Homework 2

● Graded

Student

Kulvir Singh

Total Points

37.9 / 40 pts

Question 1

Q1 **10** / 10 pts

✔ **+ 10 pts** Correct

+ 0 pts Click to write.

Question 2

Q2 **8** / 10 pts

✔ **+ 2 pts** a

✔ **+ 2 pts** b

+ 2 pts c

✔ **+ 2 pts** d

✔ **+ 2 pts** e

+ 0 pts None

Question 3

Q3 **10** / 10 pts

- ✔ **+ 2 pts** Find edge that are not in graph
- ✔ **+ 5 pts** Conclude that the two nodes on the edge are exactly one layer different.
- ✔ **+ 3 pts** Reach Contradiction.

+ 0 pts No Points Given

Question 4

Q4 **9.9** / 10 pts

✔ **– 0 pts** Correct

– 2 pts Wrong algorithm

– 2 pts 1 time complexity is wrong

– 4 pts 2 time complexity is wrong

– 6 pts 3 time complexity is wrong

– 8 pts 4+ time complexity is wrong OR no time complexity analysis

– 10 pts No attempt

– 0.1 pts Please explain more in pseudocode in the future. Also use consistent variable name in pseudocode -- mind the difference between pseudocode and actual code. For example, using B instead skip makes it clearer in pseudocode.

Question assigned to the following page: 1

Homework 2

1) $2^{\log(n)}$, $\log(n)$, $\log(n!)$, $n\log(n^2)$, $\log(\log(n^n))$, 4^{2n}, $n^{n\log(n)}$, n^{n^2}, 3^{4n}

Segregating based on polynomial, exponential & log functions we can define the increasing order of growth rate as follows

some simplifications used are :

$2^{\log(n)} = n$

$n(\log(n^2)) = 2n\log(n)$

$\log(\log(n^n)) = \log(n\log(n)) = O(\log n + \log\log n)$ ignore

$\log(n!) \simeq n\log(n)$ { $n! = n(n-1)\ldots 1$ }

~~$\log(n) < \log(n!)$~~

$\log(n) \leq \log(\log(n^n)) < 2^{\log(n)} < \log(n!) < n\log(n^2) <$

$4^{2n} < 3^{4n} < n^{n\log(n)} < n^{n^2}$

$\log(f(n)) <$ polynomial $f(n) <$ exponential$(f(n))$

Question assigned to the following page: 2

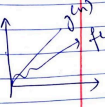
2) a) $f(n) = n!$ $\quad g(n) = 2^{2n}$

$g(n) \rightarrow$ Exponential func. (grows faster)

$f(n) \rightarrow n! = n(n-1)(n-2) \cdots 1.$

$= n^k \cdots n^{k-1} \cdots$

polynomial func.



$\boxed{f(n) \in O(g(n))}$

b) $f(n) = e^n$ $\quad g(n) = n^{\log(n)}$

↓ Exponential $\quad$ ↓ Exponential

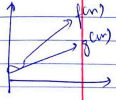
however, we know that $\log(n)$ has slowest growth rate as compared to (e) & polynomial.

$g(n) = [(\text{poly})^{(\log)}]$ Exponential

by value substitution, we can verify the same

$n = 1024$	$n = 2048$	$n = 4096$
$f(n) = e^{1024}$	$f(n) = e^{2048}$	$f(n) = e^{4096}$
$g(n) = (1024)^{\log(1024)}$	$g(n) = 2048^{11}$	$g(n) = 4096^{12}$
$= (1024)^{10}$		

the exponential component always has a greater value for $f(n)$ than in $g(n)$



$\boxed{f(n) \in \Omega(g(n))}$

Question assigned to the following page: 2

c) $f(n) = 2^n$ $\quad g(n) = 2^{3n}$

Both are exponential functions.
In this case, the exp. deg. are

$f(n) \longrightarrow n$

$g(n) \longrightarrow (3n) \longrightarrow n$ ~~can be~~ (linear)

Since both the components follow a linear ~~expon~~ polynomial,

$f(n) \in O(g(n))$ & $f(n) \in \Omega(g(n))$

$\therefore$ $\boxed{f(n) \in \Theta(g(n))}$

d) $f(n) = n^2$ $\quad g(n) = 2^{\sqrt{\log(n)}}$

n^2 ↓ polynomial

$2^{\sqrt{\log(n)}}$ ↓ exponential, however the degree is $\sqrt{\log(n)}$ which is lesser than $\log(n)$'s growth rate.

