Dynamic Programming

- **Dynamic Programming** is mainly an optimization over plain recursion.
- Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming.
- The idea is to simply store the results of subproblems, so that we do not have to re-compute them when needed later.
- Dynamic programming is both a **mathematical optimization** method and a **computer programming** method.
- The method was developed by **Richard Bellman** in the 1950s and has found applications in numerous fields, from aerospace engineering to economics.

- Like divide-and-conquer method, **Dynamic Programming** solves problems by combining the solutions of subproblems. ...
- Moreover, **Dynamic Programming** algorithm solves each sub-problem just once and then saves its answer in a table, thereby avoiding the work of re-computing the answer every time.
- Dynamic programming is an algorithm design method that can be used when the solution to a problem can be viewed as the result of a sequence of decisions.

Principle of optimality

The principle of optimality states that an optimal sequence of decisions has the property that whatever the initial state and decision are , the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision.

Applications of Dynamic Programming

- All Pairs Shortest Paths
- Single Source Shortest Paths General Weights
- Optimal Binary Search Tree
- String Edition
- 0/1 Knapsack Problem
- Reliability Design

All Pairs Shortest Paths

- The **all pair shortest path** algorithm is also known as Floyd-Warshall algorithm is used to find **all pair shortest path** problem from a given weighted graph.
- As a result of this algorithm, it will generate a matrix, which will represent the minimum distance from any node to **all** other nodes in the graph.

- Let G = (V, E) be a directed graph with n vertices.
- The all-pairs shortest-path problem is to determine a matrix A such that A(i,j)is the length of a shortest path from i toj.
- A(i,j)={ 0 if i=j

  the weight of the directed edge ⟨i,j⟩ if i≠j and

  ⟨i,j⟩∈E ∞ if i≠j and ⟨i,j⟩∉E

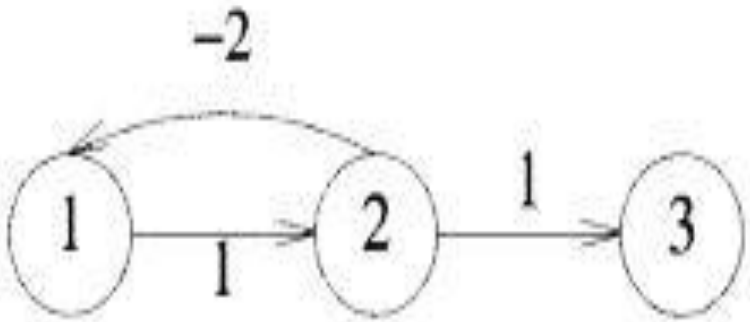
  A(i,j)= min { min {A^k(i,k) +A^k-1(k,j)},cost(i, j)}

  i<k<n

Algorithm for All pairs shortest paths

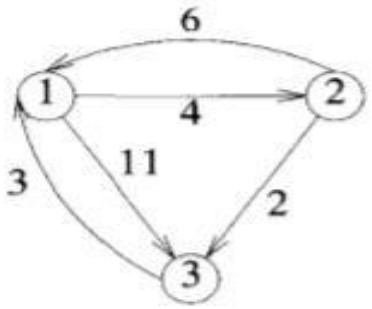
```
Algorithm AllPaths(cost, A, n)
// cost[1 : n, 1 : n] is the cost adjacency matrix of a graph with
// n vertices; A[i, j] is the cost of a shortest path from vertex
// i to vertex j. cost[i, i] = 0.0, for 1 ≤ i ≤ n.
{
     for i := 1 to n do
          for j := 1 to n do
               A[i, j] := cost[i, j]; // Copy cost into A.
     for k := 1 to n do
          for i := 1 to n do
               for j := 1 to n do
                    A[i, j] := min(A[i, j], A[i, k] + A[k, j]);
}
```

Example



$$\begin{bmatrix} 0 & 1 & \infty \\ -2 & 0 & 1 \\ \infty & \infty & 0 \end{bmatrix}$$

Example:



(a) Example digraph

A^0	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

(b) A^0

A^1	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0

(c) A^1

A^2	1	2	3
1	0	4	6
2	6	0	2
3	3	7	0

(d) A^2

A^3	1	2	3
1	0	4	6
2	5	0	2
3	3	7	0

(e) A^3

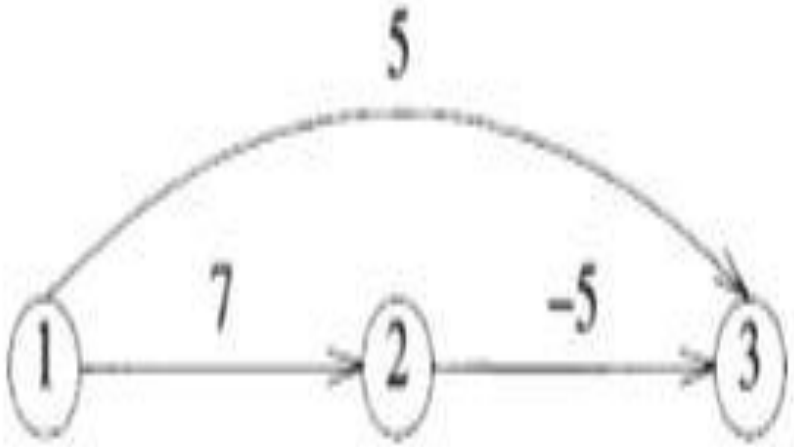
Time Complexity of APSP

- **Time complexity** of Floyd Warshall Algorithm is $\Theta(V^3)$, here V is the number of vertices in the graph.

Single Source Shortest Paths General Weights

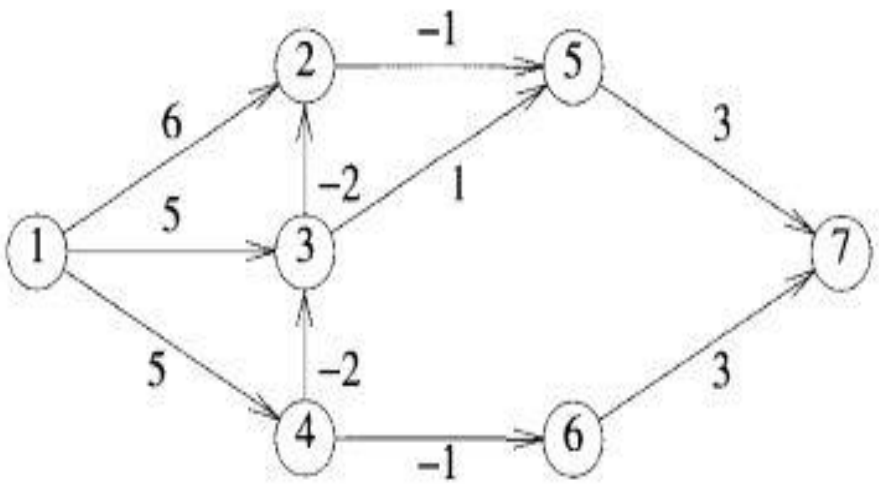
- The idea is to use **Bellman–Ford** algorithm to compute the shortest paths from a single source vertex to all of the other vertices in given weighted digraph.
- **Bellman–Ford** algorithm is slower than Dijkstra's Algorithm but it is capable of handling negative weights edges in the **graph** unlike Dijkstra's.

Single Source Shortest Paths(General Weights)



- When negative edge lengths are permitted ,we require that the graph have no cycles of negative length. This is necessary to ensure that shortest paths consist of a finite number of edge.

