to a given problem with solutions to its smaller subproblems of the same type.
2. Solve smaller instances once.
3. Record solutions in a table.
4. Extract solution to the initial instance from that table.

Dynamic Programming: The idea

Example: Computing the nth Fibonacci number

Recursive definition:

$f(n) = f(n-1) + f(n-2), \quad \text{for } n > 1$
$f(0) = 0$
$f(1) = 1$

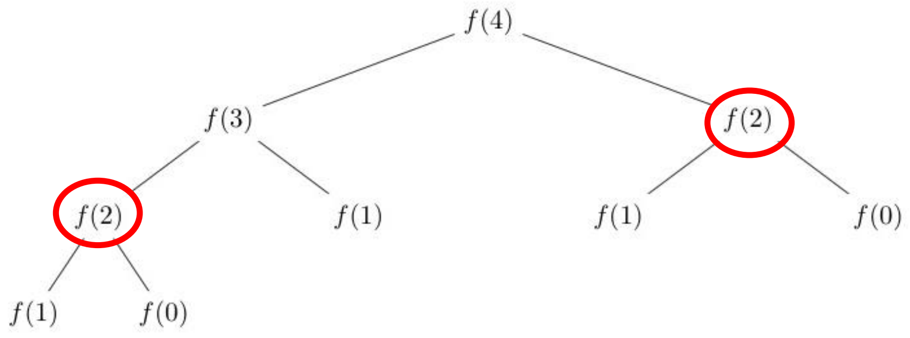
Recursive Algorithm:

```
Function f(n):
{
    if n == 0:
        return 0

    if n == 1:
        return 1

    return f(n - 1) + f(n - 2)
}
```

Visualization of the recursion using recursion tree



Here, f(2) is computed multiple times.

This could be avoided by: storing its value for the first time it is computed and use it again to reduce the number of recursive calls – The philosophy of Dynamic programming.

Time Complexity: O(2^n)

Dynamic Programming: The idea

Example: Computing the nth Fibonacci number

Recursive Algorithm (using Dynamic Programming):

Initialize **mem[0 : n]** = -1
Set **mem[0]** = 0, **mem[1]** = 1

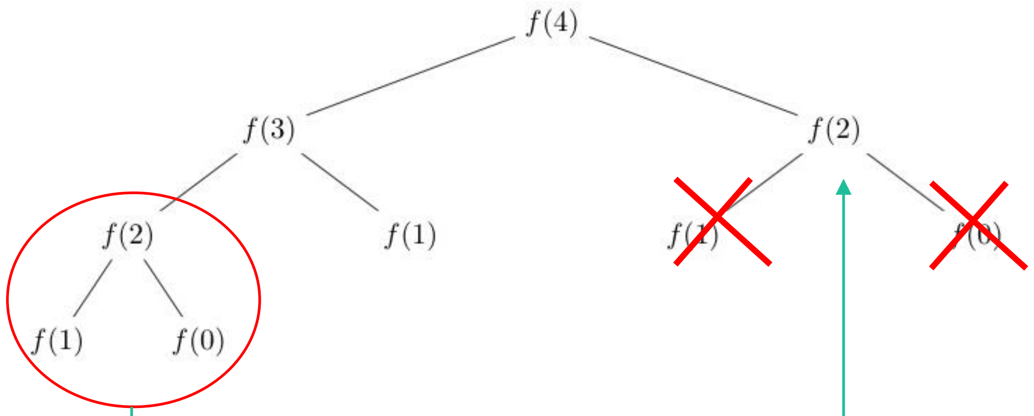
Function **f(n)**:
{
  if mem[n] != -1
    return mem[n]

  mem[n] = f(n - 1) + f(n - 2)

  return mem[n]
}

If this condition is satisfied, then : Return the already computed value of f(n) without recursing further.

