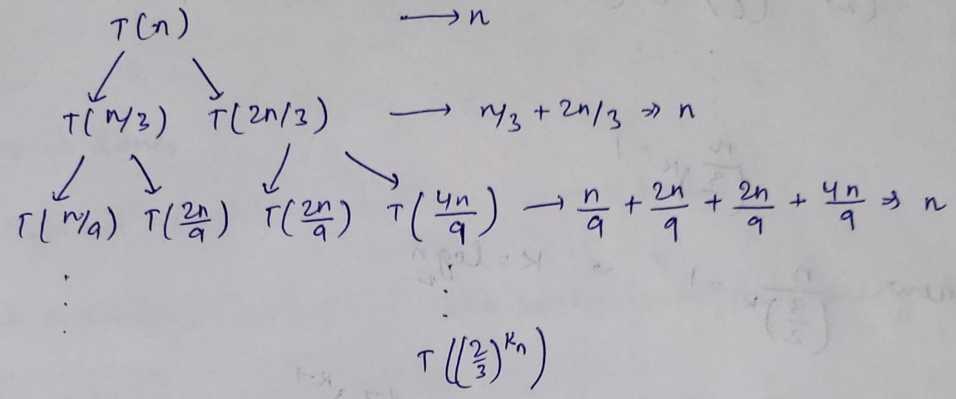
Name: Krishna Biswakarma.
Roll No.: 24CS60R71
Subject: Algorithm and Design Analysis
Date: 12/09/2024

---

① a) $T(n) = T(n/3) + T(2n/3) + n$

Sol)



This cost at each step is 'n' if we sum up the unit of all level till height 'K'.

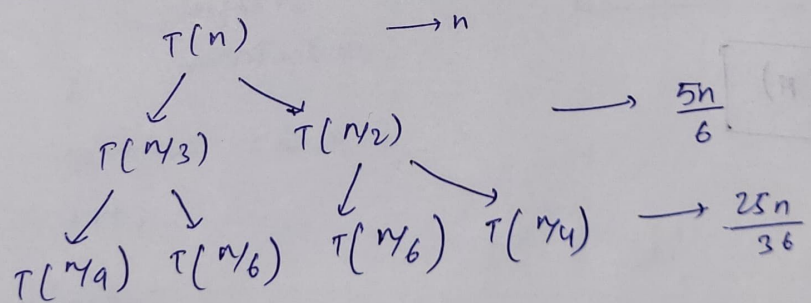
$$\left(\frac{2}{3}\right)^K n = 1$$

$$\Rightarrow n = \left(\frac{3}{2}\right)^K$$

Taking log, $K = \log_{3/2} n$

$$\boxed{T(n) = O(n \log_{3/2} n)}$$

b) $T(n) = T(n/3) + T(n/2) + n$



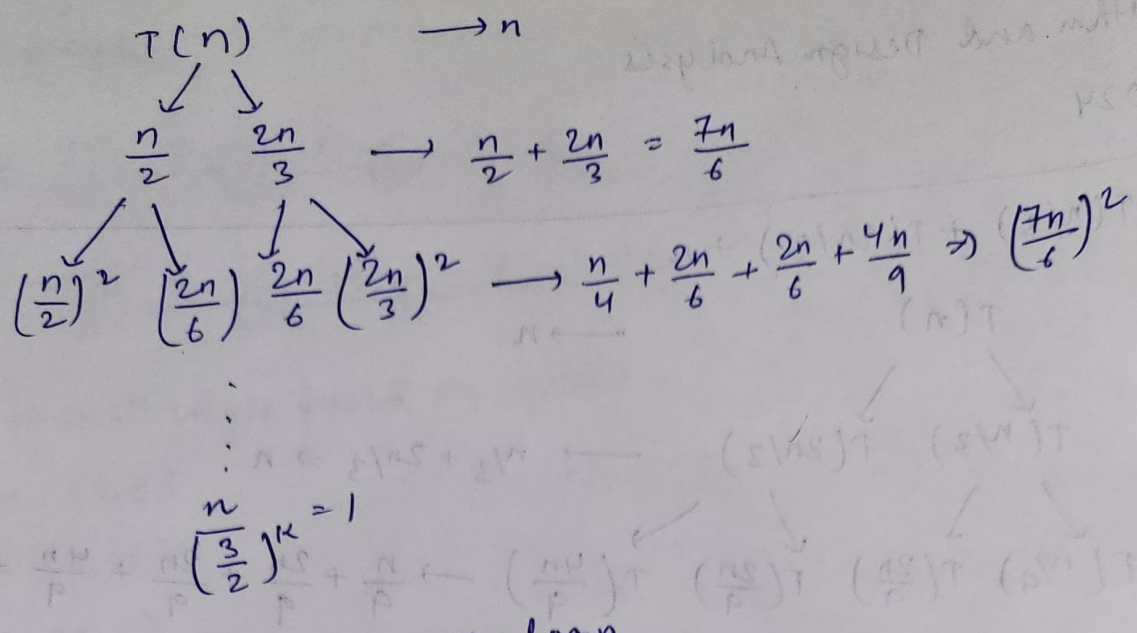
$$\text{Total Cost} = n + \frac{5}{6}n + \left(\frac{5}{6}\right)^2 n + \cdots + \left(\frac{5}{6}\right)^K n$$