Example



(a) A directed graph

	$dist^k[1..7]$						
k	1	2	3	4	5	6	7
1	0	6	5	5	∞	∞	∞
2	0	3	3	5	5	4	∞
3	0	1	3	5	2	4	7
4	0	1	3	5	0	4	5
5	0	1	3	5	0	4	3
6	0	1	3	5	0	4	3

(b) $dist^k$

Algorithm for SSSP

```
Algorithm BellmanFord(v, cost, dist, n)
// Single-source/all-destinations shortest
// paths with negative edge costs
{
     for i := 1 to n do // Initialize dist.
          dist[i] := cost[v, i];
     for k := 2 to n − 1 do
          for each u such that u ≠ v and u has
                    at least one incoming edge do
               for each ⟨i, u⟩ in the graph do
                    if dist[u] > dist[i] + cost[i, u] then
                         dist[u] := dist[i] + cost[i, u];
}
```

Time Complexity

- The over all complexity is (O^3)when adjacency matrices are used and 0(ne)when adjacency lists are used.

Optimal Binary Search Tree

- A Binary Search Tree (BST) is a tree where the key values are stored in the internal nodes.
- The external nodes are null nodes.
- The keys are ordered lexicographically, i.e. for each internal node all the keys in the left sub-tree are less than the keys in the node, and all the keys in the right sub-tree are greater.
- An optimal binary search tree is a BST, which has **minimal expected cost of locating each node.**

Optimal Binary Search Tree

- Search time of an element in a BST is ***O(n)***, whereas in a Balanced-BST search time is ***O(log n)***.
- Again the search time can be improved in Optimal Cost Binary Search Tree, placing the most frequently used data in the root and closer to the root element, while placing the least frequently used data near leaves and in leaves.
- Here we assume, the probability of accessing a key K_i is p_i.
- Some dummy keys ($d_0, d_1, d_2, \ldots d_n$) are added as some searches may be performed for the values which are not present in the Key set K.
- We assume, for each dummy key d_i probability of access is q_i.

Important steps

- OBST(i, j) denotes the optimal binary search tree containing the keys ki, ki+1, …, kj;
- Wi, j denotes the weight matrix for OBST(i, j) Wi, j can be defined using the following formula: W,
- Ci, j, 0 ≤ i ≤ j ≤ n denotes the cost matrix for OBST(i, j) Ci, j can be defined recursively, in the following manner:
- Ci, i = Wi, j
- Ci, j = Wi, j + mini (Ci,k-1+Ck,j)

$$i < k \leq j$$

- Ri, j, 0 ≤ i ≤ j ≤ n denotes the root matrix for OBST(i, j)

Algorithm for OBST

```
Algorithm OBST(p, q, n)
// Given n distinct identifiers a_1 < a_2 < ··· < a_n and probabilities
// p[i], 1 ≤ i ≤ n, and q[i], 0 ≤ i ≤ n, this algorithm computes
// the cost c[i, j] of optimal binary search trees t_ij for identifiers
// a_{i+1}, ..., a_j. It also computes r[i, j], the root of t_ij.
// w[i, j] is the weight of t_ij.
{
    for i := 0 to n − 1 do
    {
        // Initialize.
        w[i, i] := q[i]; r[i, i] := 0; c[i, i] := 0.0;
        // Optimal trees with one node
        w[i, i + 1] := q[i] + q[i + 1] + p[i + 1];
        r[i, i + 1] := i + 1;
        c[i, i + 1] := q[i] + q[i + 1] + p[i + 1];
    }
    w[n, n] := q[n]; r[n, n] := 0; c[n, n] := 0.0;
    for m := 2 to n do // Find optimal trees with m nodes.
        for i := 0 to n − m do
        {
            j := i + m;
            w[i, j] := w[i, j − 1] + p[j] + q[j];
            // Solve 5.12 using Knuth's result.
            k := Find(c, r, i, j);
                // A value of l in the range r[i, j − 1] ≤ l
                // ≤ r[i + 1, j] that minimizes c[i, l − 1] + c[l, j];
            c[i, j] := w[i, j] + c[i, k − 1] + c[k, j];
            r[i, j] := k;
        }
    write (c[0, n], w[0, n], r[0, n]);
}

Algorithm Find(c, r, i, j)
{
    min := ∞;
    for m := r[i, j − 1] to r[i + 1, j] do
        if (c[i, m − 1] + c[m, j]) < min then
        {
            min := c[i, m − 1] + c[m, j]; l := m;
        }
    return l;
}
```

