Algorithm Analysis & Design

Problem Set 1

Name: Aryaman Kolhe
Roll: 2022121007

Question 1

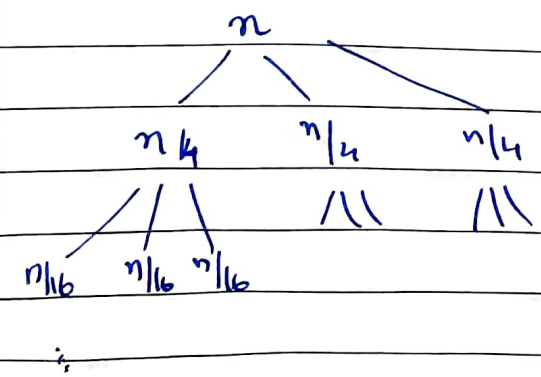
Consider the following functions

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

$$T_2(n) = b \cdot T_2(n/a) + a \cdot n$$

If $a \geq b$ and $T_1(1) = T_2(1) = 1$, how do these functions compare as n grows large.

What a recurrence relation implies algorithmically, Suppose we had $T(n) = 3T(n/4) + cn^2$. This means that our problem is being split into 3 instances of the subproblems with size $1/4^{th}$ of the original size. The cost while computing this is cn^2 for each branch.

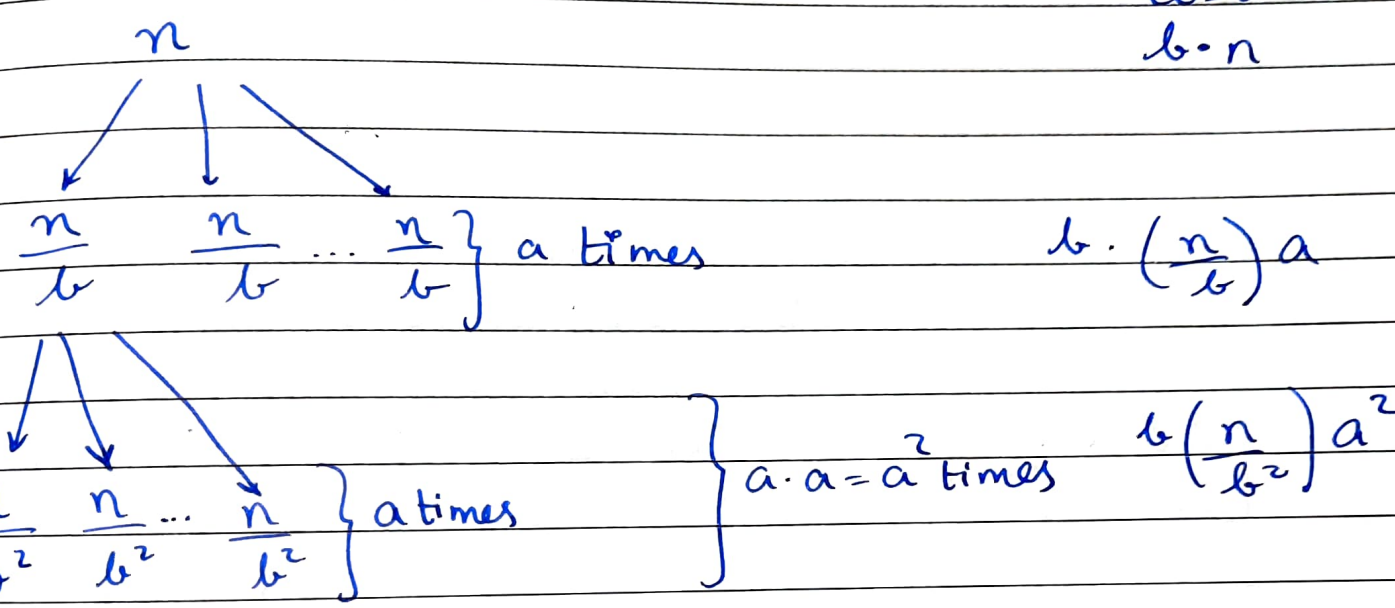


$$\frac{cn^2}{4^2}(3) = \frac{3}{16}cn^2$$

$$\left(\frac{cn}{16}\right)^2 (9) = \left(\frac{3}{16}\right)^2 cn^2$$

For Coming back to the question, we want to see function takes lesser operations.

