Notes
on
Design and Analysis
of
Algorithm

By

Dr. Satyasundara Mahapatra

(Module –IV)

Dynamic Programming with Examples Such as Knapsack. All Pair Shortest Paths – Warshal's and Floyd's Algorithms, Resource Allocation Problem. Backtracking, Branch and Bound with Examples Such as Travelling Salesman Problem, Graph Coloring, n-Queen Problem, Hamiltonian Cycles and Sum of Subsets.

Dynamic Programming (A powerful technique for Designing Algorithms.)

→ Not a specific algorithm, but a technique like divide-and-conquer.

→ Developed back in the day when "programming" meant "tabular method" (like linear Programming) (Doesn't really refer to computer programming)

→ Used for optimization problems.

→ find a solution with the optimum value.

→ for Minimization and Maximization.

It is in four-step method

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Dynamic programming used in different areas.

They are.

1. Operation Research.
2. Signal Processing.
3. Computational Biology.
4. Geometry
5. etc.

Now discuss the time complexity of recursive Fibonacci sequences. and then optimize with Dynamic Programming technique.

First we will analyze the time complexity of recursive implementation of fibonacci sequence. As we know that the simple recursive implementation of Fibonacci sequence is lot costlier than iterative implementation. (i.e using of loop)

As we know fibonacci sequence is a sequence in which the first two elements are zero and one, and all other elements are sum of previous two elements. A recursive program to find an element in the sequence goes something like the following.

0 1 1 2 3 5 8 13 . . . .

```
fibonacci (n)
{
    if n <= 1          ① unit
        return n;
    else
        return fibonacci(n-1) + fibonacci(n-2);
}
```

① unit ① unit. ① unit.

→ Let's say the time taken to calculate fibonacci of n is T(n).

→ when we try to analyze time complexity of program we make an assumption that each simple operation takes one unit of time.

→ So if we call this method fibonacci of n for n > 1 then

1. we perform a comparison ① which is one unit of cost.
2. because n is > 1, so it goes to the else

condition control. of the program and here we make two recursive calls where we pass arguments (n-1) and (n-2). So we make two subtractions, which take one unit time each. ② and then one unit of cost for addition.

So for n > 1 there are four simple operations, (i.e two simple subtractions, one addition and one comparision now time taken to calculate fibonacci of n can be written as. follows.

$$T(N) = T(N-1) + T(N-2) + 4$$

If n <= 1 which is the case for T(0) and T(1), we only have one simple question which is comparision. So here the time taken is one unit. So

$$T(0) = T(1) = 1.$$

Now Let us try to reduce T in terms of these known values (i.e T(0) and T(1)). But before that we will do an approximation.

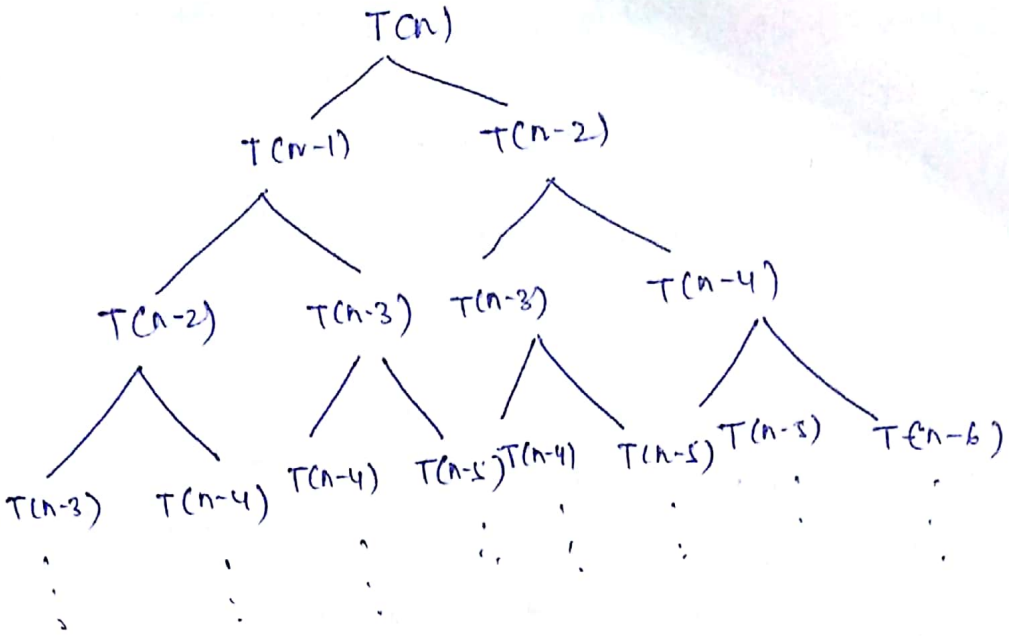
"Let's say the time taken to calculate fibonacci of (n-1) is almost equal to calculate the fibonacci of (n-2). i.e

$$T(n-1) \simeq T(n-2).$$

But in reality the time taken to calculate fibonacci(n-1) > fibonacci(n-2).

So we are trying to calculate the lower bound of fibonacci series.

Time taken T(n)



So now

$$T(n) = 2T(n-2) + c \qquad \text{where } c = 4$$

Let's try to reduce the expression.

$$\Rightarrow \quad = 2\{2T(n-4)\} + 2c + c$$

$$= 4T(n-4) + 3c.$$

$$= 4\{2T(n-6)\} + 4c + 3c$$

$$= 8T(n-6) + 7c$$

$$= 8\{2T(n-8)\} + 8c + 7c$$

$$= 16T(n-8) + 15c.$$

$$= 2^4 T(n-2\cdot 4) + (2^4 - 1)c$$

$$\vdots$$

$$\text{for } k^{th} \text{ Term} = 2^k T(n-2k) + (2^k - 1)c$$

now if we want to read these interms of T(0) which is already known to us then

$$n - 2k = 0. \qquad k = n/2$$

$$\therefore T(n) = 2^{n/2} T(0) + (2^{n/2} - 1)c$$

$$= 2^{n/2} \cdot 1 + 2^{n/2} c - c$$

$$= 2^{n/2}(c+1) - c$$

So in simple terms we can say here that

$T(n) \propto 2^{n/2}$ ← Lower bound, time taken.

Now let's try another approximation for upper bound calculation.

Let's say the time to calculate

fibonacci $(n-2) \simeq$ fibonacci $(n-1)$

This reduces the expression as.

$$T(n) = 2T(n-1) + 4c \quad (\because c = 4)$$

On reading $T(n-2) < T(n-1)$, so this expression is giving an upper bound this time. And by reducing the particular expression we get.

$$= 2\{2T(n-2)\} + 2c + c.$$

$$= 4T(n-2) + 3c$$

$$= 4\{2T(n-3)\} + 4c + 3c$$

$$= 8T(n-3) + 7c.$$

$$= 16T(n-4) + 15c$$

$$= 2^4 T(n-4) + (2^4 - 1)c$$

$$\vdots$$

$$\text{for } k^{th} \text{ term} = 2^k T(n-k) + (2^k - 1)c$$

Now read the above equation in terms of $T(0)$ which is already known to us then

$$n - k = 0 \Rightarrow k = n.$$

Hence the equation for upperbound can be written as

$$T(n) = 2^n \cdot T(0) + (2^n - 1)\, c.$$

$$= 2^n \cdot 1 + 2^n c - c$$

$$= 2^n (c+1) - c.$$

So in simplest & term in this particular case we can say that.

$T(n) \propto 2^n$ ← upper bound time taken.

It shows that the time taken by this program grows exponentially with the input. So for n input the ~~fibonacci(n)~~

$$2^{n/2} < \text{fibonacci}(n) < 2^n$$

(i.e for $n = 100$ the above program take a year to give the result..) ~~So the above algorithm~~

The above algorithm (i.e fibonacci(n)) time complexity can be represented as follows.

~~worst case~~ = $O(2^n)$ and $\Omega(2^{n/2})$.

~~Average case~~ - ~~O(n)~~.

Hence $\Theta(2^n)$

Dynamic Programming is an algorithm paradigm that solve a given complex problem by breaking it into subproblems and store the results ~~of~~ of subproblems in table to avoid computing the same results again and again. In general their are basically

two main properties of a problem that suggests that the given problem can be solved by using Dynamic programming. These properties are.

1. Overlapping Subproblems property
2. Optimal Substructure property.

1. Overlapping Subproblems. (based on Table)

The tabulated program for a given problem (i.e fibonacci) builds a table in bottom up fashion and return the last entry from table. For fibonacci number, we first calculate fibonacci(0) then fibonacci(1), then fibonacci(2), then fibonacci(3) and so on. So literally we are building the solutions bottom up fashion. following is the tabulated version of n^{th} fibonacci number.

```
int fibonacci(int n)
{
    int f[n+1], i;
    f[0] = 0;
    f[1] = 1;
    for(i = 2; i <= n; i++)
        f[i] = f[i-1] + f[i-2];
    return f[n];
}
int main()
{
    int n = 19;
    printf("Fibonacci number is %d", fibonacci(n));
    return 0;
}
```

The complexity of fibonacci number by dynamic approach is $O(N)$. which is much much less than $O(2^N)$. generated by recursive method.

0/1 Knapsack Problem by using Dynamic Programming

Knapsack problem state that:

Given a set of items, each with a mass and a value, determine the number of each item to include in a collection so that the total weight the total weight is less than or equal to a given limit and the total value is as large as possible.

In case of 0-1 knapsack, items cannot be broken which means the thief should take the item 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 x_i, can be either 0 or 1, where other constraints remain same. Lets solve it with Four step Method.

Step-1 [Characterize the structure of the optimal solution] Let there are n objects, their profit values are $\langle v_1, v_2, \ldots, v_N \rangle$, weights are $\langle w_1, w_2, \ldots, w_N \rangle$. The maximum capacity of knapsack is M.

The 0/1 knapsack problem can stated as

$$\text{Max} \sum_{i=1}^{n} v_i x_i$$ i.e Sum of the profit should be Maximize.

$$\text{Subject to} \sum_{i=1}^{n} w_i x_i \leq M$$ i.e Sum of the weights should be Less than or equal to Capacity of the bag.

where $x_i \in \{0, 1\}$

Step-2

Recursively define the value of optimal solution.

Let $c[i,j]$ is a 2D array, where

$i = 0, 1, 2, \ldots, n$ i.e no. of objects.

$j = 0, 1, 2, \ldots, M$ i.e Maximum weight of Knapsack.

$$c[i,j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0. \\ c[i-1,j] & \text{if } w_i > 0 \\ \max(c[i-1,j], c[i-1,j-w_i] + v_i) & \text{if } i>0 \text{ \& } j \geq w_i \end{cases}$$

Step-3 Compute optimal solution for 0/1 knapsack problem.

0/1 Knapsack (v, w, n, M)

1. for $j = 0$ to M
2. $c[0,j] = 0$
3. $keep[0,j] = 0$.
4. for $i = 1$ to n
5. for $j = 0$ to M
6. if $(j \geq w[i])$ && $(c[i-1, j-w[i]] + v[i] > c[i-1,j])$
7. then $c[i,j] = (c[i-1, j-w[i]] + v[i])$
8. $keep[i,j] = 1$.

9. else

$$c[i, j] = c[i-1, j]$$

10. $keep[i, j] = 0.$

11. return $c[n, M]$

Step-4 Construct / Point the optimal solution of 0/1 knapsack problem.

0/1 Knapsack Solution (n, M)

1. $K = M$
2. for $i = n$ down to 1
3. if $(keep[i, K] == 1)$
4. then print i
5. $K = K - w[i]$.

Lets solve a problem with the help of 0/1 Knapsack Problem.

Example:

Find an optimal solution for given instance of 0/1 knapsack problem.

$n = 3$ $M = 6$

	1	2	3
v	1	2	4
w	2	3	3

$x_i = \{1, 1, 1\}$ or $\{1, 1, 0\}$ or $\{0, 1, 0\}$ either we carry the object or not.

C array

P/W	W	i \ w	0	1	2	3	4	5	6
		0	0	0	0	0	0	0	0
1	2	1	0	0	1	10	01	10	01
2	3	2	0	0	1	2	2	3	3
4	3	3	0	0	0	4	48	5	6

Maximum weight of knapsack = 6

Keep. array:

i \ w	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1
2	0	0	0	1	1	1	1
3	0	0	0	1	1	1	1

So Maximum Profit = 6,

x_1	x_2	x_3
0	1	1

K = 6

If keep(3,6) == 1 Then.

Print - 3 , K = K - w[i] = 6 - 3 = 3.

If (2, 3) == 1 Then.

Print - 2 , K = K - w[i] = 3 - 3 = 0.

If (1, 0) == 1 false.

Loop terminates.

So we get maximum profit with the help of

Approach for solving Dynamic Programming.

→ Dynamic Programing is useful for solving optimization problem. (i.e such problems demand maximum result)

→ Dynamic Programming says that a problem is solving with the help of sequence of decisions.

→ find all possible solutions and pick up the best.

So how many possible solutions.

If $n = 4$ then x_i starts from

0 0 0 0

⋮ to

1 1 1 1

So this is 2^4 no.s of Possible solution. So if we take n no. of objects then the no. of possible solution is 2^n. So the time complexity is $O(2^n)$. which is too much time consuming. So Dynamic Programming provide a easy method for solving such problems. and do the same thing indirectly by using

tabular Method. So we can save maximum time. Let's see how dynamic Programming method solve a problem.

Example:

$n = 4$ $\quad M = 8$

$P/V = \{1, 2, 5, 6\}$.

$w = \{2, 3, 4, 5\}$

	1	2	3	4
P/V	1	2	5	6
w	2	3	4	5

C - array

P/V	w	i	0	1	2	3	4	5	6	7	8
		0	0	0	0	0	0	0	0	0	0
1	2	1	0	0	1	1	1	1	1	1	1
2	3	2	0	0	1	2	2	3	3	3	3
5	4	3	0	0	1	2	5	5	6	7	7
6	5	4	0	0	1	2	5	6	6	7	8

w →

→ Maximum capacity of Knapsack

Where boxes are filled for Profit.

$$C[i,j] = \max[C[i-1,j], C[i-1, j-w[i]] + V[i]]$$

$$C[4,1] = \max[V[3,1], V[3,1-5] + 6]$$

$$= \max[0, V[3,-4] + 6]$$

no such index is available.