Example

- Let n = 4 and (a1,a2,a3,a4) = (do,if, int,while).Let p(l:4) = (3,3,1,1) and q(0:4) = (2,3,1,1,1). The p's and q's have been multiplied by 16 for convenience. Initially, we have w(i,i)= q(i),c(i,i) = 0 and r(i,i)= 0,0 <i <4.
- W(I,j)=p(j)+q(j)+w(I,j-1)

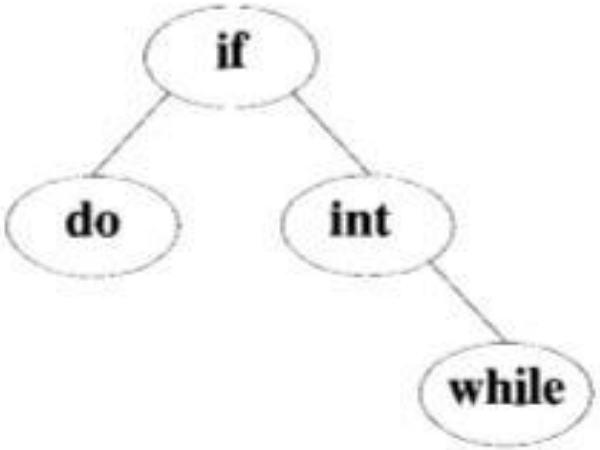
Computations

$$
\begin{aligned}
w(0,1) &= p(1) + q(1) + w(0,0) = 8 \\
c(0,1) &= w(0,1) + \min\{c(0,0) + c(1,1)\} = 8 \\
r(0,1) &= 1 \\
w(1,2) &= p(2) + q(2) + w(1,1) = 7 \\
c(1,2) &= w(1,2) + \min\{c(1,1) + c(2,2)\} = 7 \\
r(0,2) &= 2 \\
w(2,3) &= p(3) + q(3) + w(2,2) = 3 \\
c(2,3) &= w(2,3) + \min\{c(2,2) + c(3,3)\} = 3 \\
r(2,3) &= 3 \\
w(3,4) &= p(4) + q(4) + w(3,3) = 3 \\
c(3,4) &= w(3,4) + \min\{c(3,3) + c(4,4)\} = 3 \\
r(3,4) &= 4
\end{aligned}
$$

Solution Table

	0	1	2	3	4
0	$w_{00} = 2$ $c_{00} = 0$ $r_{00} = 0$	$w_{11} = 3$ $c_{11} = 0$ $r_{11} = 0$	$w_{22} = 1$ $c_{22} = 0$ $r_{22} = 0$	$w_{33} = 1$ $c_{33} = 0$ $r_{33} = 0$	$w_{44} = 1$ $c_{44} = 0$ $r_{44} = 0$
1	$w_{01} = 8$ $c_{01} = 8$ $r_{01} = 1$	$w_{12} = 7$ $c_{12} = 7$ $r_{12} = 2$	$w_{23} = 3$ $c_{23} = 3$ $r_{23} = 3$	$w_{34} = 3$ $c_{34} = 3$ $r_{34} = 4$	
2	$w_{02} = 12$ $c_{02} = 19$ $r_{02} = 1$	$w_{13} = 9$ $c_{13} = 12$ $r_{13} = 2$	$w_{24} = 5$ $c_{24} = 8$ $r_{24} = 3$		
3	$w_{03} = 14$ $c_{03} = 25$ $r_{03} = 2$	$w_{14} = 11$ $c_{14} = 19$ $r_{14} = 2$			
4	$w_{04} = 16$ $c_{04} = 32$ $r_{04} = 2$				

OBST



Time Complexity

- The algorithm requires O (n^3) **time**, since three nested for loops are used. Each of these loops takes on at most n values.