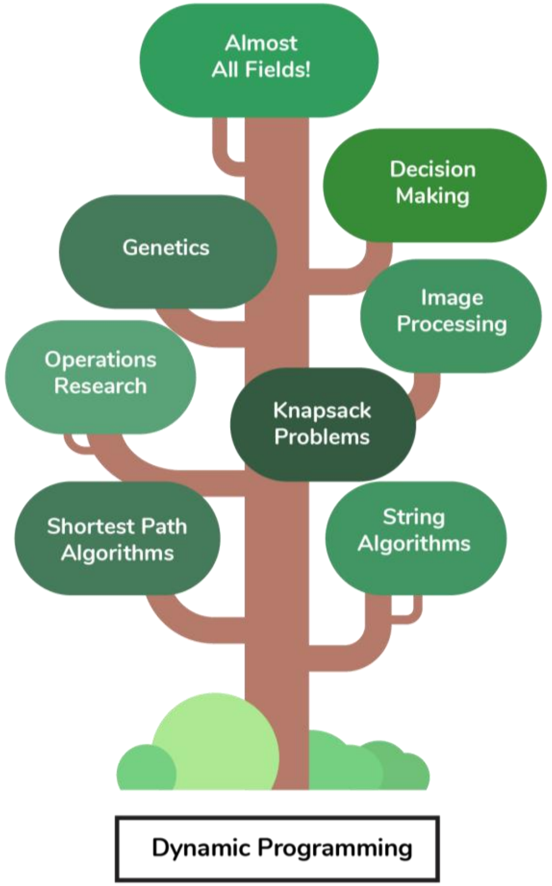
Time Complexity : O(n)



After this recursive call of f(2) is completed, the resultant value is stored in mem[2] and can be directly used when another recursive call of f(2) is encountered, without needing to recurse downwards.

Dynamic Programming: Examples & Applications

- Computing a binomial coefficient
- Warshall’s algorithm for transitive closure
- Floyd’s algorithm for all-pairs shortest paths
- Some instances of difficult discrete optimization problems:
  - Knapsack
  - Traveling salesman



Courtesy: InterviewBit

Computing a Binomial Coefficient

- The term “Binomial Coefficients” (denoted as $C(n,k)$ or ${}^{n}C_{k}$) comes from the participation of these numbers in the Binomial expansion formula:

$$(a + b)^n = C(n,0) * a^n b^0 + C(n,1) * a^{n-1} b^1 + \ldots\ldots + C(n,k) * a^{n-k} b^k + \ldots\ldots + C(n,n) * a^0 b^n$$

- Of the numerous properties of the Binomial Coefficient, we concentrate on the following recursive definition:

$\mathbf{C(n,k) = C(n-1,k-1) + C(n-1,k)}$ for $n>k>0$

and

$\mathbf{C(n,n) = C(n,0) = 1}$

Computing a Binomial Coefficient

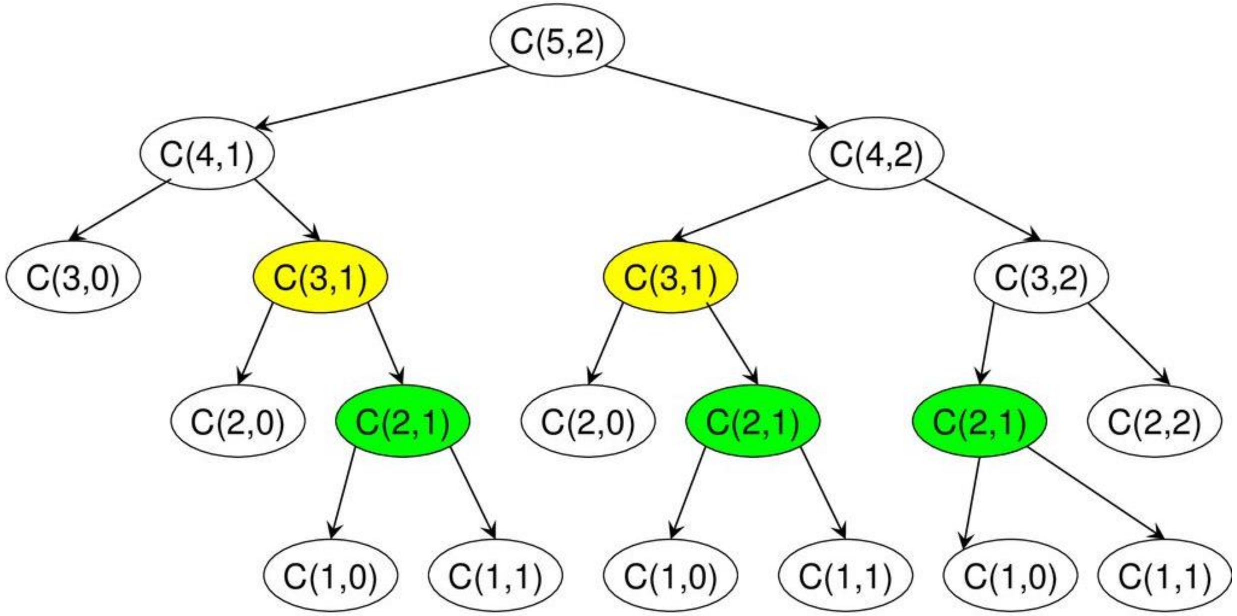
Example: Computing the Binomial Coefficient C(n,k)

Recursive Algorithm:

```
Function binomialCoeff(int n, int k)
{
    if k > n
        return 0;

    if k == 0 || k == n
        return 1;

    return binomialCoeff(n - 1, k - 1)  +  binomialCoeff(n - 1, k)
}
```

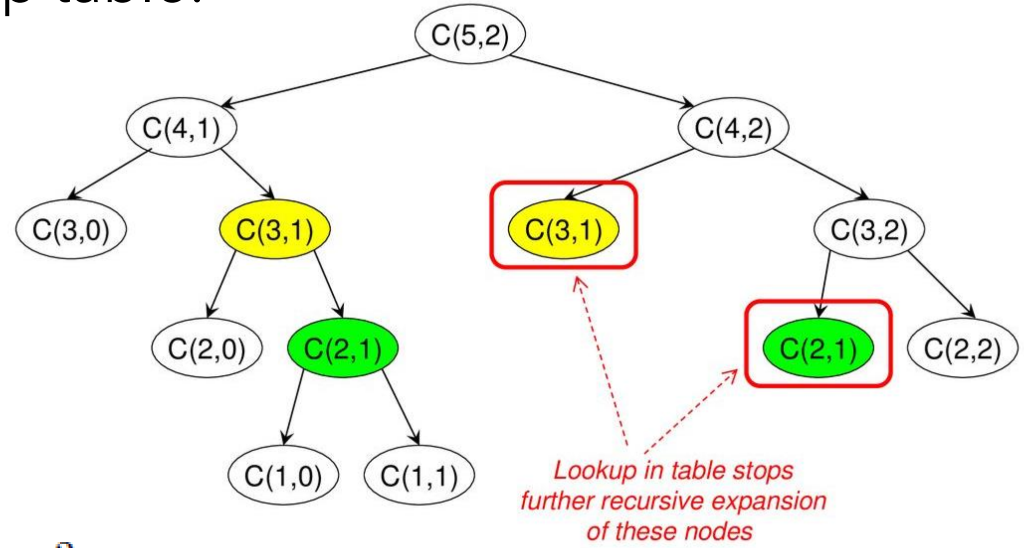


Time Complexity : $O(2^n)$

Computing a Binomial Coefficient using DP

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

	0	1	2	3	. . . .	k-1	k
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
k	1	4	6	4	1		1
.	.	.	.	.	.....	.	.
n-1	1					C(n-1,k-1)	C(n-1,k)
n	1						C(n,k)



	0	1	2	3
0	1			
1	1	1		
2	1	2	1	
3	1	3	3	1
4	1	4	6	4
5	1	5	10	10
6	1	6	15	20

C(6,3)

