DAA End Sem Exam

CS18B012
E. Nitin

1) ~~c < b < a < d~~ ~~c < a < b < d~~

1) $c < a < d < b$

2) a) $f = \Omega(g)$

b) $f = \Theta(g)$

c) $f = O(g)$

d) $f = \cancel{\Theta(g)}\ \Omega(g)$

3) a) Claim :- If $A[n/2] < A[n/2 - 1]$ then one of the first $\frac{n}{2} - 1$ elements of A is a local maxima.

Proof :- Proof by contrapositive :-

If none of the first $\frac{n}{2} - 1$ elements of A is a local maxima then $A[n/2] > A[n/2 - 1]$

We can proove this statement.

→ ~~If~~ For an element to be local maxima, it should be larger than the element to its left and right.

→ So as none of the first $\frac{n}{2} - 1$ elements of A is a local maxima so $\frac{n}{2} - 1$ th element is also not local maxima.

→ That means $A[n/2 - 1]$ is less than the element to its right and left.

$\therefore A[n/2 - 1] < A[n/2]$

b) Pseudocode :-

A is the array ; l and r represent start and end indices of the array.

```
Function findLocalMax(A, l, r):
    if l > r
        return -1
    mid = ⌊(l + r)/2⌋
    if LocalMax(A, mid)
        return A[mid]
    if A[mid] < A[mid+1]
        return findLocalMax(A, mid+1, r)
    return findLocalMax(A, l, mid)
```

```
Function LocalMax(A, mid):
    if A[mid-1] < A[mid] && A[mid+1] < A[mid]
        return true
    else
        return false
```

→ This algo is correct. We can say this by the ~~firs~~ claim in the first part (a). ~~n~~ mid i.e if A[mid] $\geq$ A(mid+1] then one of the first mid elements is a local maxima so we choose that subpart i.e ~~for first to mid we~~ first n/2 elements and find mid in that again and proceed till we get Local Maxima element

~~→ T(N) = T(N~~

→ $T(n) = T(n/2) + 1$

Scanned with CamScanner

4) Function MAJ(A, l, h)

```
if l == h
    return A[l]

mid = ⌊(l+h)/2⌋

left = MAJ(A, l, mid)
right = ~~MAJ(A, mid, h)~~ MAJ(A, mid+1, h)

  if left == right
      return left

Lcount = count number of times left is there
         between l and h in array A

Rcount = count number of times right is there
         between l and h in array A

  if Lcount > Rcount
      return left

return right
```

Scanned with CamScanner

~~Algo~~

→ Recurrence relation :-

$$T(n) = \underbrace{2T\left(\frac{n}{2}\right)}_{\text{Because we are dividing into 2 subproblems i.e from } l \text{ to mid \& mid+1 to } h.} + \underbrace{2n}_{\text{Because we are doing 2 linear scans to find the Lcount \& Rcount}}$$

→ We can solve the recurrence by master theorem.

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$$

here $a = 2$, $b = 2$, $d = 1$

$d = 1$ & $\log_b a = \log_2 2 = 1$

As $d = \log_b a$.

$\therefore T(n) = O(n^d \log_b n)$

$$\boxed{\therefore T(n) = O(n \log_2 n)} \rightarrow \text{Time Complexity}$$

5) Given company wishes to buy n specific licenses whose costs are $n_1, n_2, n_3 \dots n_n$.

Company is allowed to buy only 1 license per month.

Every month price of each license doubles

a) As every month the price of each licence doubles we should ~~first~~ always pick the licence with higher cost.

Algorithm :-

→ First we sort the licenses based on the costs.

→ For every month we pick the ~~[illegible]~~ license with higher cost.

Eg:- $n_1 < n_2 < n_3 \dots < n_n$

First we pick n_n for 1st month

After doubling

$2n_1 < 2n_2 < \dots < 2n_{n-1}$ (order is not changed by doubling)

We pick $2n_{n-1}$ for 2nd month and so on.

b) This is the optimal solution because everytime we are picking the license with higher cost so this ensures that we will not pick this license in future as ~~it~~ its cost ~~to~~ keeps on doubling.

