ANALYSIS AND DESIGN OF ALGORITHMS

UNIT-IV

CHAPTER 8:

DYNAMIC PROGRAMMING

OUTLINE

- **Dynamic Programming**
  - Computing a Binomial Coefficient
  - Warshall's Algorithm for computing the transitive closure of a digraph
  - Floyd's Algorithm for the All-Pairs Shortest-Paths Problem
  - The Knapsack Problem

Introduction

- Dynamic programming is an **algorithm design technique** which was invented by a prominent U. S. mathematician, Richard Bellman, in the 1950s.
- Dynamic programming is a technique for solving problems with overlapping sub-problems.
- Typically, these sub-problems arise from a recurrence relating a solution to a given problem with solutions to its smaller sub-problems of the same type.
- Rather than solving overlapping sub-problems again and again, dynamic programming suggests solving each of the smaller sub-problems only once and recording the results in a table from which we can then obtain a solution to the original problem.

Introduction

- The Fibonacci numbers are the elements of the sequence

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots,$$

Which can be defined by the simple recurrence

$$F(n) = F(n-1) + F(n-2) \text{ for } n \geq 2$$

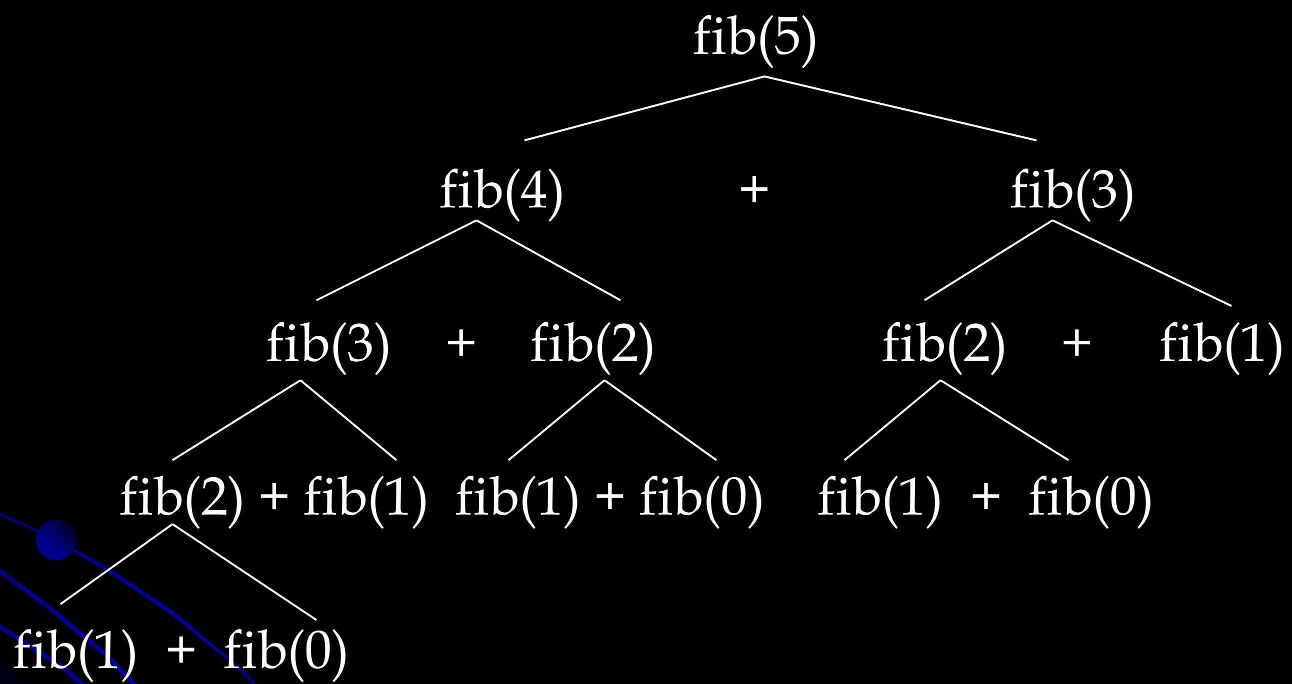
and two initial conditions, $F(0) = 0$, $F(1) = 1$.

```
Algorithm fib(n)
 if n = 0 or n = 1 return n
 return fib(n − 1) + fib(n − 2)
```

- If we try to use recurrence directly to compute the n^{th} Fibonacci number *F(n)* , we would have to recompute the same values of this function many times.

