Prakhar Jain

2022510gg

CS-203 Theory assignment - II

Q1

```
solve ( nums[], int previous, int ind, dp[][]){

    if (ind < 0) {
        return 0; }

    if (dp[ind][prev+1] != -1) {
        return dp[ind][prev+1];
    }
    int take
    if (prev == -1 || nums[ind] > nums[prev]) {
        take = 1 + solve(nums, ind, ind+1, dp);
    }
    int not take = 0 + solve (nums, prev, ind+1, dp);

    return dp[ind][prev+1] = max(take, not take);
    }

length of LIS (vector num[]) {
        return solve(nums, -1, nums.size()-1,
                    dp);
```

Q3 To construct an optimal huffman code, we need to follow some steps

① Create list of nodes, where each node represent a symbol and its freq.

② Build a min-heap of these nodes

③ ⓐ Remove two nodes of lowest frequencies

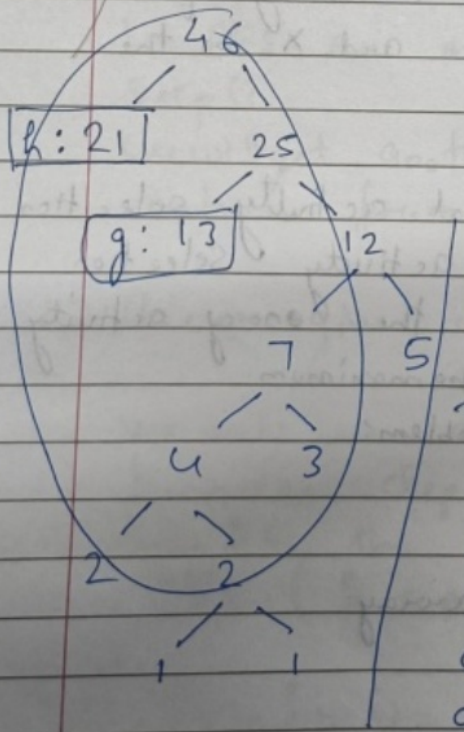
ⓑ Create a new node with freq equal to sum of two

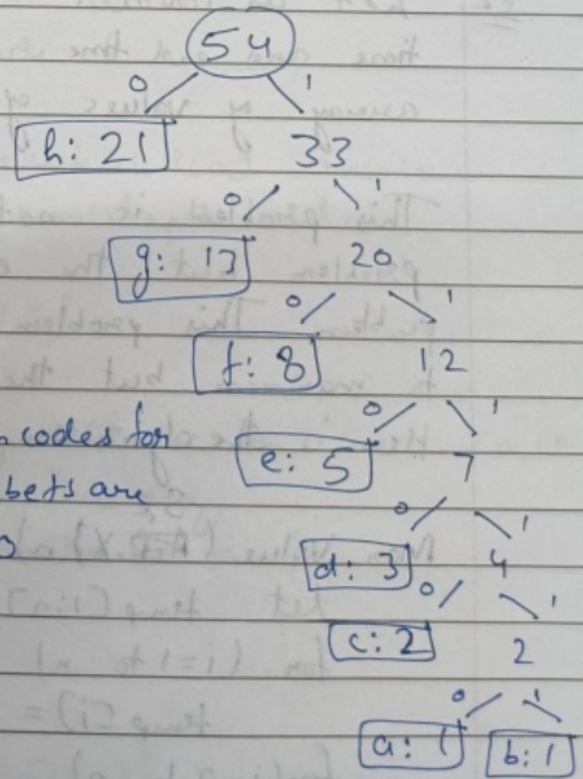
④ ⓒ Insert that new node back into min heap (P.Q)

④ Repeat the same process for all nodes

Here are given a:1, b:1, c:2, d:3, e:5, f:8, g:13, h:21.

The resulting huffman tree for the given nodes





The huffman codes for the alphabets are

a: 111111 0

b: 111111 1

c: 11110

d: 1111 0

e: 1110

f: 110

g: 10

h: 0

By looking in the pattern of the huffman code we can generalise the huffman code for n fibonacci numbers

n^{th} fib. no. = 0
$(n-1)^{th}$ " " = 10
$(n-2)^{th}$ " " = 110
$(n-3)^{th}$ " " = 1110
⋮
3^{rd} " " = 111--- (n-3) times 0
2^{nd} " " = 111-- (n-2) times 0
1^{st} " " = 111--- (n-2) times 1

Q2 Let us consider S and E as a array of start time and end time of activities and X be the array of values of activities

This problem is not the normal activity selection problem but the extension of activity selection problem. This problem don't want the no. of activity to maximum but the value to be maximum
Here is the algorithm for this problem:

```
                S,E
Max_Value ({A,B,X},n)
    Let temp[1:n] be the array
    for (i=1 to n)
        temp[i] = x[i]
    for(i=2 to n)
      for(j= 1 to i-1)
          temp[i] = max{temp[i] + X[1], temp[i]}
```

maxx = -∞
for(i=1 to n)
maxx = max{ maxx, kup[i] }

Return maxx

The required time complexity for this code will be O(n²)

