5 Dynamic Programming

- By using dynamic Programming, we can avoid recalculating the same result multiple times.
- Using the divide and conquer technique; we divide bigger problems into simpler subproblems as the algorithm requires.
- Dynamic programming works on both top-down and bottom-up approach/ technique.
- We usually start with dividing the large task into smaller subtasks, storing the results of those subtasks to be used by another subtask at the later stages of the problem solving, until we get the desired solution.
- Using this technique complexity of various algorithms can be reduced.
- The greedy method only generates one decision sequence at a time, and dynamic programming takes care of all the possibilities.
- The problems with overlapping subproblems are generally solved by the dynamic programming technique; problems that do not have a similar type of pattern are difficult to solve using the dynamic programming technique.
- Dynamic programming follows four steps:
  - Identify the pattern of a suitable solution.
  - Recursively calculate the solution.
  - Use the bottom-up technique to calculate the value of the solution.
  - Build a suitable solution from the calculated result.
- Steps 1 -3 form the foundation of a DP technique.
- If we need only the value of a solution, then skip step 4.
- During step 4, we store additional information throughout step 3 so that we can easily calculate a suitable solution.

Understanding Dynamic Programming

Before jumping to problems, let us try to understand how dynamic programming works through examples.

Fibonacci series:

A series of numbers in which the present number is the sum of the last two numbers.

The Fibonacci series is defined as follows:

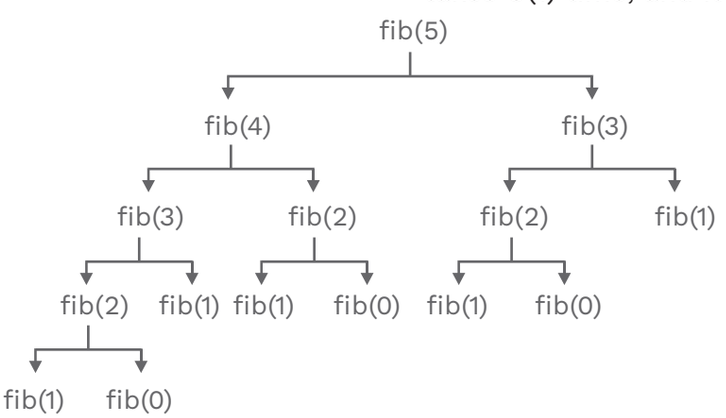
Fib(n) = 0, if n = 0

1, if n = 1

Fib(n-1) + Fib(n-2), if n > 1

Recurrence relation and time complexity:

- The recurrence call T(n) is divided into T(n-1) and T(n-2) till the base conditions T(1) and T(0).
- Therefore, the depth of the tree will be O(n).
- The number of leaves in a full binary tree of depth n gives 2^n. Since each recursion takes O(1) time, and it takes $O(2^n)$.



- In the above case, fib(2) was evaluated three-time (over-lapping of subproblems)
- If the n is large, then other values of fib (subproblems) are appraised which leads to an exponential-time algorithm.
- It is better, instead of solving the same problem, again and again, to store the result once and reuse it; it is a way to reduce the complexity.
- Memoization works like this: Begin with a recursive function and add a table that maps the parameters of the function to the results calculated by the function.
- When the function is called with identical parameter values that have already been called once, the stored value is returned from the table.

Improving:

- Now, we see how DP bring down this problem complexity from exponential to polynomial.
- We have two methods for this:
- 1. bottom-up approach: Start with the lower value of input and slowly calculate a higher value.

```
int fib[n];

int fib(int n)
{
    if (n < = 1)
        return n;
    fib[0] ← 0
    fib[1] ← 1
    for ← 2 to n
        fib[i] ← fib[i-1] + fib[i-2];
    return fib[n];
}
```

- 2. Top-down approach: We just save the values of recursive calls and use them in future.
- The implementation

```
int fib[n];

for (i ← 0 to n)
fib[i] ← 0; /* initialisation */

int fib (int n)
{
    if(n == 0) return 0;
    if(n == 1) return 1;
    if(fib[n]! = 0) /*check if already calculated*/
        return fib[n];
    return fib[n] ← fib (n-1) + fib (n-2);
}
```

- In both methods, Fibonacci Series complexity is reduced to O(n).
- Because we use already computed values from the table.
  TC: O(n).
  SC: O(n).

Note:
Both the approaches can be derived for the dynamic programming problems.

Matrix Chain Multiplication

Problem:

Let's say a series of matrices are given, $A_1 \times A_2 \times A_3 \times \ldots \times A_n$, with their value, what is the right way to parenthesize them so that it provides the minimum the number of total multiplication. Assume that we are using standard matrix multiplication and not Strassen's matrix multiplication algorithm.

Input:

Chain of matrices $A_1 \times A_2 \times A_3 \times \ldots \times A_n$, where A_i has dimensions or size $P_{i-1} \times P_i$. The value is specified in an array P.

Goal:

Finding a way of multiplying matrices(in parenthesized form) such that it gives the optimal number of multiplications.

Solution:

- We know the fact of mathematics that matrix multiplication is associative.
- Parenthesizing does not have any effect on the result.

- For example – 4 matrices A, B, C, and D, the number of ways could be:
  $(A(BC))D = ((AB)C)D = (AB)(CD)$
  $= A(B(CD)) = A((BC)(D))$
- Number of ways we can parenthesis the matrix = $\frac{(2n)!}{(n+1)!n!}$,
  Where n = number of matrix - 1
- Multiplying $A_{(p\times q)}$ matrix with $B_{(q\times r)}$ requires p*q*r multiplications, i.e. the number of scalar multiplication.
- Different ways to parenthesize matrix produce a different number of scalar multiplication.
- Choosing the best parenthesizations using brute force metthod gives $O(2^n)$.
- This time, complexity can be improved using Dynamic Programming.
- Let M[i, j] represent the minimum number of scalar multiplications required to multiply $A_i...A_j$.

$$M[i,j] = \begin{cases} 0 & \text{, if } i = j \\ \underset{i\le k<j}{\text{Min}}\{M[i,k] + M[k+1,j] + p_{i-1}p_k p_j\} & \text{if } i < j \end{cases}$$

- By using the above recursive equation, we find point k that helps to minimise the number of Scalar multiplications.
- After calculating all the possible values of k, the value which gives the minimum number of scalar multiplications is selected.
- For reconstructing the optimal parenthesization, we use a table (say, S[i, j]).
- By using the bottom-up technique, we can evaluate the M[i, j] and S[i, j].
  /* p is the size of the matrices, Matrix i has the dimension p[i-1] × p[i].
  M[i,j] is the least cost of multiplying matrices i through j.
  S[i ,j] saves the multiplication point, and we use this for backtracking, and length is the size of p array.*/
  Void MatrixChainOrder (int p[], int length)

```
{
    int n = length – 1, M[n] [n], S[n] [n];
    for i ← 1 to n
    M[i] [i] ← 0; // fills in matrix by diagonals
    for l ← 2 to n // l is chain length
    {
        for i ← 1 to n – l + 1                    {
            int j ← i + l – 1;
            M[i] [j] ← MAX VALUE;
            /* Try all possible division points i...k and k...j */
            for k ← i to j – 1
                {
                    int this.cost ← M[i] [k] + M[k+1] [j] + p[i-1] * p[k] * p[j];
                        if(this.cost <M[i] [j])
                            {
                                M[i] [j] ← this.cost;
                                S[i] [j] ← k;
                            }
                }
        }
    }
}
```

Time complexity = $O(n^3)$
Space complexity = $O(n^2)$.

Top-Down Dynamic Programming Matrix Chain Multiplication

- Top-down method is also called memoization or memoized.
- Both the top-down method and bottom-up method are going to use the same unique problem.
- The basic similarity between top-down dynamic programming and bottom-up dynamic programming is that the number of function, calls in top-down dynamic programming and the number of shells in bottom-up are almost the same.
- In the top-down method, we are not going to compute any function, twice. Whenever we compute a function, for the 1st time, we save it in the table and next time, when we want that function we take it from the table.

```
MEMOIZED_MATRIX_CHAIN(P)
{
    1. n = p . length – 1 /* p is a sequence of all dimensions, and n is equal to the number
                             of matrixces */
    2. Let m[1....n, 1....n] be a new table
                                        /* m[i, j] represents the least number of scalar
                                    multiplication needed to multiply Ai....Aj */
    3. for i = 1 to n
    4.       for j = 1 to n
    5.              m[i, j] = ∞
    6. return LOOKUP_CHAIN (m, p, 1, n)
}
    LOOKUP_CHAIN (m, p, i, j)
{
    7. if m[i, j] < ∞        /* these two line check whether it is visited for
                                                    the 1st time or not */
    8.       return m[i, j]
    9. if i == j
    10.      m[i, j] = 0
    11. else for k = i to j – 1
    12.      q = LOOKUP_CHAIN(m, p, i, k) + LOOKUP_CHAIN (m, p, k+1, j) + p_{i-1} p_k p_j
    13. if q < m[i, j]
    14.      m[i, j] = q
    15. return m[i, j]
}
```

Analysis:

- In lines 3 to 5, initially, it is placing infinity in the entire table.
- It is done so to know whether it is the 1st time program has visited some entry, or it has already computed.
- If the entry is ∞, then the it means that the program has visited that shell for the 1st time.

Time complexity:

- Number of distinct sub-problem is $O(n^2)$, and whenever the program call any distinct sub-problem, the for-loop in the worst-case run for 'n' times.
- Therefore, the time complexity is $O(n^3)$ same as the bottom-up method.
- Actual space complexity is more because of recursion, but the order of space complexity is not going to change.
- The depth of the tree is going to be O(n). When the depth of the recursion tree is O(n), then the stack size required is O(n) because, at any time, the maximum

number of elements that will be present in the stack will be equal to the depth of the recursion tree and $O(n^2)$ for table m[n, n]. Therefore, space complexity = $O(n^2) + O(n) = O(n^2)$

- Space complexity is also same as bottom-up method.

