DYNAMIC PROGRAMMING

Dynamic programming is a technique for solving problems with overlapping subproblems. Typically, these subproblems arise from a recurrence relating a given problem's solution to solutions of its smaller subproblems. Rather than solving overlapping subproblems again and again, dynamic programming suggests solving each of the smaller subproblems only once and recording the results in a table from which a solution to the original problem can then be obtained

E.g. Fibonacci Numbers

0,1,1,2,3,5,8,13,21,34,..., which can be defined by the simple recurrence

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

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

Difference between Dynamic Programming and Divide and Conquer Method

Sl No	Divide and Conquer	Dynamic Programming
1	An algorithm that recursively breaks down a problem into 2 or more sub-problems of the same or related type until it becomes simple enough to be solved directly	An algorithm that helps to efficiently solve a class of problems that have overlapping subproblems and optimal substructure property
2	Subproblems are independent of each other	Subproblems are interdependent
3	Recursive	Non-Recursive
4	More time-consuming as it solves each subproblem independently	Less time-consuming as it uses the answers of the previous subproblems
5	Less efficient	More efficient
6	Requires some memory for recursive calls	Requires a lot of memory for tabulation
7	Used by merge sort , quicksort and binary search	Used by matrix chain multiplication, optimal binary search tree
8	Top down algorithms	Bottom upalgorithms

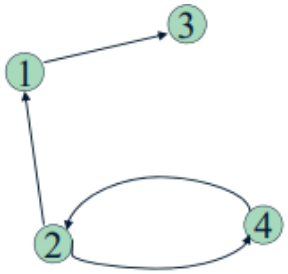
Adjacency matrix A = {aij} of a directed graph is the boolean matrix that has 1 in its ith row and jth column if and only if there is a directed edge from the ith vertex to the jth vertex .

The **transitive closure** of a directed graph with n vertices can be defined as the n × n boolean matrix T = {tij}, in which the element in the ith row and the jth column is 1 if there exists a nontrivial path (i.e., directed path of a positive length) from the ith vertex to the jth vertex; otherwise, tij is 0.

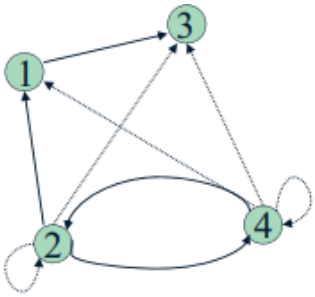
Warshall's Algorithm

Computes the transitive closure of a relation

• Example of transitive closure:



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



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

• Main idea: a path exists between two vertices i, j, iff

• there is an edge from i to j; or

• there is a path from i to j going through vertex 1; or

• there is a path from i to j going through vertex 1 and/or 2; or

• there is a path from i to j going through vertex 1, 2, and/or 3; or

•...

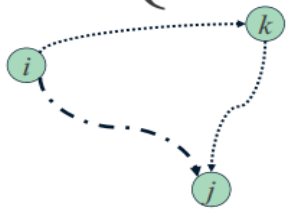
• there is a path from i to j going through any of the other vertices

Warshall's algorithm constructs the transitive closure through a series of n × n boolean matrices:

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

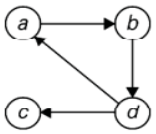
• On the k^{th} iteration, the algorithm determine if a path exists between two vertices *i, j* using just vertices among 1,…,*k* allowed as intermediate

$$R^{(k)}[i,j] = \begin{cases} R^{(k-1)}[i,j] & \text{(path using just 1 ,…,}k\text{-1)} \\ \text{or} & \\ (R^{(k-1)}[i,k] \text{ and } R^{(k-1)}[k,j]) & \text{(path from } i \text{ to } k \text{ and from } k \text{ to } i \text{ using just 1 ,…,}k\text{-1)} \end{cases}$$



k^{th} iteration

Warshall's Algorithm: Transitive Closure



(a)

$A =$

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

(b)

$T =$

	a	b	c	d
a	1	1	1	1
b	1	1	1	1
c	0	0	0	0
d	1	1	1	1

(c)

Warshall's Algorithm (matrix generation)