$$= n\left(1 + \frac{5}{6} + \left(\frac{5}{6}\right)^2 + \cdots + \left(\frac{5}{6}\right)^K\right)$$

↘ This is a decreasing G.P.

$$\boxed{T(n) = O(n)}$$

c) $T(n) = T(n/2) + T(\frac{2n}{3}) + n$



So, now $\frac{n}{(\frac{3}{2})^k} = 1 \Rightarrow k = \log_2 n$

Now, $n + \frac{7n}{6} + (\frac{7}{6})^2 n + (\frac{7}{6})^3 n + \cdots + (\frac{7}{6})^{k-1} n$

$\Rightarrow n\left(1 + \frac{7}{6} + (\frac{7}{6})^2 + \cdots + (\frac{7}{6})^{k-1}\right)$

~~$\Rightarrow n\left((\frac{7}{6})^{k-1} \cdot (\frac{7}{6})\right)$~~

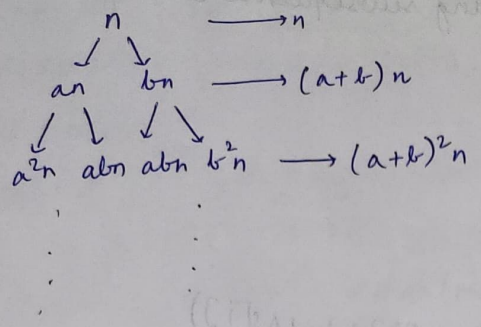
$\Rightarrow n\left(\frac{1^{7/6 - 1}}{\frac{7}{6} - 1}\right)$

$\Rightarrow n\left(\frac{1}{1/6}\right)$

$\Rightarrow 6n$

$\therefore \boxed{T(n) = O(n)}$

② $T(n) = T(an) + T(bn) + n$



Now, total work done,

$\Rightarrow n((a+b)^0 + (a+b)^1 + (a+b)^2 + \ldots\ldots + K \text{ times})$

As given in question, $a+b<1$ ∴ the series is in decreasing GP.

∴ $\boxed{T(n) = O(n)}$

③ Algorithm:

```
minStops(dist[], n, V){    // n → no. of fuel pumps
    int stops = 0;         // V → Maximum distance the car can drive.
    int currFuel = V;      // dist[] → distance between fuel pumps.
    int i = 0;
    while (i < n){
        if (dist[i] > currFuel){
            stops++;
            currFuel = V;
        }
        currFuel -= dist[i];
        i++;
    }
    return stops;
}.
```

$\boxed{T(n) = O(n)}$

•> Proof of Optimality-

The greedy strategy guarantee optimality because at each step, it chooses the furthest fuel pump that can be reached with the current fuel level. By doing this, it minimizes the no. of stops needed. If we were to stop at a closer fuel pump than necessary, it would increase the total no. of stops without providing any additional benefits.

(4) i) Recursive defination:

LAS[i] = Length of longest ascending subsequence ending at index i.

```
if (i==0)   LAS[i] = 1
else {
    for (j=0 → i-1)
        if (A[j] < A[i])
            LAS[i] = max(LAS[j]+1, LAS[i])
```

ii) Recursive Algo: $T(n) = O(2^n)$

```
int LIS(A[], n, i) {
    if (i==0) return 1;
    int maxlen = 1
    for(int j=0; j<i; j++)
        if (A[j] < A[i])
            maxlen = max(maxlen, 1+LIS(A,n,j));
    return maxlen
}
int findLIS(A[], n) {
    res = 1;
    for(int i=0; i<n; i++)
        res = max(res, LIS(A,n,i));
    return res;
}
```

Yes, there will be overlapping subproblems. The recursive calls for LIS(i) and LIS(j) for various values of 'i' and 'j' might overlap, making this inefficient without memoization.