Introduction

- Notice that if we call, say, fib(5), we produce a call tree that calls the function on the same value many different times:



- In fact, we can avoid using an extra array to accomplish this task by recording the values of just the last two elements of the Fibonacci sequence.

Introduction

- Dynamic programming usually takes one of two approaches:

- **Bottom-up approach:** All smaller subproblems of a given problem are solved.

- **Top-down approach:** Avoids solving unnecessary subproblems. This is recursion and Memory Function combined together.

COMPUTING A BINOMIAL COEFFICIENT

- Binomial coefficient, denoted as $C(n, k)$ or $\binom{n}{k}$,

is the number of combinations (subsets) of k elements from an n-element set ($0 \le k \le n$).

- The name "binomial coefficients" comes from the participation of these numbers in the binomial formula:

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

- Two important properties of binomial coefficients:
  - **$C(n, k) = C(n-1, k-1) + C(n-1, k)$ for $n > k > 0$ -------- (1)**
  - **$C(n, 0) = C(n, n) = 1$ --------** (2)

COMPUTING A BINOMIAL COEFFICIENT

- The nature of recurrence (1), which expresses the problem of computing $C(n, k)$ in terms of the smaller and overlapping problems of computing $C(n-1, k-1)$ and $C(n-1, k)$, lends itself to solving by the dynamic programming technique.

- To do this, we record the values of the binomial coefficients in a table of ***n*+1 rows** and ***k*+1 columns**, numbered from 0 to n and from 0 to k, respectively.

- To compute $C(n, k)$, we fill the table row by row, starting with row0 and ending with row n.

- Each row i $(0 \le i \le n)$ is filled left to right, starting with 1 because $C(n,0) = 1$.

COMPUTING A BINOMIAL COEFFICIENT

- Row 0 through k also end with 1 on the table's main diagonal:

  $C(i, i) = 1$ for $0 \leq i \leq k$.

- The other entries are computed using the formula

  $C(i, j) = C(i-1, j-1) + C(i-1, j)$

  i.e., by adding the contents of the cells in the preceding row and the previous column and in the preceding row and the same column.

	0	1	2	.	.	.	$k-1$	k
0	1							
1	1	1						
2	1	2	1					
.		.	.	.				
k	1							1
.		.	.	.				
$n-1$	1						$C(n-1, k-1)$	$C(n-1, k)$
n	1							$C(n, k)$

Figure: Table for computing binomial coefficient $C(n, k)$ by dynamic programming algorithm.

COMPUTING A BINOMIAL COEFFICIENT

Example: Compute C(6, 3) using dynamic programming.

j →

i ↓	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(2, 1) = C(1,0) + C(1,1) = 1+1 = 2
C(3, 1) = C(2,0) + C(2,1) = 1+2 = 3
C(3, 2) = C(2,1) + C(2,2) = 2+1 = 3
C(4, 1) = C(3,0) + C(3,1) = 1+3 = 4
C(4, 2) = C(3,1) + C(3,2) = 3+3 = 6
C(4, 3) = C(3,2) + C(3,3) = 3+1 = 4

C(5, 1) = C(4,0) + C(4,1) = 1+4 = 5
C(5, 2) = C(4,1) + C(4,2) = 4+6 = 10
C(5, 3) = C(4,2) + C(4,3) = 6+4 = 10
C(6, 1) = C(5,0) + C(5,1) = 1+5 = 6
C(6, 2) = C(5,1) + C(5,2) = 5+10 = 15
C(6, 3) = C(5,2) + C(5,3) = 10+10 = 20