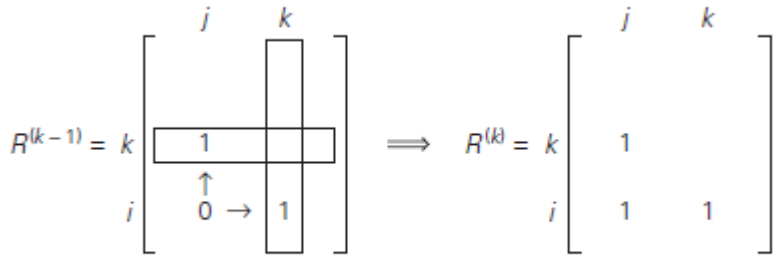
Recurrence relating elements $R^{(k)}$ to elements of $R^{(k-1)}$ is:

$$R^{(k)}[i,j] = R^{(k-1)}[i,j] \text{ or } (R^{(k-1)}[i,k] \text{ and } R^{(k-1)}[k,j])$$

It implies the following rules for generating $R^{(k)}$ from $R^{(k-1)}$:

Rule 1 **If an element in row i and column j is 1 in $R^{(k-1)}$, it remains 1 in $R^{(k)}$**

Rule 2 **If an element in row i and column j 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 $R^{(k-1)}$**



12 Rule for changing zeros in Warshall's algorithm.

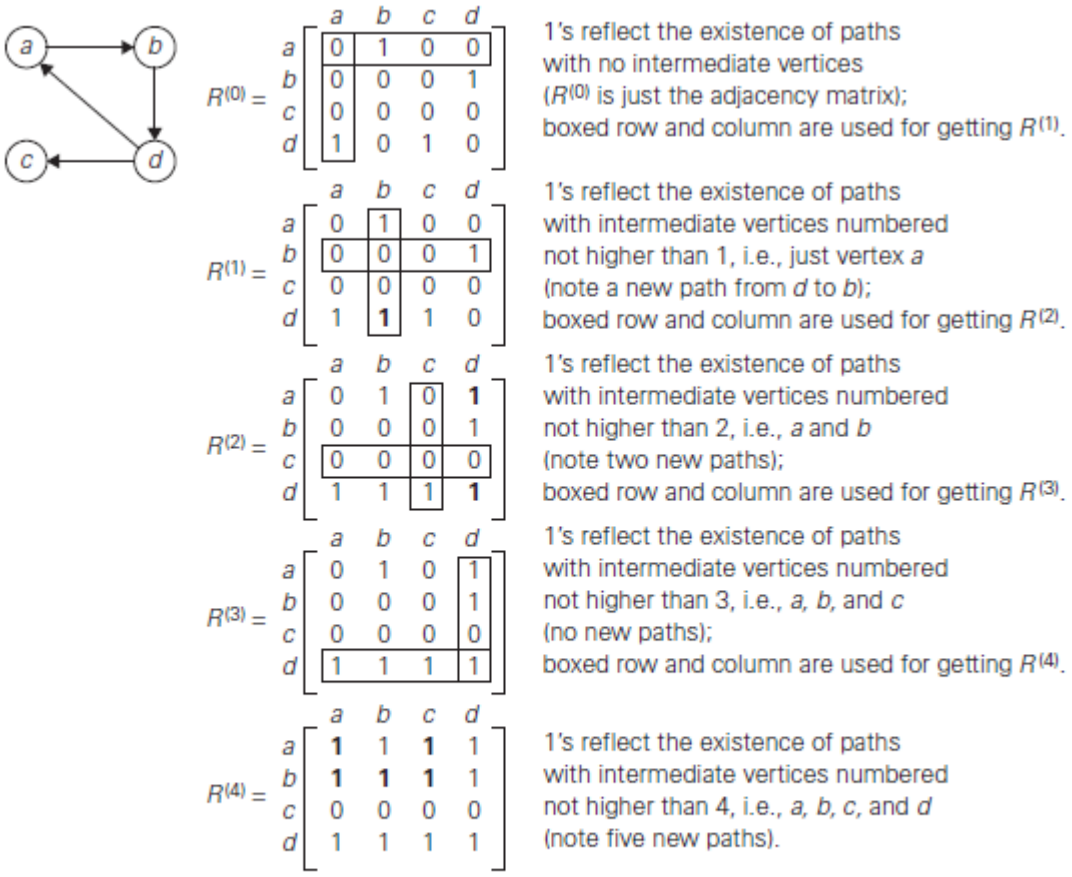


FIGURE 8.13 Application of Warshall's algorithm to the digraph shown. New 1's are in bold.