$$T_1(n) = a \cdot T_{a_1}\left(\frac{n}{b}\right) + b \cdot n$$



$$\therefore \text{Series} = bn + ab\left(\frac{n}{b}\right) + a^2 b\left(\frac{n}{b^2}\right) + \cdots$$

$$\therefore T_1(n) = bn\left[1 + \frac{a}{b} + \left(\frac{a}{b}\right)^2 + \cdots + \left(\frac{a}{b}\right)^k\right]$$

This series will terminate at the leaf nodes when $n = 1$ (subproblem has reached its base case).

$$\therefore \frac{n}{b^k} = 1 \Rightarrow \log n = k \log b$$

$$\Rightarrow k = \log_b n$$

Here k is the depth of our recurrence tree.

The cost is totalled till $k-1$, after which we reach the leaves.

$$T_1(n) = \ln\left(\sum_{i=0}^{k=\log_b n} (a/b)^i\right)$$ → can do this since $T_1(1) = 1$

$$= \ln\left(\frac{(a/b)^{\log_b n} - 1}{(a/b) - 1}\right) \quad [1]$$ [GP sum is $a_0(r^n - 1)/(r-1)$]

* Take n as a very large number

$$\Leftrightarrow \lim_{n\to\infty} T_1(n) = \lim_{n\to\infty} \ln\left[\frac{(a/b)^{\log_b n} - 1}{a/b - 1}\right]$$ * → ignored

$$= \lim_{n\to\infty} \frac{b^2 n}{a-b} \frac{a^{\log_b n}}{b^{\log_b n}}$$

$$= \lim_{n\to\infty} \frac{b^2 n}{a-b} \frac{a^{\log_a n / \log_a b}}{n} = \lim_{n\to\infty} \frac{b^2}{a-b} n^{\log_b a}$$

[since $a^{\log_b n} = a^{\log_a n / \log_a b} = (a^{\log_a n})^{1/\log_a b}$
$= n^{1/\log_a b} = n^{\log_b a}$ //]

$$\therefore \lim_{n\to\infty} T_1(n) = \lim_{n\to\infty} \frac{b^2}{a-b} n^{\log_b a}$$ //

$\therefore T_1$ is $O(n^{\log_b a})$

* For $b < 1$, $\lim_{n\to\infty} T_1(n) = \lim_{n\to\infty} \ln\frac{[0-1]}{a/b - 1} = \lim_{n\to\infty} \frac{b^2 n}{b-a}$

$\therefore T_1(n) < 0$, which is not valid $a \geq b$

$\therefore b > 1$ for this problem.

By Symmetry, we can use [1] for T_2.

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

Taking $n \to \infty$, $\log_a n \to \infty$ and $b/a \leq 1$ since $a \geq b$

$$\therefore (b/a)^{\log_a n} \to 0$$

$$\Leftrightarrow \lim_{n\to\infty} T_2(n) = \lim_{n\to\infty} an\left[\frac{0 - 1}{b/a - 1}\right] = \lim_{n\to\infty} \frac{a^2 n}{a - b}$$

$\therefore T_2$ is $O(n)$

Therefore, T_1 grows faster than T_2

(for $a \geq b$)

(and $b > 1$)

Date
Page

Question 2

Give an algorithm to detect cycles in undirected graphs.

Approach 1 - using BFS

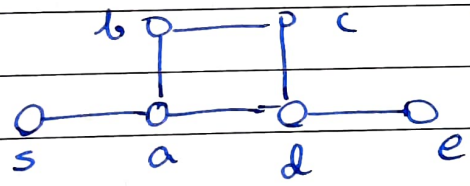
Start a BFS on some node s

While exploring the neighbours of some node u, if we find a node that has already been visited, and if that node is not the parent of u, then a cycle exists.

Proof of correctness [By construction]

Consider a graph $G = (V, E)$

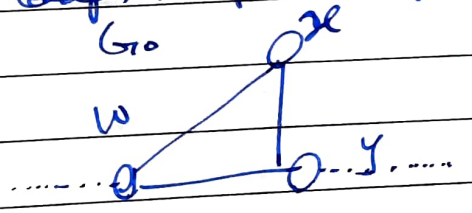
Example -



Example of graph with cycle : $a \to b \to c \to d \to a$

If a previously visited node (which is not u's parent) is seen, then a cycle exists.