So the value is 0. (Previous value)

Hence upto 4th column we get all previous value.

$$C[4,5] = \max[V[3,5], V[3,5-5] + 6]$$

$$= \max[5, 6]$$

$$= 6$$

p/w		0	1	2	3	4	5	6	7	8
	0	0	0	0	0	0	0	0	0	0
1	2	1	0	0	1	1	1	1	1	1
2	3	2	0	0	1	2	2	3	3	3
5	4	3	0	0	1	2	5	5	6	7
6	5	4	0	0	1	2	5	6	6	8

First Maximum value.

x_1	x_2	x_3	x_4	Profit	Weight
		0	1	8 - 6 = 2	8 - 5 = 3

→ The maximum profit is 8 and it is generated in last row. i.e 4th row and in previous row there is no eight. so. it gives a solution that 4th object is included. So $x_4 = 1$

→ Then calculate remaining profit left. or the weight left.

→ now check 2 in previous row, i.e 3rd row, whether available or not. if yes.

Then check in 2nd row whether #2 is available. if available then don't select 3rd row. So $x_3 = 0$.

→ Continue this process until the bag is full or profit = 0.

$$c[4,6] = \max(c[3,6], c[3,6-5]+6)$$

$$= \max(c[3,6], c[3,1]+6)$$

$$= \max(6, 0+6)$$

$$= \max(6,6) = 6.$$

$$c[4,7] = \max(c[3,7], c[3,7-5]+6)$$

$$= \max(c[3,7], c[3,2]+6)$$

$$= \max(7, 1+6)$$

$$= 7.$$

$$c[4,8] = \max(c[3,8], c[3,8-5]+6)$$

$$= \max(7, c[3,3]+6)$$

$$= \max(7, 2+6)$$

$$= 8.$$

now find the value for

x_4 x_2 x_3 x_4

for finding the x_i value we have to take sequence of decision. i.e which object should be include or which should not be included.

So Let's start with maximum Profit which is available at last cell.

Floyd - Warshall algorithm.

To solve the all-pair shortest path problem on a directed graph $G = \langle V, E \rangle$, a dynamic formulation algorithm known as Floyd - Washall algorithm is used. This algorithm runs in $\Theta(V^3)$ times. In this case negative-weight edges may be present, but we assume that there are no negative-weight cycles.

Step. 1. Characterize the structure of an optimal solution.

→ The Floyd - Warshall algorithm considers the intermediate vertices of a shortest path, where an intermediate vertex of a simple path $P = \langle v_1, v_2, \ldots, v_r \rangle$ is any vertex of p other than v_1 or v_e. i.e any vertex in the set $\{v_2, v_3, \ldots, v_{l-1}\}$.

Define d_{ij}^{k} to be the weight of shortest path p from i to j for which all intermediate vertices are in the set $\{1, 2, \ldots k\}$

→ Depending on wheather or not k is an intermediate vertex on p, we have two cases.

Case-1:

If the shortest path p (from i to j going through vertices with indices $\leq k$) does not go through vertex k then

$$d_{ij}^{k} = d_{ij}^{k-1}$$

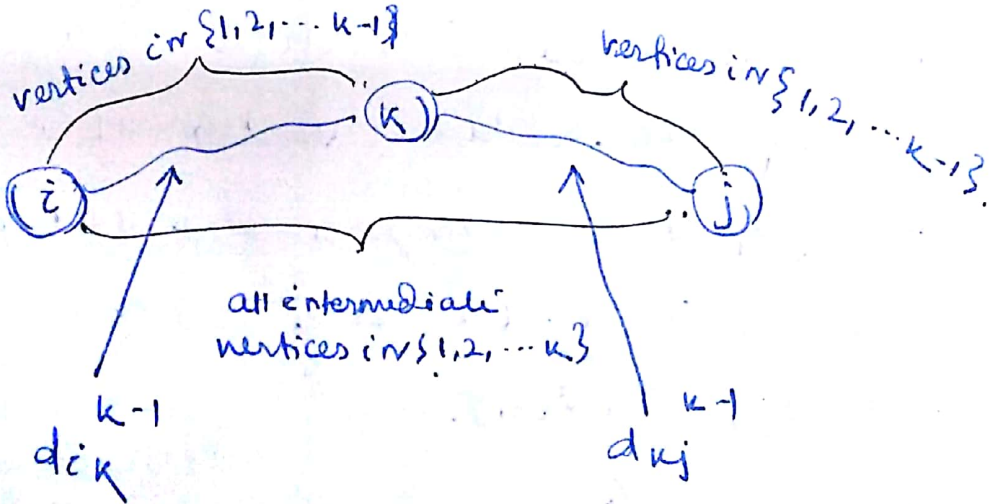
Case-2:

If the shortest path p goes through vertex k, then:

$$d_{ij}^{(k)} = d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$$

Therefore $d_{ij}^{k} = \min\left(d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}\right)$

i.e.



Step-2. Recursively define the value of an optimal solution

→ Boundary condition: for $k = 0$, a path from vertex i to j with no intermediate vertex numbered higher than 0 has no intermediate vertices, at all, hence $d_{ij}^{(0)} = w_{ij}$

→ Recursive formulation:

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

Because for any path, all intermediate vertices are in the set $\{1, 2, \ldots, n\}$, the matrix $D^n = d_{ij}^{(n)}$ gives the final answer:

$$d_{ij}^{n} = \sigma(i, j) \text{ for all } i, j \in V.$$

Step-III Compute the shortest path weights bottom up.

Floyd-Warshall (W)

1. $n = W.\text{rows}$.
2. $D^0 = W$.
3. for $k = 1$ to n
4. let $D^k = (d_{ij}^k)$ be a new $n \times n$ 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.

This procedure return the matrix D^n of the shortest path weights. and takes $\Theta(n^3)$ times for execution.

Step-IV Constructing the shortest path.

→ need to compute predecessor matrix Π from the weights matrix D.

→ Compute Π at the same time with D.

→ Compute a sequence $\Pi^{(0)}, \Pi^{(1)}, \ldots, \Pi^{(n)}$, where $\pi_{ij}^{(k)}$ is defined as the predecessor of vertex j on a shortest path from vertex i with all intermediate vertices in $\{1, 2, \ldots k\}$

→ $\Pi = \Pi^{(n)}$.

Recursive formulation of $\pi_{ij}^{(k)}$

→ when $k = 0$, a shortest path from i to j has no intermediate vertices. at all.

Hence.

$$\pi_{ij}^{(k)} = \begin{cases} \text{NULL} & \text{if } i = j \text{ or } w_{ij} = \infty. \\ i & \text{if } i \neq j \; w_{ij} < \infty. \end{cases}$$

→ when $k \geq 1$

→ If we take the path $i \rightsquigarrow k \rightsquigarrow j$, then π_{ij}^{k} is the same as the predecessor of j on the shortest path from k with intermediate vertices in $1, 2, \ldots, k-1$

$$\pi_{ij}^{k} = \pi_{kj} \quad \text{if } d_{ij}^{k-1} > d_{ik}^{k-1} + d_{kj}^{k-1}$$

→ when $k \geq 1$

→ Otherwise, Π_{ij}^{k} is the same as predecessor of j on the shortest path from i with intermediate vertices in $1, 2, \ldots, k-1$,

$$\Pi_{ij}^{k} = \Pi_{ij}^{k-1} \text{ if } d_{ij}^{k-1} \leq d_{ik}^{k-1} + d_{kj}^{k-1}$$

→ Putting these two cases together:

$$\Pi_{ij}^{k} = \begin{cases} \Pi_{ij}^{k-1} & \text{if } d_{ij}^{k-1} \leq d_{ik}^{k-1} + d_{kj}^{k-1} \\ \Pi_{kj}^{k-1} & \text{if } d_{ij} > d_{ik}^{k-1} + d_{kj}^{k-1} \end{cases}$$

The Floyd-Warshall Algorithm:

$D^0 = W$

Init-Predecessors (Π^0)

for $k = 1$ to n

  for $i = 1$ to n

    for $j = 1$ to n,

      if ($d_{ij}^{k-1} \leq d_{ik}^{k-1} + d_{kj}^{k-1}$) then

        $d_{ij}^{k} = d_{ij}^{k-1}$

        $\Pi_{ij}^{k} = \Pi_{ij}^{k-1}$

      else

        $d_{ij}^{k} = d_{ik}^{k-1} + d_{kj}^{k-1}$

        $\Pi_{ij}^{k} = \Pi_{kj}^{k-1}$

return D^n.

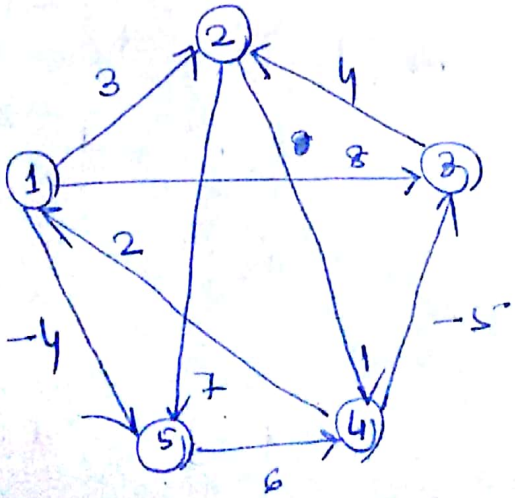
Printing shortest path with Π.

The predecessor matrix is $\Pi = \Pi^{(N)}$. The following recursive procedure prints the shortest path between vertices i and j using Π.

```
Print-all-Pairs-shortest-Path (Π, i, j)
    if i = j then
        Print i
    else
        if Πij = NULL then
            Print "no path from 'i' to 'j'"
        else
            Print-all-Pairs-shortest-Path (Π, i, Πij)
            Print j
```

Example:

Find the shortest path of the following graph. using Floyd-Warshall's Algo.



$$D^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}\begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \end{array} \qquad \Pi^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}\begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} N & 1 & 1 & N & 1 \\ N & N & N & 2 & 2 \\ N & 3 & N & N & N \\ 4 & N & 4 & N & N \\ N & N & N & 5 & N \end{pmatrix} \end{array}$$

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

$$d_{42}^1 = \min(d_{42}^0, d_{41}^0 + d_{12}^0).$$

$$= \min(\infty, 2+3) = 5$$

$$d_{45}^1 = \min(d_{45}^0, d_{41}^0 + d_{15}^0)$$

$$= \min(\infty, 2+(-4)) = -2.$$

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

$$d_{14}^2 = \min(d_{14}^1, d_{12}^1 + d_{24}^1)$$

$$= \min(\infty, 3+1) = 4$$

$$d_{34}^2 = \min(d_{34}^1, d_{32}^1 + d_{24}^1)$$

$$= \min(\infty, 4+1) = 5$$

$$d_{35}^2 = \min(d_{35}^1, d_{32}^1 + d_{25}^1)$$

$$= \min(\infty, 4+7) = 11.$$

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

$$\Pi^3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}\begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} N & 1 & 1 & 2 & 1 \\ N & N & N & 2 & 2 \\ N & 3 & N & 2 & 2 \\ 4 & 3 & 4 & N & 1 \\ N & N & N & 5 & N \end{pmatrix} \end{array}$$

$$d_{42}^{\otimes 3} = \min\left(d_{42}^{2}, d_{43}^{2} + d_{32}^{2}\right)$$

$$= \min(5, \overset{-5}{\cancel{0}} + 4) = -1$$

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

$$\Pi^4 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}\begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} N & 1 & 4 & 2 & 1 \\ 4 & N & 4 & 2 & 1 \\ 4 & 3 & N & 2 & 1 \\ 4 & 3 & 4 & N & 1 \\ 4 & 3 & 4 & 5 & N \end{pmatrix} \end{array}$$

$$d_{13}^4 = \min(d_{13}^3, d_{14}^3 + d_{43}^3)$$

$$= \min(8, 4 + (-5)) = -1$$

$$d_{21}^4 = \min(d_{21}^3, d_{24}^3 + d_{41}^3)$$

$$= \min(\infty, 1 + 2) = 3.$$

$$d_{23}^4 = \min(d_{23}^3, d_{24}^3 + d_{43}^3)$$

$$= \min(\infty, 1 + (-5) = -4$$

$$d_{31}^4 = \min(d_{31}^3, d_{34}^3 + d_{41}^3)$$

$$= \min(\infty, 5 + 2) = 7.$$

$d_{51}^{4} = \min(d_{51}^{3}, d_{54}^{3} + d_{41}^{3})$

$= \min(\infty, 6 + 2) = 8$

$d_{52}^{4} = \min(d_{52}^{3}, d_{54}^{3} + d_{42}^{3})$

$= \min(\infty, 6 + (-1)) = 5$

$D^5 =$

	1	2	3	4	5
1	0	1	-3	2	-4
2	3	0	-4	1	-1
3	7	4	0	5	3
4	2	-1	-5	0	-2
5	8	5	1	6	0

$\Pi^5 =$

	1	2	3	4	5
1	N	3	4	5	1
2	4	N	4	2	1
3	4	3	N	2	1
4	4	3	4	N	1
5	4	3	4	5	N

$d_{12}^{5} = \min(d_{12}^{4}, d_{15}^{4} + d_{52}^{4})$

$= \min(3, -4 + 5) = 1$

$d_{13}^{5} = \min(d_{13}^{4}, d_{15}^{4} + d_{53}^{4})$

$= \min(-1, -4 + 1) = -3.$

$d_{14}^{5} = \min(d_{14}^{4}, d_{15}^{4} + d_{54}^{4})$

$= \min(4, -4 + 6) = 2.$

for printing shortest path.