By using some values of n to determine the trend.

$n = 1024$, $f(n) = 1024^2$, $g(n) = 2^{\sqrt{10}}$

$n = 4096$, $f(n) = 4096^2$, $g(n) = 2^{\sqrt{12}}$

we can see that the base of 2 is always smaller than the base of (n) for large values of n.

$\therefore$ as $n \uparrow$ $f: n^2 \uparrow$ faster than $2^{\sqrt{\log(n)}}$

$\therefore$ $f(n) \in \Omega(g(n))$

Question assigned to the following page: 2

e) $f(n) = \log n$ $\quad g(n) = \log(\log(6^{2n}))$

$g(n) = \log(2n \log(6))$

$= \log((2\log(6))n)$

for log functions as the value of n increases, the value of $\log(cn)$ increases and for

Also

$\therefore f(n) = \log(n)$

$g(n) = \log(cn)$, $c = 2\log(6)$

since the degree of polynomial in $f(n)$ and $g(n)$ remain the same we can say that $f(n) \in O(g(n))$, & $g(n) = f(n) \in \Omega(g(n))$.

$\therefore$ we can say that

$$\boxed{f(n) \in \Theta(g(n))}$$

Question assigned to the following page: 3

3) given : $G(V, E)$ where the $DFS(u) = T$
$BFS(u) = T$ where $u \in V$.

$BFS(u) = DFS(u) = T$

Prove by contradiction that G is the same as T. $G = T$

To prove this we can assume there is an edge in T which is not in G.

$\therefore G \neq T$

For any edge with node (x, y) in G, but not in T

(i) BFS Tree for G has the property :

$$|Level(x) - Level(y)| \leq 1 \quad \text{— (i)}$$

as BFS looks for immediate neighbours hence the max deg level difference for x, y can be 1

(ii) DFS Tree for G has the property :

$$|Level(x) - Level(y)| \geq 1 \quad \text{— (ii)}$$

as DFS looks for the max depth disconnect for a particular node, the min. level difference for x, y will be 1.

(iii) DFS = BFS. as per the given condition. we can say (i) & (ii)

$$Level(x) - Level(y) = 1$$ overlapping part of (i) & (ii)

$$Level(x) = 1 + Level(y) \quad \text{— (iii)}$$

Question assigned to the following page: 3

~~Hence we can see that for any vertex (u, v) the edge~~

Since we can use (iii) to establish that there is only 1 level gap between x & y, we can say that x is the parent of y.

Hence x — y edge is present in the BFS / DFS tree for the graph G which CONTRADICTS our claim that x-y not in T.

Hence proved.

Question assigned to the following page: 4

4) $G = (V, E)$, Edge e → detect for cycle on e

Algorithm :

Initial variables :

```
Graph G ;
Edge e ; Node A, B ;
where A, B are the vertices in edge e ;
F ( )
{
bool visited [V] ;
dfs (A, visited, B, G)
for ( i : visited)
{
    if (i == False)
        return False ; // cycle not present on e
}
    return True ;    // cycle present on e
}
dfs (Node start, bool visited[], Node skip, Graph G)
{
    visited [start] = True ;
    for ( i : neighbours of (start) in G)
    {
        if ( i == skip) continue ;
        if (visited [i] == False)
            dfs ( i, visited, skip, G)
    }
}
```

Question assigned to the following page: 4

Time complexity

→ F() calls dfs() function once

→ F() also has 1 for loop for linear search which has a complexity of O(N) where N = V (no. of vertices)
∴ O(V)

→ dfs() execution of a recursive function which will be called a total of (V+E) times as it performs DFS for the given start node. DFS complexity = O(V+E)

→ Total T.C = F() + dfs()
= O(V) + O(V+E)
= O(V+E)

∴ **TC of algo = O(V+E)**