(Q4) D.F.S is used to calculate the ~~depth~~ diameter of tree, we start from root to calculate the depth of every node
Step ①
Calculate the max depth and ~~de~~ that node
Step ②
Use DFS on max depth node
Step ③
we will get another node which is at maximum distance from initial max depth node

We are given with T=(V,E)

we will use a adjacency matrix to store all the nodes depth in a depth array
here A be the adjacency matrix (n×n)
DFS (A, s, p, d)

for (i = 1 to n)
if (A[s][i] = A[p][i])
continue
Else if A[s][i] == 0
continue

d[i] = d[s] + 1
~~DFS(i,s)~~ // recursively call for
DFS(A, i, s, d)

Now, we get all the node's depth in the depth array. Now we have to calculate diameter of tree

Diameter_Calculation(A)
D[1:n] be the ~~depth~~ array
for i = 1 to n
D[i] = 0 // giving value 0 to all elements
DFS(A, 1, -1, D) // calling DFS func^n for root node

maxx = -1
maxx_node = 0
for (i = 1 to n)
if maxx < D[i]
maxx = D[i]
~~max_do~~
maxnode = i
D[i] = 0 // again make D[i] 0

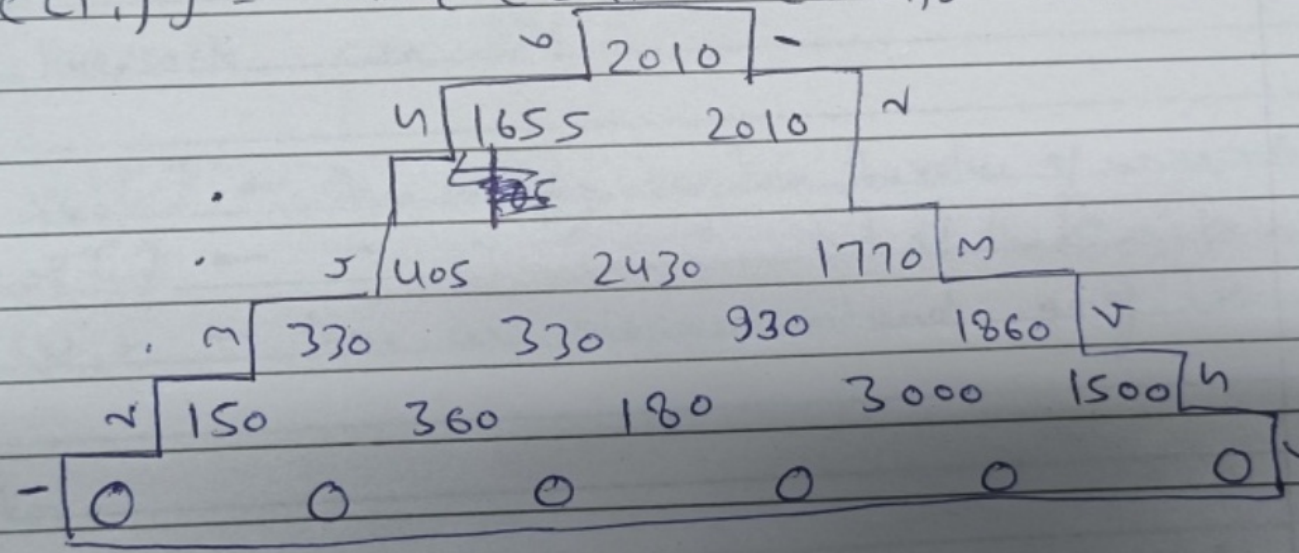
DFS(A, maxx_node, -1, D)
maxx = -1
for(i = 1 to n)
if max_ < D[i]
~~max_do~~
maxx = D[i]
max_node = i
Return maxx;

Date / /

(5) We have to calculate optimal parenthesization of matrix {5, 10, 3, 12, 5, 50, 6}

We know that ~~recursive~~ to calculate this we use a formula

$$c[i,j] = \min\{ c[i,k] + c[k+1,j] + d_{i-1} \times d_k \times d_j \}$$



Algorithm.

⑥ Fib (n)

```
let dp[1:n] be new array
dp[1] = 1
dp[2] = 1
for ( i = 3 to n)
    dp[i] = dp[i-1] + dp[i-2]

return dp[n]
```

⇒

DOMS Page No.
Date / /

(7) 0-1 knapsack is very famous problem which is used to find the maximum profit or amount that we can fill in the knapsack.
We are given two array one for the weight another is for value and W i.e maximum weight that a knapsack can fill

① val[] ← this array contain the value of corresponding wt
② wt[] ← " " " wt of the i^{th} n items
③ W, n ← these are maximum wt and no. of items.

Algorithm

```
int Knap_Sack ( int wt[], int val[], int W, int n){
    let dp[0: max W][0: max n] be a new matrix
    for i = 0 to n
        for j = 0 to W
            dp[j,i] = -∞

    if ( n == 0 || W == 0)
            return 0
    if( dp[W][n-1] != -∞)
            return dp[W][n-1]
    else if( W <= wt[n-1])
        return dp[W][n-1] = max ((val[n-1] +
                                Knap_Sack(wt,val, W-wt[n-1],
                                n-1),
                                Knap_Sack(wt,val, W,n-1))
    else {
        return dp[W][n-1] = Knap_Sack (val, wt, W, n-1)
```