CS 170 Midterm 1 Solutions

Write in the following boxes clearly and then double check.

Name :	Oski Bear
SID :	1234567890
Exam Room :	◯ Dwinelle 145 ◯ Hearst Field Annex A1 ◯ VLSB 2050 ◯ Other (Specify):
Name of student to your left :	
Name of student to your right :	

- **The exam will last 110 minutes.**
- The exam has 10 questions with a total of 100 points. You may be eligible to receive partial credit for your proof even if your algorithm is only partially correct or inefficient.
- Only your writings inside the answer boxes will be graded. **Anything outside the boxes will not be graded.** The last page is provided to you as a blank scratch page.
- Answer all questions. Read them carefully first. Not all parts of a problem are weighted equally.
- Be precise and concise.
- The problems may **not** necessarily follow the order of increasing difficulty.
- The points assigned to each problem are by no means an indication of the problem's difficulty.
- The boxes assigned to each problem are by no means an indication of the problem's difficulty.
- Unless the problem states otherwise, you should assume constant time arithmetic on real numbers. Unless the problem states otherwise, you should assume that graphs are simple.
- If you use any algorithm from lecture and textbook as a blackbox, you can rely on the correctness and time/space complexity of the quoted algorithm. If you modify an algorithm from textbook or lecture, you must explain the modifications precisely and clearly, and if asked for a proof of correctness, give one from scratch or give a modified version of the textbook proof of correctness.
- Assume the subparts of each question are **independent**.
- Please write your SID on the top of each page.
- Good luck!

1 Big O (10 points)

True or false (1 point each).

1. $n^{70} = O(n^{170})$ ◯ True ◯ False
2. $\log(n^{170}) = O(\log(n^{70}))$ ◯ True ◯ False
3. $170^n = O(70^n)$ ◯ True ◯ False
4. $70^n + 170^n = O(240^n)$ ◯ True ◯ False

For the following parts, provide the tightest big O bound that you can.

5. (1 point) If $T(n) = 170T\left(\frac{n}{70}\right) + O(n)$, then $T(n) = O(?)$

6. (1 point) If $T(n) = 70T\left(\frac{n}{170}\right) + O(n)$, then $T(n) = O(?)$

7. (4 points) If $T(n) = T(n-1) + 2 \cdot T(n-2)$ (We have $T(1) = 1$ and $T(2) = 2$), then $T(n) = O(?)$
Note: In case you need to use the quadratic formula, the solutions to the equation $ax^2+bx+c = 0$ are

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Exam continues on next page ⟹

Solution:

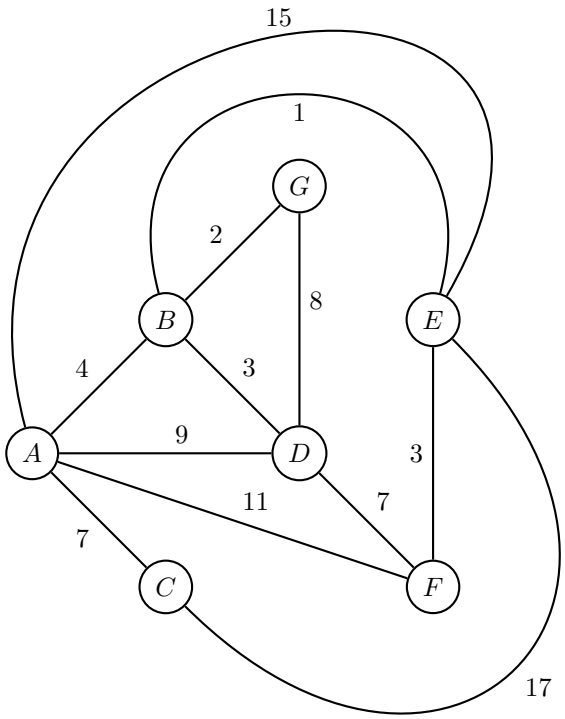
1. True.
2. True.
3. False.
4. True.
5. We can use the master theorem. Here, $a = 170, b = 70$, and $d = 1$, so $T(n) = O(n^{\log_{70}(170)})$.
6. We can use the master theorem. Here, $a = 70, b = 170$, and $d = 1$, so $T(n) = O(n)$.
7. We can lower bound this with $T(n) \geq 3T(n-2)$. This yields $T(n) = \Omega(3^{n/2}) \approx 1.732^n$. We can upper bound this with $T(n) \leq 3T(n-1)$. This yields $T(n) = O(3^n)$. Therefore, we can guess that it is an exponential. So, $T(n) = a^n$ for some $a \in [1.732, 3]$. Plugging this back in, we get $a^n = a^{n-1} + 2a^{n-2}$, or $a^2 - a - 2 = 0$. Solving this using the quadratic formula yields

$$a = -1, 2$$

Since we know $\sqrt{3}$ is a lower bound on the value of a, we must have $a = 2$, and we get that $T(n) = \boxed{\Theta(2^n)}$.