Previous Years' Question

Four matrices M_1, M_2, M_3 and M_4 of dimensions p x q, q x r, r x s and s x t respectively can be multiplied in several ways with different number of total scalar multiplications. For example, when multiplied as $((M_1 \times M_2) \times (M_3 \times M_4))$ the total number of multiplications is pqr + rst + prt.
When multiplied as $(((M_1 \times M_2) \times M_3) \times M_4)$, the total number of scalar multiplications is pqr + prs + pst.
If p = 10, q = 100, r = 20, s = 5 and t = 80, then the minimum number
of scalar multiplications needed is:
(A) 248000 (B) 44000
(C) 19000 (D) 25000
Solution: (C) **[GATE 2011]**

Longest common subsequence:

Given two strings X and Y of length m and n, respectively. Finding the longest common subsequence in both the strings from left to right, which need not be continuous. For example, if X = "ABCBDAB" and Y = "BDCABA", then LCS(X,Y) = {"BCBA", "BCAB"}. As we can see, there are several optimal solutions.

Brute force approach:

Subsequence X[1...m] (with m being the length of subsequence X) and if it is a subsequence of Y[1...n] (with n being the length of subsequence Y) checking this takes O(n) time, and there are 2^m subsequences possible for X. The time complexity thus sums up to $O(n2^m)$, which is not good for large sequence.

Recursive solution:

- Before DP Solution, we form a recursive solution for this, and later examine the LCS problem.
- Given two strings, "ABCBDAB" and "BDCABA", draw the lines from the letters in the first string to the corresponding letter in the second, no two lines should cross each other.

```
A  B  C  B  D  A  B
   |  |  |     |
   BDCAB     A
```

- Given X and Y, characters may or may not match.
- We can see that the first character of both strings are not same.
- Next, go to the second character in one of the strings and check.
- Lets go to the second character in the first string, and the remaining subproblem is followed this way, solving it recursively.

Optimal Substructure of An Lcs

- Let $X = \langle x_1, x_2, ..., x_m \rangle$ and $Y = \langle Y_1, Y_2, ..., Y_n \rangle$ be sequences, and let $Z = \langle Z_1, Z_2, ..., Z_k \rangle$ be any LCS of X and Y.
- If $x_m = y_n$, then $z_k = x_m = y_n$ and z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
- If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
- If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.
- This characterises that the LCS of two series contains within it an LCS of prefixes of the two series.
- LCS has optimal-substructure property as a problem.
- In recursive solutions, there are overlapping subproblems.

Recursive solution:

- The optimal substructure implies that we should examine either 1 or 2 subproblems when discovery LCS of $X = \langle x_1, x_2, ...,x_m \rangle$ and $Y = \langle y_1, y_2, ..., y_n \rangle$.

- If $x_m = y_n$, then we solve 1 subproblem. Evaluating longest common subsequence on X_{m-1} and Y_{n-1}.
- If X_m ! = Y_n, then we must solve two subproblems. finding an LCS of X_{m-1} and Y and finding the LCS X and Y_{n-1}.
- Whichever of these two LCSs is longer is the LCS of X and Y; these cases exhaust all possibilities.
- One of the optimal subproblem solutions seems like inside a longest common subsequence of X and Y.
- As in the matrix chain multiplication problem, a recursive solution to the LCS problems requires to initiate a recurrence for the value of an optimal solution.
- c[i, j] = length of the LCS of the sequences $<x_1, x_2, ...x_i>$ and $<y_1, y_2, ...y_j>$
- If (i = 0 or j = 0) then one of the sequence has length 0, and LCS has length 0.
- The recursive formula comes from the optimal substructure of the LCS problem.

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

When $X_i = Y_j$, we consider the subproblem of evaluating an LCS of X_{i-1} and Y_{j-1} else we evaluate two subproblems, LCS of X_i and $Y_{(j-1)}$, LCS of $X_{(i-1)}$ and Y_j

In the LCS problem; we have only O(mn) different subproblems, we can use DP to evaluate the problem.

Method LCS-LENGTH grab 2 sequences $X = <x_1, x_2, ..., x_m>$ and $Y = <Y_1, Y_2, ..., Y_n>$ as inputs.

- We store c[i, j] values in a table c[0..m, 0..n], and evaluate the entries in Row-Major order. (the procedure fills in the first row of C from left to right, then the second row, and so on).
- b[1...m, 1...n] helps us in constructing an optimal solution.
- b[i, j] points to the table entry corresponding to the optimal subproblem computed and selected when evaluating c[i, j].
- The procedure returns the table b and c, c[m, n] contains the length of an LCS of X and Y.

LCS_LENGTH(X, Y)

1. m ← X · length
2. n ← Y · length
3. Let b[1...m, 1...n] and c[0...m, 0...n] be new tables
4. for i ← 1 to m
5. c[i,0] ← 0
6. for j ← 0 to n
7. c[0,j] ← 0
8. for i ← 1 to m
9. for j ← 1 to n
10. if x_i == y_j
11. c[i,j] ← c[i-1, j-1] + 1
12. b[i,j] ← "↖"
13. elseif c[i-1, j] ≥ c[i, j-1]
14. c[i,j] ← c[i-1, j]
15. b[i,j] ← "↑"
16. else c[i,j] ← c[i, j-1]
17. b[i,j] ← "←"
18. return c and b

- The running time of the procedure is θ(mn), since each table entry takes θ(1) time to compute.
- Space complexity is θ(mn) because of tables.

Example:

Let X = <P, Q, R, Q, S, P, Q> and Y <Q, S, R, P, Q, P> be two sequences.

i		j	0	1	2	3	4	5	6
			Y_1	Q	S	R	P	Q	P
0	X_1		0	0	0	0	0	0	0
1	P		0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	Q		0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	R		0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	Q		0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	S		0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	P		0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	Q		0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

- The c and b are two dimensional matrices that stores the length of subsequence and the appropriate arrows respectively.
- Initialise the X row and Y column to zero and start with c[1,1] onwards.
- The length can be computed row wise or column wise.
- Let us consider row wise here:
  Starting with c[1, 1] P ≠ Q therefore
  c[1, 1] = max {c[0, 1], c[1, 0]} = 0
  b [1, 1] = "↑"
  c[1, 2] = max {c[1, 1], c[0, 2]} = 0
  b[1, 2] = "↑" since P ≠ S.
  c[1, 3] = max {c[0, 3], c[1, 2]} = 0
  b[1, 3] = "↑" since P ≠ R.
  c[1, 4] = c[0, 3] + 1 = 1,
  b[1, 4] = "↖" since P = P.
- Similiarly all the values in each row are calculated until c[7, 6].
- To find the possible subsequences possible, b[i, j] is used to reconstruct the subsequence from the table.
  PRINT-LCS(b, X, i, j)

  1. If i == 0 or j == 0
  2. Return
  3. If b[i, j] == "↖"
  4. PRINT-LCS(b, X, i-1, j-1)
  5. Print X_i
  6. Elseif b[i, j] == "↑"
  7. PRINT-LCS(b, X, i-1, j)
  8. Else PRINT-LCS(b, X, i, j-1)

- This procedure takes time O(m+n), since it decrements at least one of i and j each recursive call.

Previous Years' Question

A sub-sequence of a given sequence is just the given sequence with some elements (possibly none or all) left out. We are given two sequences X[m] and Y[n] of length m and n, respectively with indexes of X and Y starting from 0.
We wish to find the length of longest common sub-sequence (LCS) of X[m] and Y[n] is as l(m, n), where an incomplete recursive definition for the function l(i, j) to compute the length of the LCS of X[m] and Y[n] is given below:
l(i,j) = 0, if either i = 0 or j = 0
= expr1, if i, j>0 and X[i-1] = Y[j-1]
= expr2, if i,
j>0 and X[i-1] * Y[j-1]
Which one of the following options is correct?
(A) expr1 = l(i-1,j)+1
(B) expr1 = l(i,j-1)
(C) expr2 = max (l(i-1,j), l(i,j-1))
(D) expr2 = max(l(i-1,j-1),l(l,j))
Solution: (C) **[GATE 2009]**

Previous Years' Question

Consider the data given in the above question. The value of l(i,j) could be obtained by dynamic programming based on the correct recursive definition of l(i,j) of the form given above, using an array L[M, NJ, where M = m + 1 and N = n + 1, such that L[i,jJ = l(i,j).
Which one of the following statements would be true regarding the dynamic programming solution for the recursive definition of l(i,j)?

(A) All elements L should be initialized to O for the value of l(i, j) to be properly computed.

(B) The values of l(i, j) may be computed in a row major order or column major order of L[M, N]

(C) The values of l(i, j) cannot be computed in either row major order or column major order of L[M, NJ

(D) L[p, qJ needs to be computed before L[r, sJ if either p < r or q < s.

Solution: (B) **[GATE 2009]**

Previous Years' Question

Consider Two Strings A = "Qpqrr" And B = "Pqprqrp". Let X Be The Length Of The Longest Common Subsequence (Not Necessarily Contiguous) Between A And B And Let Y Be The Number Of Such Longest Common Subsequences Between A And B. Then X + 10y = ____

Solution: 34 **[Gate 2014 (Set-2)]**

Multistage Graph:

- A multistage graph G = (V,E) is a directed graph in which the vertices are partitioned into K ≥ 2 disjoint sets of V_i, 1 ≤ i ≤ k
- In addition, if < u, v > is an edge in E, then u is in V_i and v belongs to V_{i+1} for some i, 1 ≤ i < k.
- Let 's' and 't' be the source and destination respectively.
- The sum of the costs of the edges on a path from source (s) to destination (t) is the path's cost.
- The objective of the MULTISTAGE GRAPH problem is to find the minimum path from 's' to 't'.
- Each set V_i defines a stage in the graph. Every path from 's' to 't' starts in stage-1, goes to stage-2 then to stage-3, then to stage-4, and so on, and terminates in stage-k.
- This MULISTAGE GRAPH problem can be solved in 2 ways.
  - Forward Method
  - Backward Method

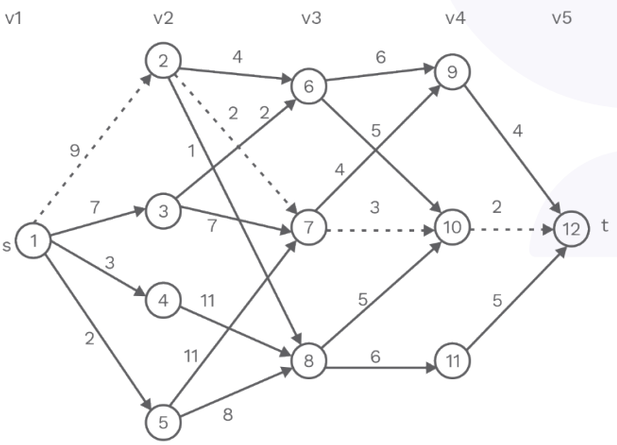
Forward method:

```
Algorithm FGraph (G,k,n,p)
// The input is a k-stage graph G=(V,E) with 'n' vertex.
// Indexed in order of stages and E is a set of edges.
// and c[i,j] is the cost of edge <i,j> is a minimum-cost path.
{
    cost[n]=0.0;
    for j=n-1 to 1 do
    {
        // compute cost[j],
        // let 'r' be the vertex such that <j,r> is an edge of 'G' &
        // c[j,r]+cost[r] is minimum.
        cost[j] = c[j,r] + cost[r];
        d[j] = r;
    }
// find a minimum cost path.
    P[1] = 1;
    P[k]=n;
    For j=2 to k-1 do
    P[j]=d[P[j-1]];
}
```

- Assume that there are 'k' stages in a graph.
- In this FORWARD approach, we find out the cost of each and every node starting from the 'k' th stage to the 1st stage.
- We will find out the path (i.e.) minimum cost path from source to the destination (i.e,) [Stage-1 to Stage-k].

