Answer the questions in the boxes provided on the question sheets. If you run out of room for an answer, add a page to the end of the document.

Name: Riya Kore Wisc id: rykore

Asymptotic Analysis

1. *Kleinberg, Jon. Algorithm Design (p. 67, q. 3, 4).* Take the following list of functions and arrange them in ascending order of growth rate. That is, if function $g(n)$ immediately follows function $f(n)$ in your list, then it should be the case that $f(n)$ is $O(g(n))$.

   (a) $f_1(n) = n^{2.5}$
   $f_2(n) = \sqrt{2n}$
   $f_3(n) = n + 10$
   $f_4(n) = 10n$
   $f_5(n) = 100n$
   $f_6(n) = n^2 \log n$

   $f(n) \leq c \cdot g(n)$

   $f_2(n) < f_3(n) < f_4(n) < f_5(n) < f_6(n) < f_1(n)$

   (b) $g_1(n) = 2^{\log n}$
   $g_2(n) = 2^n$
   $g_3(n) = n(\log n)$
   $g_4(n) = n^{4/3}$
   $g_5(n) = n^{\log n}$
   $g_6(n) = 2^{(2^n)}$
   $g_7(n) = 2^{(n^2)}$

   $g_1(n) < g_3(n) < g_4(n) < g_5(n) < g_2(n) < g_7(n) < g_6(n)$

2. *Kleinberg, Jon. Algorithm Design (p. 68, q. 5).* Assume you have a positive, non-decreasing function f and a positive, non-decreasing function g such that $g(n) \geq 2$ and $f(n)$ is $O(g(n))$. For each of the following statements, decide whether you think it is true or false and give a proof or counterexample.

(a) $\log_2 f(n)$ is $O(\log_2 g(n))$

$0 \leq f(n) \leq c \cdot g(n)$, $\forall\ n \geqslant n_0$, $\exists\ c, n_0 > 0$

taking $\log_2$ on both sides:

$0 \leq \log_2(f(n)) \leq \log_2(c) + \log_2(g(n))$. Lets assume $c' = \frac{\log_2(c)}{\log_2(g(n_0))} + 1$, where $n' = n_0$.

$\therefore\ 0 \leq \log_2(f(n)) \leq \log_2(c) + \log_2(g(n)) \leq c' \cdot \log_2(g(n))$

$0 \leq \log_2(f(n)) \leq \log_2(c) + \log_2(g(n)) \leq \left[\frac{\log_2(c)}{\log_2(g(n_0))} + \frac{\log_2(g(n_0))}{\log_2(g(n_0))}\right] \cdot \log_2(g(n))$

$0 \leq \log_2(f(n)) \leq \log_2(c) + \log_2(g(n)) \leq \log_2(c) + \log_2(g(n_0))$

Hence, proved that $\log_2(f(n))$ is $O(\log_2(g(n)))$.

This is true.

(b) $2^{f(n)}$ is $O(2^{g(n)})$

Lets assume $f(n) = 2\log_2 n$, $g(n) = \log_2 n$

$\therefore\ 2^{f(n)} = 2^{2\log_2 n} = n^2$, $2^{g(n)} = 2^{\log_2 n} = n$

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{2\log_2 n}{\log_2 n} = 2$ $\therefore$ $f(n)$ is little-oh of $g(n)$

But, $\lim_{n\to\infty} \frac{2^{f(n)}}{2^{g(n)}} = \frac{n^2}{n} = n = \infty$ $\therefore\ 2^{f(n)} > 2^{g(n)}$

Hence, since $2^{f(n)}$ is not $O(2^{g(n)})$, $2^{f(n)}$ is not $O(2^{g(n)})$.

Hence, this is false, because $2^{f(n)} \not\leq c \cdot 2^{g(n)}$ for $\exists\ c$ & $\forall\ n \geqslant n_0$.

(c) $f(n)^2$ is $O(g(n)^2)$

$0 \leq f(n) \leq c \cdot g(n)$, $f(n)$: $\exists\ c, n_0 > 0$ $\forall\ n \geqslant n_0$

squaring both sides:

$0 \leq (f(n))^2 \leq c^2 \cdot (g(n))^2$

Let $c^2 = c_0$, $c_0 \geqslant 0$ and $c_0 \geqslant c^2$

$\therefore\ 0 \leq f(n)^2 \leq c \cdot g(n)^2$ for $\forall\ n > n_0$

This tells us that $f(n)^2$ is $O(g(n)^2)$.