Path from 1 to 2 by using Print-all-pairs-shortest-path (Π, i, j)

$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 1$

$\leftarrow \leftarrow \leftarrow \leftarrow \leftarrow$

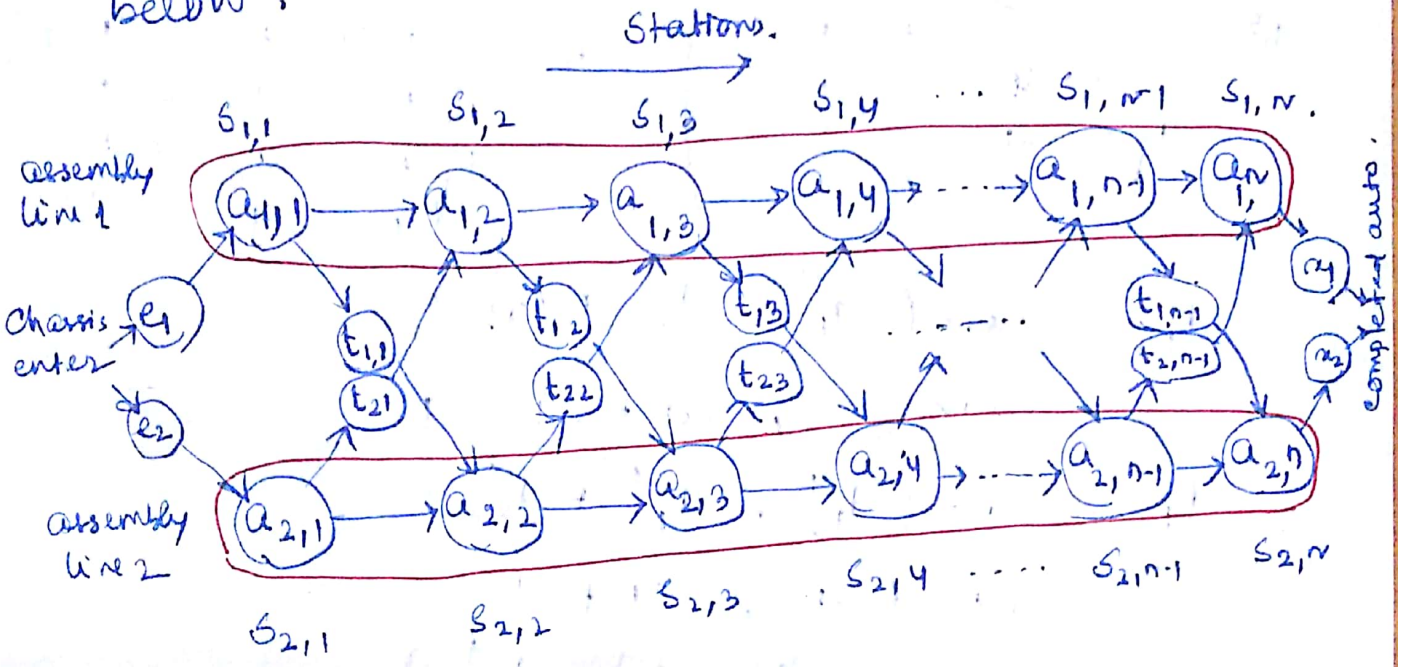
i.e 1 to 5 to 4 to 3 to 2 1. i.e 1

Path from 1 to 3

$1 \rightarrow 5 \rightarrow 4 \rightarrow 3.$ i.e = -3.

Assembly line scheduling Problem:

Assembly line scheduling Problem is a situation for manufacturing problem. The Colonel Motors Corporation produces automobiles in a factory that has two assembly lines as shown below:



Explanation:

→ there are n "stations" upon which some assembly is done.

→ there are two "lines" (think conveyor belts)

→ each station takes time s_{ij} (where i ← line number, j ← station number)

→ stations with the same j number do the same job (product is the same), but may take differing time to do it.

→ after completing a station i, j, there is choice, Either

1. Stay on the same line and progress to the next station (this has zero cost.) or
2. move to the other line and progress to the next station (this has cost $t_{i,j+1}$)

Fastest Time:

→ Normally, both lines are fully utilised and there is no swapping between lines but,

→ sometimes a rush order comes in and the goal is to find the fastest way through the factory for single item.

What is the fastest time through the factory?

We could use a brute force approach. If there are two lines, then we are essentially choosing which of the stations for line 1 to use (and therefore which of the line 2 station to use). We can think of the chosen line 1 stations as being "on" and the not chosen one as being "off". That looks like a binary number with n digits, and there are 2^n such numbers. Therefore, brute force is $O(2^n)$.

(A) From the figure it was observed that the automobile factory has two assembly line: -

→ Each line has n stations:

$S_{1,1}, S_{1,2}, \ldots, S_{1,n}$ and

$S_{2,1}, S_{2,2}, \ldots, S_{2,n}$

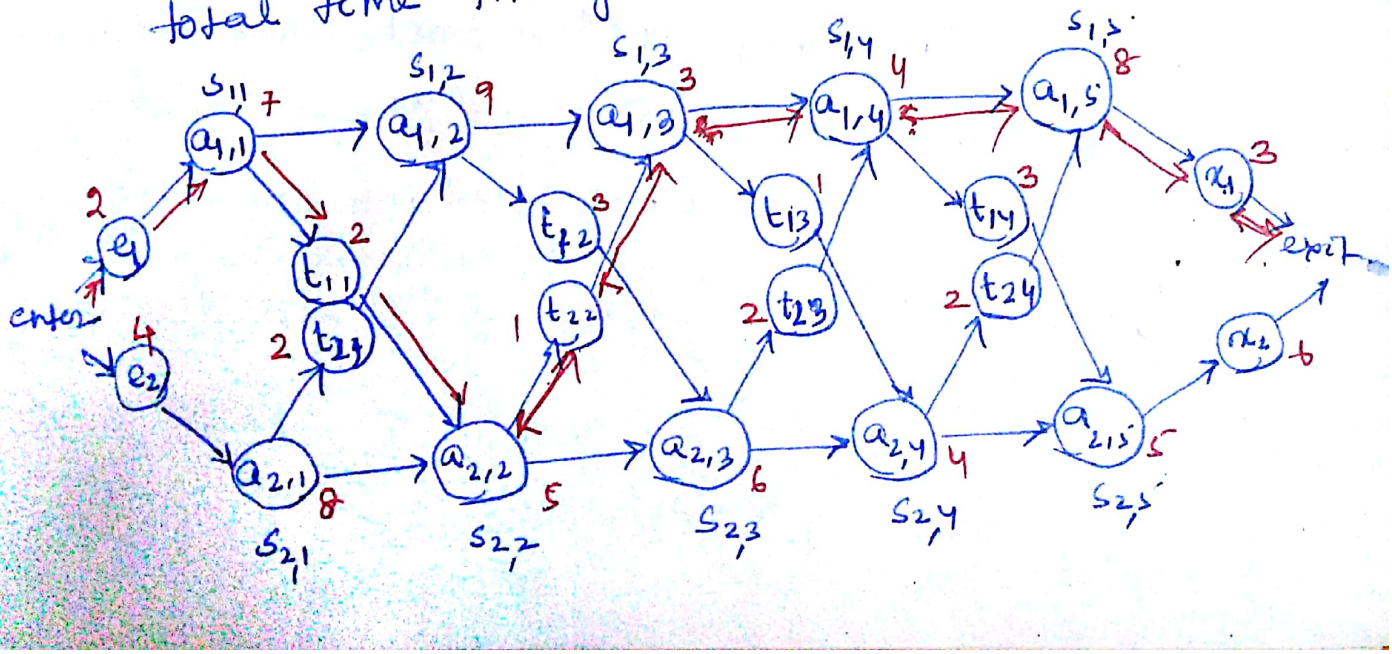
→ Corresponding stations $S_{1,j}$ and $S_{2,j}$ perform the same function but can take different amount of time $a_{1,j}$ and $a_{2,j}$

→ Entry times are: e_1 & e_2;
exit times are: x_1 and x_2

(A) → After going through a station, can either
→ stay on same line at no cost or
→ transfer to other line:
cost after S_{ij} is t_{ij} $j = 1 \ldots n-1$

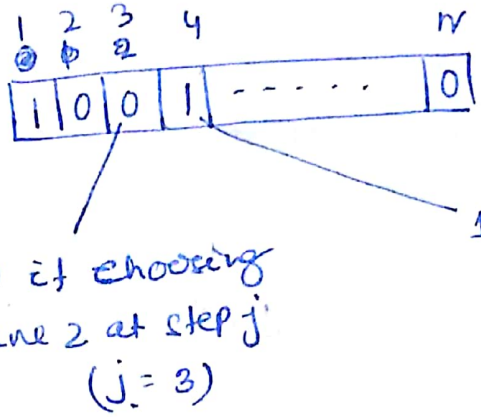
Problem:

What stations should be chosen from line 1 and which from line 2 in order to minimize the total time through the factory for one car?



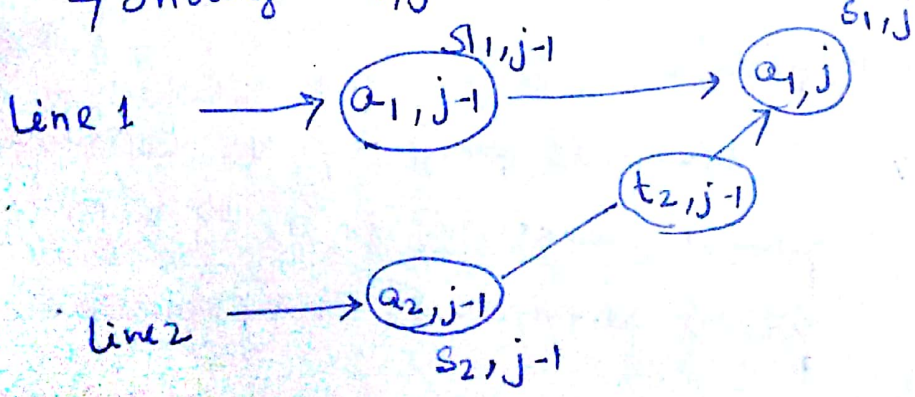
→ Brute force
  → Enumerate all possibilities of selecting stations
  → Compute how long it takes in each case and choose the best one.

Soln.



— Hence there are 2^n possible ways to choose stations
— Infeasible when n is large.

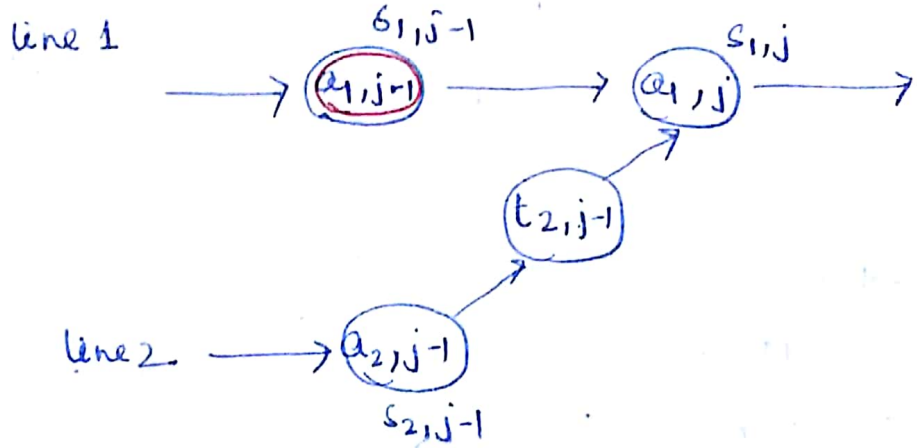
1. Characterize the structure of the optimal solution:-
  → How do we compute the minimum time of going through a station?
  → Let's consider all possible ways to get from the starting point through station $S_{1,j}$:
    → Through $S_{1,j-1}$ then directly through $S_{1,j}$
    → Through $S_{2,j-1}$ then transfer over to $S_{1,j}$



→ Suppose that the fastest way through $S_{1,j}$ is through $S_{1,j-1}$

→ We must have taken a fastest way from entry through $S_{1,j-1}$

→ If there were a faster way through $S_{1,j-1}$ we would used it instead.



Similarly for $S_{2,j-1}$ ✓

Generalization: an optimal solution to the problem "find the fastest way through $S_{1,j}$" contains within it an optimal solution to subproblems: "find the fastest way through $S_{1,j-1}$ or $S_{2,j-1}$".

→ This is reffered as the optimal substructure property.

→ We use this property to construct an optimal solution to a problem from optimal solutions to subproblems.

Step-2 A Recursive Solution:-

→ Define the value of an optimal solution in terms of the optimal solution to subproblems

Definition:

- f^* : the fastest time to get through the entire factory.
- $f_i[j]$: the fastest time to get from the station point through station $S_{i,j}$

$$f^* = \min\left(f_1[n] + x_1 \,,\, f_2[n] + x_2\right)$$

Base Case:

$j = 1$, $i = 1, 2$ (getting through station 1)

$$f_1[1] = e_1 + a_{1,1}$$

$$f_2[1] = e_2 + a_{2,1}$$

General Case:

$j = 2, 3, \ldots, n$ & $i = 1, 2$

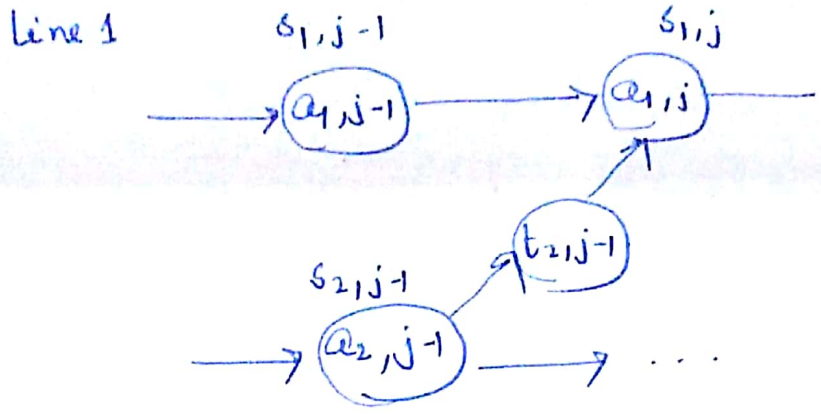
Fastest way through $S_{1,j}$ is either

→ the way through $S_{1,j-1}$, then directly through $S_{1,j}$ or

$$f_1[j-1] + a_{1,j}$$

→ the way through $S_{2,j-1}$, transfer from line 2 to line 1, then through $S_{1,j}$

$$f_2[j-1] + t_{2,j-1} + a_{1,j}$$



$$f_1[j] = \min\left(f_1[j-1] + a_{1,j},\ f_2[j-1] + t_{2,j-1} + a_{1,j}\right)$$

So.

$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{if } j = 1 \\ \min\left(f_1[j-1] + a_{1,j},\ f_2[j-1] + t_{2,j-1} + a_{1,j}\right) & \text{if } j \geq 2 \end{cases}$$

$$f_2[j] = \begin{cases} e_2 + a_{2,1} & \text{if } j = 1 \\ \min\left(f_2[j-1] + a_{2,j},\ f_1[j-1] + t_{1,j-1} + a_{2,j}\right) & \text{if } j \geq 2 \end{cases}$$

Step 3. Computing Optimal Solution.