G_0 = Simple Graph with smallest cycle.

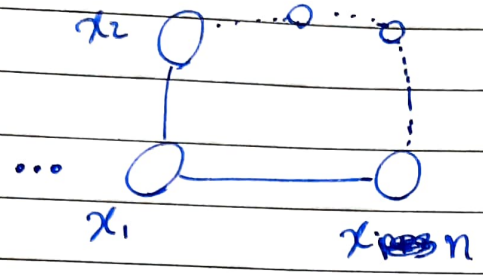


After exploring w, we go to x.
After exploring x, we go to y
Since y is the second neighbor of w. Now, while exploring y, we come across x, which is not y's parent (y's parent = x).

During BFS, the search naturally closes the cycle (wraps around it)

G_{i+1} = Simple undirected graph with $|V_{cycle}(G_{0i})| + 1$ nodes in the cycle

$|V_{cycle}(G_0)| = 3$



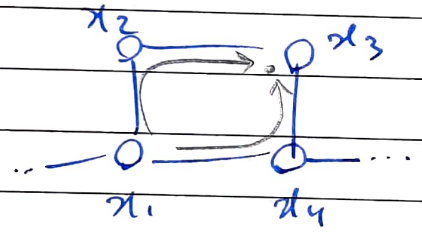
$|V_{cycle}(G_1)| = 4$

and so on.

$n = i + 3$

In node $\lfloor \frac{n}{2} \rfloor + 1$, the BFS will encounter a node that has been traversed already.

Example. $n = 4$



BFS: $x_1 \rightarrow x_2 \rightarrow x_4 \rightarrow x_3$

and $\lfloor \frac{4}{2} \rfloor + 1 = 3$

This proves that cycle detection works for any $n \geq 3$

Q. E. D

Algorithm -

```
function cycleDetection ( G ):
    V = V(G)
    E = E(G)
    s = V[0]  //choose any node as the starting point
    let discovered be a list of length n
    discovered[s] = True
    For all v ∈ V - {s}:
        discovered[v] = False
    let parent be a list of length n, all initialised to NULL
```

Let Q be a queue data structure, which supports two operations : enqueue, dequeue
Initially,
Q ← ∅
Q.enqueue(s).

While Q is not empty :
    u = Q.dequeue()

    For each edge (v,u) ∈ E :
        if discovered[v] == False :
            discovered[v] = True
            Q.enqueue() ; parent[v] = u
        else if discovered[v] == True
            and parent[u] != v :
            // cycle found
            return True

// no cycle
return False // O(m+n) time. Same as BFS

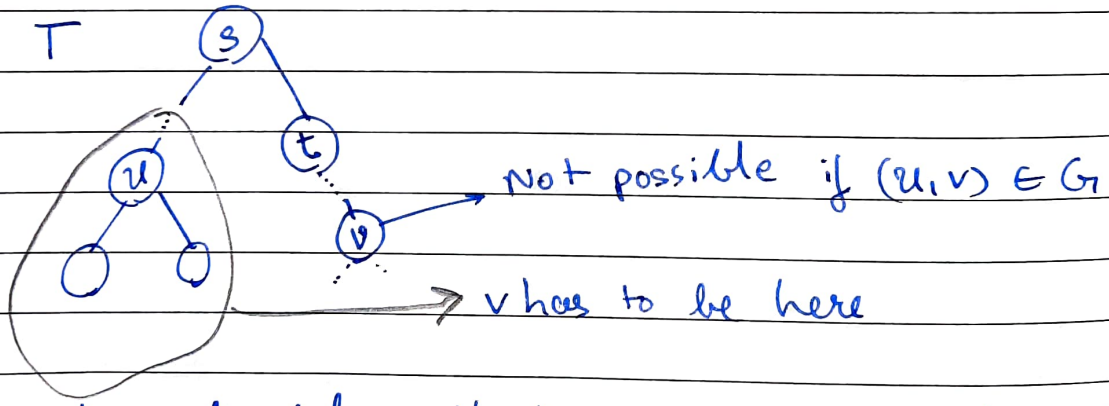
Approach 2 - using DFS.
If an undirected graph's DFS tree has a back edge, then it contains a cycle.

Simple Proof of Correctness -
A DFS tree consists of tree edges, forward edges, cross edges, and back edges.

Claim: A DFS tree of an undirected graph contains no cross edges.