Hence, this is true.

3. *Kleinberg, Jon. Algorithm Design (p. 68, q. 6).* You're given an array A consisting of n integers. You'd like to output a two-dimensional n-by-n array B in which $B[i, j]$ (for $i < j$) contains the sum of array entries $A[i]$ through $A[j]$ — that is, the sum $A[i] + A[i + 1] + ... + A[j]$. (Whenever $i \geq j$, it doesn't matter what is output for $B[i, j]$.) Here's a simple algorithm to solve this problem.

```
for i = 1 to n — n
  for j = i + 1 to n — n
    add up array entries A[i] through A[j] — n
    store the result in B[i, j]
  endfor
endfor
```

(a) For some function f that you should choose, give a bound of the form $O(f(n))$ on the running time of this algorithm on an input of size n (i.e., a bound on the number of operations performed by the algorithm).

$O(n^3)$

(b) For this same function f, show that the running time of the algorithm on an input of size n is also $\Omega(f(n))$. (This shows an asympto tically tight bound of $\Theta(f(n))$ on the running time.)

To show that the running time of this algorithm for input n is $\Omega(n^3)$.
The outer for loop (for i=1 to n) runs n times. Hence, it is $\Omega(n)$. The inner for loop (for j=i+1 to n) runs (n-1) times. But, since n is the dominating term, running time would be $\Omega(n)$. For the first step inside the second for loop, you are iterating through A to find A[i] and A[j], so this step also runs n times. Hence, $\Omega(n)$. ∴ Total running time is $\Omega(n^3)$.

(c) Although the algorithm provided is the most natural way to solve the problem, it contains some highly unnecessary sources of inefficiency. Give a different algorithm to solve this problem, with an asymptotically better running time. In other words, you should design an algorithm with running time $O(g(n))$, where $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$.

```
for i = 1 to n
    j = i+1
    temp = A[i]
    while j < n
        add A[j] to temp
        store result of temp in B[i,j]
        increment j by 1.
    end while
end for
```

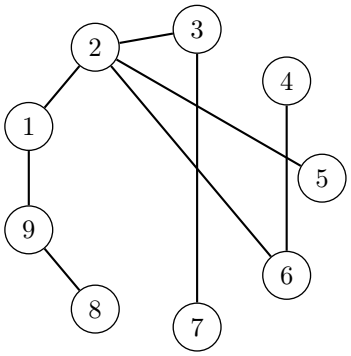
running time = $O(n^2)$

$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \lim_{n\to\infty} \frac{n^2}{n^3}$

$= \lim_{n\to\infty} \frac{1}{n} = 0$.

Graphs

4. Given the following graph, list a possible order of traversal of nodes by breadth-first search and by depth-first search. Consider node 1 to be the starting node.



Breadth-first search: 1, 2, 9, 3, 5, 6, 8, 7, 4

Depth-first search: 1, 2, 3, 7, 5, 6, 4, 9, 8

5. *Kleinberg, Jon. Algorithm Design (p. 108, q. 5).* A binary tree is a rooted tree in which each node has at most two children. Show by induction that in any binary tree the number of nodes with two children is exactly one less than the number of leaves.

$(n = m-1)$.

Let m = no. of leaves and n = no. of nodes with 2 children.
Lets perform induction on n.

Base case: A binary tree with only one node. (not with two children).
The binary tree will have only one node (the root node). Here, the value of $n = 0$ while $m = 1$. You want $(n = m-1)$ to hold true. In this case, $0 = 1-1$ does hold true.
Hence, the base case holds true that $n = m-1$.

Inductive hypothesis: Lets assume that for a positive integer k, where k is the no. of nodes with two children in the binary tree, there are $k+1$ leaves.
$\therefore k = (k+1) - 1 = k+1-1 = k$.