Fastest-way (a, t, e, x, n)

1. $f_1[1] = e_1 + a_{1,1}$
2. $f_2[1] = e_2 + a_{2,1}$
3. for $j = 2$ to n
4. do if $f_1[j-1] + a_{1,j} \leq f_2[j-1] + t_{2,j-1} + a_{1,j}$
5. then $f_1[j] = f_1[j-1] + a_{1,j}$
6. $l_1[j] = 1$
7. else $f_1[j] = f_2[j-1] + t_{2,j-1} + a_{1,j}$
8. $l_1[j] = 2$

9. if $f_2[j-1] + a_{2,j} \leq f_1[j-1] + t_{1,j-1} + a_{2,j}$

10. then $f_2[j] = f_2[j-1] + a_{2,j}$

11. $l_2[j] = 2$

12. else $f_2[j] = f_1[j-1] + t_{1,j-1} + a_{1,j}$

13. $l_1[j] = 1$

14. if $f_1[n] + x_1 <= f_2[n] + x_2$

15. then $f^* = f_1[n] + x_1$

16. $l^* = 1$

17. else $f^* = f_2[n] + x_2$

18. $l^* = 2$

Computed in $O(n)$ time.

$j \rightarrow$	1	2	3	4	5	6
$f_1[j]$	9	18	20	24	32	
$f_2[j]$	12	16	22	25	30	

$f^* = 35$

$j \rightarrow$	2	3	4	5	6
$l_1[j]$	1	2	1	1	
$l_2[j]$	1	2	1	2	

$l^* = 1$

Step 4: Construct the optimal solution:

```
Print_stations (l, n)
1. i = l*
2. Print line "i", station "n".
3. for j ← n down to 2
4.     do i = l_i[j]
5.        Print line "i" station "j-1"
```

	1	2	3	4	5
$f_1[j]$	9	18	20	24	32
$f_2[j]$	12	16	22	25	30

$f^* = 35$

	2	3	4	5
$l_1[j]$	1	2	1	1
$l_2[j]$	1	2	1	2

$l^* = 1$

$i = 1$

for $j = 5$ down to 2

$i = l_1[j] : l_1 5 = 1$ — line 1, station 5

$i = l_1[j] : l_1 4 = 1$ — line 1, station 4

$i = l_1[j] : l_1 3 = 2$ — line 1, station 3

$i = l_1[j] : l_1 2 = 1$ — line 2, station 2

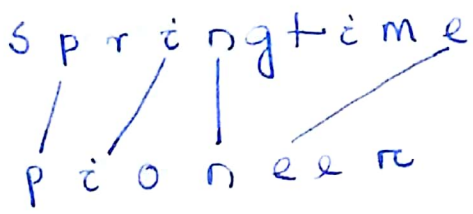
line 1, station 1

Longest Common Subsequence: (LCS)

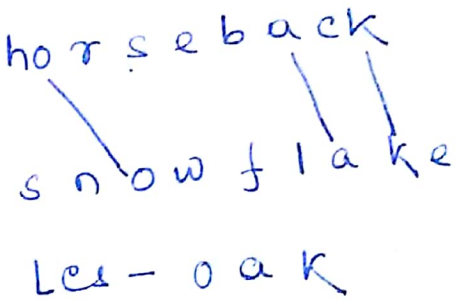
Problem: Given 2 sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$. Find a subsequence common to both whose length is longest.

A subsequence doesn't have to be consecutive, but it has to be in order.

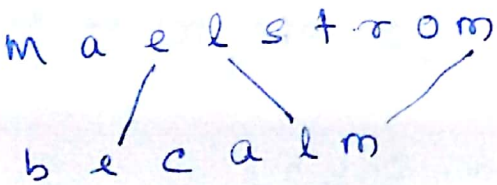
Example:



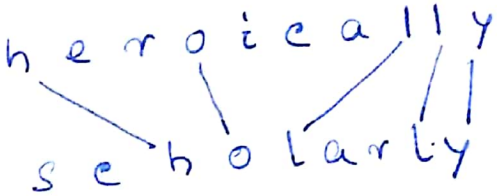
LCS - p i o n e



LCS - o a k



LCS :- e l m



LCS: h o l l y

1. Characterize the structure of an optimal solution.

notation: X_i = prefix $\langle x_1, \ldots, x_i \rangle$

Y_i = prefix $\langle y_1, \ldots, y_i \rangle$

Let $Z = \langle z_1, \ldots, z_k \rangle$ be any LCS of X and Y

1. 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}
2. If $x_m \neq y_n$, then $z_k \neq x_m \Rightarrow Z$ is an LCS of X_{m-1} and Y.

3. If $X_m \neq Y_n$, then $Z_k \neq Y_n \Rightarrow Z$ is an LCS of X and Y_{n-1}

Step-2 Recursive formulation: (i.e Recursively define the value of an optimal solution)

Define $c[i,j]$ = length of LCS of X_i and Y_j

We want $c[m,n]$

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

Step-3 Computing the value of an optimal solution.

LCS-Length (X, Y, m, n)

1. let $b[1..m, 1..n]$ and $c[0..m, 0..n]$ be new table.
2. for $i = 1$ to m
3. $\quad c[i,0] = 0$.
4. for $j = 0$ to n
5. $\quad c[0,j] = 0$.
6. for $i = 1$ to m.
7. $\quad$ for $j = 1$ to n.
8. $\quad\quad$ if $x_i = y_i$
9. $\quad\quad\quad c[i,j] = c[i-1,j-1] + 1$
10. $\quad\quad\quad b[i,j] = "\nwarrow"$
11. $\quad\quad$ else if $c[i-1,j] \geq c[i,j-1]$
12. $\quad\quad\quad c[i,j] = c[i-1,j]$
13. $\quad\quad\quad b[i,j] = "\uparrow"$

14. else c[i,j] = c[i, j-1]

15. b[i,j] = "←"

16. return c & b. → $\Theta(m \times n)$

X = A B C B D A B = 7

Y = B D C A B A. = 6

c Table. with b table.

		Yi	B	D	C	A	B	A
	Xi		0	0	0	0	0	0
	A		0 ↑	0 ↑	0 ↑	1 ↖	1 ←	1 ↖
✓	B		0 ↑	1 ↖	1 ←	1 ↑	2 ↖	2 ←
✓	C		0	1 ↑	1 ↑	2 ↖	2 ↑	2 ↑
✓	B		0	1 ↖	1 ↑	2 ↑	2 ↑	3 ←
	D		0	1 ↑	2 ↖	2 ↑	2 ↑	3 ↑
✓	A		0	1 ↑	2 ↑	2 ↑	3 ↖	3 ↑
	B.		0	1 ↖	2 ↑	2 ↑	3 ↑	4 ↖

So LCS → A B C B

Rev (LCS) = B C B A.

Time required to execute the Algo is $\Theta(mn)$

Step-4 Construct an optimal solution from computed information.

```
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 xi
```

6. else if $b[i,j] == "\uparrow"$

7. Print-LCS(b, X, i-1, j)

8. else print-LCS(b, X, i, j-1)

This algorithm required $\Theta(m+n)$ time for execution.

Matrix-Chain-Multiplication Problem.

Given
- → dimensions $p_0, p_1, p_2, \ldots, p_n$
- → Corresponding to matrix sequence $A_1, A_2, \ldots, A_n$ where A_i has dimension $p_{i-1} \times p_i$

determine the "multiplication sequence" that minimizes the number of scalar multiplications in computing $A_1, A_2, \ldots, A_n$. That is determine how to parenthesize the multiplications.

$A_1 A_2 A_3 A_4 = ((A_1 A_2)(A_3 A_4))$
$= (A_1 (A_2 (A_3 A_4)))$
$= (A_1 ((A_2 A_3) A_4))$
$= (((A_1 A_2) A_3) A_4)$

Step-1 characterize the structure of subproblems.

→ Decompose the problem into subproblems:

For each pair $1 \le i \le j \le n$, determine the multiplication sequence for $A_{i..j} = A_i A_{i+1} \cdots A_j$ that minizes the number of multiplications.

→ Clearly, $A_{i..j}$ is a $p_{i-1} \times p_j$ matrix

→ High level parenthesization for $A_{i..j}$

For any optimal multiplication sequence, at the last step you are multiplying two matrices $A_{1..k}$ and $A_{k+1..j}$ for some k. i.e

$A_{i..j} = (A_{i..k} A_k)(A_{k+1} \cdots A_j)$
$= A_{i..k} A_{k+1..j}$

Example:

$$A_{3..6} = (A_3 (A_4 A_5))(A_6)$$
$$= (A_{3..5})(A_{6..6})$$
$$= A_{3..5} A_{6..6}$$

Here $k = 5$.

Hence the problem of determining the optimal sequence of multiplications is broken down into 2 questions:

→ How do we decide where to split the chain?
→ (i.e search all possible values of k)
→ How do we parenthesize the subchains $A_{i..k}$ and $A_{k+1..j}$?
(i.e problem has optimal substructure property that is $A_{i..k}$ and $A_{k+1..j}$ must be optimal so we can apply the same procedure recursively)

What is optimal substructure property?
Ans: If final "optimal" solution of $A_{i..j}$ involves splitting into $A_{i..k}$ and $A_{k+1..j}$ at final step then parenthesization of $A_{i..k}$ and $A_{k+1..j}$ in final optimal solution must also be optimal for the subproblems "standing alone".

If parenthisization of $A_{i..k}$ was not optimal we could replace it by a better parenthesization and get a cheaper final solution, leading to a contradiction.

Similarly, if parenthisization of $A_{k+1..j}$ was not optimal we could replace it by a better parenthesization and get a cheaper final solution, also leading to a contradiction.

Step-2 (Recursively define the value of an optimal solution). Let $m[i,j]$ = Minimum no. of scalar multiplication.

→ We can define $m[i,j]$ recursively as follows.

If $i = j$, the problem is ~~trivial~~ trivial; i.e the chain consists of just one matrix $A_{i..i} = A_i$. So that no scalar multiplications are necessary to compute the product. Thus $m[i,i] = 0$ for $i = 1, 2, \cdots n$.

→ To compute $m[i,j]$ when $i < j$, let us assume that to optimally parenthesize we split the product $A_i A_{i+1} \cdots A_j$ between A_k and A_{k+1}, where $i \le k < j$. Then $m[i,j]$ equals to the minimum cost for computing subproducts $A_{i..k}$ and $A_{k+1..j}$ plus the cost of multiplying these two matrices together.

i.e

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j$$

This recursive equation assumes that we know the value k. There are only $j-i$ possible values of k, however $k = i, i+1, \ldots, j-1$.

Our recursive definition for the minimum cost of parenthesizing the product $A_i A_{i+1} \cdots A_j$ becomes.

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

Step-3 Computing the optimal cost.

Matrix-Chain-Order (p)

1. $n = p.length - 1$
2. Let $m[1..n, 1..n]$ and $s[1..n-1, 2..n]$ be new tables.
3. for $i = 1$ to n
4. $\quad m[i,i] = 0$
5. for $l = 2$ to n
6. $\quad$ for $i = 1$ to $n - l + 1$
7. $\quad\quad j = i + l - 1$
8. $\quad\quad m[i,j] = \infty$
9. $\quad\quad$ for $k = i$ to $j - 1$
10. $\quad\quad\quad q = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j$

```
11.                 if q < m[i, j]
12.                     m[i, j] = q
13.                     s[i, j] = k
14. Return m & s.
```

Lets illustrate the algorithm with the help of an example.

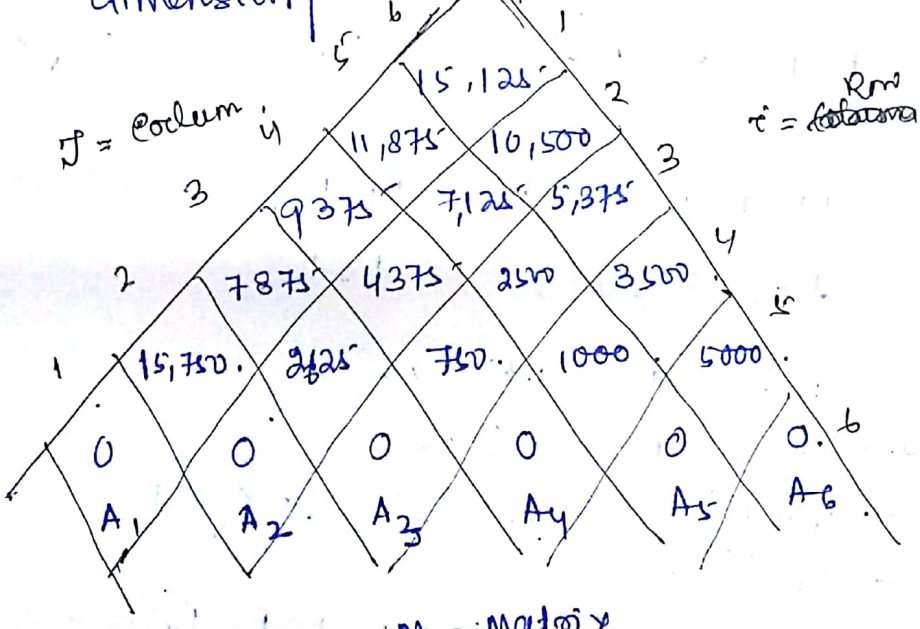
Question.

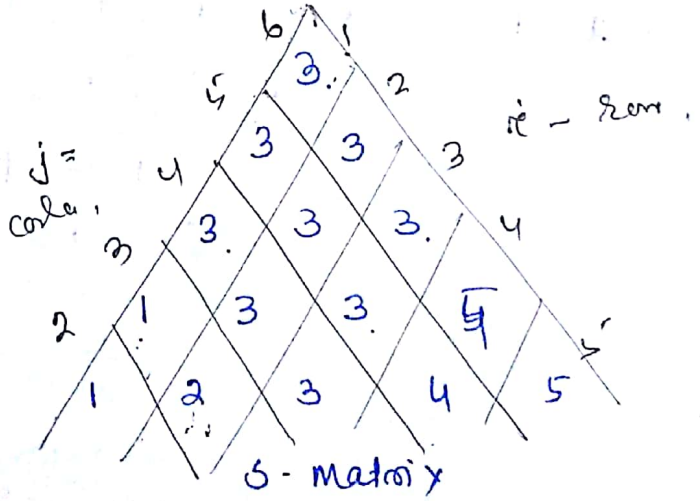
Find an optimal parenthesization of a matrix-chain product whose sequence of dimension is

P_0 P_1 P_2 P_3 P_4 P_5 P_6

$\langle 30, 35, 15, 5, 10, 20, 25 \rangle$.