- Maintain a cost matrix cost[n] which stores the distance from any vertex to the destination.
- If a vertex is having more than one path, then we have to choose the minimum distance path and the intermediate vertex, which gives the minimum distance path and will be stored in the distance array 'd'.
- Thus, we can find out the minimum cost path from each and every vertex.
- Finally cost(1) will give the shortest distance from source to destination.
- For finding the path, start from vertex-1 then the distance array D(1) will give the minimum cost neighbour vertex which in turn give the next nearest vertex and proceed in this way till we reach the destination.
- For a 'k' stage graph, there will be 'k' vertex in the path.

Example:



- In the above graph V1...V5 represent the stages. This 5 stage graph can be solved by using forward approach as follows,

STEPS: DESTINATION, D

- Cost (12) = 0 D (12) = 0
- Cost (11) = 5 D (11) = 12
- Cost (10) = 2 D (10) = 12
- Cost (9) = 4 D (9) = 12

For forward approach,

Cost (i,j) = min {c (j,l) + Cost (i+1,l) }
$l \in V_i + 1$
$(j,l) \in E$

cost(8) = min {c (8,10) + cost (10), c(8,11) + Cost(11)}
= min (5+2, 6+5)
= 7
cost(8) = 7 =>D(8) = 10
cost(7) = min(c (7,9) + cost(9), c(7,10) + cost(10))
= min(4+4, 3+2)
= min(8,5)
= 5
cost(7) = 5 => D(7) = 10
cost(6) = min(c (6,9) + cost(9), c(6,10) + cost(10))
= min(6+4, 5+2)
= min(10, 7)
= 7
cost(6) = 7 =>D(6) = 10
cost(5) = min(c (5,7) + cost(7), c(5,8) + cost(8))
= min(11+5, 8+7)
= min(16,15)
= 15
cost(5) = 15 =>D(5) = 8
cost(4) = min(c (4,8) + cost(8))
= min(11+7)
= 18
cost(4) = 18 =>D(4) = 8
cost(3) = min(c (3,6) + cost(6), c(3,7) +cost(7))
= min(2+7, 7+5)
= min(9, 12)
= 9
cost(3) = 9 =>D(3) = 6
cost(2) = min(c (2, 6) + cost(6), c(2,7) + cost(7), c(2,8) + cost(8))
= min(4+7, 2+5, 1+7)
= min(11, 7, 8)
= 7
cost(2) = 7=>D(2) = 7
cost(1)= min(c (1,2)+cost(2), c(1,3) + cost(3), c(1,4) + cost(4), c(1,5) + cost(5))
= min(9+7, 7+9, 3+18, 2+15)
= min(16, 16, 21, 17)
= 16
cost(1)= 16 =>D(1) = 2
Start from vertex -2
D (1) = 2
D (2) = 7

D (7) = 10
D (10) = 12

So, the minimum-cost path is,

Backward method:

- It is similar to forward approach, but differs only in two or three ways.
- Maintain a cost matrix to store the cost of every vertices and a distance matrix to store the minimum distance vertex.
- Find out the cost of each and every vertex starting from vertex 1 up to vertex k.
- To find out the path start from vertex 'k', then the distance array D (k) will give the minimum cost neighbour vertex which in turn gives the next nearest neighbour vertex and proceed till we reach the destination.

Algorithm: backward method:

```
Algorithm BGraph (G,k,n,p)

// The input is a k-stage graph G=(V,E) with 'n' vertex.

// Indexed in order of stages and E is a set of edges.

// and c[i,j] is the cost of edge <i,j> (i,j are the vertex number), p[k] is a minimum cost path.

{
    bcost[1]=0.0;
    for j=2 to n do
    {
            // compute bcost[j]
            // let 'r' be the vertex such that <r,j> is an edge of 'G' &
            // bcost[r]+c[r,j] is minimum.
    bcost[j] = bcost[r] + c[r,j];
            d[j]=r;
    }
// find a minimum cost path.
    P[1]=1;
    P[k]=n;
    For j= k-1 to 2 do
    P[j]=d[P[j+1]];
}
```

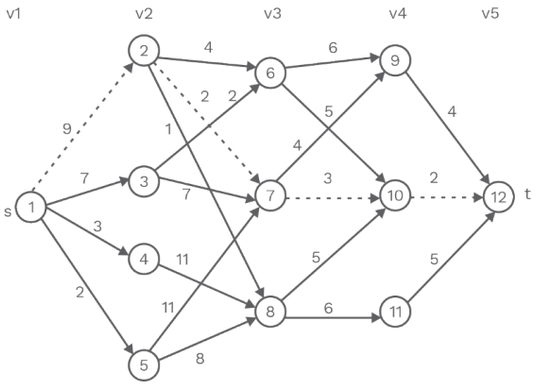


Fig. 5.1

Cost(1) = 0 => D(1)= 0
Cost(2) = 9 => D(2)=1
Cost(3) = 7 => D(3)=1
Cost(4) = 3 => D(4)=1
Cost(5) = 2 => D(5)=1
Cost(6) = min(c (2,6) + cost(2), c(3,6) + cost(3))
= min(13,9)
cost(6) = 9 => D(6)=3
Cost(7) = min(c (3,7) + cost(3), c(5,7) + cost(5), c(2,7) + cost(2))
= min(14,13,11)
cost(7) = 11 =>D(7)=2
Cost(8) = min(c (2,8) + cost(2), c(4,8) + cost(4), c(5,8) + cost(5))
= min(10,14,10)
cost(8) = 10 =>D(8)=2
Cost(9) = min(c (6,9) + cost(6), c(7,9) + cost(7))
= min(15, 15)
cost(9) = 15 =>D(9)=6
Cost(10) = min(c(6,10) + cost(6), c(7,10) + cost(7)), c(8,10) + cost(8))
= min(14,14,15)
cost(10) = 14 =>D(10)=6
Cost(11) = min(c (8,11) + cost(8))
cost(11) = 16 =>D(11)=8
Cost(12) = min(c(9,12) + cost(9), c(10,12) + cost(10), c(11,12) + cost(11))
= min(19,16,21)
cost(12) = 16 =>D(12) = 10

Start from vertex-12:

D(12) = 10
D(10) = 6
D(6) = 3
D(3) = 1

So the minimum cost path is,

$$1 \xrightarrow{7} 3 \xrightarrow{2} 6 \xrightarrow{5} 10 \xrightarrow{2} 12$$

The cost is 16.

Travelling Salesman Problem

- The traveling salesman problem (TSP) is to find the shortest possible route that visits each city exactly once and returns to the starting point given a set of cities and the distance between each pair of cities.
- Take note of the distinction between the Hamiltonian cycle and the TSP. The Hamiltonian cycle problem entails determining whether a tour exists that visits each city exactly once. The problem is to find a minimum weight Hamiltonian cycle. We know that Hamiltonian tours exist (because the graph is complete), and there are many of them.
- Let the number of vertices in the given set be 1, 2, 3, 4,...n. Let's use 1 as the starting and ending points for the output. We find the minimum cost path with 1 as the starting point, i as the ending point, and all vertices appearing exactly once.
- Let's say the cost of this path is cost(i), then the cost of the corresponding cycle is cost(i) + dist(i, 1), where dist(i, 1) is the distance between from i to 1. Finally, we return the value that is the smallest of all [cost(i) + dist(i, 1)]. So far, this appears to be straightforward. The question now is how to obtain cost(i).
- We need a recursive relation in terms of sub-problems to calculate the cost(i) using dynamic programming.
- Let's say C(S, i) is the cost of the minimum cost path visiting each vertex in set S exactly once, starting at 1 and ending at i.
- We begin with all subsetds of size 2 and calculate C(S, i) for all subsets where S is the subset, then calculate C(S, i) for all subsets of size 3, and so on. It's worth noting that 1 must appear in each subset. For a subset of cities $S \subseteq \{1, 2, \ldots, n\}$ that includes 1, and $j \in S$, let $C(S, j)$ be the length of the shortest path visiting each node in S exactly once, starting at 1 and ending at j.

When |S| > 1, we define C(S,1) = ∞ since the path cannot both start and end at 1.

Now, let's express C(S,j) in terms of smaller subproblems. We need to start at 1 and end at j; what should we pick as the second-to-last city? It has to be some $i \in S$, so the overall path length is the distance from 1 to i, namely, C(S – {j}, i), plus the length of the final edge. d_{ij}. We must pick the best such i:

$$C(S, j) = \min_{i \in S: i \neq j} C(S - \{j\}, i) + d_{ij}.$$

The subproblems are ordered by |S|. Here's the code.