Proof -
For an undirected graph, forward edges and back edges are the same. (not directed, therefore ancestry is not well defined).

If edge $e = (u, v) \in E(G)$, but $e \notin E(T)$, then e must be a back edge, since we would have come across v while exploring u.
Basically, we can't get the case where v is found in a separate branch.



∴ An undirected graph's DFS tree can only contain tree edges and back edges.

Key Argument!
If a DFS tree has 0 back edges, then all edges explored are tree edges. This means that for $|V(G)| = n$; there are n vertices in $T = DFSTree(G)$. and $n-1$ edges in T.

0 back edges implies that out of n-1 edges in T, all n-1 edges are in G_T (and are present in T).

This implies that G_T is also a tree and we know that a graph is a tree iff it contains no cycles.

What it means if no. of back edges > 0.
This means that not all edges in G_T are tree edges, and G_T is not a tree (and therefore, G_T has a cycle)

Q.E.D

~~Algorithm~~
~~Recall~~
~~Edges DFS(u)~~
~~Explored[u] ← True~~

Algorithm -
Stack S ← ϕ ; parent is a list of n elements ; explored[n]
DFS(s):
    S.push(s)
    While S is not empty:
        u ← S.pop()
        if explored[u] == False:
            explored[u] = True
            T ← T ∪ {(parent[u], u)} //add edge to tree
            For Each (u, v) ∈ E(G)
                S.push(v)
                parent[v] = u

// Now check if there is a 1-1 correspondence b/w T, Gr

if $E(G_r) \neq E(T)$:

    There is a cycle

else :

    No cycle in Gr

Interestingly, we also ended up proving that DFSTree(Gr) = Gr iff Gr is an acyclic undirected graph.

Question 3

A binary tree is a rooted tree in which each node has at most two children. Show by any means possible that in any binary tree, the number of nodes with two children is exactly one less than the number of leaves.

Proof by construction:

Start with a graph $G_1 = \{\{v_1\}, \{\}\}$

G_1 v_1 This graph is trivially a binary tree.

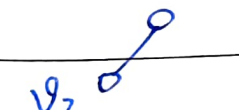
Number of leaves = 1 (trivially), root = leaf

Number of nodes with 2 children = 0

Number of children of v_1 = $nchild(v_1) = 0$ ↳ $nchild2(G_1) = 0$

∴ The statement holds for G_1.

To get G_2, add a node v_1 to G_1. WLOG, let v_2 be the left child of v_1.



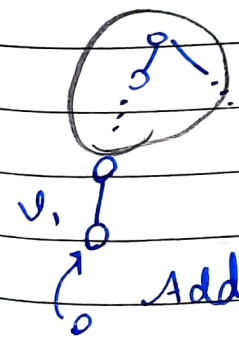
As we can see,

$nleaves(G_2) = 1$

$nchild(v_1) = 1$, $nchild(v_2) = 0$

∴ $nchild2(G) = 0$ ∴ The statement holds.

Consider some arbitrary binary tree T.



If we try to add a new node v_2 to v_1 and if v_1 initially has no children, then the number of leaf nodes in T will not change.

Let L_i be the set of leaf nodes L_i,

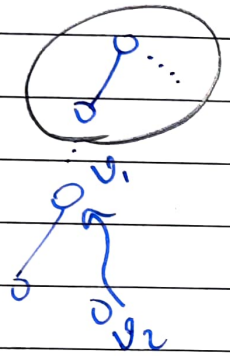
$L_{i+1} \leftarrow L_i - \{v_1\} + \{v_2\}$

Note $|L_i| = |L_{i+1}|$

Seeing what happens

On adding a child to a node with no children, the number of leaves will not change. Also, the number of nodes with 2 children will also not change.

Seeing what happens when we add a child(v_2) to a node with 1 child(v_1)



Set of leaves L_i will turn to L_{i+1}, where

$L_{i+1} \leftarrow L_i + \{v_2\}$

$\therefore |L_{i+1}| = |L_i| + 1$

At the same time, v_1 turns into a node with 2 children.

$\therefore$ nchild2(T_{i+1}) $\leftarrow$ nchild2(T_i)

Cases: (operations)

[1] Add a child to a node with no children [covered]

[2] Add a child to a node with 1 child [covered]

Add a child to a node with 2 children - not possible, as this would disrupt the binary tree.
(same for adding a child to a node with >2 children).