Exam continues on next page ⟹

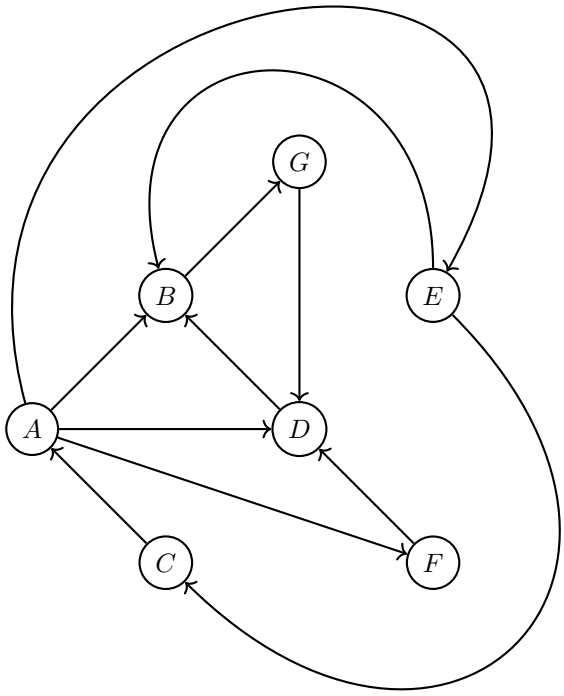
2 Parth's Paths (7 points)



1. (4 points) Run Dijkstra's Algorithm on the above graph for 5 iterations, starting from node C. For each iteration, fill in the below table with the weights of the nodes in the heap as well as the current node that was popped. For vertices that have been popped from the heap already, just fill in the distance to that vertex. The first iteration has been filled in for you already. Break ties in alphabetical order (so node Y before node Z).

Iteration	Curr Node	A	B	C	D	E	F	G
Start	N/A	∞	∞	0	∞	∞	∞	∞
1	C	7	∞	0	∞	17	∞	∞
2								
3								
4								
5								

Exam continues on next page



2. (3 points) Compute all of the strongly connected components (SCC) of the above graph. For ease of grading, please fill in the SCCs in alphabetical order (and the vertices in each SCC also in alphabetical order). For example, if the SCCs were $\{U, V\}$ and $\{T, W\}$, you should write $\{T, W\}$ before $\{U, V\}$. If there are extra boxes left over, you may leave them blank.

SCC 1

SCC 2

SCC 3

SCC 4

SCC 5

SCC 6

SCC 7

SCC 8

Exam continues on next page ⟹

Solution:

1.

Iteration	Curr Node	A	B	C	D	E	F	G
Start	N/A	∞	∞	0	∞	∞	∞	∞
1	C	7	∞	0	∞	17	∞	∞
2	A	7	11	0	16	17	18	∞
3	B	7	11	0	14	12	18	13
4	E	7	11	0	14	12	15	13
5	G	7	11	0	14	12	15	13

2. $\{A, C, E\}, \{B, D, G\}, \{F\}$

Exam continues on next page ⟹

3 Guess the Number (8 points)

Ajit has a positive integer n, and James has to figure out what it is. James may guess an integer x, and Ajit will say whether $n > x$, $n = x$, or $n < x$. Please help James use the least amount of guesses possible to figure out Ajit's integer.

1. (3 points) If we know $n \leq B$ for some positive integer B, describe an algorithm that uses $O(\log B)$ guesses to guess n.

2. (5 points) If there are no bounds on n (n can be any positive integer), describe an algorithm that uses $O(\log n)$ of guesses to guess n. Please explain its runtime as well.

Exam continues on next page ⟹

Solution:

1. We can use binary search; we start with the range $[0, B]$ and divide it in half with each guess. This takes around $\log B$ guesses which is sub-linear.

2. First, we guess $1, 2, 4, 8, \ldots$, the powers of 2, until we get to something that is greater than or equal to n. At this stage, we know n is less than our last guess, so we can transition to using binary search to guess n. We have used a total of around $2 \log n$ guesses, which is sub-linear.