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25





$$m[1,2] = \min \begin{bmatrix} m[1,1] + m[2,2] + p_0 p_1 p_2 \end{bmatrix}$$

$$= \min [\, 0 + 0 + 30 \times 35 \times 15 \,] = 15,750. \quad \text{—— } k = 1$$

$$m[2,3] = \min [\, m[2,2] + m[2,3] + p_1 p_2 p_3 \,]$$

$$= \min [\, 0 + 0 + 35 \times 15 \times 5 \,] = 2,625 \quad \text{—— } k = 2$$

$$m[3,4] = \min [\, m[3,3] + m[4,4] + p_2 p_3 p_4 \,]$$

$$= \min [\, 0 + 0 + 15 \times 5 \times 10 \,] = 750. \quad \text{—— } k = 3$$

$$m[4,5] = \min [\, m[4,4] + m[5,5] + p_3 p_4 p_5 \,]$$

$$= \min [\, 0 + 0 + 5 \times 10 \times 20 \,] = 1000. \quad \text{—— } k = 4$$

$$m[5,6] = \min [\, m[5,5] + m[6,6] + p_4 p_5 p_6 \,]$$

$$= \min [\, 0 + 0 + 10 \times 20 \times 25 \,] = 5000. \quad \text{—— } k = 5$$

$$m[1\ 3] = \min \begin{bmatrix} m[1,1] + m[2,3] + p_0 p_1 p_3. \\ m[1,2] + m[3,3] + p_0 p_2 p_3. \end{bmatrix}$$

$$= \min \begin{bmatrix} 0 + 2625 + 30 \times 35 \times 5 \\ 15750 + 0 + 30 \times 15 \times 5 \end{bmatrix}$$

$$= \min \begin{bmatrix} 7,875 \\ 18,000 \end{bmatrix} = 7,875 \quad \text{for } k = 1.$$

$$m[2\ 4] = \min \begin{bmatrix} \min[2,2] + m[3,4] + p_1 p_2 p_4 \\ \min[2,3] + m[4,4] + p_1 p_3 p_4 \end{bmatrix}$$

$$= \min \begin{bmatrix} 0 + 750 + 35 \times 15 \times 10. \\ 2625 + 0 + 35 \times 5 \times 10. \end{bmatrix} = \begin{bmatrix} 6000. \\ 4375. \end{bmatrix} \text{—K}$$

$$m[3,5] = \min \begin{bmatrix} m[3,3] + m[4,5] + p_2 p_3 p_5 \\ m[3,4] + m[5,5] + p_2 p_4 p_5 \end{bmatrix}$$

$$= \min \begin{bmatrix} 0 + 1000 + 15 \times 5 \times 20 \\ 750 + 0 + 15 \times 10 \times 20 \end{bmatrix} = \begin{bmatrix} 2500 \\ 3750 \end{bmatrix} \text{ for } k = 3.$$

$$m[4,6] = \min \begin{bmatrix} m[4,4] + m[5,6] + p_3 p_4 p_6 \\ m[4,5] + m[6,6] + p_3 p_5 p_6 \end{bmatrix}$$

$$= \min \begin{bmatrix} 0 + 5000 + 5 \times 10 \times 25 \\ 1000 + 0 + 5 \times 20 \times 25 \end{bmatrix} = \begin{bmatrix} 6250 \\ 3500 \end{bmatrix} \text{ for } k = 5$$

---

i.

$$m[1,4] = \min \begin{bmatrix} m[1,1] + m[2,4] + p_0 p_1 p_4 \\ m[1,2] + m[3,4] + p_0 p_2 p_4 \\ m[1,3] + m[4,4] + p_0 p_3 p_4 \end{bmatrix}$$

3.

$$= \min \begin{bmatrix} 0 + 4375 + 30 \times 35 \times 10 \\ 1750 + 750 + 30 \times 15 \times 10 \\ 7875 + 0 + 30 \times 5 \times 10 \end{bmatrix}$$

$$= \min \begin{bmatrix} 14,875 \\ 18,000 \\ 9,375 \end{bmatrix} \text{ for } k = 3.$$

$$m[2,5] = \min \begin{bmatrix} m[2,2] + m[3,5] + p_1 p_2 p_5 \\ m[2,3] + m[4,5] + p_1 p_3 p_5 \\ m[2,4] + m[5,5] + p_1 p_4 p_5 \end{bmatrix}$$

$$= \min \begin{bmatrix} 0 + 2500 + 35 \times 15 \times 20 \\ 2625 + 1000 + 35 \times 5 \times 20 \\ 4375 + 0 + 35 \times 10 \times 20 \end{bmatrix} =$$

$$= \min \begin{bmatrix} 10,500 \\ 7,125 \\ 11,375 \end{bmatrix} \qquad k = 3.$$

$$m[3,6] = \min \begin{bmatrix} m[3,3] + m[4,6] + P_2 P_3 P_6 \\ m[3,4] + m[5,6] + P_2 P_4 P_6 \\ m[3,5] + m[6,6] + P_2 P_5 P_6 \end{bmatrix}$$

$$= \min \begin{bmatrix} 0 + 3500 + 15 \times 5 \times 25 \\ 750 + 5000 + 15 \times 10 \times 25 \\ 2500 + 0 + 15 \times 20 \times 25 \end{bmatrix}$$

$$= \min \begin{bmatrix} 5,375 \\ 9,500 \\ 10,000 \end{bmatrix} \quad \text{for } k = 3.$$

$$M[1,5] = \min \begin{bmatrix} M[1,1] + M[2,5] + P_0 P_1 P_5 \\ M[1,2] + M[3,5] + P_0 P_2 P_5 \\ M[1,3] + m[4,5] + P_0 P_3 P_5 \\ m[1,4] + m[5,5] + P_0 P_4 P_5 \end{bmatrix}$$

$$= \min \begin{bmatrix} 0 + 7125 + 30 \times 25 \times 20 \\ 15750 + 2500 + 30 \times 15 \times 20 \\ 7875 + 1000 + 30 \times 5 \times 20 \\ 9375 + 0 + 30 \times 10 \times 20 \end{bmatrix}$$

$$= \min \begin{bmatrix} 28,125 \\ 27,250 \\ 11,875 \\ 15,375 \end{bmatrix} \quad k = 3.$$

$$M[2,6] = \min \begin{bmatrix} m[2,2] + m[3,6] + P_1 P_2 P_6. \\ m[2,3] + m[4,6] + P_1 P_3 P_6. \\ m[2,4] + m[5,6] + P_1 P_4 P_6. \\ m[2,5] + m[6,6] + P_1 P_5 P_6. \end{bmatrix}$$

$$= \min \begin{bmatrix} 0 + 5375 + 35 \times 15 \times 25 \\ 2625 + 3500 + 35 \times 5 \times 25 \\ 4375 + 5000 + 35 \times 10 \times 25 \\ 7125 + 0 + 35 \times 20 \times 25 \end{bmatrix}$$

$$= \min \begin{bmatrix} 15,375 \\ 10,500 \\ 18,125 \\ 24,625 \end{bmatrix} \longrightarrow k = 3.$$

$$m[1,6] = \min \begin{bmatrix} m[1,1] + m[2,6] + P_0 P_1 P_6. \\ m[1,2] + m[3,6] + P_0 P_2 P_6 \\ m[1,3] + m[4,6] + P_0 P_3 P_6. \\ m[1,4] + m[5,6] + P_0 P_4 P_6 \\ m[1,5] + m[6,6] + P_0 P_5 P_6. \end{bmatrix}$$

$$= \min \begin{bmatrix} 0 + 10,500 + 30 \times 35 \times 25 \\ 15750 + 5375 + 30 \times 15 \times 25 \\ 7875 + 3500 + 30 \times 5 \times 25. \\ 9375 + 5000 + 30 \times 10 \times 25 \\ 11,875 + 0 + 30 \times 20 \times 25 \end{bmatrix}$$

$$= \min \begin{bmatrix} 36,750 \\ 32,375 \\ 15,125 \\ 21,875 \\ 26,875 \end{bmatrix} \longrightarrow k = 3.$$

A simple inspection of the nested loop structure of Matrix-chain-order yields a running time of $O(n^3)$ for the algorithm.

Step-4 Constructing an optimal solution.

Print-Optimal-Parens (s, i, j)

1. if i == j
2. Print "A_i"
3. else print "("
4. Print-Optimal-Parens (s, i, s[i,j])
5. Print-Optimal-Parens (s, s[i,j]+1, j)
6. Print ")"

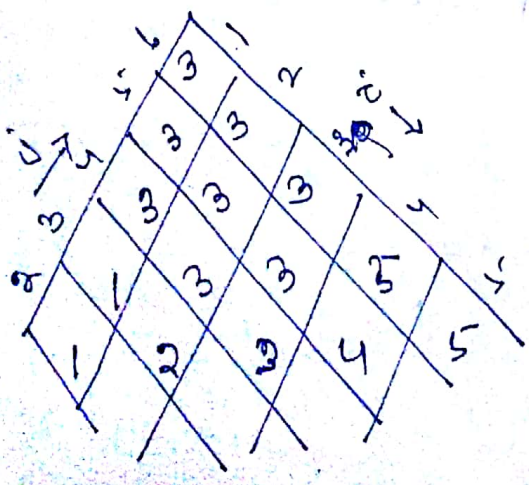
s [1,6] = 3 · $(A_1\ A_2\ A_3)(A_4\ A_5\ A_6)$

A [1,3] = 1 $(A_1 (A_2 A_3))$

A [4,6] = 5 $((A_4 A_5) A_6)$

Hence the final Multiplication sequence is

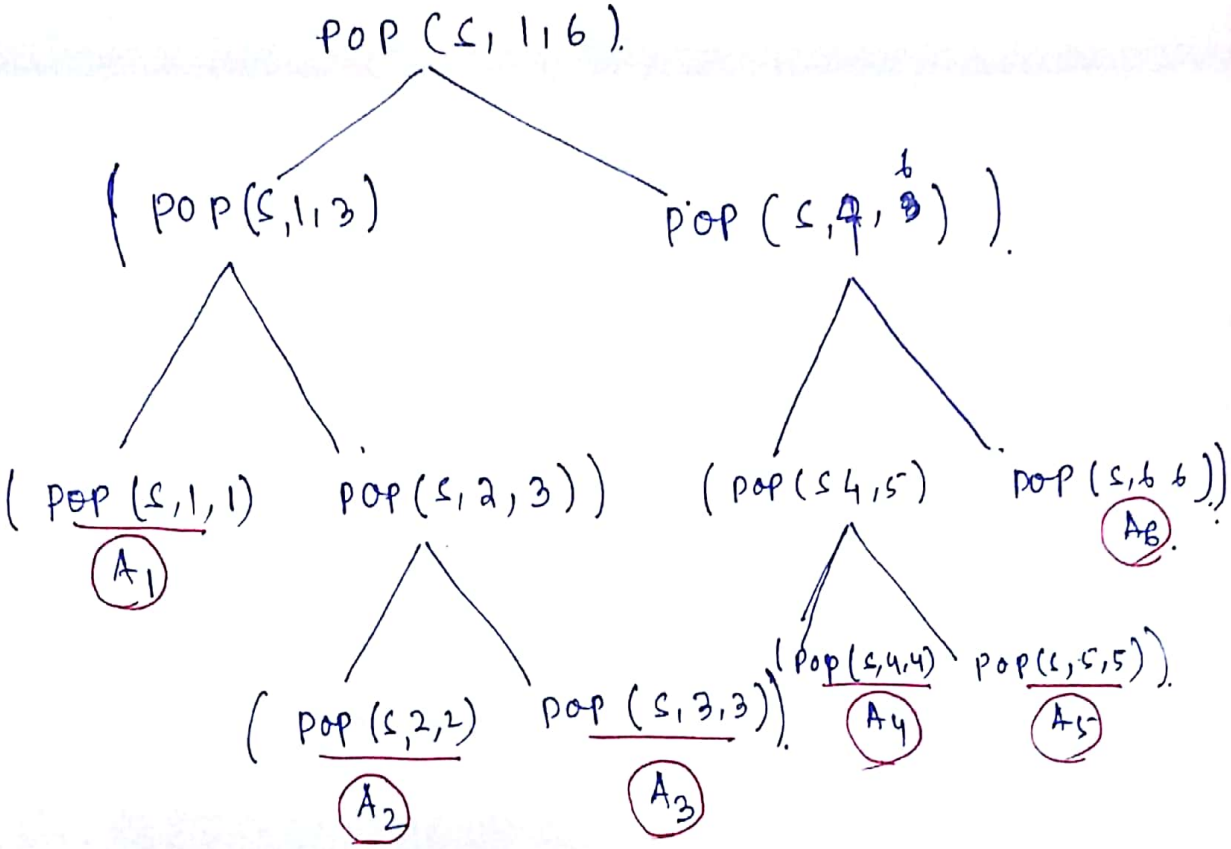
$(A_1 (A_2 A_3))((A_4 A_5) A_6)$.



	2	3	4	5	6
1	1	1	3	3	3
2		2	3	3	3
3			3	3	3
4				4	5
5					5

s - array.

execution of pop (S, 1, j)



Backtracking:

→ Backtracking is a general algorithmic technique that consider searching every possible combination in order to solve an optimization problem.

→ Backtracking Algorithm is based on depth-first recursive search.

→ Approach.

1. Tests whether solution has been found
2. If found solution, return it.
3. Else for each choice that can be made
   a) Make that choice
   b) Recure
   c) If recursion returns a solution, return it.
4. If no choices remain, return failure.

→ Some time called "search tree.