(5) i) Algorithm –

Step 1: tasks $\{a_1, a_2, \ldots, a_n\}$ with corresponding processing times $\{p_1, p_2, \ldots, p_n\}$

Step 2: Sort tasks based on processing time in ascending order.

Step 3: For each task in the sorted order:
a. Schedule the task.

Step 4: Scheduled task order.

Time Complexity:

Sorting : $O(n \log n)$
Scheduling the tasks in linear i.e; $O(n)$.

$\therefore$ $T(n) = O(n \log n)$

ii) Proving Optimality of the Algorithm –

- Completion Time Formula – Let the completion time of a task a_j be c_j. The objective is to minimize the average completion time, which is,

$$\frac{1}{n} \sum_{i=1}^{n} c_i$$

- Exchange Argument – Suppose the tasks are scheduled in an order that is not the SPT order and at least two tasks, say a_i and a_j are out of order where $p_i > p_j$ and a_i is scheduled before a_j.
By swapping these two tasks, the completion time of a_j decrease and that of a_i increase. However since $p_i < p_j$ ~~and a_i is scheduled before~~ a_j the total reduction in the completion time for a_j is greater than the increase in the completion time for a_i. Hence, the average completion time decrease by this swap.

iii) Memoized Solution -

```
int LIS(A[], n, i, dp[])
    if (dp[i] != -1) return dp[i];
    int maxlen = 1;
    for (int j=0; j<i; j++)
        if (A[j] < A[i])
            maxlen = max(maxlen, 1 + LIS(A, n, j, dp));

    dp[i] = maxlen;
    return dp[i]
}
int findLIS (int A[], int n){
    dp[n];
    fill(dp, dp+n, -1);
    int res = 1;
    for (int i=0; i<n; i++)
        res = max(res, LIS(A, n, i, dp));

    return res;
}.
```

$T(n) = O(n^2)$

v) Iterative solution -

```
int LIS(int A[], int n) {
    int dp[n];
    for (int i=0; i<n; i++)
        dp[i] = 1
    for (int i=1; i<n; i++)
        for (int j=0; j<i; j++)
            if (A[j] < A[i])
                dp[i] = max(dp[i], dp[j]+1);

    int maxLIS = 0
    for (int i=0; i<n; i++)
        maxLIS = max(maxLIS, dp[i]);

    return maxLIS;
}.
```

$T(n) = O(n^2)$

- Inductive Argument - By repeatedly applying this swapping argument, we can show that any scheduled that is not sorted by ascending processing times can be improved by performing such swaps, leading to the optimal solution where tasks are scheduled in increasing order of ~~swaps~~ processing time.

∴ The SPT algorithm is optimal for minimizing the average completion time.

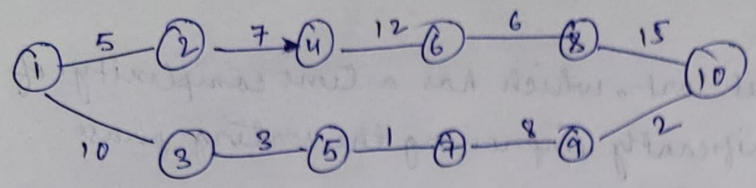
(6) For only single -ve weight we can use modified version of Dijkstra algo modified version.

```
Dijkstra_mod(G, s, g) {
  1. dist[] = ∞
  2. dist[s] = 0
  3. Run dijkstra algo on G starting from s:
        Create a priority_queue Q.
        Insert node s into Q with priority 0.
        while Q is not empty:
            u = extract_min(Q)
            for each neighbour v of u:
                if dist[u] + weight(u,v) < dist[v]:
                    dist[v] = dist[u] + weight(u,v);
                    update Q with new dist[v] value
  4. Identify the -ve edge (u,v) with weight ~~w~~ w_uv < 0
  5. Perform ~~one~~ one relaxation step for the -ve edge:
        if dist[u] + weight(u,v) < dist[v]:
            dist[v] = dist[u] + weight(u,v)
  6. Return dist[g]
}
```

Time Complexity: O(ElogV)

⊙ Working –

Assume a graph,



Now, we bucket edges based on weight:

1 → (5,7)
2 → (9,10)
3 → (3,5)
5 → (1,2)
6 → (6,8)
7 → (2,4)
8 → (7,9)
10 → (1,3)
12 → (4,6)
15 → (8,10)

Now, we will be adding edges to MST:

Processing Bucket 1:
Edge (5,7) – Weight: 1

Processing Bucket 2:
Edge (9,10) – Weight: 2

Processing Bucket 3:
Edge (3-5) – Weight: 3

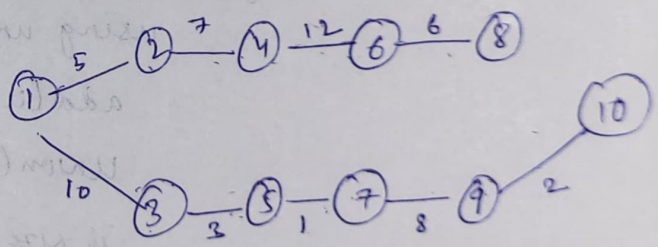
Processing Bucket 5:
Edge (1-2) – Weight: 5

Processing Bucket 6:
Edge (6-8) – Weight: 6

Processing Bucket 7:
Edge (2-4) – Weight: 7

Processing Bucket 8:
Edge (7-9), Weight: 8

Final MST –



Processing Bucket 10–
Edge (1,3), Weight – 10

Processing Bucket 12–
Edge (4,6) Weight – 12

Total weight = 54

(7) We can use an efficient approach other than traditional algorithm like Prims and Kruskal's for finding MST. We can use Bucket-based Kruskal's Algorithm.

Here, we will be using Bucket sort, which has a time complexity of O(E) hence will help in significantly improving the sorting phase.

•> Algorithm -

```
Kruskals_modification(V, edges){
    // V → Vertices.
    // Edges {u, v, weight}
1. Create 101 empty buckets for weight 1 to 100
2. For each edge (u,v, weight) in edges:
        place edge in bucket corresponding to the weight.
3. Initialize unionfind structure for 'V'-vertices
4. Initialize empty list MST to store the edges of the MST.
5. for (weight i=1 ⟶ 100){
        For each edge (u,v) in current bucket:
            if (u and v are in different components
                using unionfind):
                add (u,v) to MST
                Union(u,v) in Union find
                if size of MST equals V-1:
                    return MST
6. return MST.
}.
```

•> Time Complexity - (Amortized TC.)

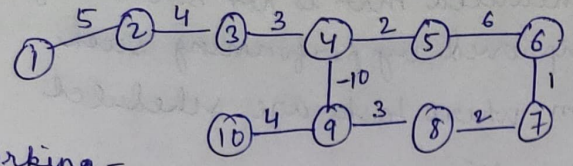
i> Union Find = $O(\alpha(n))$ where α = Inverse Ackermann function.

ii> Edge sorting = $O(E)$

iii> Total time taken by Union Find = $O(E\alpha(V))$

∴ Total TC = $O(E + E\alpha(V))$

Assume a graph,



⊙ Working -

- Staring from node 1 :
- Find the shortest path to all other nodes while ignoring the -ve edge.

Initial Dijkstra result -

dist[1] = 0, dist[2] = 5, dist[3] = 9, dist[4] = 12, dist[5] = 14,
dist[6] = 20, dist[7] = 21, dist[8] = 23, dist[9] = 26, dist[10] = 30.

Having the -ve edge (4, 9, -10) -
Relax this edge : New distance to node 9 through -ve edge is
dist[4] + (-10) = 12 - 10 = 2.
As a result new path through node 9 will update the distance to node 10 as well dist[10] = dist[9] + 4 = 2 + 4 ⇒ 6.

·> Final Distance -

dist[1] = 0, dist[2] = 5, dist[3] = 9, dist[4] = 12, dist[5] = 14
dist[6] = 20, dist[7] = 21, dist[8] = 23, dist[9] = 2, dist[10] = 6

∴ Shortest path from 1 to node 10 = (6).

⊙ Proof of correctness :

- Dijkstra algo correctly finds the shortest path in graph with +ve edges -
- The single relaxation for -ve edge can only affect path that pass through it. By performing one additional relaxation step for the -ve edge, we ensure that all paths that could be affected by it are updated.
- Bellman Ford would typically be used for graphs with arbitary -ve edges but since we only have one -ve edges, a single relaxation step after Dijkstra is sufficient to guarantees correctness.

∴ The algo. is correct in all cases.