C({1},1) = 0

for s = 2 to n:

for all subsets $S \subseteq \{1, 2, \ldots, n\}$ of size s and containing 1:

C(S,1) = ∞

for all $j \in S, j \neq 1$:

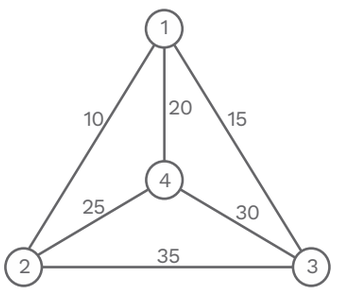
$$C(S, j) = \min\{C(S - \{j\}, i) + d_{ij} : i \in S, i \neq j\}$$

return $\min_j C(\{1, \ldots, n\}, j) + d_{j1}$

There are at most $2^n \cdot n$ subproblems, and each one takes linear time to solve. The total running time is, therefore $O(n^2 2^n)$.

Example:

- Consider the graph



- Matrix representation of the above graph

	1	2	3	4
1	0	10	15	20
2	10	0	35	25
3	15	35	0	30
4	20	25	30	0

- Lets start from node 1
  C(4, 1) = 20 C(3,1) = 15 C(2,1) = 10
  C({3},2) = d(2,3) + C(3,1) = 50
  C({4},2) = d(2,4) + C(4,1) = 45
  C({2},3) = d(3,2) + C(2,1) = 45
  C({4},3) = d(3,4) + C(4,1) = 50
  C({2},4) = d(4,2) + C(2,1) = 35
  C({3},4) = d(4,3) + C(3,1) = 45
  C({3,4,2}) = min(d(2,3) + C({4},3), d(2,4) + C({3},4))
  = min(85,70)
  = 70
  C({2,4},3) = min(d(3,2) + C({4},2), d(3,4) + C({2},4))
  = min(80,65)
  = 65
  C({2,3},4) = min(d(4,2) + C({3},2), d(4,3) + C({2},3))
  = min(75, 75)
  = 75
- Finally
  C({2,3,4},1) = min(d(1,2) + C({3,4},2), d(1,3) + C({2,4},3), d(1,4)+C({2,3},4))
  = min(80,80,95)
  = 80
  ∴ Optimal tour length is 80
  Optimal tour 1-2-4-3-1

0-1 Knapsack problem:

The 0-1 knapsack problem is as follows. A thief robbing a store finds n items. The ith item is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers. The thief wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack, for some integer W. Which items should he take? (We call this the 0-1 knapsack problem because, for each item, the thief must either take it or leave it behind; he cannot take a fractional amount of an item or take an item more than once.)

Dynamic Programming Approach

- Suppose we know that a particular item of weight w is in the solution. Then we must solve the subproblem on n – 1 items with maximum weight W – w.
- Thus, to take a bottom-up approach, we must solve the 0-1 knapsack problem for all items and possible weights smaller than W.
- We'll build an n + 1 by W + 1 table of values where the rows are indexed by item, and the columns are indexed by total weight.
- For row i column j, we decide whether or not it would be advantageous to include item i in the knapsack by comparing the total value of a knapsack including items 1 through i – 1 with max weight j, or the total value of including items 1 through i – 1 with max weight j – i.weight and also item i.
- To solve the problem, we simply examine the [n,W] entry of the table to determine the maximum value we can achieve.
- To read the items we include, start with entry [n,W]. In general, proceed as follows: If entry [i,j] equals entry [i–1, j], don't include item i, and examine entry [i–1, j], next.
- If entry [i,j] doesn't equal entry [i – 1, j], include item i and examine entry [i – 1, j – i].weight next.

Algorithm 0-1 Knapsack(n,W)

1. Initialize a 2D matrix KS of size (n+1) x (W+1)
2. Initialize 1st row and 1st column with zero
3. For itr<- 1 to n
4. For j<-1 to W
5. If(j< W[itr])
6. KS[itr,j]<- K[itr-1][j]
7. Else
8. KS[itr,j]<-max(KS[itr-1,j] , KS[itr-1 , j])
9. End for
10. End for

End

Analysis:

- Since the time to calculate each entry in the table KS[i,j] is constant; the time complexity is Θ(n x W). Where n is the number of items and W is the capacity of the Knapsack.

Recurrence relation:

- In a Knapsack problem, we are given a set of n items where each item i is specified by a size/weight w_i and a value P_i. We are also given the size bound W, the size (capacity) of our Knapsack.

$$KS(i,w) = \begin{cases} \max(P_i + KS(i-1, w-w_i), KS(i-1,w)); \text{if } w_i \le w \\ KS(i-1,w); \text{if } w_i > w \\ 0 \quad ; \text{if } i = 0 \text{ or } w = 0 \end{cases}$$

Where KS (i,w) is the best value that can be achieved, for instance, with only the first i items and capacity w.

Example:

Item	1	2	3
Weight (in kgs)	1	2	3
Values (In rupees)	10	15	40

Capacity of bag = W = 6 kgs

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0	10	15	25	25	25	25
3	0	10	15	40	50	55	65

$$KS(1,1) = \max\begin{cases} P_1 + KS(0,0), \\ KS(0,1) \end{cases} = \max\begin{cases} (10+0), \\ 0 \end{cases} = 10$$

$$KS(1,2) = \max\begin{cases} P_1 + KS(0,1), \\ KS(0,2) \end{cases} = \max\begin{cases} (10+0), \\ 0 \end{cases} = 10$$

$$KS(1,3) = \max\begin{cases} P_1 + KS(0,2), \\ KS(0,3) \end{cases} = \max\begin{cases} (10+0), \\ 0 \end{cases} = 10$$

$$KS(1,4) = \max\begin{cases} P_1 + KS(0,3), \\ KS(0,4) \end{cases} = \max\begin{cases} (10+0), \\ 0 \end{cases} = 10$$

$$KS(1,5) = \max\begin{cases} P_1 + KS(0,4), \\ KS(0,5) \end{cases} = \max\begin{cases} (10+0), \\ 0 \end{cases} = 10$$

$$KS(1,6) = \max\begin{cases} P_1 + KS(0,5), \\ KS(0,6) \end{cases} = \max\begin{cases} (10+0), \\ 0 \end{cases} = 10$$

$$KS(2,1) = KS(1,1) = 10$$

$$KS(2,2) = \max\begin{cases} P_2 + KS(1,0), \\ KS(1,2) \end{cases} = \max\begin{cases} (15+0), \\ 10 \end{cases} = 15$$

$$KS(2,3) = \max\begin{cases} P_2 + KS(1,1), \\ KS(1,3) \end{cases} = \max\begin{cases} (15+10), \\ 10 \end{cases} = 25$$

$$KS(2,4) = \max\begin{cases} P_2 + KS(1,2), \\ KS(1,4) \end{cases} = \max\begin{cases} (15+10), \\ 10 \end{cases} = 25$$

$$KS(2,5) = \max\begin{cases} P_2 + KS(1,3), \\ KS(1,5) \end{cases} = \max\begin{cases} (15+10), \\ 10 \end{cases} = 25$$

$$KS(2,6) = \max\begin{cases} P_2 + KS(1,4), \\ KS(1,6) \end{cases} = \max\begin{cases} (15+10), \\ 10 \end{cases} = 25$$

$$KS(3,1) = KS(2,1) = 10$$

$$KS(3,2) = KS(2,2) = 15$$

$$KS(3,3) = \max\begin{cases} P_3 + KS(2,0), \\ KS(2,3) \end{cases} = \max\begin{cases} (40+0), \\ 25 \end{cases} = 45$$

$$KS(3,4) = \max\begin{cases} P_3 + KS(2,1), \\ KS(2,4) \end{cases} = \max\begin{cases} (40+10), \\ 25 \end{cases} = 50$$

$$KS(3,5) = \max\begin{cases} P_3 + KS(2,2), \\ KS(2,5) \end{cases} = \max\begin{cases} (40+15), \\ 25 \end{cases} = 55$$

$$KS(3,6) = \max\begin{cases} P_3 + KS(2,3), \\ KS(2,6) \end{cases} = \max\begin{cases} (40+25), \\ 25 \end{cases} = 65$$

Subset Sum Problem

Problem:

Given a sequence of n positive numbers A_1, A_2.....A_n, give an algorithm which checks whether there exists a subset of A whose sum of all numbers is T.

- This is a variation of the Knapsack problem. As an example, consider the following array: A = [3, 2, 4, 19, 3, 7, 13, 10, 6, 11] Suppose if we want to check whether there is any subset whose sum is 17. The answer is yes, because the sum of 4 + 13 = 17 and therefore {4, 13} is such a subset.
- Greedy method is not applicable in subset-sum. If we try to be greedy, then we don't know whether to take the smallest number or the highest number. So, there is no way we could go for greedy.

Example:

Let set = {6, 2, 3, 1}. Check whether there is a subset whose sum is 5.

Solution:

Here, there is a subset {3, 2} whose sum is 5 Here greedy method fails; if we try to be greedy on a smaller number, then it will take 1, and so if we take 1, then the remaining sum is going to be 4. So, there is no number or subset which makes to 4.

So, greedy doesn't work in subset-sum.

Brute-fore method:

- If there are 'n' numbers, then any number can be present or absent in subset. So, every a number has 2 options. Hence, the number of subsets is equal to 2^n.
- Brute force method examines every subset, i.e, equal to 2^n. To examine each sub-problem, it will take O(n) time.
- Therefore, time complexity = number of subproblem * time taken by each sub-problem. = $0(2^n) * O(n) = O(n2^n)$

Recursive equation:

- Let us assume that SS(i, S) denote a subset-sum from a_1 to a_i whose sum is equal to some number 'S'.
- First, we check the base condition. If there are no elements i.e., i = 0, and we want to produce some sum S then it is not possible. So, it is false.
- If there are no elements i.e., i = 0, and sum S = 0 is possible. So, it is true because Sum = 0 is always possible.
- This above two are base conditions.
- Using i^{th} element if this sum 'S' has to be possible, then there are two cases:

  Case 1: If we include the i^{th} element in our subset, then a sum of $(S - a_i)$ should be possible with the remaining (i-1) elements.