Computing a Binomial Coefficient using DP

Iterative Algorithm (using Dynamic Programming):

```
Function binomialCoeff(int n, int k)
{
   for i=0 to n
         for j=0 to min(i,k)

            if j = 0 || j = i
                    C[i,j] = 1

            else C[i,j] = C[i-1,j-1] + C[i-1,j]

   return C[n,k]
}
```

If A(n,k) represents the total number of additions made by algorithm in computing C(n,k) , then:

$$A(n,k) = \underbrace{\sum_{i=1}^{k}\sum_{j=1}^{i-1} 1}_{\text{Triangle}} + \underbrace{\sum_{i=k+1}^{n}\sum_{j=1}^{k} 1}_{\text{Rectangle}} = \sum_{i=1}^{k}(i-1) + \sum_{i=k+1}^{n} k$$

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

	0	1	2	3	. . . .	k-1	k
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
k	1	4	6	4	1		1
.	.	.	.	.	.....	.	.
n-1	1					C(n-1,k-1)	C(n-1,k)
n	1						C(n,k)

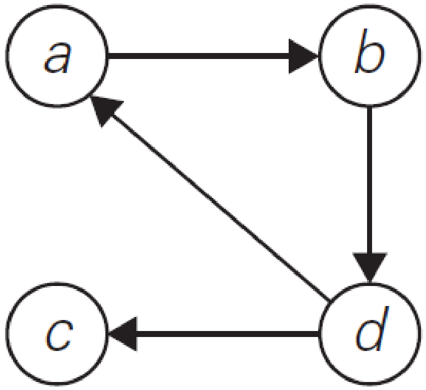
Time Complexity : O(n*k)

Warshall's Algorithm for Transitive Closure

Definition:

The **transitive closure** of a directed graph with n vertices can be defined as :

The $n \times n$ boolean matrix $\mathbf{T = \{t_{ij}\}}$, in which the element in the i^{th} row and the j^{th} column is 1 if there exists a nontrivial path (i.e., directed path of a positive length) from the i^{th} vertex to the j^{th} vertex; otherwise, t_{ij} is 0.



DAG

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

Adjacency Matrix

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

Transitive Closure

Warshall's Algorithm for Transitive Closure

- Introduced by Stephen Warshall in 1962, this algorithm constructs the transitive closure through a series of **n × n boolean matrices**:

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

- Specifically, the element $\mathbf{r_{ij}^{(k)}}$ in the i^{th} row and j^{th} column of matrix $\mathbf{R^{(k)}}$ $(i, j = 1, 2, \ldots, n, k = 0, 1, \ldots, n)$ is equal to 1 if and only if there exists a directed path of a positive length from the i^{th} vertex to the j^{th} vertex with each intermediate vertex, if any, numbered not higher than k.

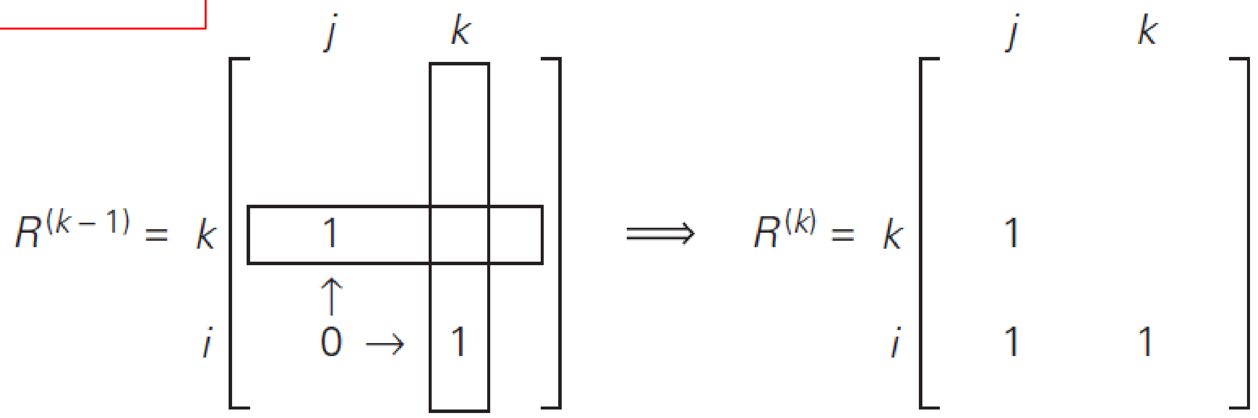
- Thus $\mathbf{R^{(0)}}$ will represent the adjacency matrix of the DAG.