COMPUTING A BINOMIAL COEFFICIENT

```
ALGORITHM    Binomial(n, k)
  //Computes C(n, k) by the dynamic programming algorithm
  //Input: A pair of nonnegative integer n ≥ k ≥ 0
  //Output: The value of C(n, k)
  for i ← 0 to n do
      for j ← 0 to min(i, k) do
          if j = 0 or j = i
              C[i, j] ← 1
          else C[i, j] ← C[i - 1, j - 1] + C[i - 1, j]
  return C[n, k]
```

Time Efficiency of Binomial Coefficient Algorithm

- The basic operation for this algorithm is **addition.**
- Let $A(n, k)$ be the total number of additions made by the algorithm in computing $C(n, k)$.
- To compute each entry by the formula, $\mathbf{C(i, j) = C(i-1, j-1) + C(i-1, j)}$ requires just one addition.
- Because the **first $k + 1$ rows** of the table **form a triangle** while the **remaining $n - k$ rows form a rectangle**, we have to split the sum expressing $A(n, k)$ into two parts:

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

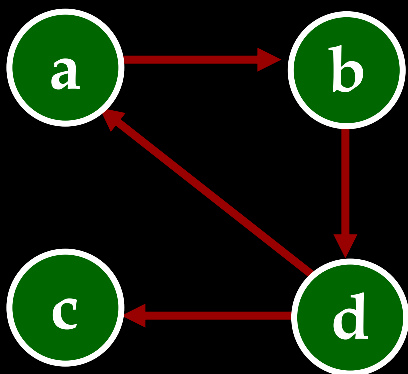
$$= \frac{(k-1)k}{2} + k(n-k) \in \Theta(nk).$$

WARSHALL'S ALGORITHM

- **Warshall's algorithm** is a well-known algorithm for computing the transitive closure (path matrix) of a directed graph.

- **Definition of a transitive closure:** The ***transitive closure*** of a directed graph with n vertices can be defined as the n-by-n boolean matrix T = $\{t_{ij}\}$, in which the element in the i^{th} row ($1 \le i \le n$) and the j^{th} column ($1 \le j \le n$) is 1 if there exists a nontrivial directed path (i.e., a directed path of a positive length) from the i^{th} vertex to the j^{th} vertex; otherwise, t_{ij} is 0.

WARSHALL'S ALGORITHM



(a) Digraph.

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

(b) Its adjacency matrix.

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

(c) Its transitive closure.

WARSHALL'S ALGORITHM

- We can generate the transitive closure of a digraph with the help of depth-first search or breadth-first search.

- Performing traversal (BFS or DFS) starting at the i^{th} vertex gives the information about the **vertices reachable from the i^{th} vertex** and hence the columns that contain ones in the i^{th} row of the transitive closure.

- Thus, doing such a traversal for every vertex as a starting point yields the transitive closure in its entirety.

- Since this method traverses the same digraph several times, we should have a better algorithm.

- It is called Warshall's algorithm after S. Warshall.

WARSHALL'S ALGORITHM

- Warshall's algorithm constructs the transitive closure of a given digraph with n vertices through a series of n-by-n boolean matrices:

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

- Each of these matrices provide certain information about directed paths in the digraph. Specifically, the element $r_{ij}^{(k)}$ in the i^{th} row and j^{th} column of matrix $\boldsymbol{R}^{(k)}$ (k=0, 1, . . . ,n) is equal to 1 if and only if there exists a directed path from the i^{th} vertex to the j^{th} vertex with each intermediate vertex if any, numbered not higher than k.

WARSHALL'S ALGORITHM

- The series starts with $R^{(0)}$, which does not allow any intermediate vertices in its path; hence, $R^{(0)}$ is nothing else but the adjacency matrix of the graph.

- $R^{(1)}$ contains the information about paths that can use the first vertex as intermediate; so, it may contain more ones than $R^{(0)}$.

- In general, each subsequent matrix in series (1) has **one more vertex to use as intermediate for its path than its predecessor**.

- The last matrix in the series, $R^{(n)}$, reflects paths that can use all n vertices of the digraph as intermediate and hence is nothing else but the **digraph's transitive closure**.