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) ← A
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            R^(k)[i, j] ← R^(k−1)[i, j] or (R^(k−1)[i, k] and R^(k−1)[k, j])
return R^(n)
```

Time efficiency: $\Theta(n^3)$

Floyd's Algorithm for the All-Pairs Shortest-Paths Problem

Given a weighted connected graph (undirected or directed), the all-pairs shortest paths problem asks to find the distances—i.e., the lengths of the shortest paths—from each vertex to all other vertices. This is one of several variations of the problem involving shortest paths in graphs. Because of its **important applications** to communications, transportation networks, and operations research, it has

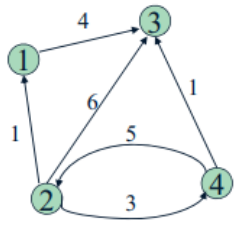
been thoroughly studied over the years. Among recent applications of the all-pairs shortest-path problem is precomputing distances for motion planning in computer games.

Floyd's Algorithm: All pairs shortest paths

Problem: **In a weighted (di)graph, find shortest paths between every pair of vertices**

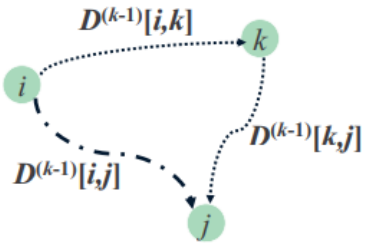
Same idea: construct solution through series of matrices $D^{(0)}$, …, $D^{(n)}$ using increasing subsets of the vertices allowed as intermediate

• **Example:**

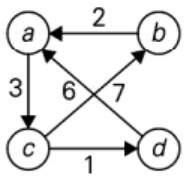


On the *k*-th iteration, the algorithm determines shortest paths between every pair of vertices *i, j* that use only vertices among 1,…,*k* as intermediate

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



Floyd's Algorithm: All pairs shortest paths



(a)

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

(b)

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

(c)

FIGURE 8.5 (a) Digraph. (b) Its weight matrix. (c) Its distance matrix.

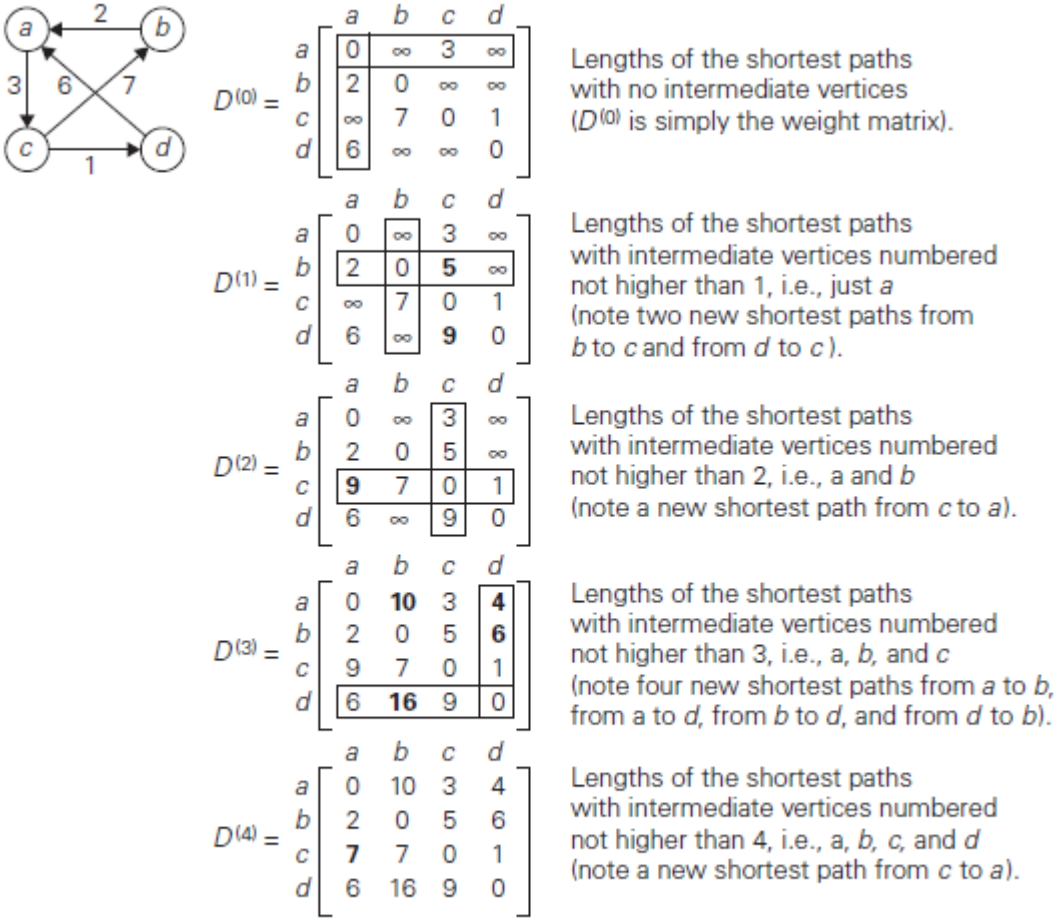


FIGURE 8.16 Application of Floyd's algorithm to the digraph shown. Updated elements are shown in bold.

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
```