Inductive step: Lets prove that for a binary tree with $(k+1)$ nodes having two children, there are $k+2$ leaves.
We have a binary tree where we add two nodes to a leaf node. Doing this would increase the number of nodes with two children by 1 and the number of leaves increases by 1. Similarly, if we add a node to a binary tree where a node has only one child, adding the new node as the second child of the node would also increase the no. of nodes with two children by 1, and the no. of leaves by 1. The above covers both cases of a possible addition of a node to the binary tree.
From the inductive hypothesis, when you add two leaf nodes to a leaf node in the tree with $(k+1)$ nodes, it would mean that there are $(k+1)+1$ leaf nodes $= (k+2)$. When you add a single node to a tree with $(k+1)$ nodes, where one node has one child (the leaf node), the total no. of leaves is $k+2$.

Hence, by induction, the no. of nodes with two children $n = k+1$ is exactly one less than the no. of leaves $m = k+2$ in a binary tree.

$k+1 = k+2-1 \Rightarrow n = m-1$. Hence, proved. ■

6. *Kleinberg, Jon. Algorithm Design (p. 108, q. 7).* Some friends of yours work on wireless networks, and they're currently studying the properties of a network of n mobile devices. As the devices move around, they define a graph at any point in time as follows:

   > There is a node representing each of the n devices, and there is an edge between device i and device j if the physical locations of i and j are no more than 500 meters apart. (If so, we say that i and j are "in range" of each other.)

   They'd like it to be the case that the network of devices is connected at all times, and so they've constrained the motion of the devices to satisfy the following property: at all times, each device i is within 500 meters of at least $\frac{n}{2}$ of the other devices. (We'll assume n is an even number.) What they'd like to know is: Does this property by itself guarantee that the network will remain connected?

   Here's a concrete way to formulate the question as a claim about graphs:

   Claim: Let G be a graph on n nodes, where n is an even number. If every node of G has degree at least $\frac{n}{2}$, then G is connected.

   Decide whether you think the claim is true or false, and give a proof of either the claim or its negation.

We can use proof by contradiction to check this claim.
Let's assume that the graph G with n nodes is not connected. This means that there exists at least two nodes $x, y \in V$ such that there is no path between x and y.

Let X and Y be the set of nodes reachable from x and y respectively. As x and y have no paths between them, $X \cap Y = \phi$.

The nodes in graph G have a degree of atleast $\frac{n}{2}$, we can say $|X| \geq \frac{n}{2}+1$ and $|Y| \geq \frac{n}{2}+1$.

$\therefore |X \cup Y| = |X| + |Y| - |X \cap Y|$ ... (as x, y have no path between) 0

$\therefore |X \cup Y| = |X| + |Y| = \frac{n}{2}+1+\frac{n}{2}+1 = n+2$

This is a contradiction because we said graph G has a total of n nodes but $|X \cup Y|$ is coming out to be $(n+2)$.
Hence, our assumption is false. Graph G is a connected graph with every node of G having a degree of at least $\frac{n}{2}$.

Coding Question: DFS

7. Implement depth-first search in either C, C++, C#, Java, Python, or Rust. Given an undirected graph with n nodes and m edges, your code should run in $O(n+m)$ time. Remember to submit a makefile along with your code, just as with the first coding question.

Input: the first line contains an integer t, indicating the number of instances that follows. For each instance, the first line contains an integer n, indicating the number of nodes in the graph. Each of the following n lines contains several space-separated strings, where the first string s represents the name of a node, and the following strings represent the names of nodes that are adjacent to node s.

The input order of the nodes is important as it will be used as the tie-breaker. For example, consider two consecutive lines of an instance:

① Graph to put edges in.
② Adjacency List

```
0, F
B, C, a
```

The tie break priority is 0 < F < B < C < a.

Input constraints:

- $1 \leq t \leq 1000$
- $1 \leq n \leq 100$
- Strings only contain alphanumeric characters
- Strings are guaranteed to be the names of the nodes in the graph.

Output: for each instance, print the names of nodes visited in depth-first traversal of the graph, *with ties between nodes visiting the first node in input order*. Start your traversal with the first node in input order. The names of nodes should be space-separated, and each line should be terminated by a newline.

Sample Input:

```
2
3
A B
B A
C
9
1 2 9
2 1 6 5 3
4 6
6 2 4
5 2
3 2 7
7 3
8 9
9 1 8
```

Sample Output:

```
A B C
1 2 6 4 5 3 7 9 8
```

The sample input has two instances. The first instance corresponds to the graph below on the left. The second instance corresponds to the graph below on the right.