Exercise:

- Use function OBST to compute w(i,j),r(i,j),and c(i,j), 0 < i < j < 4, for the identifier set (01,02,03,04) = (cout , float , if, while) with p(l)= 1/20,p(2) = 1/5,p(3) = 1/10,p(4) = 1/20,q(0)= 1/5,q(l) = 1/10,q(2)= 1/5,q(3)= 1/20,and q(4)= 1/20.Usingthe r(i,j)'s, construct the optimal binary search tree.

STRING EDITING

- Given two strings (sequences) return the “distance” between the two strings as measured by...
- The minimum number of “character edit operations” needed to turn one sequence into the other is known as **String Edit Distance**.
- X=Andrew, Y=Amdrewz

1. substitute m to n
2. delete the z

There fore Distance = 2

STRING EDITING

- String distance metrics:• Given strings s and t
- Distance is the shortest sequence of edit commands that transform s to t, (or equivalently s to t).
- Simple set of operations:
  - Copy character from s over to t (cost 0)
  - Delete a character in s (cost 1)
  - Insert a character in t (cost 1)
  - Substitute one character for another (cost 1)
  - This is “Levenshtein distance”

Levenshtein distance - example

1.Distance(“William Cohen” , “Willliam Cohon”)

2. “Another fine day in the park “

“Anybody can see him pick the ball”

Applications

- **File Revision**
- **Spelling Correction**
- **Plagiarism Detection**
- **Speech Recognition**
- **Molecular Biology(DNA test)**

Approach:

- For i = j = 0,cost(i,j) = 0,since the two sequences are identical(and empty).
- Also, if j = 0 and i >0,we can transform X into Y by a sequence of deletes.
- Thus, cost(i,0) = cost(i-1,0) +D(xi).
- Similarly, if i = 0 and j >0, we get cost(0, j) = cost(0, j -1)+I(yj).
- If i >0 and j > 0,x1,x2, … xi can be transformed into yi,y2,---,yj in one of three ways:

Approach:

1. Transform $x_1, x_2, \ldots, x_{i-1}$ into $y_1, y_2, \ldots, y_j$ using a minimum-cost edit sequence and then delete x_i. The corresponding cost is $cost(i-1, j) + D(x_i)$.

2. Transform $x_1, x_2, \ldots, x_{i-1}$ into $y_1, y_2, \ldots, y_{j-1}$ using a minimum-cost edit sequence and then change the symbol x_i to y_j. The associated cost is $cost(i-1, j-1) + C(x_i, y_j)$.

3. Transform $x_1, x_2, \ldots, x_i$ into $y_1, y_2, \ldots, y_{j-1}$ using a minimum-cost edit sequence and then insert y_j. This corresponds to a cost of $cost(i, j-1) + I(y_j)$.

Formula:

$$cost(i,j) = \begin{cases} 0 & i = j = 0 \\ cost(i-1,0) + D(x_i) & j = 0,\ i > 0 \\ cost(0,j-1) + I(y_j) & i = 0,\ j > 0 \\ cost'(i,j) & i > 0,\ j > 0 \end{cases}$$

$$\text{where } cost'(i,j) = \min \{ \begin{array}{l} cost(i-1,j) + D(x_i), \\ cost(i-1,j-1) + C(x_i,y_j), \\ cost(i,j-1) + I(y_j) \ \} \end{array}$$

Example:

- Consider the string editing problem of X = a,a,b,a,b and Y = b,a,b,b. Each insertion and deletion has a unit cost and a change costs 2 units. For the cases i = 0,j >1,and j = 0, i >1,cost(i,j) can be computed first .Let us compute the rest of the entries in row-major order. The next entry to be computed is cost(1,1).

