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: Zinnia Nie Wisc id: 908 319 4044

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

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

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

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

2. *Kleinberg, Jon. Algorithm Design (p. 68, q. 5).* Assume you have a positive, non-decreasing function f and a positive, increasing 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 $O(\log_2 g(n))$

Def of Big-Oh: $\exists c, n_0 > 0 \mid 0 \leq f(n) \leq c\, g(n) \;\; \forall n \geq n_0$

Since $f(n)$ is $O(g(n))$ that means there exists a c that satisfies $f(n) \leq c\, g(n)$.

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

$\log_2 f(n) \leq \log_2 (c\, g(n))$

$\log_2 f(n) \leq (\log_2 c) + (\log_2 g(n))$ — Since $g(n) \geq 2$, $\log_2 g(n) \geq 1$

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

$\log_2 f(n) \leq (\log_2 c + 1)(\log_2 g(n))$

Thus $c' = \log_2 c + 1$ and $\log_2 g(n)$ is an increasing function, which means there is a large n that makes $\log_2(f(n)) \leq c' \log_2(g(n))$ true.

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

Let $f(n) = \log_2(n^2)$ and $g(n) = \log_2(n)$

$\log_2(n^2) = O(\log_2 n)$

Then $2^{f(n)} = 2^{\log_2(n^2)} = n^2$, $2^{g(n)} = 2^{\log_2(n)} = n$ $\longrightarrow n^2 \neq O(n)$

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

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

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

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

Then $c' = c^2$ which makes $f(n)^2 \leq c' g(n)^2$

so $f(n)^2 = O(g(n)^2)$

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).

$O(n^3)$

The nested for loop would take n^2 time in the worst case. Then adding array entries also takes a maximum of n time for a worst case of n^3. Thus the upper bound on run time is $O(n^3)$.

(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.)

$\Omega(f(n))$: $\exists c, n_0 > 0 \mid 0 \leq cg(n) \leq f(n) \quad \forall n \geq n_0$

Summing will take $j - i + 1$ time. When $i \leq n/4$ and $j \geq 3n/4$, $j - i + 1 \leq 3n/4 - n/4 + 1 < n/2$. There are $(\frac{n}{4})^2$ pairs fulfilling that. Thus, the algorithm will perform at least $\frac{n}{2} \cdot (\frac{n}{4})^2 = \frac{n^3}{32}$ operations $c = \frac{1}{32}$ making $0 \leq cn^3 \leq n^3$, so the algorithm is $\Omega(n^3)$.

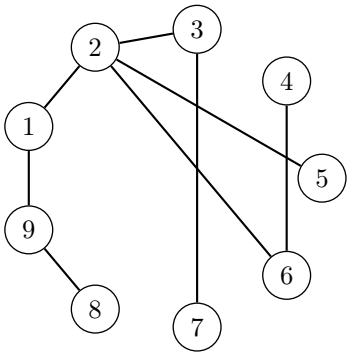
(c) Although the algorithm provided is the most natural way to solve the problem, it contains some highly unnecessary sources of inefficiency. Give a different algorithm to solve this problem, with an asymptotically better running time. In other words, you should design an algorithm with running time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

```
for i=1 to n
    sum = A[i]
    for j=i+1 to n
        sum = sum + A[j]
        store sum in B[i,j]
    endfor
end for
```

now only takes $O(1)$ time since we don't have to find the sum of the array entries each iteration.

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.



BFS: 1 2 9 3 5 6 8 4 7

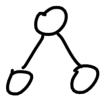
DFS: 1 2 3 7 5 6 4 9 8

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.

C = # nodes with 2 children L = # of leaves

Prove $C = L - 1$

Base case: $C = 1$, one node has 2 children means



the figure on the left appears only once in the tree, thus there are 2 leaves in the tree and $C = L - 1 = 2 - 1 = 1$ is true.

Induction hypothesis: $C = k$, a tree with k nodes with 2 children has k+1 leaves

→ Prove a tree with k+1 nodes with 2 children has k+2 leaves.

In order to add another node with 2 children it has to be appended to a leaf. Thus the number of leaves is reduced by 1. However, the 2 children become leaves, so the total number of leaves is $k+1-1+2 = k+2$. ■

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.

The claim is true.

Proof by contradiction:

Assume G is not connected. This means that there are at least 2 connected components in G. If every node of G has a degree of at least $\frac{n}{2}$, then a node has to connect to at least $\frac{n}{2}$ other nodes. Thus, each connected component has $\frac{n}{2}+1$ nodes. Since there are at least 2 connected components, the graph G has at least $2(\frac{n}{2}+1)$. Thus the minimum number of nodes in G is $n+2$ which is not equal to n. Therefore by contradiction, G must be a connected graph. ■

Coding Question

7. Implement depth-first search in either C, C++, C#, Java, or Python. 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 week 1's 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. You can assume that the nodes are listed line-by-line in lexicographic order (0-9, then A-Z, then a-z), and the adjacent nodes of a node are listed in lexicographic order. For example, consider two consecutive lines of an instance:

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

   Note that 0 < B and 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:

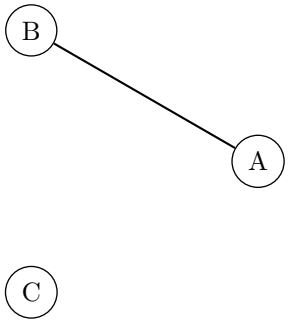
   Input:

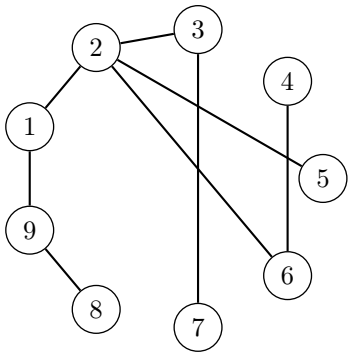
   ```
   2
   3
   A B
   B A
   C
   9
   1 2 9
   2 1 6 5 3
   4 6
   6 2 4
   5 2
   3 2 7
   7 3
   8 9
   9 1 8
   ```

   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.


B
A
C


3
2
4
1
5
9
6
8
7