  Case 2: If we don't include the i^{th} element in our subset, then a sum of (S) should be possible with the remaining (i-1) elements. So, the recursive equation look like as shown below:

$$SS(i,s) = \begin{cases} SS(i-1,S); S < a_i \\ SS(i-1,S-a_i) \ V \ S \ S(i-1,S); S \geq a_i \\ true; S = 0 \\ False; i = 0, S \neq 0 \end{cases}$$

- If the problem is a subset-sum of SS(n,w) (where n positive numbers $A_1A_2...A_n$ and w is a sum). Then the number of unique sub-problem is O(nw).

Solved Examples

1. Given the set S = {6, 3, 2, 1}. Is there any subset possible whose sum is equal to 5.

Solution:

Since here, the number of elements = 4 and the sum is going to be 5 then the number of problem = 4 × 5 = 20.

Sum →

number of element ↓	0	1	2	3	4	5
0	T	F	F	F	F	F
1	T	F	F	F	F	F
2	T	F	F	T	F	F
3	T	F	T	T	F	T
4	T	T	T	T	T	T

- Whenever we want the sum = 0 then it is always possible with whatever element because a null set is going to be sum = 0.
- Index (i, j) indicate, with ith element is sum j possible.
  SS(1,1) = SS(0,1) = False
  [Since the 1st element weight is '6'. So, it can't lead to sum of 1 then we have to go for SS(i-1,S) i.e., SS(0, 1)]
  Similarly,
  SS(2,1) = SS(1,1) = False
  SS(2,2) = SS(1,2) = False
  SS(2,3) = SS(1,0) VSS(1,3) = True V False = True
  SS(2,4) = SS(1,1) V SS(1,4) = False V False = False
  SS(2,5) = SS(1,2) V SS(1,5) = False V False = False
  SS(3,1) = SS(2,1) (Here, S < i) = False
  SS(3,2) = SS(2,0) V SS(2,2) = True V False = True
  SS(3,3) = SS(2,1) V SS(2,3) = False V True = True
  SS(3,4) = SS(2,2) V SS (2,4) = False V False = False
  SS(3,5) = SS(2,3) V SS (2,5) = True V False = True
  SS(4,1) = SS(3, 0) V SS(3,1) = True V False = True
  SS(4,2) = SS(3,1) V SS(3,2) = False V True = True
  SS(4,3) = SS(3,2) V SS (3,3) = True V True = True
  SS(4,4) = SS(3,3) V SS(3,4) = True V False = True
  SS(4,5) = SS(3,4) V SS(3,5) = False V True = True
- Since, the final answer is in shell (4,5). So, the final answer is true. The final answer will always be present in (n, w).
- The subset sum can be computed either in row major order or column major order.

Time complexity:

- Here, the number of the subproblem is (nw), and the time required to calculate each subproblem is O(1). Hence, time complexity = (nw) × O(1) = O(nw).

Note:

- Either to use dynamic programming or not depends on the value of w.
- If 'w' a is big number, then the brute force method gives better time complexity i.e. $O(2^n)$ otherwise dynamic programming.

Conclusion:

Time complexity = $\min \begin{cases} O(2^n) \\ O(nw) \end{cases}$

- If w is n! then 2^n is going to be better than O(nw).

Space complexity:

Space complexity is required for the table. So, O(nw) is the space complexity.

All Pairs Shortest Path Floyd Warshall

Problem:

Given a weighted directed graph G = (V, E), where V = {1, 2, ...,n}. Find the shortest path between all pair of nodes in the graph.

- We can solve an all-pairs shortest-paths problem by running a single-source shortest-paths algorithms |V| times, once for each vertex as the source.
- If all the edges of the graph contain the positive weight, then apply Dijkstra's algorithm.
- If we use the linear-array implementation of the min-priority queue, the running time is $O(V^3 + VE)$, and we know, $E = V^2$ so, $O(V^3)$.
- The binary min-heap implementation of the min-priority queue yields a running time of O(V+ElogV), which is an improvement if the graph is sparse.
- Alternatively, we can implement the min-priority queue with a Fibonacci heap, yielding a running time of $O(V^2 \log V + VE)$.
- Instead, we must run the slower Bellman-ford algorithm once from each vertex.
- The resulting running time is $O(V^2E)$, which for the dense graph is $O(V^4)$
- In the Floyd-Warshall algorithm, the negative-weight edge is allowed.

- The Floyd-Warshall algorithm considers the intermediate vertices of a shortest path, where an intermediate vertex of a simple path $P = \langle V_1, V_2, \ldots, V_l \rangle$ is any vertex of P other than V_1 or V_l, that is, any vertex in the set $\{V_2, V_3, \ldots, V_{l-1}\}$
- The Floyd-Warshall algorithm relies on the following observation.
- Under our assumption that the vertices of G are V = {1,2,...,n}, let us consider a subset {1,2,...,k} of vertices for some k.
- For any pair of vertices $i, j \in V$, consider all paths from i to j whose intermediate vertices are all drawn from {1,2,...k} and Let p be a minimum-weight path from among them. (Path P is simple)
- The Floyd-Warshall algorithm exploits a relationship between path p and shortest paths from i to j with all intermediate vertices in the set {1,2,...,k-1}.
- The relationship depends on whether or not k is an intermediate vertex of path p.
- If k is not an intermediate vertex of path p, then all intermediate vertices of path p are in the set {1,2,...,k-1}. Thus, a shortest path from vertex i to vertex j with all intermediate vertices in the set {1,2,...,k-1} is also the shortest path from i to j with all intermediate vertices in the set {1,2,...,k}.
- If k is an intermediate vertex of path p, then we decompose p into $i \xrightarrow{P_1} k \xrightarrow{P_2} j$, as shown below:

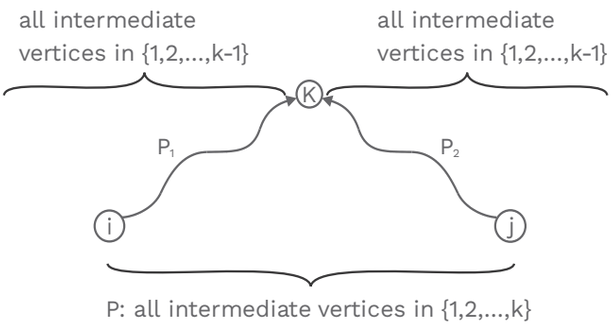


Fig. 5.2

- P_1 is a shortest path from vertex i to vertex k with all intermediate vertices in the set {1,2,....,k}. In fact, we can make a slightly stronger statement. Because vertex k is not an intermediate vertex of path p_1, all intermediate vertices of p_1 are in the set {1,2,...,k-1}. Therefore, P_1 is a shortest path from i to k with all intermediate vertices in the set {1,2,...,k-1}.
- Similarly, P_2 is a shortest path from vertex k to vertex j with all intermediate vertices in the set {1,2,...,k-1}.

Recurrence relation:

- Let $d_{ij}^{(k)}$ be the weight of a shortest path from vertex i to vertex j for which all intermediate vertices are in the set {1, 2,...,k}.
- When k=0, a path from vertex i to vertex j with no intermediate vertex.
- Such a path has at most one edge, and hence $d_{ij}^{(0)} = w_{ij}$.

  We define $d_{ij}^{(k)}$ recursively by

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1 \end{cases}$$

- Because for any path, all intermediate vertices are in the set {1,2,...,n}, the matrix $D^{(n)} = \left(d_{ij}^{(n)}\right)$ gives the final answer: $d_{ij}^{(n)} = \delta(i, j)$ for $i, j \in V$.
- Based on recurrence relation, we can us the following bottom-up procedure to compute the value $d_{ij}^{(k)}$ in order of increasing values of k.

Floyd-Warshall(W)

1. $n = W \cdot rows$
2. $D^{(0)} = W$
3. For k = 1 to n
4. Let $D^{(k)} = \left(d_{ij}^{(k)}\right)$ be a new n × n size matrix
5. For i = 1 to n
6. For j=1 to n
7. $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
8. Return $D^{(n)}$

- Its input is an n × n matrix W. The procedure returns the matrix $D^{(n)}$ of shortest-path weights.
- The running time of the Floyd-Warshall algorithm is determined by the triply nested for loops of lines 3-7 because each execution of line 7 takes O(1) time, the algorithm runs in time $\theta(n^3)$.

eg:

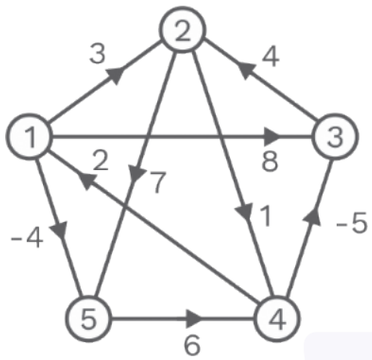


Fig. 5.3

$$D^{(0)}=\begin{array}{c} \\ 1\\2\\3\\4\\5\end{array}\begin{array}{c}\begin{array}{ccccc}1&2&3&4&5\end{array}\\\begin{bmatrix}0&3&8&\infty&-4\\\infty&0&\infty&1&7\\\infty&4&0&\infty&\infty\\2&\infty&-5&0&\infty\\\infty&\infty&\infty&6&0\end{bmatrix}\end{array}$$

$$D^{(1)}=\begin{array}{c} \\ 1\\2\\3\\4\\5\end{array}\begin{array}{c}\begin{array}{ccccc}1&2&3&4&5\end{array}\\\begin{bmatrix}0&3&8&\infty&-4\\\infty&0&\infty&1&7\\\infty&4&0&\infty&\infty\\2&5&-5&0&-2\\\infty&\infty&\infty&6&0\end{bmatrix}\end{array}$$

$$D^{(2)}=\begin{array}{c} \\ 1\\2\\3\\4\\5\end{array}\begin{array}{c}\begin{array}{ccccc}1&2&3&4&5\end{array}\\\begin{bmatrix}0&3&8&4&-4\\\infty&0&\infty&1&7\\\infty&4&0&5&11\\2&5&-5&0&-2\\\infty&\infty&\infty&6&0\end{bmatrix}\end{array}$$

$$D^{(3)}=\begin{array}{c} \\ 1\\2\\3\\4\\5\end{array}\begin{array}{c}\begin{array}{ccccc}1&2&3&4&5\end{array}\\\begin{bmatrix}0&3&8&4&-4\\\infty&0&\infty&1&7\\\infty&4&0&5&11\\2&-1&-5&0&-2\\\infty&\infty&\infty&6&0\end{bmatrix}\end{array}$$

$$D^{(4)}=\begin{array}{c} \\ 1\\2\\3\\4\\5\end{array}\begin{array}{c}\begin{array}{ccccc}1&2&3&4&5\end{array}\\\begin{bmatrix}0&3&-1&4&-4\\3&0&-4&1&-1\\7&4&0&5&3\\2&-1&-5&0&-2\\8&5&1&6&0\end{bmatrix}\end{array}$$

$$D^{(5)}=\begin{array}{c} \\ 1\\2\\3\\4\\5\end{array}\begin{array}{c}\begin{array}{ccccc}1&2&3&4&5\end{array}\\\begin{bmatrix}0&1&-3&2&-4\\3&0&-4&1&-1\\7&4&0&5&3\\2&-1&-5&0&-2\\8&5&1&6&0\end{bmatrix}\end{array}$$

Space complexity:

Here, space complexity is $O(n^3)$, but we can reduce the space complexity to $O(n^2)$.