Time efficiency: $\Theta(n^3)$

The Knapsack Problem and Memory Functions

Given n items of known weights $w_1, w_2, \ldots, w_n$ and values $v_1, v_2, \ldots, v_n$ and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack.

To design a dynamic programming algorithm, we need to derive a recurrence relation that expresses a solution to an instance of the knapsack problem in terms of solutions to its smaller subinstances.

Let us consider an instance defined by the first i items, $1 \le i \le n$, with

weights w1, . . . , wi, values v1, . . . , vi , and knapsack capacity j, $1 \le j \le W$.

Let F(i, j) be the value of an optimal solution to this instance, i.e., the value of the most valuable subset of the first i items that fit into the knapsack of capacity j.

We can divide all the subsets of the first i items that fit the knapsack of capacity j into two categories: those that do not include the ith item and those that do.

Note the following:

1. Among the subsets that do not include the ith item, the value of an optimal subset is, by definition, F(i − 1, j).

2. Among the subsets that do include the ith item (hence, j – wi ≥ 0), an optimal subset is made up of this item and an optimal subset of the first i − 1 items that fits into the knapsack of capacity j − wi . The value of such an optimal subset is vi + F(i − 1, j − wi).

Thus, the value of an optimal solution among all feasible subsets of the first i items is the maximum of these two values. Of course, if the ith item does not fit into the knapsack, the value of an optimal subset selected from the first i items is the same as the value of an optimal subset selected from the first i − 1 items.

These observations lead to the following recurrence:

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

initial conditions : F(0, j) = 0 for $j \ge 0$ and F(i, 0) = 0 for $i \ge 0$

EXAMPLE 1 Let us consider the instance given by the following d

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**

Thus, the maximal value is F(4, 5) = $37.

Optimal solution {item 1,item 2, item 4}

Time Complexity of 0/1 Knapsack Problem:

The time efficiency and space efficiency of this algorithm are both in **Θ(nW).**

The time needed to find the composition of an optimal solution is in **O(n).**

Memory Functions:

This method solves a given problem in the top-down manner but, in addition, maintains a table of the kind that would have been used by a bottom-up dynamic programming algorithm. Initially, all the table's entries are initialized with a special "null" symbol to indicate that they have not yet been calculated. Thereafter,whenever a new value needs to be calculated, the method checks the corresponding entry in the table first: if this entry is not "null," it is simply retrieved from the table; otherwise, it is computed by the recursive call whose result is then recorded in the table.

The following algorithm implements this idea for the knapsack problem. After initializing the table, the recursive function needs to be called with i = n (the number of items) and j = W (the knapsack capacity).

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 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**

FIGURE 8.6 Example of solving an instance of the knapsack problem by the memory function algorithm.

Only 11 out of 20 nontrivial values (i.e., not those in row 0 or in column 0) have been computed. Just one nontrivial entry, V (1, 2), is retrieved rather than being recomputed.

Note :

- In 0-1 Knapsack(solved using Dynamic Programming), items cannot be broken which means that the items should be taken as a whole or should leave it. This is reason behind calling it as 0-1 Knapsack. Hence, in case of 0-1 Knapsack, the value of xi can be either 0 or 1, where other constraints remain the same.
- Greedy approach gives an optimal solution for Fractional Knapsack.
- 0-1 Knapsack cannot be solved by Greedy approach.