Exam continues on next page ⟹

4 Pairing Param's Pears (8 points)

Param loves pears. He also loves pairing up his pears. Param has $2N$ pears, each located at a distinct integer location on the number line. The locations have already been sorted in increasing order. He wishes to pair up his pears, forming N pairs. However, he would like to minimize the sum of distances between the pears in each pair. Design an efficient algorithm to pair up the pairs while minimizing the sum of distance between each of the pairs, and prove its correctness using an exchange argument. **You do not have to analyze its runtime.**

Exam continues on next page ⟹

Solution: We want to pair up the first two pears, then the next two, etc. Proof of correctness. Assume the leftmost pear is not paired with the second left most pear. Let the leftmost pear is located at a and the second left most pear be located at b, so $b > a$. Let c be the location of a's partner and d be the location of b's partner. Currently, the distance is $c - a + d - b$. However, if we pair up the left two pears, the distance would be $b - a + |c - d|$. If we assume that $d > c$, then we know that $d > c > b > a$, so

$$c - a + d - b > c - a + d - b + 2 * (b - c) = d - c + b - a$$

On the other hand, if we assume $c > d$, then we know that $c > d > b > a$, so

$$c - a + d - b > c - a + d - b + 2 * (b - d) = c - d + b - a$$

Therefore, pairing up the two leftmost pears yields a smaller total distance! This shows that the leftmost two pears must be paired together. Applying induction proves the correctness of our algorithm. For the runtime, we have to sort the pears, so the runtime is $O(n \log n)$.

Exam continues on next page ⟹

5 Maximize the Dot Product (10 points)

Richard has an array A of n integers and an array B of m integers with $\log n < m < n$. He wishes to compute the maximum dot product between B and every contiguous subarray of A that has length m. The dot product of 2 arrays p and q (both of length r) is defined as

$$p \cdot q = \sum_{i=1}^{r} p_i q_i$$

where p_i is the i-th element of p (and q_i is the i-th element of q).

1. (1 point) Compute the maximum dot product for $A = [1, 4, 2, 5, 1]$ and $B = [2, -1]$.

2. (9 points) Describe an algorithm to calculate the maximum dot product. Your algorithm must run faster than $O(mn)$ time. Also prove its correctness and analyze its runtime.

Exam continues on next page

Solution:

1. 9

2. We can use FFT. We reverse v, encode both v and a into a degree-n polynomial, and then multiply them using FFT in time $O(n \log n)$. Finally, we take the maximum coefficient of the middle $n - m + 1$ terms, which gives us our answer. Therefore, the overall time complexity is $O(n \log n)$.

Exam continues on next page ⟹

6 Igloo Part 1 (7 points)

PNPenguin is in a rectangular igloo, which consists of a $m \times n$ grid of rooms. PNPenguin must travel through all of the rooms. However, each room is separated from each of the 4 adjacent rooms by a door, which requires some amount of energy to open. Once a door is opened, it stays open forever. Rooms on the edge of the grid or the corner of the grid are only connected to the 3 or 2 adjacent rooms, respectively. Let c_{AB} denote the cost of opening a door between rooms A and B (they must be adjacent). PNPenguin starts at the upper left room, located at $(0, 0)$, but doesn't care where he ends up as long as he has visited every room. Design an efficient algorithm to help PNPenguin compute the minimum amount of energy needed to open some set of doors such that PNPenguin can travel through all of the rooms. **You do have to prove its correctness but you do not have to analyze its runtime.**