WARSHALL'S ALGORITHM

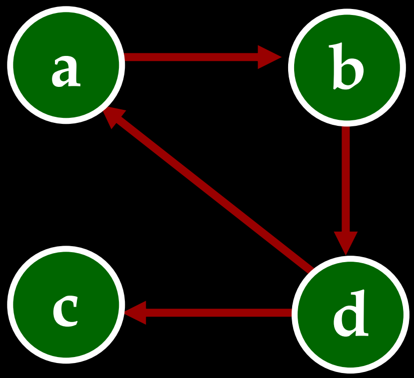
- We have the following formula for generating the elements of matrix $R^{(k)}$ from the elements of matrix $R^{(k\text{-}1)}$:

$$r_{ij}{}^{(k)} = r_{ij}{}^{(k-1)} \text{ or } (r_{ik}{}^{(k-1)} \text{ and } r_{kj}{}^{(k-1)}). \qquad \text{------------------ (3)}$$

- This formula implies the following rule for generating elements of matrix $R^{(k)}$ from elements of matrix $R^{(k\text{-}1)}$:
  - If an element r_{ij} is 1 in $R^{(k-1)}$, it remains 1 in $R^{(k)}$.
  - If an element r_{ij} is 0 in $R^{(k-1)}$, it has to be changed to 1 in $R^{(k)}$ 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 $R^{(k-1)}$.

$$R^{(k-1)} = \begin{array}{c} \\ k \\ i \\ \\ \end{array} \begin{array}{c} \begin{array}{cc} j & k \end{array} \\ \left(\begin{array}{cc} & \\ 1 & \\ 0 & 1 \\ & \end{array} \right) \end{array} \qquad R^{(k)} = \begin{array}{c} \\ k \\ i \\ \\ \end{array} \begin{array}{c} \begin{array}{cc} j & k \end{array} \\ \left(\begin{array}{cc} & \\ 1 & \\ 1 & 1 \\ & \end{array} \right) \end{array}$$

WARSHALL'S ALGORITHM



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

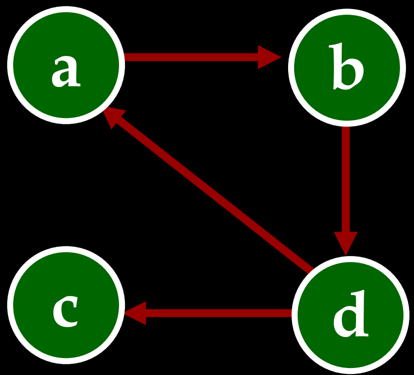
Ones reflect the existence of paths with no intermediate vertices ($R^{(0)}$ is just the adjacency matrix); boxed row and column are used for getting $R^{(1)}$.

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

Ones reflect the existence of paths with intermediate vertices numbered not higher than 1. i.e., just vertex a (note a new path from d to b); boxed row and column are used for getting $R^{(2)}$.

Figure: Application of Warshall's algorithm to the digraph shown. New ones are in bold.

WARSHALL'S ALGORITHM



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

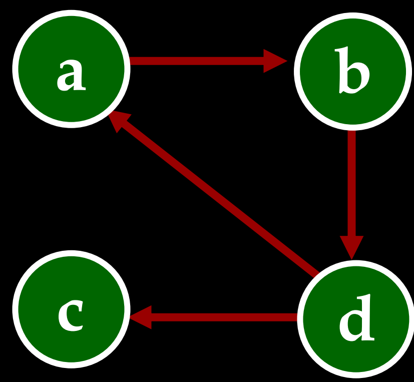
Ones reflect the existence of paths with intermediate vertices numbered not higher than 2. i.e., a and b (note two new paths); boxed row and column are used for getting $R^{(3)}$.

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

Ones reflect the existence of paths with intermediate vertices numbered not higher than 3. i.e., a, b and c (no new paths); boxed row and column are used for getting $R^{(4)}$.

Figure: Application of Warshall's algorithm to the digraph shown. New ones are in bold.

WARSHALL'S ALGORITHM



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

Ones reflect the existence of paths with intermediate vertices numbered not higher than 4. i.e., a, b, c, and d (note five new paths) .

