Problem set - 1

Advanced Algorithm Design.

Name :- MALLASALLESH
Roll number - 2021101106.

① $T_1(n) = a \cdot T_1(n/b) + b \cdot n.$

$T_1(n/b) = a \cdot T_1(n/b^2) + n.$

$T_1(n/b^2) = a \cdot T_1(n/b^3) + n/b.$

$\vdots$

$T_1(n) = a^2 T_1(n/b^2) + an + bn.$

$= a^3 T_1(n/b^3) + \frac{a^2 n}{b} + an + bn.$

$\vdots$

Let say this goes until $n/b^K \simeq 1$

$T_1(n) = n(b + a + \frac{a^2}{b} + \cdots$ (K terms) $\cdots) + a^K \cdot T_1(n/b^K).$

(Geometric progression with common ratio a/b.)

$n \approx b^K$

$K = \log_b n.$

$T_1(n) = n \cdot b \left(\frac{(a/b)^K - 1}{a/b - 1} \right) + a^{\log_b n} \cdot T_1(1)$

given $T_1(1) = 1$

so $T_1(n) = a^{\log_b(n)} + bn\left[\frac{(a/b)^{\log_b n} - 1}{a/b - 1}\right]$

~~Hy~~ ~~$T_2(n)$~~ ~~$\log_a(n)$~~ ~~$an\left[(b/a)^{\log_a n} - 1\right]$~~ ~~$b/a = 1$~~

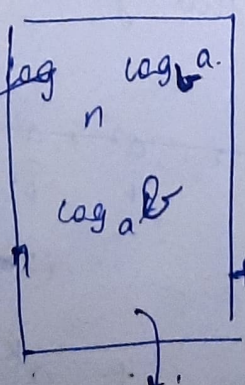
because only a, b are interchanged from T_1 to T_2.

$$T_2(n) = b^{\log_a n} + an\left[\frac{1 - (b/a)^{\log_a n}}{1 - (b/a)}\right].$$

$$\left(\frac{a}{b}\right)^{\log_b n} \rightarrow n^{\log_b a/b}$$

$$= n^{\log_b a - 1}.$$

$$\left(\frac{b}{a}\right)^{\log_a n} \rightarrow n^{\log_a b/a} \Rightarrow n^{\log_a b - 1}.$$

$T_1(n) =$  $+ n(b + a + a^2/b + \dots$ k terms)

$T_2(n) =$ $n(a + b + b^2/a + \dots$ x terms).

$\log a \geq b$

$\log a \geq \log b$.

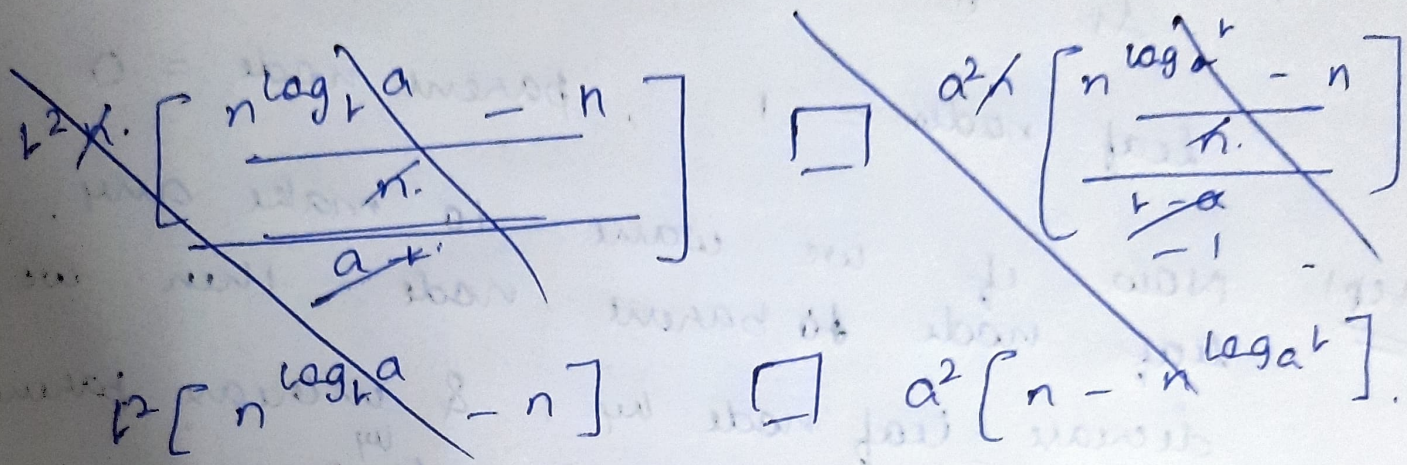
$\log a \cdot \log a \geq \log b \cdot \log b$

$\frac{\log a}{\log b} \geq \frac{\log b}{\log a}$

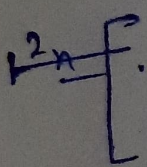
$\log_b a \geq \log_a b$

$n^{\log_b a} \geq n^{\log_a b}$ — (1)

$a \geq b \Rightarrow \log_a b \leq 1$.