[3] Add no child - the tree remains the same.

In this way, we have covered all ways of creating a binary tree from G_1.

The key argument is that the number of leaf nodes increases by 1 iff the number of nodes with 2 children increases by 1.

Since the statement

$nchild2(T_0) = |L_0| + 1$ holds for the tree with 1 node (G_0), it holds for any other tree T_i

$\therefore nchild2(T_i) = |L_i| + 1$, since any tree T_i can be constructed from T_0 using operations [1], [2], [3].

Q.E.D

Question 4

We have a connected graph $G = (V, E)$ and a specific vertex $u \in V$. Suppose we compute depth-first search rooted at u and obtain a tree T, that includes all $v \in V$. Suppose then we compute a breadth first search tree rooted at u and obtain the same tree T. Prove that $G = T$, i.e. if T is both a depth first tree and a breadth first tree rooted at u, then G cannot contain any more edges than in T.

Assumption:
Tie breaking order is fixed (say lexicographical) and not arbitrary.
Then, DFS tree(G, u) = BFS tree(G, u) is possible.

Proof by contradiction:

Assume that there is some edge e_0 in $E(G)$ such that $e_0 = (s, t)$ for some $s, t \in V(G)$. Now, assume that this edge is not in $E(T)$. [statement 0]

~~We know that~~

Statement 1 -

Claim If $(s,t) \in E(G)$, then, in the BFS tree T, if layer$(s) = L(s) = L_i$ and layer of t is L_j then $|i-j| \leq 1$. That is, s is either in the same layer as t, or one layer above t, or one layer below t.

Proof :

WLOG, assume $i < j$

BFS gaurantees that s is at a distance i from the root u, and y is at a distance j from u.

Suppose $j - i > 1$

From BFS, after exploring s in L_i, t is added to L_{i+1} as $e_0 = (s,t) \in E(G)$.

But this means that $i+1 < j$ gets contradicted

$\therefore i \geq j-1$, or i and j differ by 1 (at most).

Q.E.D.

Statement 2 -

In BFS tree (G, u), if $(s,t) \in E(G)$, then s is an ancestor of t, or t is an ancestor of s in T.

WLOG assume s is an ancestor of t.

Now, from statement 0, we can say that s cannot be a parent of t in T, since the edge $e_0 = (s,t) \notin E(T)$.

$\therefore$ In the BFS tree, if $s \in L_i$ and $t \in L_j$ then :

$i < j$ (s is an ancestor of t)

$i \neq j-1$ (s is not t's parent).

$\therefore i < j-1$

$\Rightarrow i+1 < j$ But this contradicts statement 1.

∴ The initial assumption that such an edge exists in G_T, but not T was wrong.

∴ DFS tree(G,u) = BFS tree$(G,u) = T \longrightarrow G_T = T$

~~Let $e \in T$ DFS tree$(G,u) =$ BFS tree~~

Infact, the question statement is even stronger.

If G_T is some tree T, then

BFS on G_T and DFS on G_T will both give the same tree rooted at u.

(imagine holding up the tree from the root, and letting the other nodes fall down).

More formally, since G_T is connected,

$G_T = T \longrightarrow$ DFS tree(G,u) = BFS tree(G,u)

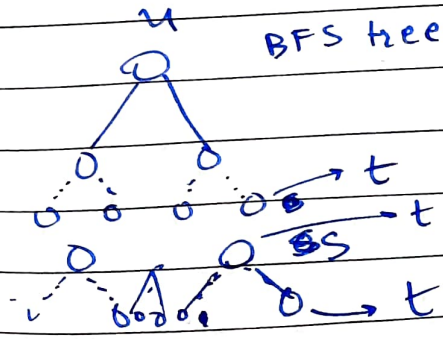
∴ We have proved that

DFS tree(G,u) = BFS tree$(G,u) = T \iff G_T = T$

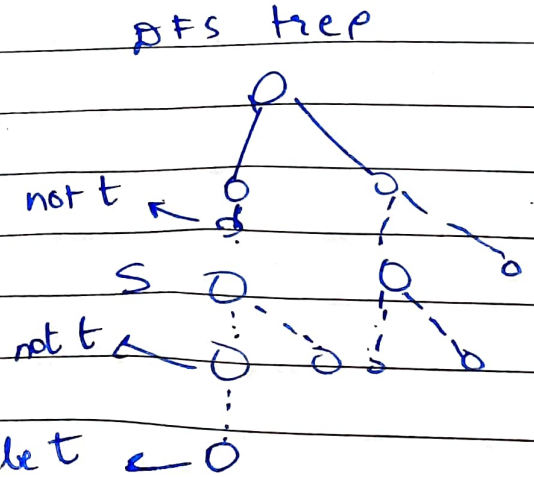
Q.E.D

~~programatic~~

Visualization of the proof:



But



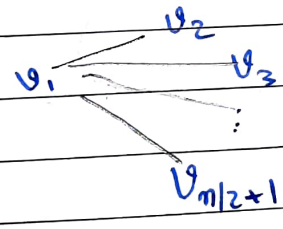
Question 5 -

Prove or disprove the following claim.
Let G_1 be a graph on n nodes, where n is an even number. If every node of G_1 has degree at least $n/2$, then G_1 is connected.

Proof [By contradiction]

Let's assume that graph G_1 can be constructed in such a way that $\cdot$ degree $(v) \geq \frac{n}{2}$ $\forall v \in V(G_1)$ and it is disconnected. (n is even).

Let's start with node $v_1 \in V(G_1)$.
If v_1 is to be connected ∃ no. of edges $= n/2$, then it will be connected to $n/2$ vertices. ($(n/2+1)^{th}$ vertice)

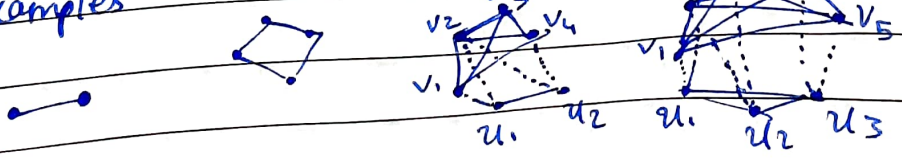


This means that we already have a component with $n/2+1$ nodes.

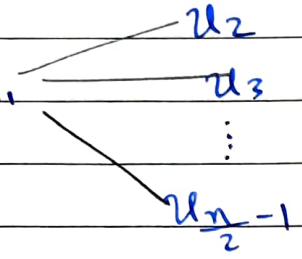
Number of nodes left $= n - \left(\frac{n}{2}+1\right) = \frac{n}{2} - 1$.

Without loss of Generality, it will be enough to prove that we can satisfy degree $(v) \geq n/2$ $\forall v \in V$ AND stay within the leftover vertices to satisfy the assumption. That is, we're trying to split G_1 into 2 components G_{11} and G_{12}, where G_{11} consists of v_i from v_1 to $v_{\frac{n}{2}+1}$ and G_{12} consists of u_i from u_1 to $u_{\frac{n}{2}-1}$.

Some examples



Now, take u_1 from $V(G_2)$.
To satisfy $\text{degree}(u_1) \geq \frac{n}{2}$, take the lower bound and show that u_1 is only connected to u_i, $i \neq 1$ and not any $v_i \in V(G_1)$. [Here, $u_i \in V(G_2)$]



u_1 can form only $\left(\frac{n}{2}-1\right)-1 = \frac{n}{2}-2$ edges within G_2.

This means that to make $\deg(u_1) \geq \frac{n}{2}$, we need to form 2 more connections.
(u_1, v_i), (u_1, v_j) where $v_i \neq v_j$, $v_i, v_j \in V(G_1)$.
With this, the component G_1 gets one more vertex $v_{\frac{n}{2}+2} = u_1$.

Now u_2 has only from u_3 to $u_{\frac{n}{2}-1}$ to make $\deg(u_2) \geq \frac{n}{2}$, which is again not possible.
Another way of thinking of this is by invoking the pigeonhole principle. Initially, if $\deg(u_1) \geq \frac{n}{2}$, then we have $\frac{n}{2}$ edges, but $\frac{n}{2}-2$ vertices to connect u_1 with, which means that 2 nodes would have multi-edges with u_1, which is not allowed in a simple graph.
∴ To maintain $\deg(u_i) \geq \frac{n}{2}$, or at least $\deg(u_i) = \frac{n}{2}$, we will have to use edges from $v_i \in V(G_2)$, thus contradicting our initial assumption that we can construct a disconnected graph satisfying the constraint $\deg(v) \geq \frac{n}{2}$ $\forall v \in V(G_1)$.