Warshall's Algorithm for Transitive Closure

- According to the algorithm, the elements of matrix $\mathbf{R^{(k)}}$ (denoted as $\mathbf{r_{ij}^{(k)}}$) are generated from the elements of matrix $\mathbf{R^{(k-1)}}$ using the following formula:

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

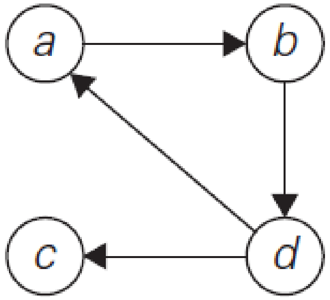
- The formula implies the following:
  - If an element $\mathbf{r_{ij}}$ is 1 in $\mathbf{R^{(k-1)}}$, it remains 1 in $\mathbf{R^{(k)}}$.
  - If an element $\mathbf{r_{ij}}$ is 0 in $\mathbf{R^{(k-1)}}$, it has to be changed to 1 in $\mathbf{R^{(k)}}$ if and only if the element in its row i and column k and the element in its column j and row k are both 1's in $\mathbf{R^{(k-1)}}$



Warshall's Algorithm for Transitive Closure

Warshall's Algorithm (working):

- Compute the Transitive Closure of the input directed graph through the Construction of n x n boolean matrices $R^{(k)}$'s obtained from $R^{(k-1)}$'s



Input graph

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

Find path via 'a'

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

Find path via 'b'

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

Find path via 'c'

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

Find path via 'd'

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

Transitive closure

Warshall's Algorithm for Transitive Closure

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 $R^{(n)}$

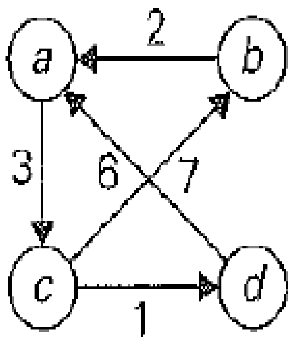
Time Complexity : O(n^3)

Floyd's All-Pairs Shortest Path Algorithm

Problem: To find the lengths of shortest path from any given vertex to all other vertices.

Input: A weighted connected graph (directed or undirected).

Initially W_{ij} = Infinity, when there is no direct path from i to j.



Input graph

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

Weighted adjacency matrix

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

Final Distance (Shortest path) matrix

Applications of the algorithm:

- Communication, Transport Networks.
- Operations Research
- Motion Planning in Computer games.

Floyd's All-Pairs Shortest Path Algorithm

- This algorithm follows the same principle as the Warshall's Algorithm for Transitive Closure.

- Floyd's algorithm computes the distance matrix of a weighted graph with n vertices through a series of $n \times n$ matrices:

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

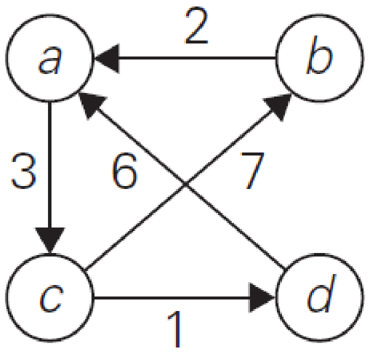
- $\mathbf{D^{(0)}}$ is the weighted adjacency matrix for the input graph. If a direct edge does not exist between any two vertices v_i and v_j, then its corresponding entry $\mathbf{d_{ij}^{(0)}}$ is set as equal to INFINITY.