$b^2 \left[n^{\log_b a} - n \right]$ $\square$ $a^2 \left[n - n^{\log_a b} \right]$.



2nd term of $T_1(n) \Rightarrow$ $n(b + a \cdot + a^2/b + \text{---} \cdot \text{k terms})$

$T_2(n) \Rightarrow$ $n(a + b + b^2/a + \text{---} \text{ k terms})$

$k < k$.

because.

$a \geq b$.

so $\frac{n}{a^2} \approx 1$ for smaller value.

$a \geq b$.

Thus 2nd term also for $T_1(n)$ is greater

Thus $T_1(n)$ is greater than $T_2(n)$ &
$T_1(n)$ grows larger as $n \uparrow$ than $T_2(n)$

(2)

step (1) Initialise one vector ~~two arrays~~ or ~~arrays~~, - visited with all value set to 0 ~~dfsvisited~~

step (2) Then call the function dfs taking some node as starting node. (we will also store parent node as a parameter) → source.

step (3) Now we will make that node as visited (i.e. make it equal to 1). Now we will check for elements adjacent to it.

step (4) If not visited. then call the dfs again repeating previous steps else if check the element is equal to parent ~~if not else~~ return 1 ~~true false~~ else continue.

step (5) If nothing returned return ~~false~~ (or 0)

In step (4) while calling dfs taken parent as previous source node & source.node is x (element adjacent to previous source node)

↳ int dfs (vector<int> adj[], int parent, int source ~~int visited[~~ : vector<int> &visited)

```
{
    visited[source] = ~~true~~ 1;
    for(auto x : source)
    {
        if (visited[x] == 0)
        {
            dfs (adj, source, x, visited);
        }
        else if (x != parent) return 1;
        ~~return true;~~ else continue;
        ~~break;~~
    }
    return ~~false~~ 0;
}
```

③

Let us first consider 1 node.

· As it has no children.

leaf nodes = 1 , parent node = 0

Step1 Now if we want to make any leaf node to parent node then we decrease leaf node by 1 & increase parent node by 1 & increase leaf node by 2.

leaf nodes = 1 − 1 + 2 ⇒ 2
parent nodes = 0 + 1 ⇒ 1.

so now we go on repeating step ① — (1)

from step1 ~~node~~ : increase in leaf node = −1 + 2 ⇒ 1
increase in parent node = 1 with 2 children.

for every step the increase in leaf, parent nodes are same. — (2)

Initially there is 1 leaf node & 0 parent nodes

Now by repeating how many step we want

we get leaf nodes = 1 + x = 1 + no. of parent nodes.

Thus any binary tree with parent node having exactly 2 children is number of nodes with two children is exactly one less than number of leaf nodes.

we may think there can be parent node with 1 child.

for these,

If we make any leaf node parent with 1 child the increase in leaf node = -1+1 = 0

Thus no affect to ~~proof~~ by parent node with 1 child. to the relation below.

Number of leaf node = 1 + number of parent node with two children

Thus proved =.

(4) ~~tt~~ ~~ttt~~ ~~call Bfs~~, ~~Dfs~~ ~~from~~ ~~same~~ ~~node for~~ the same node ~~it~~.

~~clarify~~. we called bfs, dfs from

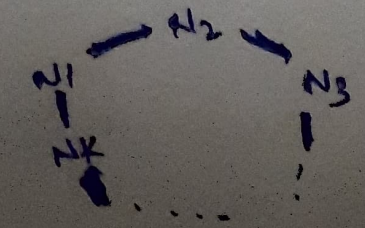
✓ Bfs gives a tree :- first gives level 1 of tree. (which ~~given~~ contains root

then it gives level 2 of tree

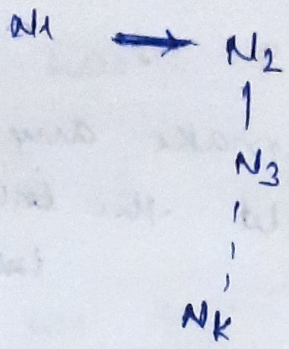
and so on ---

✓ Dfs :- recursive code which uses dfs function call if node isn't visited.