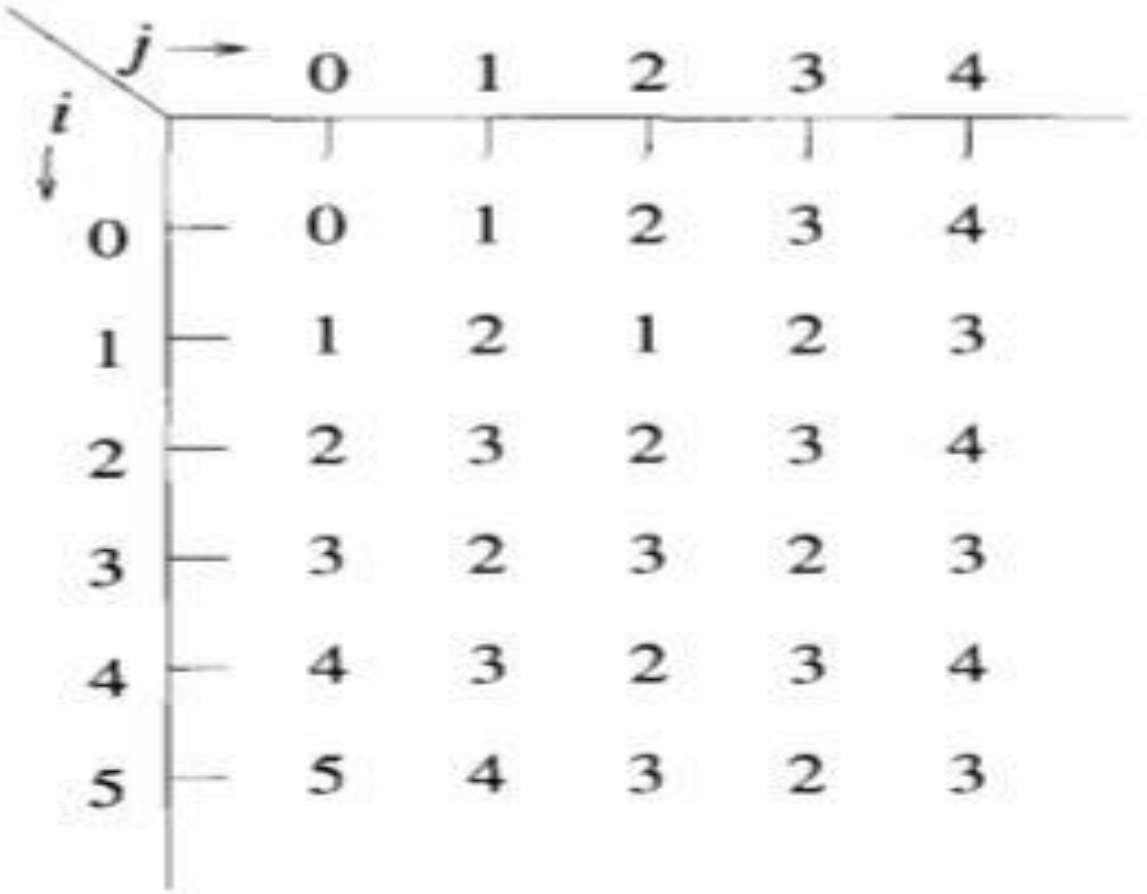
Example:

$$\begin{aligned} cost(1,1) &= \min\{cost(0,1)+D(x_1), cost(0,0)+C(x_1,y_1), cost(1,0)+I(y_1)\} \\ &= \min\{2,2,2\} = 2 \end{aligned}$$

Next is computed $cost(1,2)$.

$$\begin{aligned} cost(1,2) &= \min\{cost(0,2)+D(x_1), cost(0,1)+C(x_1,y_2), cost(1,1)+I(y_2)\} \\ &= \min\{3,1,3\} = 1 \end{aligned}$$

Table of Values



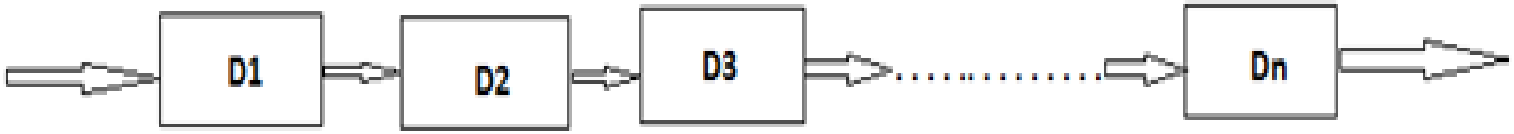
i \ j	0	1	2	3	4
0	0	1	2	3	4
1	1	2	1	2	3
2	2	3	2	3	4
3	3	2	3	2	3
4	4	3	2	3	4
5	5	4	3	2	3

Time Complexity

- The time-complexity of the algorithm is O(|s1|*|s2|), i.e. $O(n^2)$ if the lengths of both strings is about `n'.