Examples of Backtracking:

1. Sum of Subsets
2. Graph Coloring
3. N-Queen Problem.
4. Hamiltonian Cycle.

Sum of Subset:

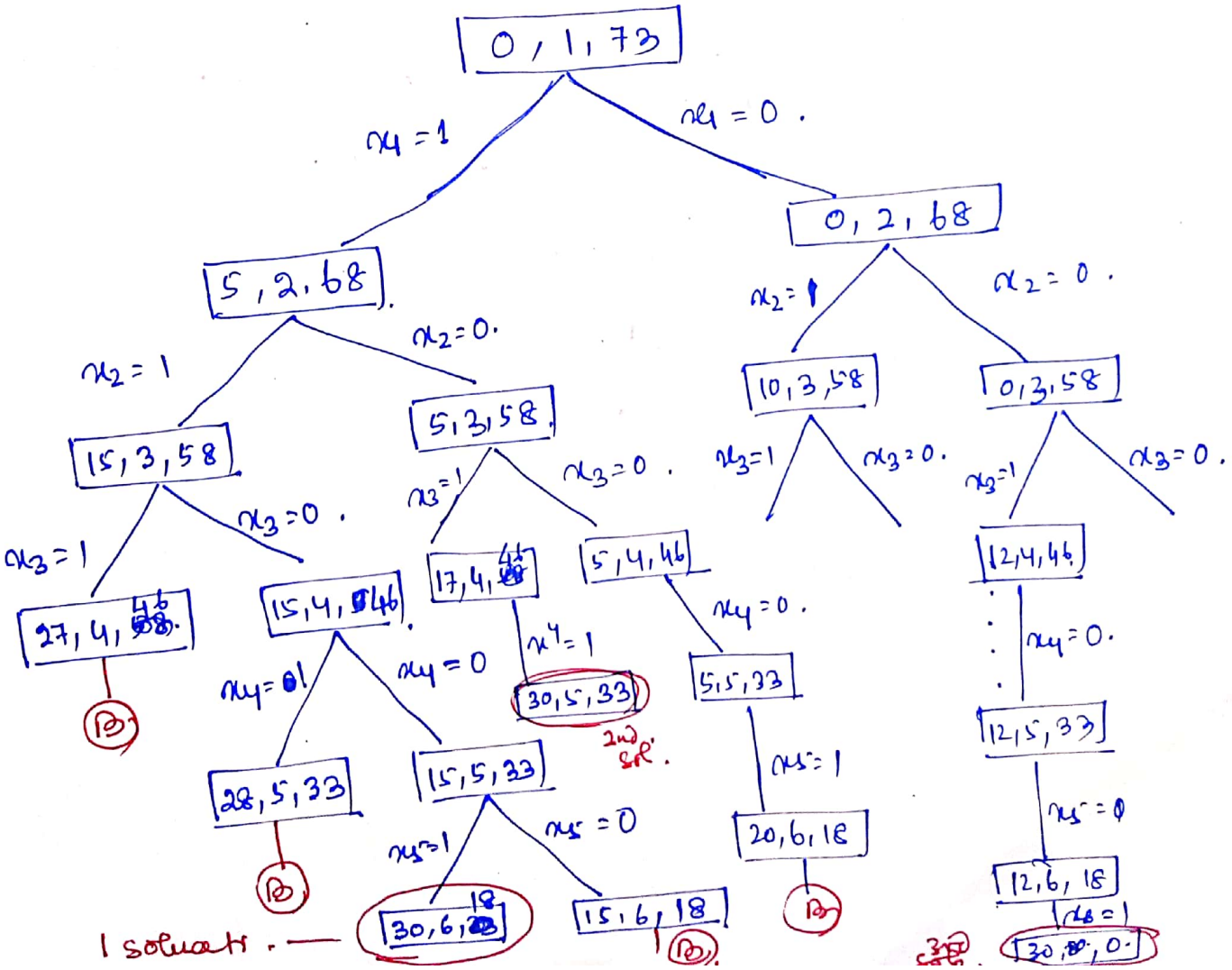
Suppose we are given n distinct positive numbers and we desired to find all combinations of these numbers whose sums are 'm'. is known as Sum of subset problem.

Let us solve a problem for understanding the concept.

Problem:

Let w = {5, 10, 12, 13, 15, 18} and m = 30. Find all possible subsets of w that sum to m.

First generate the state space tree with the help of SumOfSubset() {discussed letter}



The Recursive backtracking algorithm for Sum of subsets problem.

```
Sum Of Subset (s, k, r)
1. x[k] = 1.
2. if (s + w[k] = m) then
3.         Print x[1:k]  // subset found.
4. else if (s + w[k] + w[k+1] ≤ m)
5.         then Sum Of Subset (s, w[k], k+1, r-w[k]);
6. if ((s + r - w[k] ≥ m) and (s + w[k+1] ≤ m) then.
7.         x[k] = 0.
8.         SumOfSubset (s, k+1, r-w[k]);
```

Analysis:

The all possible solution in above example is

```
0 0 0 0 0 0
:
1 1 1 1 1 1
```

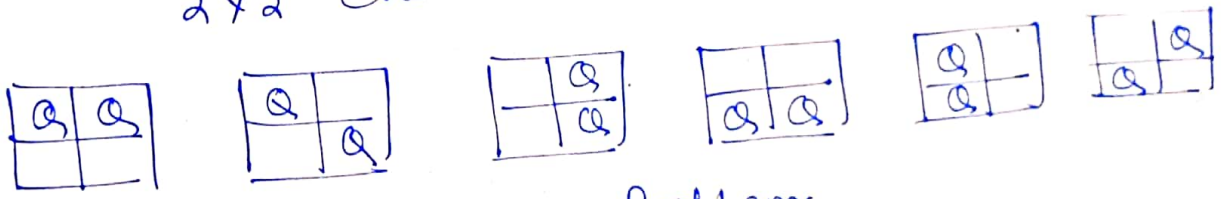
i.e. 2^6.

So if there are n numbers then the running time is $O(2^n)$.

N. Queen Problem.

1. N-Queen Problem is based on chess games.
2. The problem is based on arranging the queens on chessboard in such a way that no two queens can attack each other.
3. The N Queen problem states as consider a n x n chessboard on which we have to place n queens, so that no two queens attack each other by being in the same diagonal.

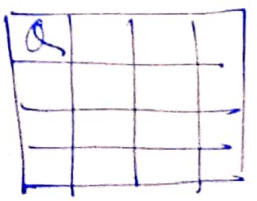
(Note: The 2 Queen problem is not solvable because 2-Queens can be placed on 2x2 chess board as shown below.



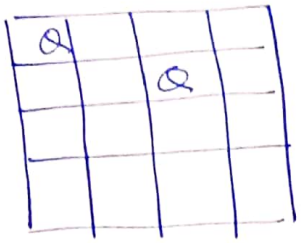
How to solve N-Queen Problem.

1. Let us take the example of 4-Queen and 4x4 chessboard.
2. Start with an empty chess board.
3. place queen 1 in the first possible position of its row i.e 1st row and 1st column.

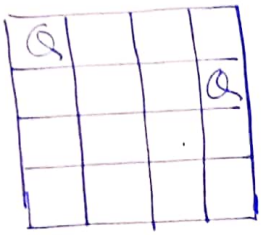
(a)



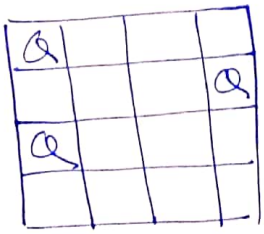
4. Then place queen 2 after trying unsuccessful place (1,2), (2,1) (2,2) at (2,3) i.e 2nd row and 3rd column.



5. This is a dead end because a 3rd queen cannot be placed in next column, as there is no acceptable position for queen 3. Hence algorithm backtracks and places the 2nd queen at (2,4) position.

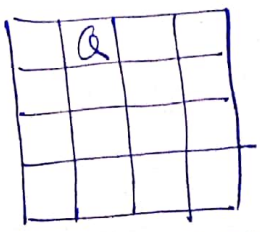
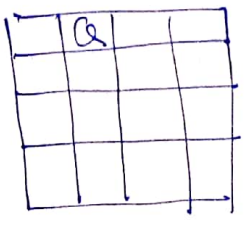
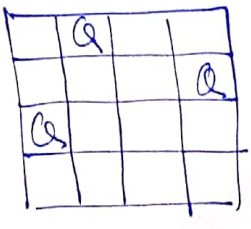
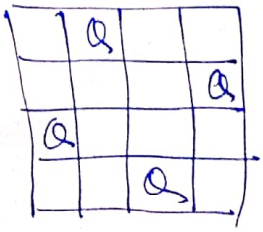


6. Then place 3rd queen at (3,2) but it again another dead lock end as next queen (4th queen) cannot be placed at permissible position.

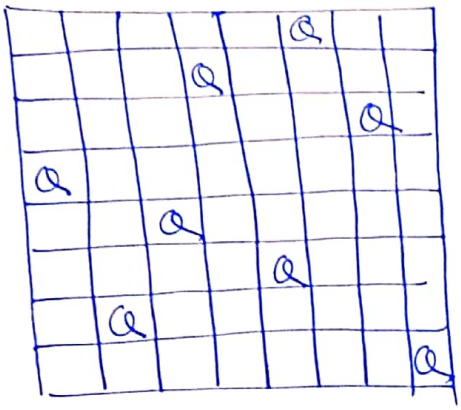


7. Hence we need to backtrack all the way up to queen 1 and move it to (1,2)

8. place queen 1 at (1,2), queen 2 at (2,4), queen 3 at (3,1) and queen 4 at (4,3),

(ix) Thus the solution is obtained (2,4,1,3) in row wise manner (x). The one of the solution to 8-Queen Problem is shown below.



(xi) The solution is obtained (6,4,7,1,3,5,2,8) in row wise manner.

Algorithm nQueens(k,n)
(Using backtracking this procedure prints all possible placements of n queens on n x n chessboard so that they are non attacking.)

1. for i = 1 to n do.
2. if Place(k,i) then
3. x[k] = i
4. if (k=n) then
5. write (x[1:n]))
6. else
8. nQueens (k+1, n)

Algorithm Place (k, i)

(Returns true if a queen can be placed in kth row and ith column. Otherwise it returns false. x[] is a global array whose (k-1) values have been set. Abs(r) returns the absolute value of r.)

1. for j = 1 to k-1 do.
2. if ((x[j] = i) or (Abs[x[j] - i) = Abs(j-k)))
3. then return false.
4. return true.

~~Analysis~~:

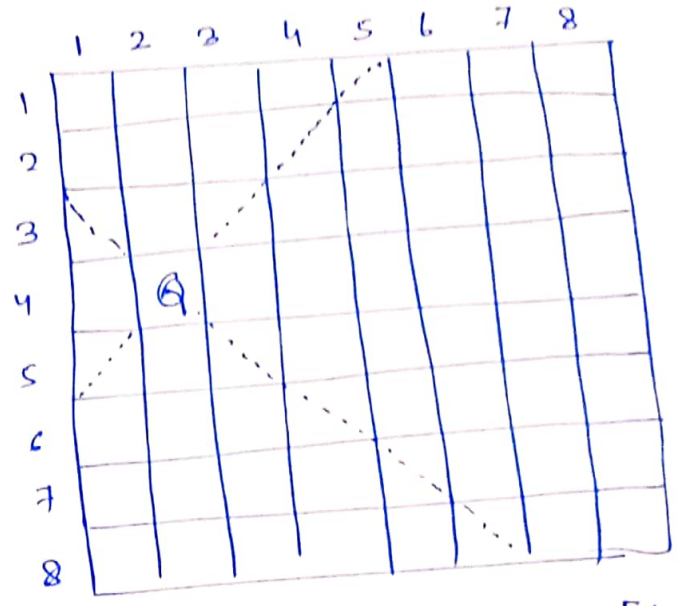
~~...~~ place ~~...~~

~~...~~.

The execution of algorithm is based on following principle.

If we imagine the chessboard squares being numbered as the indices of the two-dimensional array a[1..n, 1..n], then we observe that every element on the same diagonal that runs from the upper left to the lower right has the same row column value. Let's execute with an example of 8x8 matrix (i.e 8-Queen) problem.

a.



Let us consider the queen at a[4,2] position. The squares that are diagonal to that queen running from upper left to upper right are a[3,1], a[5,3], a[6,4], a[7,5] and a[8,6]. All these squares have a row − column value of 2. Also every element on the same diagonal that goes from upper right to lower left are a[1,5], a[2,4], a[3,3], a[5,1] has the same row + column value of 6.

Suppose two queens are placed at position (i, j) and (k, l), then they lie above they are on the same diagonal if and only if

$$i - j = k - l \quad \text{or} \quad i + j = k + l$$

The first equation implies

$$j - l = i - k$$

The second equation implies

$$j - l = k - i$$

Therefore two queens lie on the same diagonal if and only if $|j-l| = |i-k|$.

Analysis.

Place (k, i) algorithm returns a boolean value that is true if the k^{th} queen can be placed in column i. It tests both whether i is distinct from all previous values $x[1] \ldots x[k-1]$ and whether there is no other queen on the same diagonal. Its computing time is $O(k-1)$.

The NQueen (k, n) algorithm required $O(n!)$ time for execution.

Graph Coloring: (m coloring Problem)

Given an undirected graph and a number m, determine if the graph can be colored with at most m colors such that no two adjacent vertices of the graph are colored with same color. Here coloring of a graph means assignment of colors to all vertices.

Input:

1) A 2D array graph [V][V] where V is the number of vertices in graph and graph [V][V] is adjancy matrix representation of the graph. A value graph [i] [j] is 1 if there is a direct edge from i to j, otherwise graph [i] [j] = 0.

2) An integer m which is maximum number of colors that can be used.

O/P:

An array x[v] that should have numbers from 1 to m. x[i] should represent the color assigned to the ith vertex. The code should also return false if the graph cannot be colored with m colors.