Q.E.D

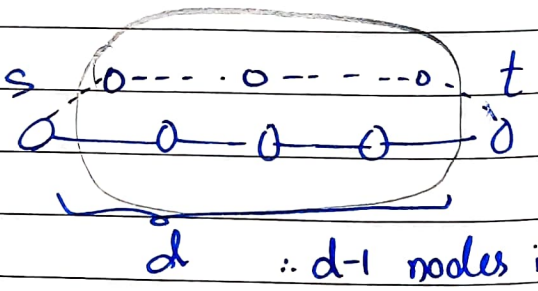
Question 6 -

$G = (V, E)$, $|V| = n$ G is undirected.
There are s, t $\in V(G)$ such that the distance between s and t is strictly greater than $n/2$.
Show that there must exist some node v, $v \neq s$, $v \neq t$ such that deleting v from G destroys all s to t paths.
(The graph obtained from G by deleting v contains no paths from s to t).
Give an algorithm in $O(m+n)$ to find v.

Proof by contradiction.

Assume that no such v exists in G.
Let the distance between s and t be d.

Now, $d > \lfloor \frac{n}{2} \rfloor$



$\Rightarrow d-1 \geq \lfloor \frac{n}{2} \rfloor$

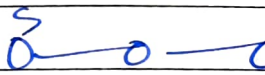
$\Rightarrow 2(d-1) \geq n$ [1]

Now, for some alternative path of at least length d to exist, we need to ~~add~~ use $2(d-1)$ nodes (at least).
But from [1], $2(d-1)$ is at least n.

This forms a contradiction, since if $2(d-1)$ is n, the nodes s and t would not exist / or we would need $n+2$ nodes in G_1. (But we have only n).

* Another line of thought is proof by construction.

Start with $n=3$

G_2 

$$\left\lfloor \frac{n}{2} \right\rfloor = \left\lfloor \frac{3}{2} \right\rfloor = 1$$

$\therefore d > 1 \Rightarrow d = 2$

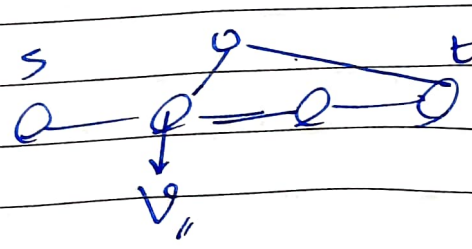
But if try to add a node to G_2, to form an alternate path, the numbers of nodes will change, $n: 3 \rightarrow 4$ and

$$\left\lfloor \frac{n+1}{2} \right\rfloor = \left\lfloor \frac{4}{2} \right\rfloor = 2$$

for which the $d > \frac{n}{2}$ property fails.

This argument can be generalised for any $n \geq 3$

* $\therefore$ There must exist a node v, where the attempted alternate path meets the first path. On removing v, we get 2 connected components.

E.g. s 

G_3 

On looking at G_3, we can picture the existence of v. And since we have proven that v exists under the given constraints, we can locate it by performing a BFS rooted at s.

Then we can look at the path from s to t in the BFS tree $= T$

The layer in T with only one node is the one that can be cut off.

$\therefore$ if $|L_i| = 1$ then $v \in L_i$.

BFS is a natural choice because of this property

Algorithm -

$O(m+n)$ time.

```
locate_V ( G, s, t ).
visited = list of length n, All visited[i] = False , i=1,2,..,n
visited[s] = True
L = layers list
L[0] = { s }   // first layer
i ← 0

while L[i] is not empty :
    L[i+1] ← { }
    For each u ∈ L[i] :
        For each edge (u,v) ∈ E incident on u :
            if visited[v] == False :
                visited[v] = True
                L[i+1].append (v)
    if length of L[i+1] is 1 :
        return k , where k ∈ L[i+1]  // v found
    i ← i+1
return -1   // v not found (not possible under given constraints)
```

References -

Q1 - Harshit Aggarwal 2021111015
Found a mistake in my approach (finite series)
Yatharth Gupta 2021101093
Consider the case where $b < 1$.

Q2 - geeksforgeeks.org/tree-back-edge-and-cross-edges-in-dfs-of-graph
Used to learn about back edge, forward edge, cross edge

Q4 - Referenced Prof Suryajith's lecture for statement 1

Q6 - Referenced & Prof Suryajith's lecture for BFS details.