Reliability Design

- In **reliability design**, the problem is to design a system that is composed of several devices connected in series.



- If we imagine that r1 is the reliability of the device.
- Then the reliability of the function can be given by **πr1**.
- So, if we duplicate the devices at each stage then the reliability of the system can be increased.

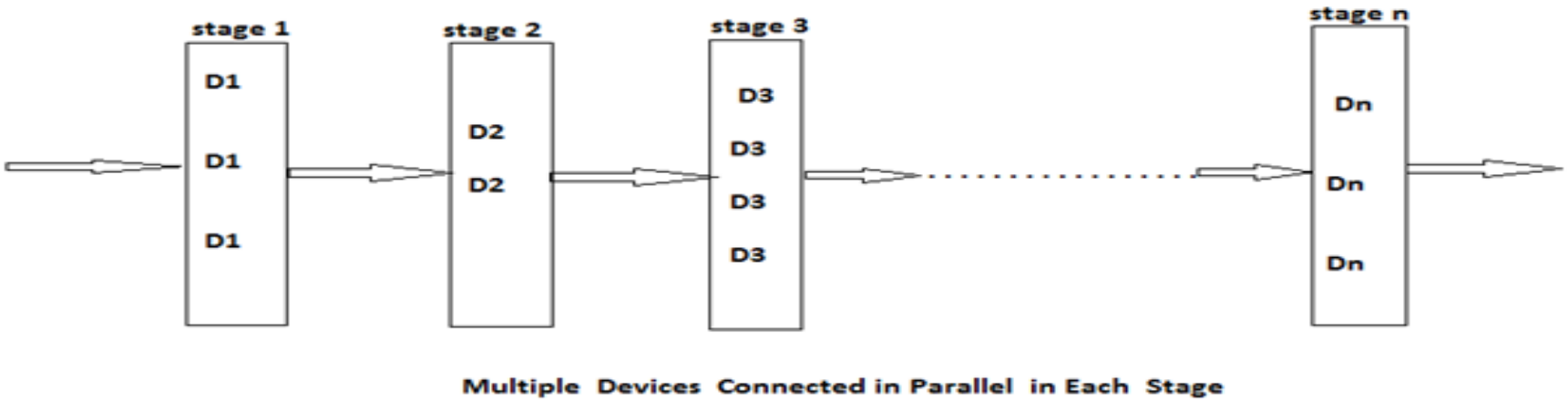
Reliability Design

- It can be said that multiple copies of the same device type are connected in parallel through the use of switching circuits.
- Here, switching circuit determines which devices in any given group are functioning properly.
- Then they make use of such devices at each stage, that result is increase in reliability at each stage.
- If at each stage, there are **mi** similar types of devices **Di**, then the probability that all **mi** have a malfunction is **(1 - ri)^mi**, which is very less.

Reliability Design

- If at each stage, there are **mi** similar types of devices **Di**, then the probability that all **mi** have a malfunction is **(1 - ri)^mi**, which is very less.
- And the reliability of the stage **1** becomes **(1 – (1 - ri) ^mi)**.
- Thus, if **ri = 0.99** and **mi = 2**, then the stage reliability becomes **0.9999** which is almost equal to **1**.
- Which is much better than that of the previous case or we can say the reliability is little less than **1 - (1 - ri) ^mi** because of less reliability of switching circuits.

Reliability Design



Multiple Devices Connected in Parallel in Each Stage

In reliability design, we try to use device duplication to maximize reliability. But this maximization should be considered along with the cost.