Algorithm

mColoring (k)
(This algorithm was formed using the recursive backtracking schema. The graph is represented by its boolean adjacency matrix G[1..n, 1..n]. All assignments of 1, 2, ..., m to the vertices of the graph such that adjacent vertices are assigned distinct integers are printed. k is the next vertex to color.)

```
1. repeat
2.     nextValue (k);
3.     if (x[k] = 0) then
4.         return;
5.     if (k == n) then
6.         print x[1..n] );
```

```
    else mColoring (k+1);
} until false.
```

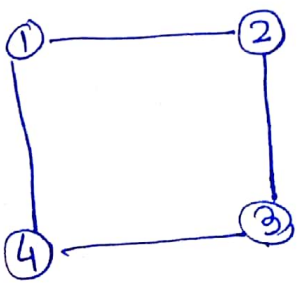
Algorithm

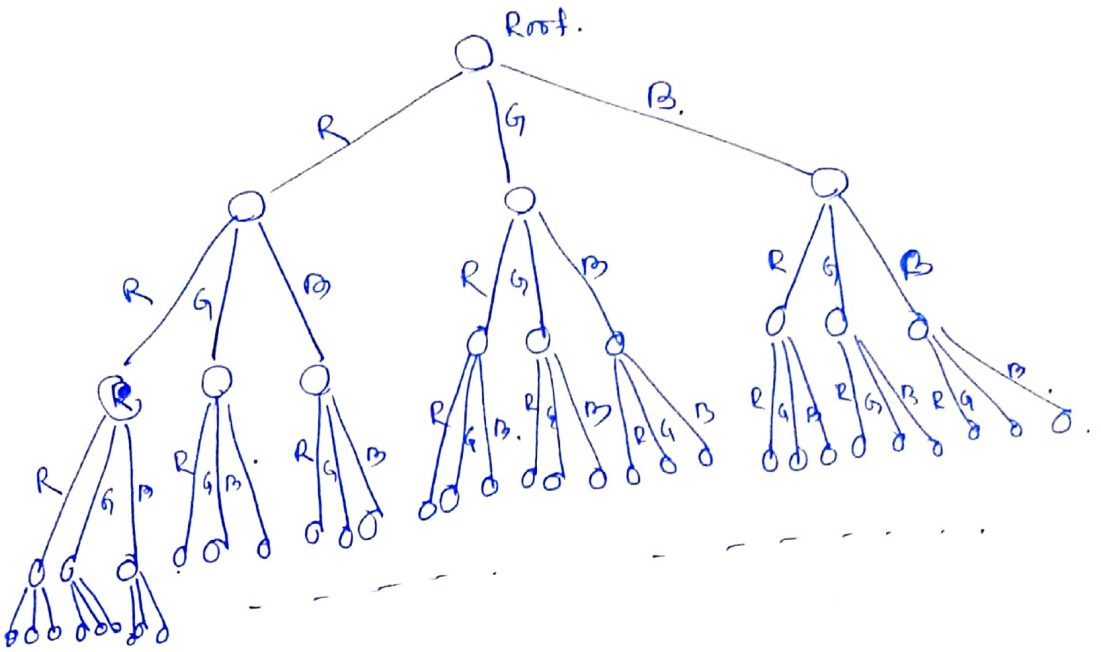
```
NextValue (k)
1. repeat.
2. { x[k] = (x[k]+1) mod (m+1)
3.   if (x[k] == 0)
4.       return ;
5.   for (j = 1 to n) do
6.       if ( (G[j,k] ≠ 0 and x[k] = x[j]))
7.           break;
8.   if (j = n+1)
9.       return;
10. } until (false)
```

Analaysis.

Below there is a simple graph and three color is given. The work is colored all the vertices such that no two adjacent vertices are same color. The method of solving this problem in Backtracking is state space tree.

So Let's first generate the state space tree without any condition and checking.





The above state space tree shows all possibility to which we can color them without checking the adjacency checking condition. i.e neighbouring vertices have same condition.

Let's check how many nodes are created without checking condition.

$$= 1 + 3 + 3\times3 + 3\times3\times3 + 3\times3\times3\times3 \cdots$$

$$= 1 + \underbrace{3^1 + 3^2 + 3^3 + 3^4 \cdots}_{\text{GP series.}} = \frac{3^{4+1} - 1}{3-1} = \frac{3^{n+1} - 1}{2} \approx 3^{n+1}$$

$$\vdots \approx c^{n+1}$$

$$3$$

Now generate state space tree with the help of Backtracking.

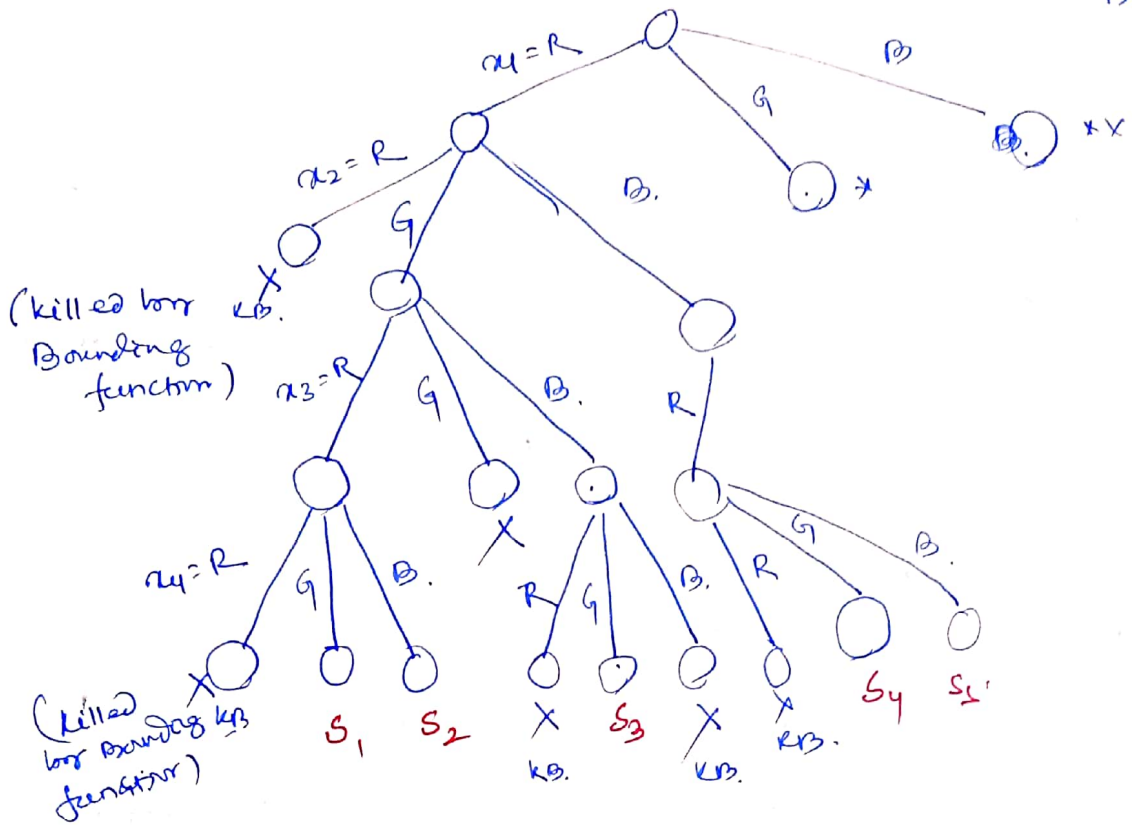
$S_1 = R\ G\ R\ G$

$S_2 = R\ G\ R\ B$

$S_3 = R\ G\ B\ G$

$S_4 = R\ B\ R\ G$

$S_5 = R\ B\ R\ B.$



* find all possible soln. for G *

** find all possible soln. for B.

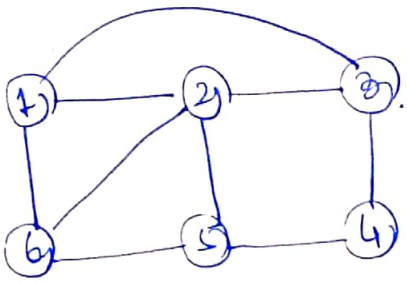
Hence the complexity is e^n.

Hamiltonian Cycles.

→ Let $G = \langle V, E \rangle$ be a connected graph with n vertices. → A Hamiltonian cycle is a round-trip path along n edges of G that visits every vertex once and returns to its starting position.

→ In other words a Hamiltonian cycle begins at some vertex $v_1 \in G$ and the vertices of G are visited in the order $v_1, v_2, \ldots, v_{n+1}$, then the edges (v_i, v_{i+1}) are in E, $1 \leq i \leq n$, and the v_i are distinct except for v_1 and v_{n+1}, which are equal.

Some Hamiltonian cycle with graph given below.

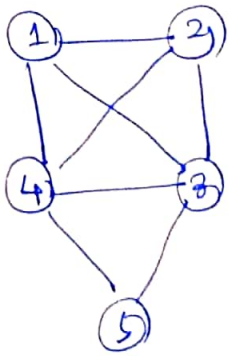


→ Hamiltonian Cycle.

1 2 3 4 5 6 1

1 2 6 5 4 3 1

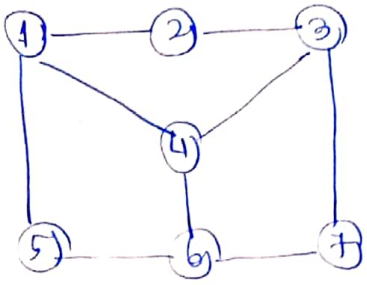
1 6 2 5 4 3 1



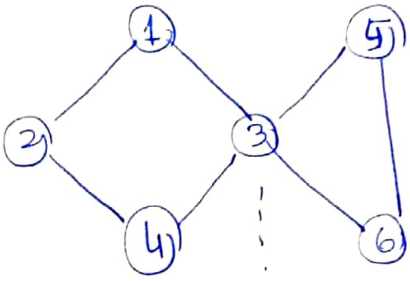
→

1 2 3 5 4 1

1 2 4 5 3 1

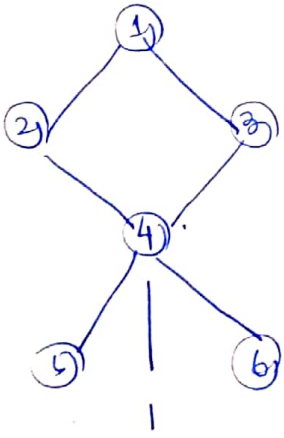


→ no Hamiltonian Cycle. is not existance in this graph.



→ Hamiltonian cycle is not possible because of Articulation point vertex

junction Point
or
Conecting Point.
or
Articulation Point.



→ Hamiltonian cycle is not possible because of Pendent ~~vertex~~ vertex.

Pendent
Vertices.

Let's find how Backtracking is help us for finding the hamiltonian cycle. in next Page.

Algorithm Hamiltonian (k).

(The algorithm uses the recursive formulation of backtracking to find all the Hamiltonian cycles of a graph. The graph is stored as an adjacency matrix G[1..n, 1..n]. All cycle begin at node 1)

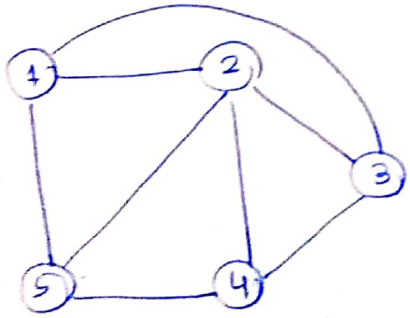
```
1. {
2.   repeat.
3.   {// Generate values for x[k].
4.     nextvalue(k); // Assign legal next value to x[k]
5.     if (x[k] = 0) then
6.           return.
7.     else.
8.           Hamiltonian(k+1);
9.   } until (false).
10. }
```

Algorithm nextValue(k).

(x[1..k-1] is a path of k-1 distinct vertices. If x[k] = 0, then no vertex has as yet been assigned to x[k]. After execution x[k] is assigned to the next highest numbered vertex which does not already appear in x[1..k-1] and is connected by an edge to x[k-1]. Otherwise x[k] = 0, if k == n, then in addition x[k] is connected to x[1].)

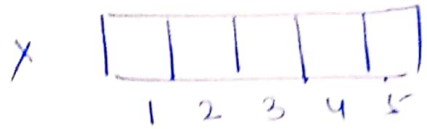
```
1.  {
2.      repeat.
3.      {
4.          x[k] = (x[k] + 1) mod (n+1) ;   // next value of vertex
5.          if (x[k] = 0) then
6.              return ;
7.          if (G[x[k-1], x[k]]) ≠ 0) then   // check vertex exists or not.
8.          {   for j := 1 to k-1 do
9.              if (x[j] = x[k]) then   // duplicate check
10.                 break.
11.         if (j = k) then
12.             if ((k < n) or (k = n) and G[x[n], x[1]] ≠ 0))
                    then return.
13.         }
14.     } until (false).
15. }
```

Lets execute the above algorithm with the help of an example.

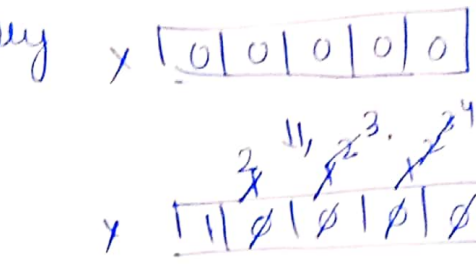


$$G = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

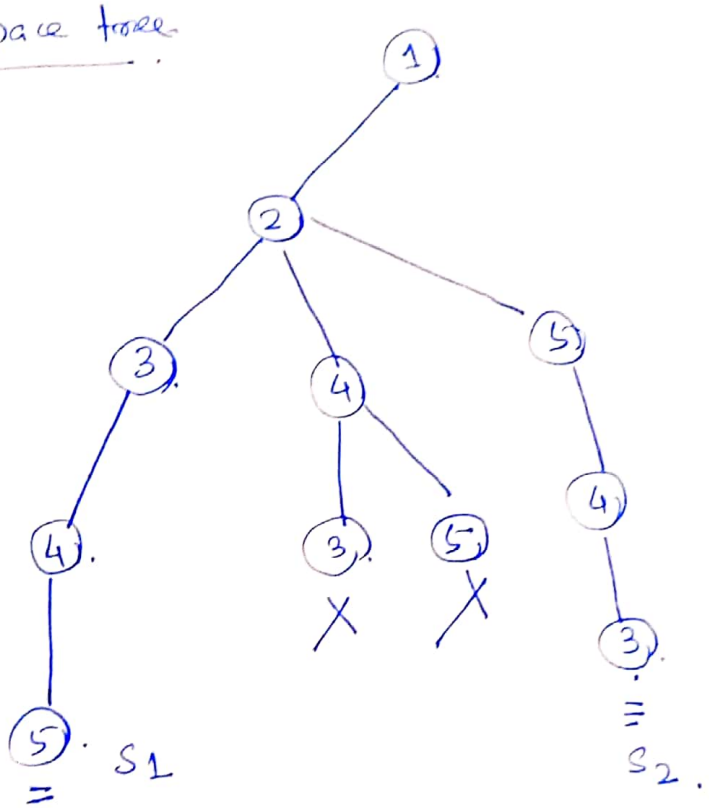
Use x array for finding the cycle.