Previous Years' Question

The Floyd-Warshall algorithm for all-pair shortest paths computation is based on:
(A) Greedy paradigm.
(B) Divide-and-Conquer paradigm
(C) Dynamic Programming paradigm
(D) Neither Greedy nor Divide-and Conquer nor Dynamic programming paradigm
Solution: (C) **[GATE 2016 (Set-2)]**

Optimal Binary Search Trees

Problem:

- Given a set of n(sorted) keys A[1...n], build the best binary search tree for the elements of A. Also, assume that, each element is associated with the frequency, which indicates the number of times that particular item is searched in the binary search tree.
- To reduce the total search time, we need to construct BST (Binary search tree).
- Understand the problem by taking 5 elements in the array. Let us assume that the given array is A=[3, 12, 21, 32, 35]. To represent these elements, there are many ways and below are two of them.

- The search time for an element depends which level node is present.
- The average number of comparisons for the first tree is: $\frac{1+2+2+3+3}{5} = \frac{11}{5}$ and for the second tree, the average number of comparisons: $\frac{1+2+3+3+4}{5} = \frac{13}{5}$.

  Among the two, the first tree is giving better results.
- Here, frequencies are not given, and if we want to search all elements, then the above simple calculation is enough for deciding the best tree.
- If the frequencies are given, then the selection depends on the frequencies of the elements and also the depth of the elements.
- For simplicity, let us assume that, the given array is A and the corresponding frequencies are in array F. F[i] indicates the frequency of i^{th} element A[i].
- With this, the total search time S(root) of the tree with root can be defined as

$$S(root) = \sum_{i=1}^{n} (depth(root, i) + 1) \times F[i])$$

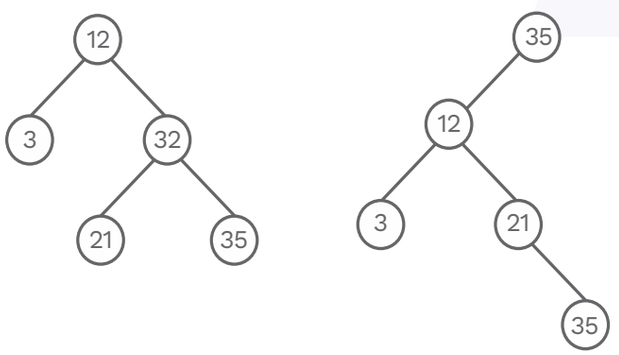


Fig. 5.4

- In the above expression, depth (root, i) + 1 indicates the number of comparisons for searching the i^{th} element.
- Since we are trying to create a binary search tree, the left subtree elements are less than the root element, and the right subtree elements are greater than the root element.
- If we separate the left subtree time and right subtree time, then the above expressions can be written as:

$$S(root) = \sum_{i=1}^{r-1} (depth(root, i) + 1) \times F[i]) + \sum_{r}^{n} F[i] + \sum_{i=r+1}^{n} (depth(root, i) + 1) \times F[i])$$

  Where "r" indicates the positions of the root element in the array.
- If we replace the left subtree and right subtree times with their corresponding recursive calls, then the expression becomes:

$$S(root) = S(root \rightarrow left) + S(root \rightarrow right) + \sum_{i=1}^{n} F[i]$$

Implementation:

```
Node optimalBST(int keys[], int freq[])
{
      int n = keys.length;
      int cost[][] = new int[n][n];
      int root[][] = new int[n][n];
      for(int i=0;i<n;i++)
      for(int j=0;j<n;j++)
      cost[i][j] = inf;
      /*
      cost[i][j] is the minimum cost of optimal
      subtree formed by the vertices[i to j]
      root[i][j] represents the index of the root
      in the subtree with vertices [i to j]
      */
      // cost of the optimal binary search tree
      // index starts at 0
intminCost=optimalCost(0,n-1,freq,cost,root);
NodeoptimalTreeRoot=buildOptimalTree(0,n-
1,keys,root);
      return optimalTreeRoot;
}
int optimalCost(int i, int j , int freq[], int cost[]
[], int root[][]){
            // base conditions
            if(i<j) return 0;
            else if(i==j) return freq[i];
            // using stored values
            if(cost[i][j] < inf) return cost[i][j];
            int minCost = inf;
            int minRoot = i;
```

```
        for(int r=i; r<=j; r++){
            // root can be any vertex from i to j
            int c = optimalCost(i,r-1,freq,cost,root) + optimalCost(r+1,j,freq,cost,root);
            if(c<minCost){
                minCost=c;
                minRoot = r;
            }
        }
        int freqSum = 0;
        for(int k=i;k<=j;k++)
            freqSum+=freq[k];
            cost[i][j] = minCost + freqSum;
            root[i][j] = minRoot;
            return cost[i][j];
}
Node buildOptimalTree(int i,int j,int keys[], int root[][]){
        // base conditions
        if(i<j) return null;
        if(i==j) return new Node(keys[i]);
        // getting the index of optimal root of subtree[i,j] stored in the matrix
        int rindex = root[i][j];
        Node node = new Node(keys[rindex]);
        node.left = buildOptimalTree(i,rindex-1,keys,root);
        node.right = buildOptimalTree(rindex+1,j,keys,root);
        return node;
}
```

Conclusion:

- We can determine whether the given problem can be solved using a dynamic approach based on the two properties:-
  - Optimal substructure: An optimal solution to a problem contains the optimal solution to subproblem.
  - Overlapping subproblems: A recursive solution the contains a small number of similar subproblems repeated many times.
- Bottom-up programming depend on values to calculate and order of evaluation. In this approach, we solve the sub-problems first only and based on the solution of the sub-problems we determine the solution to the larger problem.
- In top-down programming, the recursive structure of the original code is preserved, but unnecessary recalculation is avoided. The problem is broken into subproblems, and these subproblems are solved, and the solutions are remembered, in case they need to be solved again.

Note:
Some problems can be solved with both techniques.

Solved Examples

1. Let A_1, A_2, A_3 and A_4 be four matrices of dimensions 1 × 2, 2 × 1, 1 × 4 and 4 × 1 respectively. The minimum number of scalar multiplications required to find the product $A_1A_2A_3A_4$ using the basic matrix multiplication method is _____

Solution: 7

The unique function call which are made in m[1, 4] are given below:

0 (1, 1)	0 (2, 2)	0 (3, 3)	0 (4, 4)
2 (1, 2)	8 (2, 3)	4 (3, 4)	
6 (1, 3)	6 (2, 4)		
7 (1, 4)			

$$M[i, j] = \begin{cases} 0 & \text{, if } i = j \\ \min\limits_{i \le k < j} \{M[i,k] + M[k+1, j] + p_{i-1}p_k p_j\} & \text{if } i < j \end{cases}$$

Number of scalar multiplication

$A_1A_2 = 1 \times 2 \times 1 = 2$

$A_2A_3 = 2 \times 1 \times 4 = 8$

$A_3A_4 = 1 \times 4 \times 1 = 4$

$$A_1A_2A_3 = m(1,3) = \min \begin{cases} m(1,1) + m(2,3) + 1 \times 2 \times 4 \\ m(1,2) + m(3,3) + 1 \times 1 \times 4 \end{cases}$$

$$= \min \begin{cases} 0 + 8 + 8 \\ 2 + 0 + 4 \end{cases} = 6$$

$$A_2A_3A_4 = m(2,4) = \min \begin{cases} m(2,2) + m(3,4) + 2 \times 1 \times 1 \\ m(2,3) + m(4,4) + 2 \times 4 \times 1 \end{cases}$$

$$= \min \begin{cases} 0 + 4 + 2 \\ 8 + 0 + 8 \end{cases} = 6$$

$$A_1A_2A_3A_4(1,4) = \min \begin{cases} m(1,1) + m(2,4) + 1 \times 2 \times 1 \\ m(1,2) + m(3,4) + 1 \times 1 \times 1 \\ m(1,3) + m(4,4) + 1 \times 4 \times 1 \end{cases}$$

$$= \min \begin{cases} 0 + 6 + 2 \\ 2 + 4 + 1 \\ 6 + 0 + 4 \end{cases} = \min \begin{cases} 8 \\ 7 \\ 10 \end{cases} = 7$$

Hence, 7 is the minimum number of scalar multiplication required to find the product $A_1A_2A_3A_4$.

2. Let us define A_iA_{i+1} as an explicitly computed pair for a given parenthesization if they are directly multiplied. For example, in the matrix multiplication chain $A_1A_2A_3A_4$ using parenthesization $\mathbf{A_1((A_2A_3)A_4)}$, A_2 A_3 are only explicitly computed pairs.

Consider the matrix given in question number 1 that minimises the total number of scalar multiplication, the explicitly computed pairs is/are

(A) A_1A_2 and A_3A_4 only

(B) A_2A_3 only

(C) A_1A_2 only

(D) A_3A_4 only

Solution: (A)

In question 1, we got the optimal scalar as 7 in (A_1A_2), (A_3A_4). So, A_1A_2, and A_3A_4 are explicitly computed pairs.