Figure: Application of Warshall's algorithm to the digraph shown. New ones are in bold.

WARSHALL'S ALGORITHM

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** $R^{(k-1)}[k, j])$

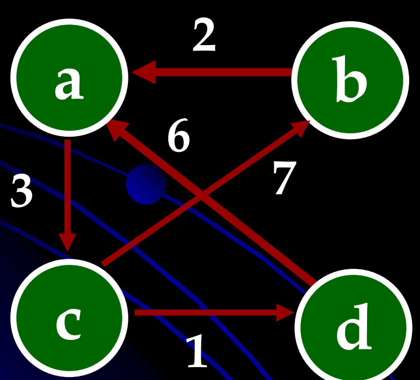
return $R^{(n)}$

Note: The time efficiency of Warshall's algorithm is $\Theta(n^3)$

FLOYD'S ALGORITHM

- Given a weighted connected graph (undirected or directed), the all-pairs shortest-paths problem asks to find the distance (lengths of the shortest paths) from each vertex to all other vertices.

- The Distance matrix D is an n-by-n matrix in which the lengths of shortest paths is recorded; the element d_{ij} in the i^{th} row and the j^{th} column of this matrix indicates the length of the shortest path from the i^{th} vertex to the j^{th} vertex $(1 \le i, j \le n)$.



(a) Digraph.

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

(b) Its weight matrix.

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

(c) Its distance matrix.

FLOYD'S ALGORITHM

- Floyd's algorithm is a well-known algorithm for the all-pairs shortest-paths problem.
- Floyd's algorithm is named after its inventor R. Floyd.
- It is applicable to both undirected and directed weighted graphs provided that they do not contain a cycle of a negative length.
- Floyd's algorithm computes the distance matrix of a weighted graph with n vertices through a series of n-by-n matrices:

$$D^{(0)}, \ldots, D^{(k-1)}, D^{(k)}, \ldots, D^{(n)}. \qquad \text{-----------} \quad (1)$$

- Each of these matrices contains the lengths of shortest paths with certain constraints on the paths considered for the matrix in question. Specifically, the element $d_{ij}^{(k)}$ in the i^{th} row and j^{th} column of matrix $D^{(k)}$ (k=0, 1, . . . , n) is equal to the length of the shortest path among all paths from the i^{th} vertex to the j^{th} vertex with each intermediate vertex, if any, numbered not higher than k.

FLOYD'S ALGORITHM

- The series starts with $D^{(0)}$, which does not allow any intermediate vertices in its path; hence, $D^{(0)}$ is nothing but the weight matrix of the graph.

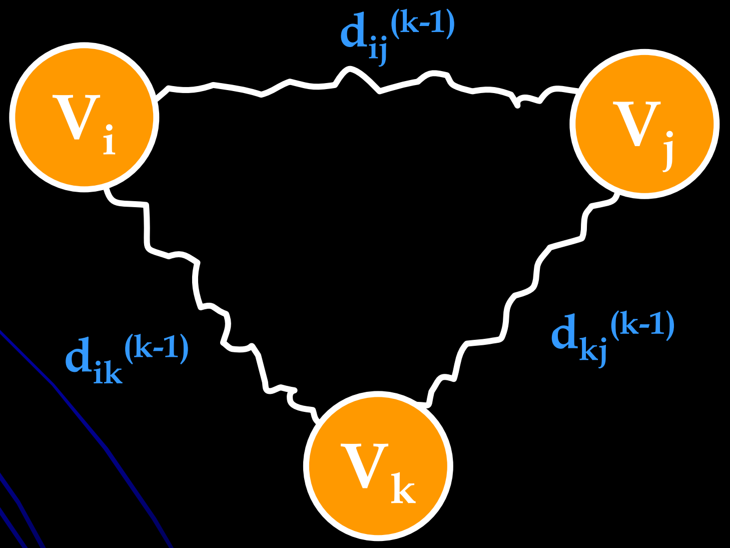
- The last matrix in the series, $D^{(n)}$, contains the lengths of the shortest paths among all paths that can use all n vertices as intermediate and hence is nothing but the distance matrix being sought.