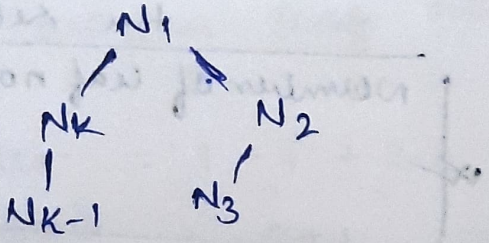
Let say, there is a cycle in G



In this when we do DFS we get



by doing bfs we get



clearly we didnot get same result.

so no cycles in G

$\Rightarrow$ G is a tree.

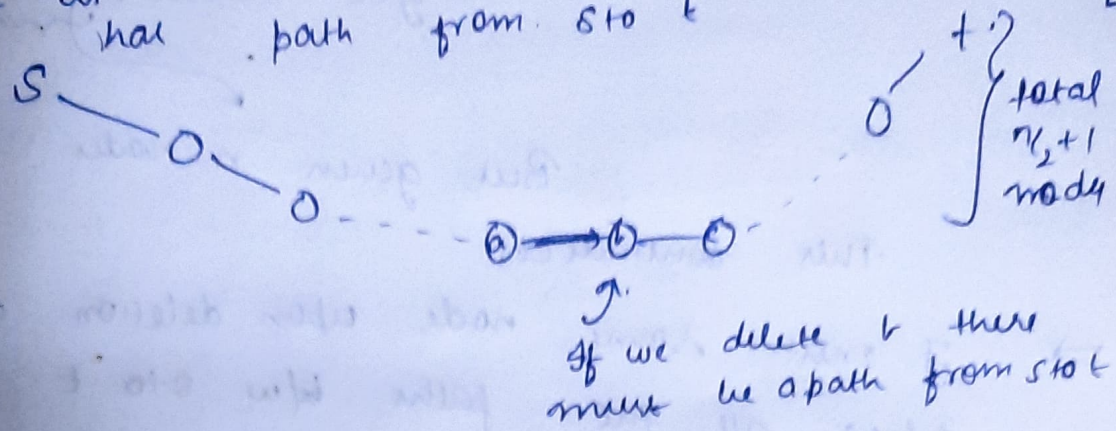
As G is tree. its dfs, bfs will be clearly same as it

$$\boxed{G = T}$$

Hence proved

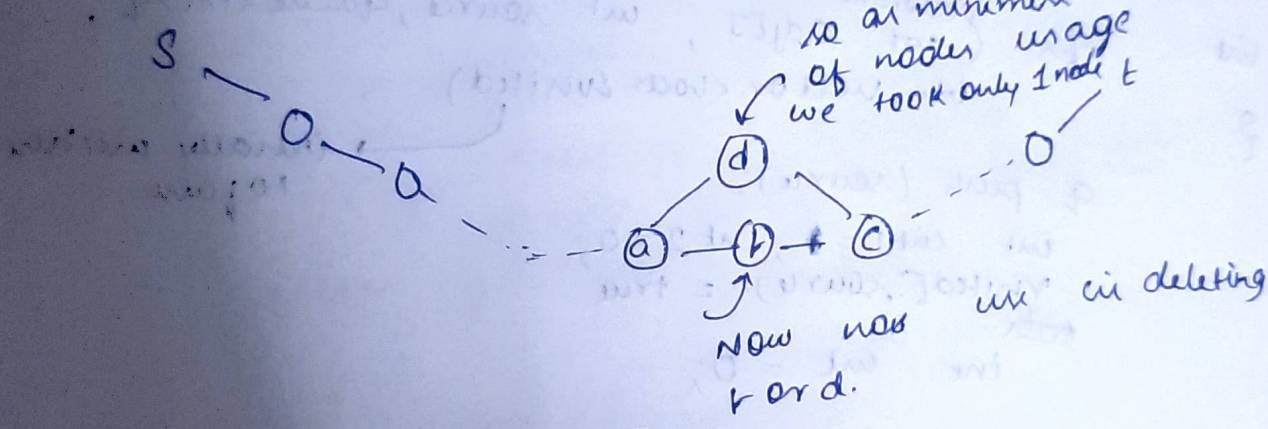
⑥ It was given that distance b/w s and t is greater than $n/2$. ⇒ more than $n/2 - 1$ nodes in the path from s to t