3. Consider two strings X = "ABCBDAB" and Y = "BDCABA". Let u be the length of the longest common subsequences (not necessarily contiguous) between X and Y and let V be the number of such longest common subsequences between X and Y. then V + 10u is_____

Solution: 43

		0	1	2	3	4	5	6
		Y	B	D	C	A	B	A
0	X	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
7	B	0	1	2	2	3	4	4

Here, the subsequence are shown below:

X(2 3 4 6) X(4 5 6 7) X(2 3 6 7)

Y(1 3 5 6) Y(1 2 4 5) Y(1 3 4 5)

B C B A B D A B B C A B

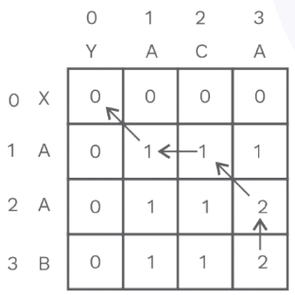
The length of the longest common subsequence is 4 and there are 3 subsequence of length 4.

So, u = 4 and v = 3

So, v + 10u = 3 + 40 = 43

4. Consider two string X = "AAB" and Y = "ACA". Let u be the length of the longest common subsequence (not necessarily contiguous) between X and Y and let v be the number of such longest common subsequences between X and Y.
Then u + 10v is_____

Solution: 12

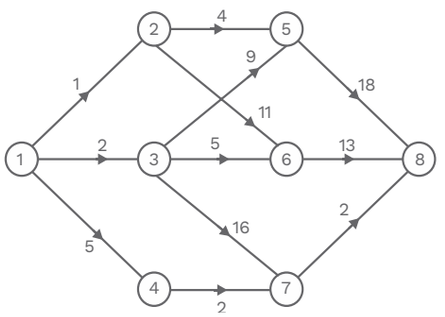


Here, the subsequence is AA. So, the length of subsequence is 2, and there is only 1 occurrence of subsequence.

Hence, u = 2, v = 1

u + 10v = 2 + 10*1 = 12.

5. Consider the multistage graph K = 4 then find out the minimum cost path from node 1 to node 8?



Solution: 9

The path is from $1 \rightarrow 4 \rightarrow 7 \rightarrow 8$, which incur a minimum cost as 9.

T[i] denotes the minimum cost from ith node to node 8.

$$\text{So, } T[i] = \underset{(\text{for all } j=(i+1) \text{ to } n)}{\text{minimum}} \{\text{cost}(i, j) + T[j]\}$$

$$T[8] = 0 \quad j = (i+1) \text{ to } n$$

$$T[7] = \text{minimum}\{\text{cost}[7,8] + T[8]\}$$

$$= \text{minimum}\{2+0\} = 2$$

$$T[6] = \text{minimum}\begin{cases}\text{cost}[6,7] + T[7] \\ \text{cost}[6,8] + T[8]\end{cases}$$

$$T[6] = \text{minimum}\begin{cases}\infty + 2 \\ \qquad = 13 \\ 13 + 0\end{cases}$$

$$T[5] = \text{minimum}\begin{cases}\text{cost}[5,6] + T[6] \\ \text{cost}[5,7] + T[7] \\ \text{cost}[5,8] + T[8]\end{cases}$$

$$T[5] = \text{minimum}\begin{cases}\infty + 13 \\ \infty + 2 \quad = 18 \\ 18 + 0\end{cases}$$

$$T[4] = \text{minimum}\begin{cases}\text{cost}[4,5] + T[5] \\ \text{cost}[4,6] + T[6] \\ \text{cost}[4,7] + T[7] \\ \text{cost}[4,8] + T[8]\end{cases}$$

$$T[4] = \text{minimum}\begin{cases}\infty + 18 \\ \infty + 13 \quad = 4 \\ 2 + 2 \\ \infty + 0\end{cases}$$

$$T[3] = \text{minimum}\begin{cases}\text{cost}[3,4] + T[4] \\ \text{cost}[3,5] + T[5] \\ \text{cost}[3,6] + T[6] \\ \text{cost}[3,7] + T[7] \\ \text{cost}[3,8] + T[8]\end{cases}$$

$$= \text{minimum}\begin{cases}\infty + 4 \\ 9 + 18 \\ 5 + 13 = 18 \\ 16 + 2 \\ \infty + 0\end{cases}$$

$$T[2] = \text{minimum}\begin{cases}\text{cost}[2,3]+T[3]\\ \text{cost}[2,4]+T[4]\\ \text{cost}[2,5]+T[5]\\ \text{cost}[2,6]+T[6]\\ \text{cost}[2,7]+T[7]\\ \text{cost}[2,8]+T[8]\end{cases}$$

$$= \text{minimum}\begin{cases}\infty+18\\ \infty+4\\ 4+18\\ 11+13\\ \infty+2\\ \infty+0\end{cases} = 22$$

$$T[1] = \text{minimum}\begin{cases}\text{cost}[1,2]+T[2]\\ \text{cost}[1,3]+T[3]\\ \text{cost}[1,4]+T[4]\\ \text{cost}[1,5]+T[5]\\ \text{cost}[1,6]+T[6]\\ \text{cost}[1,7]+T[7]\\ \text{cost}[1,8]+T[8]\end{cases}$$

$$= \text{minimum}\begin{cases}1+22\\ 2+18\\ 5+4\\ \infty+18\\ \infty+13\\ \infty+2\\ \infty+0\end{cases} = 9$$

Hence, 9 is the minimum cost path from node 1 to node 8.

6. Shortest path in the multistage graph can be found by using

(A) Greedy method

(B) Dynamic method

(C) Either by greedy method or dynamic method

(D) None of above

Solution: (B)

Greedy method fails in finding the shortest path in the multistage graph.

By using dynamic programming, we can get the shortest path in the multistage graph.

7. Consider the weights and values of the items listed below.

Item Number	Weight (in kgs)	Values (in Rupees)
1	1	10
2	2	12
3	4	28

The task is to pick a subset of these items such that their total weight is no more than 6kg and their total value is maximised. Moreover, no item may be split. The total values of items picked by 0/1 knapacks is_____

Solution: 40

object \ weight	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0	10	12	22	22	22	22
3	0	10	12	22	28	38	40

If there is no object, then the maximum profit (value) is going to be 0. This is indicated by 1st row.

Similarly, if the weight is 0, then also maximum profit is 0. This is indicated by 1st column.

Let ks(i,w) indicate maximum profit if considering i's number of elements and maximum weight occupied is w.

By using recurrence equation:

$$ks(i,w) = \begin{cases}\max(p_i + ks(i-1, w-w_i), ks(i-1,w)), w_i \le w\\ 0; i = 0 \text{ or } w = 0\\ ks(i-1,w); w_i > w\end{cases}$$

$$ks(1,1) = \max\begin{cases}p_1 + ks(0,0)\\ ks(0,1)\end{cases} = \max\begin{cases}10+0\\ 0\end{cases} = 10$$

$$ks(1,2) = \max\begin{cases}p_1 + ks(0,1)\\ ks(0,2)\end{cases} = \max\begin{cases}10+0\\ 0\end{cases} = 10$$

$$ks(1,3) = \max\begin{cases}p_1 + ks(0,2)\\ ks(0,3)\end{cases} = \max\begin{cases}10+0\\ 0\end{cases} = 10$$

$$ks(1,4) = \max\begin{cases} p_1 + ks(0,3) \\ ks(0,4) \end{cases} = \max\begin{cases} 10+0 \\ 0 \end{cases} = 10$$

$$ks(1,5) = \max\begin{cases} p_1 + ks(0,4) \\ ks(0,5) \end{cases} = \max\begin{cases} 10+0 \\ 0 \end{cases} = 10$$

$$ks(1,6) = \max\begin{cases} p_1 + ks(0,5) \\ ks(0,6) \end{cases} = \max\begin{cases} 10+0 \\ 0 \end{cases} = 10$$

$$ks(2,1) = ks(1,1) = 10$$

$$ks(2,2) = \max\begin{cases} p_2 + ks(1,0) \\ ks(1,2) \end{cases} = \max\begin{cases} 12+0 \\ 10 \end{cases} = 12$$

$$ks(2,3) = \max\begin{cases} p_2 + ks(1,1) \\ ks(1,3) \end{cases} = \max\begin{cases} 12+10 \\ 10 \end{cases} = 22$$

$$ks(2,4) = \max\begin{cases} p_2 + ks(1,2) \\ ks(1,4) \end{cases} = \max\begin{cases} 12+10 \\ 10 \end{cases} = 22$$

$$ks(2,5) = \max\begin{cases} p_2 + ks(1,3) \\ ks(1,5) \end{cases} = \max\begin{cases} 12+10 \\ 10 \end{cases} = 22$$

$$ks(2,6) = \max\begin{cases} p_2 + ks(1,4) \\ ks(1,6) \end{cases} = \max\begin{cases} 12+10 \\ 10 \end{cases} = 22$$

$$ks(3,1) = ks(2,1) = 10$$

$$ks(3,2) = ks(2,2) = 12$$

$$ks(3,3) = ks(2,3) = 22$$

$$ks(3,4) = \max\begin{cases} p_3 + ks(2,0) \\ ks(2,4) \end{cases} = \max\begin{cases} 28+0 \\ 22 \end{cases} = 28$$

$$ks(3,5) = \max\begin{cases} p_3 + ks(2,1) \\ ks(2,5) \end{cases} = \max\begin{cases} 28+10 \\ 22 \end{cases} = 38$$

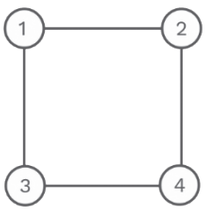
$$ks(3,6) = \max\begin{cases} p_3 + ks(2,2) \\ ks(2,6) \end{cases} = \max\begin{cases} 28+12 \\ 22 \end{cases} = 40$$

8. Subset-sum can be computed by
(A) Row-major order only
(B) Column-major order only
(C) Either by row-major for column-major only
(D) None of the above

Solution: (C)

It can be computed by eighter row-major or column-major only.