Reliability Design

- Let **c** is the maximum allowable cost and **ci** be the cost of each unit of device **i**. Then the maximization problem can be given as follows:

$$\text{maximize } \Pi_{1 \le i \le n} \ \phi_i(m_i)$$

$$\text{subject to } \sum_{1 \le i \le n} c_i m_i \le c$$

$$m_i \ge 1 \text{ and integer, } 1 \le i \le n$$

- The upper bound U_i follows from the observation that $m_j >= 1$.

$$u_i = \left\lfloor (c + c_i - \sum_{1}^{n} c_j)/c_i \right\rfloor$$

Reliability Design

- The principal of optimality holds and

$$f_n(c) = \max_{1 \le m_n \le u_n} \{\phi_n(m_n) f_{n-1}(c - c_n m_n)\}$$

- For any fi(x), i >1,this equation generalizes

$$f_i(x) = \max_{1 \le m_i \le u_i} \{\phi_i(m_i) f_{i-1}(x - c_i m_i)\}$$

Problem

- We are to design a three stage system with device types D1,D2and D3. The costs are$30,$15,and $20 respectively. The cost of the system is to be no more than $105.The reliability of each device type is 0.9,0.8 and 0.5 respectively.

Procedure

- We assume that if stage i has m_i devices of type i in parallel, then $Øi(m_i)= 1-(1-r_i)^{mi}$. In terms of the notation used earlier, c1 = 30,C2 = 15,C3= 20,c = 105, r1 = .9,r2 = .8,r3 = .5,u1=2,U2 = 3, and u3 = 3.

Solution

We use S^i to represent the set of all undominated tuples (f, x) that may result from the various decision sequences for $m_1, m_2, \ldots, m_i$. Hence, $f(x) = f_i(x)$. Beginning with $S^0 = \{(1, 0)\}$, we can obtain each S^i from S^{i-1} by trying out all possible values for m_i and combining the resulting tuples together. Using S_j^i to represent all tuples obtainable from S^{i-1} by choosing $m_i = j$, we obtain $S_1^1 = \{(.9, 30)\}$ and $S_2^1 = \{(.9, 30), (.99, 60)\}$. The set

Solution

$S_1^2 = \{(.72, 45), (.792, 75)\}$; $S_2^2 = \{(.864, 60)\}$. Note that the tuple (.9504, 90) which comes from (.99, 60) has been eliminated from S_2^2 as this leaves only \$10. This is not enough to allow $m_3 = 1$. The set $S_3^2 = \{(.8928, 75)\}$. Combining, we get $S^2 = \{(.72, 45), (.864, 60), (.8928, 75)\}$ as the tuple (.792, 75) is dominated by (.864, 60). The set $S_1^3 = \{(.36, 65), (.432, 80), (.4464, 95)\}$, $S_2^3 = \{(.54, 85), (.648, 100)\}$, and $S_3^3 = \{(.63, 105)\}$. Combining, we get $S^3 = \{(.36, 65), (.432, 80), (.54, 85), (.648, 100)\}$.

The best design has a reliability of .648 and a cost of 100. Tracing back through the S^i's, we determine that $m_1 = 1, m_2 = 2$, and $m_3 = 2$. □

0/1 KNAPSACK

$$S_1^i = \{(P,W) | (P - p_i, W - w_i) \in S^i\}$$

0/1 KNAPSACK Problem:

- Consider the knapsack instance n = 3,(w_1,w_2,w_3) = (2,3,4), (p_1,p_2,p_3) = (1,2,5), and m = 6.

Solution

$$
\begin{aligned}
S^0 &= \{(0,0)\}; S_1^0 = \{(1,2)\} \\
S^1 &= \{(0,0),(1,2)\}; S_1^1 = \{(2,3),(3,5)\} \\
S^2 &= \{(0,0),(1,2),(2,3),(3,5)\}; S_1^2 = \{(5,4),(6,6),(7,7),(8,9)\} \\
S^3 &= \{(0,0),(1,2),(2,3),(5,4),(6,6),(7,7),(8,9)\}
\end{aligned}
$$

Note that the pair (3, 5) has been eliminated from S_3 as a result of the purging rule stated above.