↳ we consider ∃ no node on deletion also that path from s to t



If we delete b there must be a path from s to t

so for this there is a node from a to c other than b



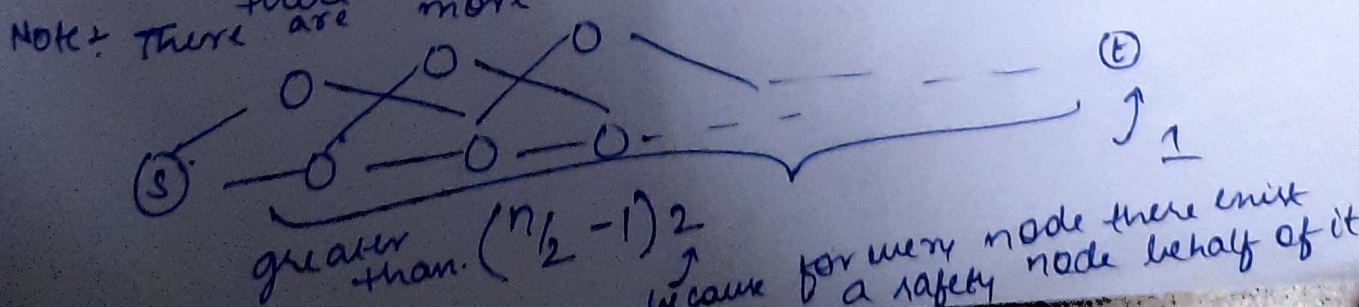
Now we can delete ~~a~~ c node c right

so for all nodes in path s to t

by we can delete any node b/w s & t

there must another node to travel through

Note: There are more than $n/2 - 1$ nodes in the path



greater than $(n/2 - 1)2$

because for every node there exist a safety node behalf of it.

No. of nodes greater than $1 + (n/2 - 1)2 + 1$

(s, t)

$\Rightarrow$ greater than n nodes

But given n nodes

This contradicts

$\Rightarrow$ $\exists$ exist some node upon deletion will delete all the paths b/w s to t.

proved

Algorithm

adjacency list

nodes

```
int bfs(vector<int> adj[], int n, int source,
        vector<bool> &visited, queue<int> q)
```

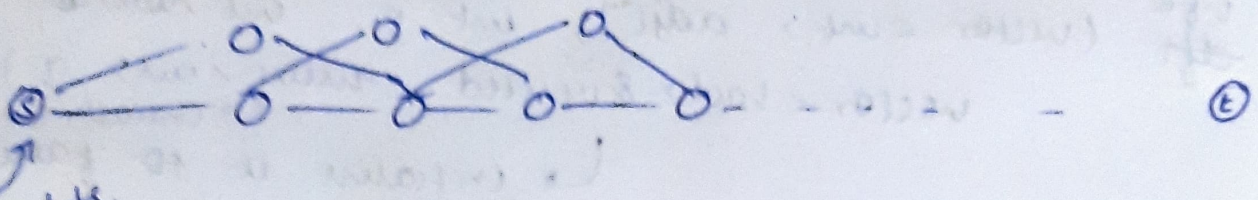
↳ initially empty

↳ initialise it to false initially

```
{
    q.push(source);
    int cnt1 = 1, cnt2 = 0;
    visited[source] = true;

    while(!q.empty())
    {
        int u = q.front();
        q.pop();
        for(auto x : adj[u])
        {
            if(visited[x] == false)
            {
                visited[x] = true;
                q.push(x);
                cnt2++;
            }
        }
        cnt1--;
        if(cnt1 == 0)
        {
            swap(cnt1, cnt2);
            if(cnt1 == 1)
            return 't'-'a';
        }
    }
}
```

explanation next page.



Bfs.

of every node given two nodes but at a point it will give 1 node. i.e there is no safety node for it. So that's the node we want & we will return it

=

cnt 1 → Keep the number of elements in current level.

After reading every node, we do cnt 1--

cnt 2 → ~~Keep~~ stores the number of elements in next level

when cnt 1 == 0

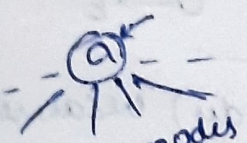
swap(cnt 1, cnt 2) ⟹ the present level has the nodes that are previously stored by cnt 2

=

✓

(5) let us first consider two disconnected components. say. C_1, C_2 & say graph is disconnected

~~atleast~~ Graph G → C_1 C_2



atleast $n/2$ edges.

so C_1 has atleast $n/2 + 1$ (including a) nodes

As there are n nodes in total

$\Rightarrow$ C_2 has $n/2 - 1$ nodes

which means it can have atmost $n/2 - 2$ edges for every node.

∴ contradiction

~~Hence.~~ it ~~is a connected component~~ thus if every node has atleast $n/2$ degree. then it is a connected graph.