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: Ki Min Kang Wisc id: 908 404 5062

Asymptotic Analysis $1 < \log(n) < \sqrt{n} < n < n\log(n) < n^2 < 2^n < n! < n^n$

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

   $\sqrt{2n} < n+10 = 10n = 100n < n^2\log n < n^{2.5}$

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

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

$f(n) = O(g(n))$
$f(n) \le c \cdot g(n)$

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

$2^{f(n)} = O(2^{g(n)})$
$2^{f(n)} \le c \cdot 2^{g(n)}$

Let $f(n) = \log_2 n^2$, $g(n) = \log_2 n$ (counter example)

$\lim_{n\to\infty} \frac{f(n)}{g(n)}$ can be expressed as $\lim_{n\to\infty} \frac{2\log_2 n}{\log_2 n} = 2$

$\lim_{n\to\infty} \frac{2^{f(n)}}{2^{g(n)}} = \lim_{n\to\infty} \frac{2^{\log_2 n^2}}{2^{\log_2 n}} = \lim_{n\to\infty} n = \infty$

Since it radiates infinitely, this is a counter example of the statement.
Therefore, it is false.

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

$f(n)^2 = O(g(n)^2)$
$f(n)^2 \le c \cdot g(n)^2$

We know that when $f(n) = O(g(n))$, we can get $f(n) \le c \cdot g(n)$.
When squaring both sides, we get:
$f(n)^2 \le (c \cdot g(n))^2$
$f(n)^2 \le c^2 \cdot g(n)^2$
Since c^2 is a positive number, we can express it as a c_0 for some positive constant.
$f(n)^2 \le c_0 \cdot g(n)^2$
This proves that $f(n)^2$ is $O(g(n)^2)$, since $f(n)^2$ can be bounded above by a constant multiple of $g(n)^2$.
Therefore, the statement is true.

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

$\log_2 f(n) = O(\log_2 g(n))$
$\log_2 f(n) \le c \cdot \log_2 g(n)$

We get $f(n) \le c_1 \cdot g(n)$
$\log_2 f(n) \le \log_2 (c_1 \cdot g(n))$
$\log_2 f(n) \le \log_2 c_1 + \log_2 g(n)$
Since $\log_2 c_1$ and n_1 are constant,
$c_2 \ge \frac{\log_2 c_1}{\log_2 g(n)} + 1$ for another constant c_2.
This ensures that $\log_2 c_1 + \log_2 g(n)$ is less than or equal to $c_2 \log_2 g(n)$ for all n greater than n_1.

Therefore, the statement 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
  for j = i + 1 to n
    add up array entries A[i] through A[j]
    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).

$n \times n \times n = n^3$

(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 asymptotically tight bound of $\Theta(f(n))$ on the running time.)

The algorithm above have a triple-nested loop.

The outer loop : $\Omega(n)$
The middle loop: $\Omega(n)$
The inner loop: $\Omega(n)$

Therefore, the running time of the algorithm is $\Omega(n^3)$.

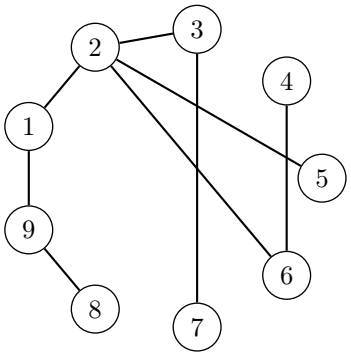
(c) Although the algorithm provided is the most natural way to solve the problem, it contains some highly unnecessary sources of inefficiency. Give a different algorithm to solve this problem, with an asymptotically better running time. In other words, you should design an algorithm with running time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

```
Initialize the first element B[1,1] = A[1]

for i from 2 to n:
   B[1, i] = B[1, i-1] + A[i]    // Fill in the first row    O(n)

for row from 2 to n:
   for col from row to n:
      B[row, col] = B[row-1, col] - A[row-1]    //Fill the rest of the array
                                                              O(n²)
```

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

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.



Starting node : 1

Breadth-First-Search : 1, [2, 9], [3, 5, 6, 8], [4, 7]

Depth-First-Search : 1, 2, 3, 7, 6, 4, 5, 9, 8
1, 2, 6, 4, 5, 3, 7, 9, 8
1, 9, 8, 2, 3, 7, 6, 4, 5

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.

Let the number of nodes with two children t_n,
the number of leaves l_n

Assume that for any binary tree, the number of nodes with two children is exactly one less than the number of leaves. ($t_n = l_n - 1$)

Base case: The binary tree with a single node has just one node and it is also a leaf (since it has no children)
Then, $l_1 = 1$ and $t_1 = 0$, $t_1 = l_1 - 1$.
The base case holds.

Inductive Step: Let's prove the property holds for a binary tree with k+1 nodes.
Since the tree has more than one node, leaf must have a parent.
Let leaf = f, parent = P.
We will now consider 2 scenarios.

1. If f was the only child of P, then removing f makes P a leaf.
The number of leaves l_{k+1} remains the same as l_k, and the number of two-child nodes t_{k+1} is still t_k.
$t_{k+1} = t_k = l_k - 1 = l_{k+1} - 1$

2. If P had two children, then removing one child from P decreases the number of leaves by one. Also, the number of two-child nodes decreases by one.
$t_{k+1} - 1 = t_k$ and $l_{k+1} - 1 = l_k$. $t_{k+1} = l_{k+1} - 1$

Therefore, the number of nodes with two children is always exactly one less than the number of leaves by induction

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.

Assume there exists a graph G not connected,
which indicates that there are nodes from G not connected by any edge.

Let A and B be the sets of nodes from G.
Since we assumed that G is not connected, A and B are disjoint.
Both A and B must have more than $\frac{n}{2}$ nodes, because
A or B must be connected to at least $\frac{n}{2}$ other nodes.

Then, $|A| \geq \frac{n}{2}+1$, $|B| \geq \frac{n}{2}+1$.

$|A \cup B| = |A|+|B| \geq n+2$ (since A and B are disjoint)

This indicates that there are at least $n+2$ nodes in G, which contradicts our statement.

Therefore, by contradiction, the statement is true.

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 an instance

```
4
xy v0 b
b xy
v0 xy a
a v0
```

The tie break priority is `xy` < `v0` < `b`, so your code should produce output

```
xy v0 a b
```

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, the output of each instance should be terminated by a newline, and the lines should have **no trailing spaces**.

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.