- We can compute all the elements of each matrix $D^{(k)}$ from its immediate predecessor $D^{(k-1)}$ in series (1).

FLOYD'S ALGORITHM

- The lengths of the shortest paths is got by the following recurrence:

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



FLOYD'S 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 ← W // is not necessary if W can be overwritten

for $k \leftarrow 1$ **to** n **do**

  for $i \leftarrow 1$ **to** n **do**

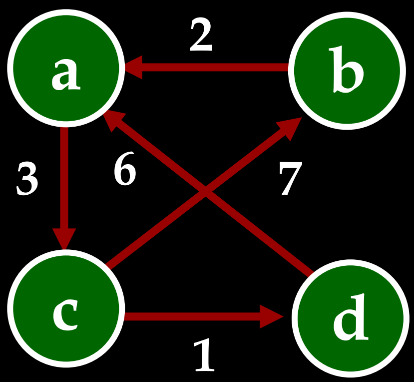
    for $j \leftarrow 1$ **to** n **do**

      $D[i, j] \leftarrow \min \{ D[i, j], D[i, k] + D[k, j] \}$

return D

Note: The time efficiency of Floyd's algorithm is cubic i.e., $\Theta(n^3)$

FLOYD'S ALGORITHM



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

Lengths of the shortest paths with no intermediate vertices ($D^{(0)}$ is simply the weight matrix); boxed row and column are used for getting $D^{(1)}$.

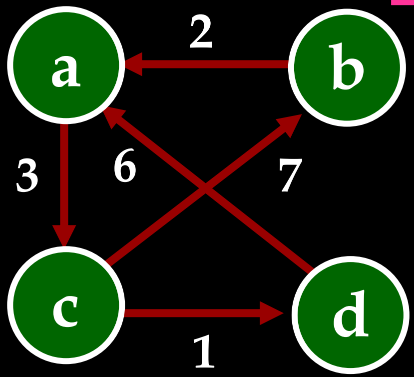
$$D^{(1)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{pmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \mathbf{5} & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \mathbf{9} & 0 \end{pmatrix} \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 1, i.e., just a. (Note two new shortest paths from b to c and from d to c); boxed row and column are used for getting $D^{(2)}$.

Figure: Application of Floyd's algorithm to the digraph shown.

Updated elements are shown in bold.

FLOYD'S ALGORITHM



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

Lengths of the shortest paths with intermediate vertices numbered not higher than 2, i.e., a and b. (Note a new shortest path from c to a); boxed row and column are used for getting $D^{(3)}$.

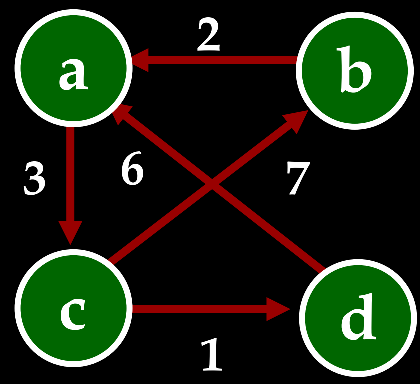
$$D^{(3)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{pmatrix} 0 & \mathbf{10} & 3 & \mathbf{4} \\ 2 & 0 & 5 & \mathbf{6} \\ 9 & 7 & 0 & 1 \\ 6 & \mathbf{16} & 9 & 0 \end{pmatrix} \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 3, i.e., a, b, and c. (Note four new shortest paths from a to b, from a to d, from b to d, and from d to b); boxed row and column are used for getting $D^{(4)}$.

Figure: Application of Floyd's algorithm to the digraph shown.

Updated elements are shown in bold.

FLOYD'S ALGORITHM



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

Lengths of the shortest paths with intermediate vertices numbered not higher than 4, i.e., a, b, c, and d. (Note a new shortest path from c to a).

Figure: Application of Floyd's algorithm to the digraph shown.

Updated elements are shown in bold.