- According to the algorithm, the elements of matrix $\mathbf{D^{(k)}}$ (denoted as $\mathbf{d_{ij}^{(k)}}$) are generated from the elements of matrix $\mathbf{D^{(k-1)}}$ using the following formula:

$$\boxed{d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} \quad \text{for } k \geq 1, \quad d_{ij}^{(0)} = w_{ij}}$$

Floyd’s All-Pairs Shortest Path Algorithm

Floyd’s Algorithm (working):



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

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

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

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

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

Floyd's All-Pairs Shortest Path Algorithm

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
$D \leftarrow W$ //is not necessary if W can be overwritten
for $k \leftarrow 1$ **to** n **do**
    for $i \leftarrow 1$ **to** n **do**
        for $j \leftarrow 1$ **to** n **do**
            $D[i, j] \leftarrow \min\{D[i, j], D[i, k] + D[k, j]\}$
return D

Time Complexity : O(n^3)

Knapsack Problem

- Re-visiting the problem:

Given **n items** of :

Integer weights: $w_1 \quad w_2 \quad \ldots \quad w_n$

Values: $v_1 \quad v_2 \quad \ldots \quad v_n$

and

A knapsack of integer capacity W

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

Knapsack Problem

Recursive Pseudocode (Top-down):

```
int knapSack(int W, int weights[], int values[], int n)
{
        if (n == 0 || W == 0)
                return 0;

        if (weights[n - 1] > W)
                return knapSack(W, weights, values, n - 1)

        else
        {
                include  = values[n - 1] + knapSack( W - weights[n - 1],  weights,  values,  n - 1);

                exclude = knapSack(W,  weights,  values,  n – 1)

                return max(include, exclude);
        }
}
```

Time Complexity : $O(2^n)$

Knapsack Problem by DP

Tabulation Approach (Bottom-up):

- Let us consider an instance defined by the first i items, $1 \le i \le n$, with weights $w_1, \ldots, w_i$, values $v_1, \ldots, v_i$, and knapsack capacity j, $1 \le j \le W$.

- Let $F(i, j)$ be the value of an optimal solution to this instance.

- Construct the table F(i,j) as follows:

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

		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 by DP

- Example:

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	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**

Maximum possible profit under the given constraints

Time Complexity : O(W*n)

Knapsack Problem by DP: Memory Functions

- The direct top-down approach to finding a solution to such a recurrence leads to an algorithm that solves common subproblems more than once.

- The tabulation-based approach discussed previously, works bottom-up: it fills a table with solutions to all smaller subproblems. But not all the smaller solutions are required to get the solution for the problem given.

- **The goal is to get a method that solves only subproblems that are necessary and does so only once. This method is based on using memory functions.**

- **Solves the problem in top-down manner but maintains the table that works on bottom-up dynamic programming.**

Knapsack Problem by DP: Memory Functions

Memory Function-Based Approach (Top-down):

In the initial function call the value of:
i = W (Capacity), and j = n (no. of items)

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 $j < Weights[i]$
    $value \leftarrow MFKnapsack(i-1, j)$
  else
    $value \leftarrow \max(MFKnapsack(i-1, j),$
        $Values[i] + MFKnapsack(i-1, j - Weights[i]))$
  $F[i, j] \leftarrow value$
return $F[i, j]$

Input data

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

capacity $W = 5$.

The constructed table

	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	—	12	22	—	22
$w_3 = 3, v_3 = 20$	3	0	—	—	22	—	32
$w_4 = 2, v_4 = 15$	4	0	—	—	—	—	**37**

Only necessary values are computed here. Whereas in the tabulation approach all the values in the table are computed. So, this approach is computationally better than the tabulation method

Time Complexity : O(W*n)

Thank you!

Any queries?