Say there are $n_1 < n_2 < n_3 \dots < n_n$

If we pick ~~n_1~~ any ~~of~~ licence other than n_n then in future n_n ~~[illegible]~~ will be doubled & doubled and if we pick ~~[illegible]~~ n^{th} license in future then it will cost so much for us.

Solving recurrence :-

$T(n/2) = T(n/4) + 1$ $\rightarrow$ $T(n) = T(n/4) + 2$

$T(n/4) = T(n/8) + 1$ $\rightarrow$ $T(n) = T(n/8) + 3$

$\vdots$ $\vdots$

~~T(n~~

$T(n) = T\left(\frac{n}{2^k}\right) + k$

$\frac{n}{2^k} = 1$

$\log n = k \log_2 2$

$k = \log n$

$\therefore T(n) = T(1) + \log n$

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

c) → First we sort the n licenses so that takes $O(n\log n)$

& then for every month we pick highest cost licence

So $O(1)$ for that.

→ For picking all the licences for all ~~moth~~ months

$$\underbrace{O(1) + O(1) + \cdots + O(1)}_{n \text{ times}}$$

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

i.e

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

b) Greedy Algo for Max. spanning tree :-

→ Let G be graph.

→ Sort edges of G in decreasing order of weights.

→ Let H be set of edges ~~containing max~~ of max spanning tree.

→ Add first edge to H

→ Add next edge to H if and only if it doesnot form a cycle in H.

→ If H has n-1 edges then ~~output~~ stop & output H.

This is basically applying Kruskals after sorting edges in descending order.

Proof of Correctness:-

→ We can use cut property by ~~neg~~ multiplying ~~negative~~ −1 to all the edges ~~&~~ inorder to proove.

→ In the end we get a minimum spanning tree but now again multiply all edges of the obtained MST by −1 then we can see that we have got a maximum spanning tree.

Cut property tells that :-

~~If we make a cut in~~

The minimum cost edge e in ~~MST~~ cut $(S, \bar{S})$ should be present in a MST.

Time Complexity :-

Time Complexity = $O(m \log n)$ where m is no. of edges
n is no. of vertices.

Scanned with CamScanner

7) DP Algorithm :-

```
Function LCS(x, y, n, m)
    mat [n+1] [m+1]
    max_length = 0
    for i = 0 to n
        for j = 0 to m
            if i == 0 or j == 0
                mat [i][j] = 0
            else if x[i-1] == y[j-1]
                mat [i][j] = mat[i-1][j-1] + 1
                ~~max_length~~
                max_length = max(max_length, mat[i][j])
            else
                mat [i][j] = 0
    return max_length
```

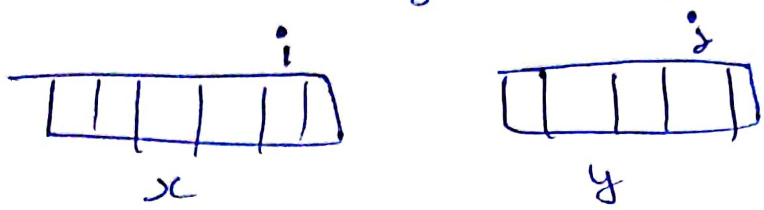
Time Complexity = $O(n \times m)$

As there are 2 for loops, outer loop runs from 0 to n i.e n times & inner loop runs ~~for~~ from 0 to m i.e m times

$\therefore$ Time Complexity = $O(nm)$

Subproblems :-

→ There are 2 strings x & y



There are 2 sub problems :-

→ if $x[i-1]$ ~~=~~ is equal ~~to~~ to $y[j-1]$ i.e last but one character of x (just before i) and y (just before j) are equal