PRINCIPLE OF OPTIMALITY

- It is a general principle that underlines dynamic programming algorithms for optimization problems.
- Richard Bellman called the principle as the ***principle of optimality.***
- It says that an optimal solution to any instance of an optimization problem is ***composed of optimal solutions to its subinstances***.

THE KNAPSACK PROBLEM

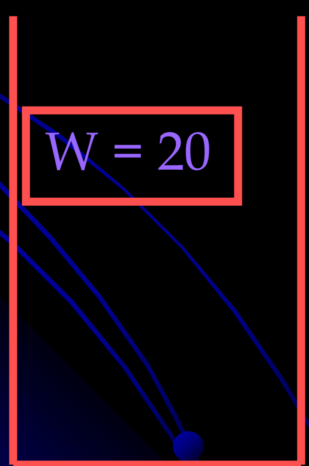
- Given a knapsack with maximum capacity W, and n items.

- Each item i has some weight w_i and value v_i (all w_i, v_i and W are positive integer values).

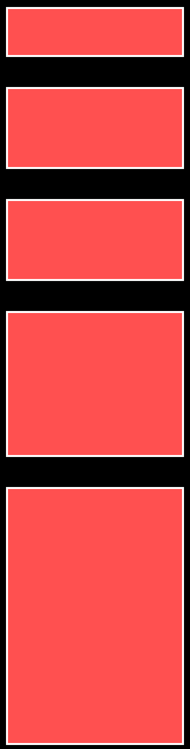
- **Problem:** How to pack the knapsack to achieve maximum total value of packed items? i.e., find the most valuable subset of the items that fit into the knapsack.

Knapsack problem: a picture

This is a knapsack
Max weight: **W** = 20



Items



Weight w_i	value v_i
2	3
3	4
4	5
5	8
9	10

Knapsack problem

- The problem is called a "*0-1 Knapsack problem*", because each item must be entirely accepted or rejected.
- Just another version of this problem is the "*Fractional Knapsack Problem*", where we can take fractions of items.

Knapsack problem: brute-force approach

- Since there are n items, there are 2^n possible combinations of items.
- We go through all combinations and find the one with the most total value and with total weight less or equal to W
- Running time will be $O(2^n)$

KNAPSACK PROBLEM

- To design a **dynamic programming algorithm**, we need to have a recurrence relation that expresses a solution to an instance of the knapsack problem in terms of solutions to its smaller sub-instances.

- The following recurrence is used for the Knapsack problem:

$$V[i, j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ V[i-1, j] & \text{if } j-w_i < 0 \\ \max\{V[i-1], j], v_i + V[i-1, j-w_i]\} & \text{if } j-w_i \geq 0 \end{cases}$$

Our goal is to find V[n, W], the maximal value of a subset of the n given items that fit into the knapsack of capacity W, and an optimal subset itself.

KNAPSACK PROBLEM

- Below figure illustrates the values involved in recurrence equations:

		0	$j\text{-}w_i$	j	W
	0	0	0	0	0
	$i\text{-}1$	0	$V[i\text{-}1, j\text{-}w_i]$	$V[i\text{-}1, j]$	
w_i, v_i	i	0		$V[i, j]$	
	n	0			goal

Figure: Table for solving the knapsack problem by dynamic programming.

- For $i, j > 0$, to compute the entry in the i^{th} row and the j^{th} column, $V[i, j]$, we compute the maximum of the entry in the previous row and the same column and the sum of v_i and the entry in the previous row and w_i columns to the left. The table can be filled either row by row or column by column.

i.e., $V[i, j] = \max\{V[i\text{-}1], j], v_i + V[i\text{-}1, j\text{-}w_i]\}$

Example : Let us consider the instance given by the following data

Build a Dynamic Programming Table for this Knapsack Problem

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

Capacity $W = 5$

Example – Dynamic Programming Table

Item ↓ Capacity → j

	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1=2, v_1=12$	1						
$w_2=1, v_2= 10$	2						
$w_3=3, v_3=20$	3						
$w_4=2, v_4=15$	4						

Entries for Row 0:

V[0, 0]= 0 since i and j values are 0

V[0, 1]=V[0, 2]=V[0, 3]=V[0,4]=V[0, 5]=0 Since i=0

Example – Dynamic Programming Table

Item ↓ i; 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						
w_3=3, v_3=20	3						
w_4=2, v_4=15	4						

Entries for Row 1:

V[1, 0] = 0 since j=0

V[1, 1] = V[0, 1] = 0 (Here, V[i, j]= V[i-1, j] since $j-w_i < 0$)

V[1, 2] = max{V[0, 2], 12 + V[0, 0]} = max(0, 12) = 12

V[1, 3] = max{V[0, 3], 12 +V[0, 1]} = max(0, 12) = 12
V[1, 4] = max{V[0, 4], 12 + V[0, 2]} = max(0, 12) = 12
V[1, 5] = max{V[0, 5], 12 + V[0, 3]} = max(0, 12) = 12

Example – Dynamic Programming Table

Item ↓   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						
$w_4=2, v_4= 15$	4						

Entries for Row 2:

V[2, 0] = 0 since j = 0

V[2, 1] = max{V[1, 1], 10 + V[1, 0]} = max(0, 10) = 10

V[2, 2] = max{V[1, 2], 10 + V[1, 1]} = max(12, 10) = 12

V[2, 3] = max{V[1, 3], 10 +V[1, 2]} = max(12, 22) = 22

V[2, 4] = max{V[1, 4], 10 + V[1, 3]} = max(12, 22) = 22

V[2, 5] = max{V[1, 5], 10 + V[1, 4]} = max(12, 22) = 22

Example – Dynamic Programming Table

Item ↓   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						

Entries for Row 3:

V[3, 0] = 0   since j = 0

V[3, 1] = V[2, 1] = 10 (Here, V[i, j]= V[i-1, j]   since $j-w_i < 0$)

V[3, 2] = V[2, 2] = 12 (Here, V[i, j]= V[i-1, j]   since $j-w_i < 0$)

V[3, 3] = max{V[2, 3], 20 +V[2, 0]} = max(22, 20) = 22
V[3, 4] = max{V[2, 4], 20 + V[2, 1]} = max(22, 30) = 30
V[3, 5] = max{V[2, 5], 20 + V[2, 2]} = max(22, 32) = 32

Example – Dynamic Programming Table

Item ↓ 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

Entries for Row 4:

$V[4, 0] = 0$ since $j = 0$

$V[4, 1] = V[3, 1] = 10$ (Here, $V[i, j]= V[i-1, j]$ since $j-w_i < 0$)

$V[4, 2] = \max\{V[3, 2], 15 + V[3, 0]\} = \max(12, 15) = 15$

$V[4, 3] = \max\{V[3, 3], 15 +V[3, 1]\} = \max(22, 25) = 25$

$V[4, 4] = \max\{V[3, 4], 15 + V[3, 2]\} = \max(30, 27) = 30$

$V[4, 5] = \max\{V[3, 5], 15 + V[3, 3]\} = \max(32, 37) = 37$

Example: To find composition of optimal subset

Item ↓ 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

- Thus, the maximal value is $V[4, 5]$= \$37. We can ***find the composition of an optimal subset by tracing back the computations of this entry in the table.***

- Since $V[4, 5]$ is not equal to $V[3, 5]$, item 4 was included in an optimal solution along with an optimal subset for filling 5 - 2 = 3 remaining units of the knapsack capacity.

Example: To find composition of optimal subset

Item ↓ 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

- The remaining is V[3, 3]
- Here V[3, 3] = V[2, 3] so item 3 is not included
- V[2, 3] $\neq$ V[1, 3] so item 2 is included

Example: To find composition of optimal subset

Item ↓   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

- The remaining is V[1,2]
- $V[1, 2] \neq V[0, 2]$ so item 1 is included

- **The solution is {item 1, item 2, item 4}**
- **Total weight is 5**
- **Total value is 37**

The Knapsack Problem

- The time efficiency and space efficiency of this algorithm are both in **$\theta(nW)$.**

- The time needed to find the composition of an optimal solution is in $O(n + W)$.

End of Chapter 8