Initially x



State space tree



If we are in last vertex then check there is a path from last vertex to first vertex is present or not. If present print the path.

S_1 = 1 2 3 4 5

S_2 = 1 2 5 4 3.

The time complexity of n vertices to visit all permutations of the vertices which is n!, i.e O(n!)

∴ [redacted]

Branch and Bound.

→ Branch and Bound is a systematic method for solving optimization problems.

→ Branch and Bound is a general optimization technique that applies where the greedy method and dynamic programming fail.

→ However, it is much slower. Indeed, it often leads to exponential time complexities in the worst case.

→ On the other hand, if applied carefully, it can lead to algorithms that run reasonably fast on average.

→ The general idea of Branch and Bound is a BFS (Breadth first search) - like search for the optimal solution, but not all nodes get expanded (i.e. their children expanded). Rather a carefully selected criterion determines which node to expand and when, and another criterion tells the algorithm when an optimal solution has been found.

Travelling Salesman Problem using Branch & Bound.

Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visit every city exactly once and returns to the starting point.

The term Branch and Bound refer to all state space search methods in which all the children of E-node (i.e expandable node) are generated before any other live node can become the E-node. E-node is the node which is being expanded. State space tree can be expanded in any method. i.e BFS or DFS. Both start with the root node and generate other nodes.

A node which has been generated and all of whose children are not yet been expanded is called live node.

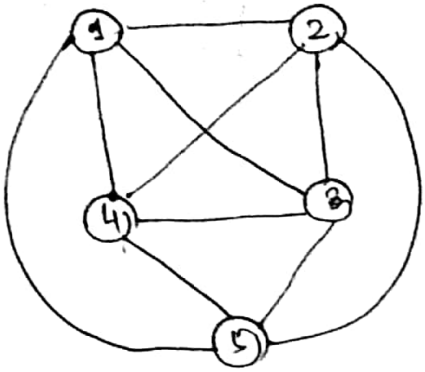
A node is called dead node, which has been generated but it cannot be expanded further.

With the help of this method we expand the node which is most promising, means the node which promises that expanding or choosing, it will give us the optimal solution. Hence we prepare the tree starting from the root and then we expand it.

The cost matrix defined by

$$c(i,j) = \begin{cases} w(i,j) & \text{if there is a direct path between } e_i \text{ to } e_j \\ \infty & \text{if there is no direct path between } e_i \text{ to } e_j. \end{cases}$$

Let us solve with an example.



The cost matrix of this graph is given below.

	1	2	3	4	5
1	∞	20	30	10	11
2	15	∞	16	4	2
3	3	5	∞	2	4
4	19	6	18	∞	3
5	16	4	7	16	∞

Step-1 First reduced the cost of the table matrix. (i.e. Reducing Matrix)

Find the minimum value of each row and then create the resultant matrix by subtracting the minimum value from each element of the same row (i.e Reduce row)

∞	20	30	10	11	10
15	∞	16	4	2	2
3	5	∞	2	4	2
19	6	18	∞	3	3
16	4	7	16	∞	4

21

11

∞	10	20	0	1
13	∞	14	2	0
1	3	∞	0	2
16	3	15	∞	0
12	0	3	12	∞

1 0 3 0 0 = 4

Find the minimum value of each column and create the resultant matrix by subtracting the minimum value from each element of the same column. (i.e Reduce Column)

Hence, the Reduced cost matrix is :

$$\begin{bmatrix} \infty & 10 & 13 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

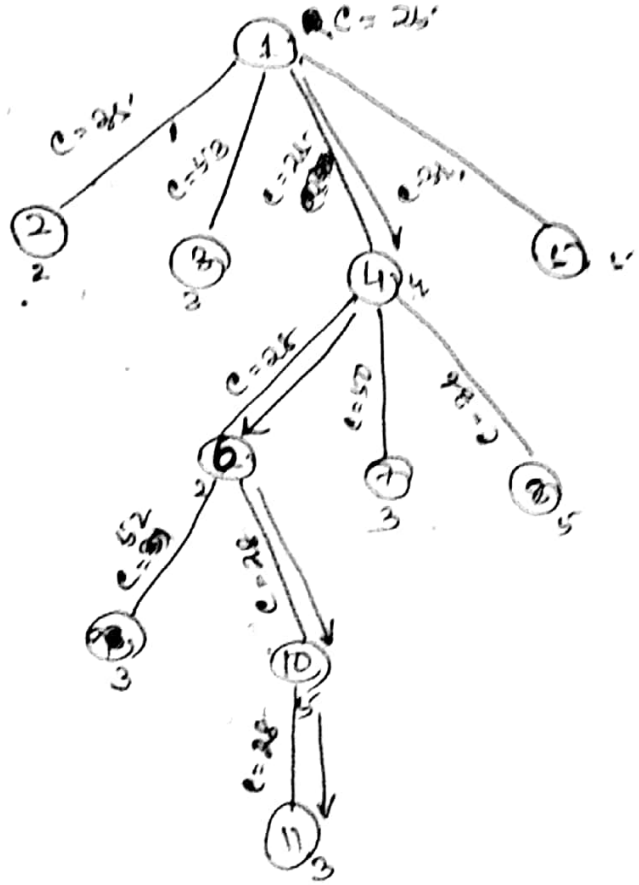
Total cost of reduction of all the rows = 10+2+3+2+1 = 18

Total cost of reduction of all column = 1+0+0+0+0 = 1

Hence the reduced cost = 21+4+1 = 26

Hence, reduce matrix is one in which every row and coloumn is reduced, i.e at least one zeros in each column and each row.

Lets solve the problem using branch and bound with the help of state space tree.



The cost of the 1st node is = 25.

Find the cost from 1 to 2 node.

→ Make ∞ to 1st row.

Make ∞ to 2nd column.

Make 2 to 1 also ∞.

Then check the matrix reduced or not

If not then reduced it by using row & reduce and column reduced formula.

For 1 − 2.

∞	∞	11	2	0	0
0	∞	∞	0	2	0
15	∞	12	∞	0	0
11	∞	0	12	∞	0
0	0	0	0	0	

Then check the Matrix is reduced or not.

← This matrix is now is reduced.

Then calculate the cost.

$c(1)$

The cost = $c(1,2) + r + \hat{r}$

= 10 + 25 + 0.

∴ = 35.

Where -

r ← Reduced cost of 1st node i.e (25).

$\hat{r}$ ← Reduction cost of this Matrix i.e 0.

For 1 − 3.

1 ∞	∞	∞	2	0	0
∞	3	∞	0	2	0
4 ∞	3	∞	∞	0	0
0 ∞	0	∞	12	∞	0
11	0		0	0.	= 0 + 11 = 0.

The cost =

$= c(1,3) + r + \hat{r}$ (11)

= 17 + 25 + 0.11)

= 53.

For Path 1-4.

∞	∞	∞	∞	∞
12	∞	11	∞	0
0	3	∞	∞	2
∞	3	12	∞	0
11	0	0	∞	∞
0	0	0	0	0

cost = $c(1,4) + r^{c(1)} + \hat{r}$

= 30 + 2 + 0.

= 40.2

For Path 1-5

∞	∞	∞	∞	∞	
12 (10)	∞	11 (9)	2 (0)	∞	2
0	3	∞	0	∞	0
12 (9)	3 (0)	12 (9)	∞	∞	3
∞	0	0	12	∞	0
0	0	0	0		5+0=5

cost = $c(1,5) + r^{c(1)} + \hat{r}$.

= 1 + 2 + 5

= 21

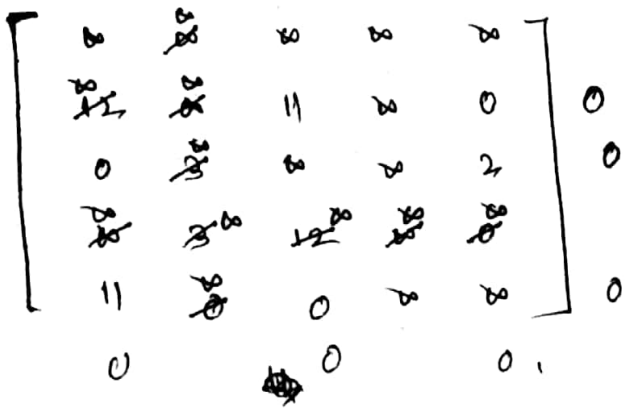
So on the state space tree it was found that the minimum cost is 25 i.e Path 1-4 will be explored next. Hence the 1-4 matrix will be treated as next Reduced matrix. i.e.

Reduced Matrix of Path 1 to 4.

∞	∞	∞	∞	∞
12	∞	11	∞	0
0	3	∞	∞	2
∞	3	12	∞	0
11	0	0	∞	∞

For path 4-2

Make 4th row ∞
2nd col. ∞.



The path is $1 \to 4 \to 2 \to 1$
& $2 \to 1$ also ∞.
i.e $(2,1) = \infty$.

Then check Reduced Matrix. (it is ok in this case).

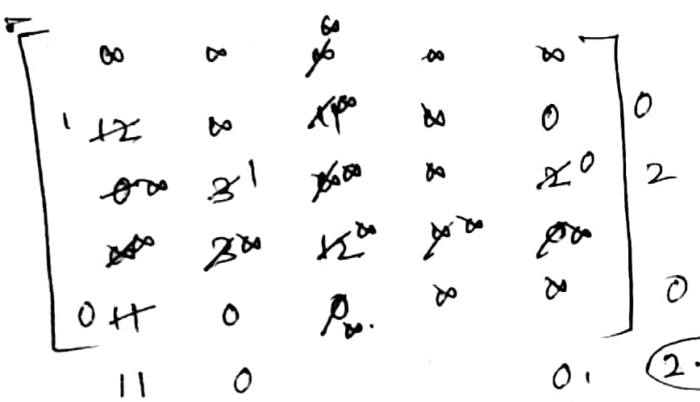
$$\text{Cost} = c(4,2) + r + c(4) + \hat{r}$$

$$= 3 + 25 + 0 = 28$$

$r \leftarrow$ cost of vertex 4 i.e = 25

$\hat{r} \leftarrow$ current reduction cost.

For Path 4-3.



Make 4th row ∞.
3rd col ∞.

$1 \to 4 \to 3 \to 1$

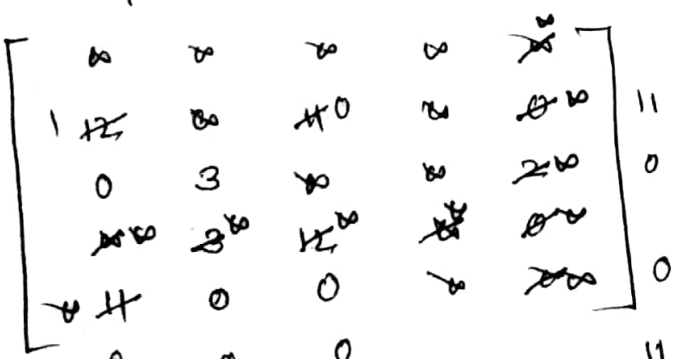
$(i.e)(3,1) = \infty$

$2 + 11 = 13 = \hat{r}$

$$\text{Cost} = c(4,3) + c(4) + \hat{r}$$

$$= 12 + 25 + 13 = 50.$$

For Path 4-5



Make 4th row ∞
5th col ∞.

$1 \to 4 \to 5 \to 1$
$(i.e)(5,1) = \infty$.

$11 + 0 = 11 (\hat{r})$

$$\text{Cost} = c(4,5) + c(4) + \hat{r} = 0 + 25 + 11 = 36$$

So on the state space tree it was found that the minimum cost is 28 i.e path from 4-2 will be explored next. So the Matrix of 4-2 will be treated as next Reduced Matrix. i.e.

Reduced Matrix of path 4-2.

∞	∞	∞	∞	∞
∞	∞	11	∞	0
0	∞	∞	∞	2
∞	∞	∞	∞	∞
11	∞	0	∞	∞

for 2-3

∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	3	∞	∞	3
∞	∞	∞	∞	
∞	0	∞	∞	0.
	0			

Make - 2nd row - ∞
3rd col - ∞.

1 → 4 → 2 → 3 → 1

(3, 1) - ∞.

3 + 0 = 3 ($\hat{r}$)

cost = c(2,3) + c(2) + $\hat{r}$.

= 0 + 28 + 3 = 31

r = cost of vertex 2 i.e = 28

$\hat{r}$ = Current reduction cost.

for 2-5

∞	∞	∞	∞	∞
1	∞	1	0	∞
0	3	∞	∞	∞
∞	∞	∞	∞	∞
∞	0	0	∞	∞

Make - 2nd row = ∞
5th col = ∞.

for 2-3.

Make - 2nd row ← ∞
3rd col ← ∞.

1 → 4 → 2 → 3 → 1

So (3,1) ← ∞.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & \infty & \infty & \infty \end{bmatrix} 2$$

11 2+11 = 13. ($\hat{r}$)

r = cost of vertex 2
i.e = 28

$\hat{r}$ = current Reduction cost.

$$\text{cost} = c(2,3) + \overset{r}{c(2)} + \hat{r}$$

$$= 11 + 28 + 13 = 52$$

for 2-5

Make - 2nd row ← ∞
5th col ← ∞

1 → 4 → 2 → 5 → 1

So (5,1) ← ∞

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix} 0$$

0 0 0 ($\hat{r}$)

$$\text{cost} = c(2,5) + c(2) + \hat{r}$$

$$= 0 + 28 + 0 = 28$$

So from the state space tree it was found that the minimum cost is 28, (i.e Path 2→5) will be explored next. So the matrix of 2→5 will be treated as next Reduced Matrix. i.e.

Reduced Matrix for path 2 → 5

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 6 & \infty & \infty & \infty & 2 \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix}$$

Path ⑤ for 5 to 3

Make 5th row ∞.
3rd col ∞.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ 4 & \infty & \infty & 4 & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

$1 \to 4 \to 2 \to 5 \to 3 \to 1$

& (3, 1) = ∞.

$O(\hat{r})$

r = cost of the vertex 5 i.e = 28

$\hat{r}$ = current reduction cost.

cost := $c(5,3) + c(5) + \hat{r}$.

$= 0 + 28 + 0 = (28)$

due to $(n-1)!$ paths → the complexity of TSP by using BB is $\Omega(2^n)$