9. What is the minimum cost of the travelling salesman problem, if starting vertex is 1?



Cost matrix:–

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

Solution: 5

$$T(1,\{2,3,4\}) = \min\begin{cases} (1,2) + T(2,\{3,4\}) \\ (1,3) + T(3,\{2,4\}) \\ (1,4) + T(4,\{2,3\}) \end{cases}$$

$$= \min\begin{cases} 1+4 \\ 2+7 \\ 3+4 \end{cases}$$

$$= 5$$

$$T(2,\{3,4\}) = \min\begin{cases} (2,3) + T(3,\{4\}) \\ (2,4) + T(4,\{3\}) \end{cases} = \min\begin{cases} 4+8 \\ 2+2 \end{cases} = 4$$

$$T(3,\{2,4\}) = \min\begin{cases} (3,4) + T(4,\{2\}) \\ (3,2) + T(2,\{4\}) \end{cases} = \min\begin{cases} 5+5 \\ 2+5 \end{cases} = 7$$

$$T(4,\{2,3\}) = \min\begin{cases} (4,2) + T(2,\{3\}) \\ (4,3) + T(3,\{2\}) \end{cases} = \min\begin{cases} 4+5 \\ 1+3 \end{cases} = 4$$

$$T(3,\{4\}) = (3,4) + T(4,1) = 5+3 = 8$$

$$T(4,\{3\}) = (4,3) + T(3,1) = 1+1 = 2$$

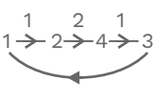
$$T(2,\{4\}) = (2,4) + T(4,1) = 2+3 = 5$$

$$T(4,\{2\}) = (4,2) + T(2,1) = 4+1 = 5$$

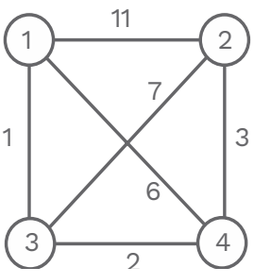
$$T(2,\{3\}) = (2,3) + T(3,1) = 4+1 = 5$$

$$T(3,\{2\}) = (3,2) + T(2,1) = 2+1 = 3$$

So, minimum cost is 5 and the path is



10. Find the length of the shortest path between all pair vertices for the given graph G.



(A)
$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 6 & 1 & 3 \\ 6 & 0 & 5 & 3 \\ 1 & 5 & 0 & 2 \\ 3 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

(B)
$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 11 & 1 & 3 \\ 11 & 0 & 7 & 3 \\ 1 & 7 & 0 & 2 \\ 3 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

(C)
$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 11 & 1 & 6 \\ 11 & 0 & 7 & 3 \\ 1 & 7 & 0 & 2 \\ 6 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

(D) None of above

Solution: (A)

Let D^i = set of all shortest path between every pair in such a way that the path is allowed to go through node 0 to node i.

So,

$D^0=$
$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 11 & 1 & 6 \\ 11 & 0 & 7 & 3 \\ 1 & 7 & 0 & 2 \\ 6 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

$D^1=$
$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 11 & 1 & 6 \\ 11 & 0 & 7 & 3 \\ 1 & 7 & 0 & 2 \\ 6 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

$D^2=$
$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 11 & 1 & 6 \\ 11 & 0 & 7 & 3 \\ 1 & 7 & 0 & 2 \\ 6 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

$D^3=$
$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 8 & 1 & 3 \\ 8 & 0 & 7 & 3 \\ 1 & 7 & 0 & 2 \\ 3 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

$D^4=$
$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 6 & 1 & 3 \\ 6 & 0 & 5 & 3 \\ 1 & 5 & 0 & 2 \\ 3 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

Chapter Summary

- The major components of dynamic programming are:

  Recursion: Solves subproblems recursively.
  Memoization: Stores the result of sub-problems.
  Dynamic programming = Recursion + Memoization
- The two dynamic programming properties, which tells whether it can solve the given problem or not are: Optimal substructure: An optimal solution to a problem contains optimal solutions to subproblems.

  Overlapping subproblems: When a large problem is break down into smaller sub-problems, then there are many some sub-problems which are the same and repeated many times; these are known as overlapping sub problem.
- Basically, there are two approaches for solving DP problems:
  - Bottom-up dynamic programming
  - Top-down dynamic programming

  Bottom-up dynamic programming: In this method, we evaluate the function starting with the smallest possible input argument value, and then we step through possible values, and slowly increase the input argument value.
  While computing the values, we store all computed values in a table (memory).
  As larger arguments are evaluated, pre-computed values for smaller arguments can be used.
- Top-down dynamic programming: In this method, the problem is broken into subproblems, and these subproblems are solved, and the solutions are remembered, in case they need to be again solved. Also, we save the each computed value as the final action of a recursive function, and as the first action we check if the pre-computed value exists.
- Example of dynamic programming
  - Fibonacci series
  - Matrix chain multiplication
  - Longest common subsequence
  - Subset sum problem
  - 0/1 knapnack
  - Multistage graph
  - Travelling salesman problem
  - All-pairs shortest path Floyd–Warshall
- Fibonacci Series:
  Recurrence equation:
  Fib(n) = 0, if n = 0
  = 1, if n = 1
  = Fib(n-1) + Fib(n-2), if n>1
  Solving the fibonacci series by using dynamic programming takes time and space complexity as O(n).
- For all problems, it may not be possible to find both top-down and bottom-up programming solution.

Matrix Chain Multiplication

Problem: Given a series of matrices: $A_1 \times A_2 \times A_3 \times \ldots \times A_n$ with their dimensions, what is the best way to parenthesize them so that it produces the minimum scalar multiplication.

Number of ways we can parenthesis the matrix = $\frac{(2n)!}{(n+1)!n!}$

Where n = number of matrix – 1

Let M[i, j] represents the least number of multiplications needed to multiply $A_i \ldots A_j$.

$$M[i, j] = \begin{cases} 0, \text{if } i = j \\ \text{Min}\{M[i,K] + M[K+1, j] + P_{i-1}P_KP_j\} \text{ if } i < j \end{cases}$$

- Bottom-up matrix chain multiplication
  Time complexity = $O(n^3)$
  Space Complexity = $O(n^2)$.
  Top down dynamic programming of matrix chain multiplication
  Time complexity = $O(n^3)$
  Space Complexity = $O(n^2)$
- Longest common subsequence: Given two strings: string X of length m [X(1...m)], and string Y of length n [Y(1...n)], find longest common subsequence: The longest sequence of characters that appear left-to-right (but not necessarily in a contiguous block) in both strings for example X = "ABCBDAB" and Y = "BDCABA", the LCS(X,Y) = {"BCBA", "BDAB", "BCAB"}
- Brute force approach of longest common subsequence is $O(n2^m)$
- Recursive equation of the LCS is

$$C[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ C[i-1, j-1] + 1 \text{ if } i, j > 0 \text{ and } x_i = y_j, \\ \max\left(C[i, j-1], C[i-1, j]\right), \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

- Dynamic programming approach of longest common subsequence takes $\theta(mn)$ as both time complexity and space complexity.
- **Multistage graph:** It is a directed graph in which the nodes can be divided into a set of stages such that all edges are from one stage to the next stage only, and there will be no edges between vertices of the same stage.
  Greedy methods fails to find the shortest path from source to destination in multistage graph.
- Let 'S' be the source node and 'T' be the target node and let at stage 2 have nodes A, B and C. So, the recursive equation look like as shown below:

$$M(N-1, i) = i \rightarrow T$$

$$M(1, S) = \text{minimum} \begin{cases} S \rightarrow A + M(2, A) \\ S \rightarrow B + M(2, B) \\ S \rightarrow C + M(2, C) \end{cases}$$

  M is a function which represents the shortest path with minimum cost.
- Time complexity of bottom-up dynamic programming of multistage graph problem is O(E).

0/1 knapsack problem: In this problem, 'n' distinct objects are there, which are associated with two integer value s and v where, s represents the size of the object and v represents the value of the object. We need to fill a knapsack of total capacity w with items of maximum value. Hence, we are not allowed to take partial of items.
Let us say KS(i,w) represent knapsack. Here, i is the ith elements to be considered, and the capacity of knapsack remaining is w.
The recurrence equation is:

$$KS(i,w)=\begin{cases}\max\left(P_i+KS(i-1,w-w_i),KS(i-1,w)\right);w_i\le w\\0;i=0\ \text{or}\ w=0\\KS(i-1,w);w_i>w(w_i\ \text{is the weight of}\ i^{th}\ \text{element})\end{cases}$$

Time complexity of 0/1 knapsack = minimum $\begin{cases}O(2^n)\\O(nw)\end{cases}$ (n is the number of object and w is the total capacity of the knapsack.)

- **Subset sum problem:** Given a sequence of n positive numbers A_1, A_2,....A_n, give an algorithm which checks whether there exists a subset of A whose sum of all numbers is T.
- Brute force method time complexity of subset sum problem is $O(2^n)$.
- Let us assume that SS(i,S) denote sum from a_1 to a_i whose sum is equal to some number 'S'.
- Recursive equation of subset-sum is given below:

$$SS(i,S)=\begin{cases}SS(i-1,S);S<a_i\\SS(i-1,S-a_i)\ \text{VSS}(i-1,S);S\ge a_i\\\text{true};\ S=0\\\text{False};i=0,S\ne 0\end{cases}$$

- Time complexity of subset-sum = minimum $\begin{cases}O(2^n)\\O(nw)\end{cases}$

All Pair Shortest Path Floyd Warshall

Problem: Given a weighted directed graph G=(V,E), where V = {1, 2, ...,n}. Find the shortest path between all pair of nodes in the graph.

- The running time of Dijkstra's algorithm if we use to find all pairs shortest path is O(VElogV).
- The running time of Bellman-Ford algorithm if we use to find all pair shortest path is $O(V^2E)$.
- Let $d_{ij}^{(k)}$ be the weight of a shortest path from vertex i to vertex j for which all intermediate vertices are in the set {1, 2, ...,k}
The recurrence equation is:

$$d_{ij}^{(k)}=\begin{cases}w_{ij} & ,\text{if}\ k=0\\\min\left(d_{ij}^{(k-1)},d_{ik}^{(k-1)}+d_{kj}^{(k-1)}\right),\text{if}\ k\ge 1\end{cases}$$

- The time complexity of all pairs shortest path Floyd–Warshall in is $O(V^3)$. (where V = number of vertices)
- The space complexity of all pairs shortest pat Floyd–Warshall is $O(V^3)$, but it can be reduced to $O(V^2)$. (where V = number of vertices).