→ if $x[i-1]$ is not equal to $y[j-1]$

8) a) ~~DP~~ DP Algorithm :-

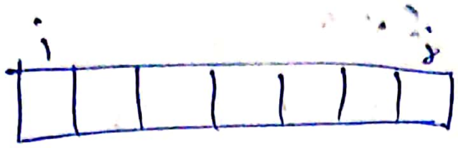
Function LPS(S, n)

S is binary string
n is length of S.

```
mat[n][n]

for i=0 to n
    mat[i][i] = 1

for k=2 to n
    for i=0 to n-k+1
        j = i+k-1
        if S[i] == S[j] && k==2
            mat[i][j] = 2
        else if S[i] == S[j]
            mat[i][j] = mat[i+1][j-1] + 2
        else
            mat[i][j] = max(mat[i][j-1],
                            mat[i+1][j])

return mat[0][n-1]
```

b) ~~Sup~~ Subproblems :-

→ S is a string



S

→ If S[i] is equal to S[j]
we increase the length by 2 & check for LPS(i+1, j-1) ~~S[i+1]~~
~~LPS(S[i+1], S[j-1])~~

→ If S[i] is not equal to S[j] there are 2 possibilities :-

LPS(i, j-1) and LPS(i+1, j)

We take max of them.

↳ this checks for j^{th} element by ignoring element at i^{th} index.

this ~~answers~~ checks for i^{th} element by ignoring element at j^{th} index.

d) Time Complexity :-

→ run 2 for loops

Time Complexity = $O(n) + O(n^2)$

↓

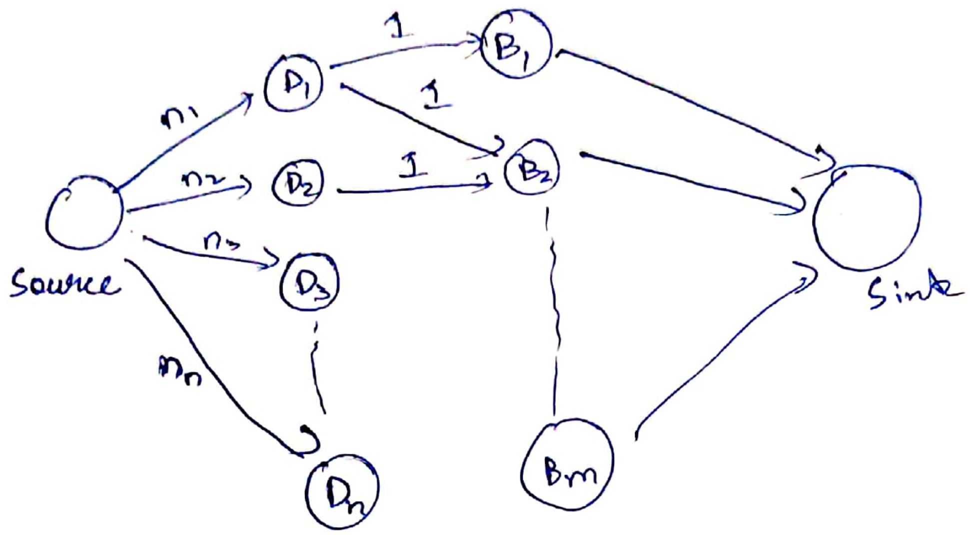
running a for loop for assigning lengths as 1 for each character. As a single character itself is a palindrome.

$\boxed{\therefore \text{Time Complexity} = O(n^2)}$

9) ~~a~~

Let us denote nodes of doctor as $D_1, D_2 \dots D_n$

Let us denote nodes of days as $B_1, B_2, \dots B_m$



→ The edge indicates doctor requesting holidays for that day

We can assign their capacities as 1.

1 means ~~assigned~~ holiday assigned

0 " " not assigned

→ We restrict days to sink capacity to $n/2$ so that atleast $n/2$ doctors are covered each day.

→ Source to doctor has capacity of n_i

→ Edge D to B indicates doctor requesting for that day.

We should maximize the flow.