Exam continues on next page ⟹

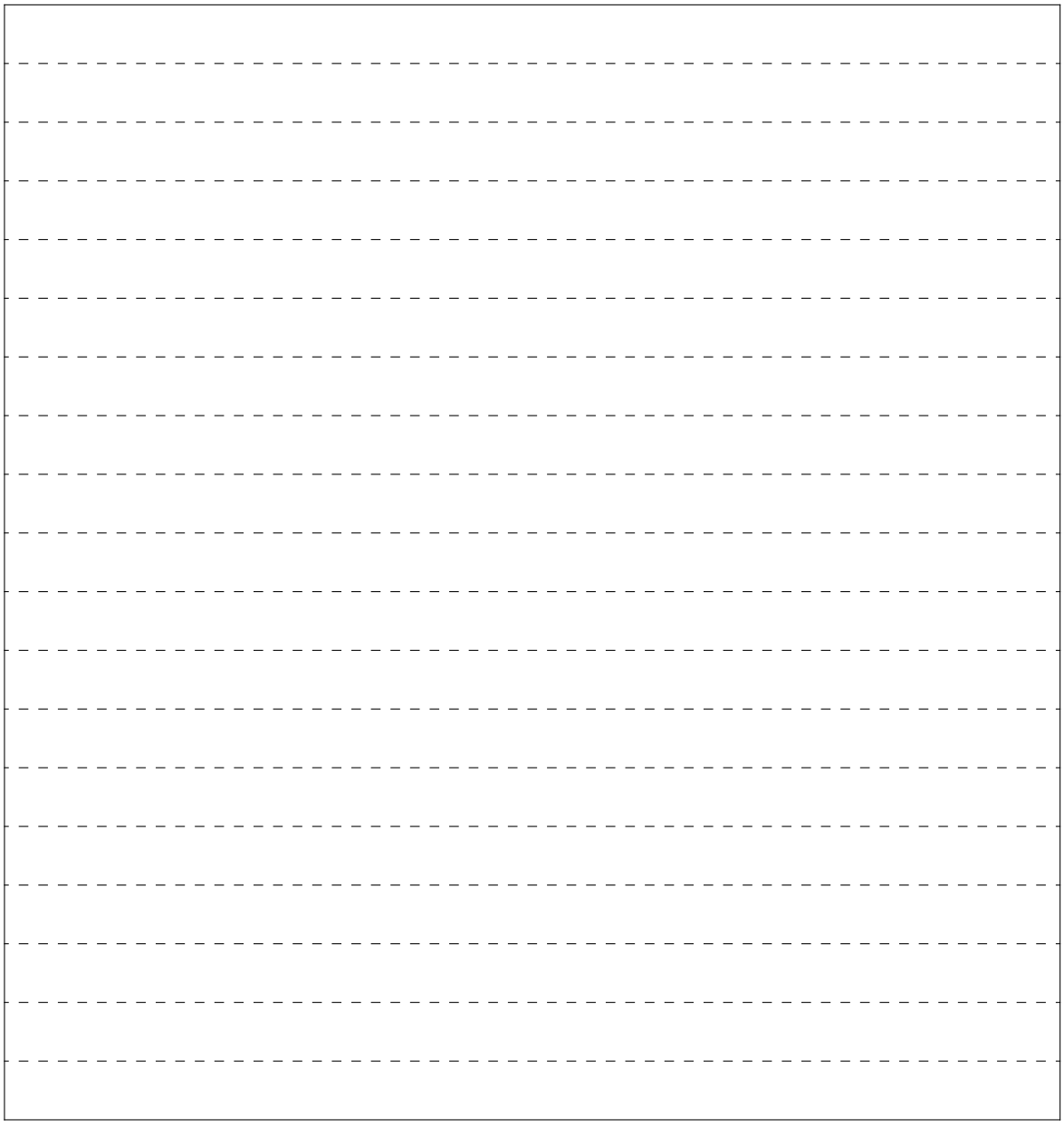
Solution: We can construct a minimum spanning tree where the vertices are each of the rooms and the edges represent the doors between adjacent rooms. The weight of each edge is the cost to open each door. Since the MST is the lowest-weight connected graph that spans all vertices, the set of doors opened from this algorithm corresponds to the lowest cost to connect all rooms.

Exam continues on next page ⟹

7 Igloo Part 2 (10 points)

PNPenguin has a rectangular igloo, which consists of a $m \times n$ grid of rooms; each room is separated from each of the 4 adjacent rooms by a door. Rooms on the edge or the corner of the grid are only connected to the 3 or 2 adjacent rooms. Every minute, PNPenguin chooses one of the doors and opens it; now, the two rooms are connected. Every time PNPenguin opens a door, he would like to know how many **new** pairs of rooms are connected (i.e. rooms A and B were not connected before the door was opened, but are connected after the door was opened). Rooms A and B are connected if and only if it is possible to get from room A to room B through a sequence of open doors.

Describe an algorithm that can compute the number of pairs of rooms that are connected. Your algorithm must run in $O(\log(mn))$ time every time a door is opened. Also prove its correctness and analyze its runtime. Your algorithm can store the state of the grid (it doesn't have to start from scratch every time a door is opened), along with other variables if necessary.



Exam continues on next page

Solution: For this problem we can represent the grid as a graph where the vertices are rooms and the edges are opened doors between two rooms. We can use Union Find. Every time a door is opened, we union the two rooms (unless they are already in the same connected component). Now, we have to compute how many new pairs of rooms are connected through the opening of this door. If the door is between rooms A and B, then the number of new pairs of rooms is the number of rooms that was previously connected to A times the number of rooms that was previously connected to B. We can compute this in $\log(mn)$ through our union find data structure.
Every time a door is opened, we only need to run 2 finds and at most 1 union, as well as some arithmetic operations. Finds and unions take $O(\log(mn))$ time since there are mn vertices. Therefore, that is the runtime of the algorithm.

Exam continues on next page ⟹

8 Shalin's Shuffles (12 points)

Shalin is playing a game where he shuffles a deck of n cards, numbered from 1 to n. Then, for each card c drawn from top of the deck, he gets points equal to the number of cards drawn before c that are of smaller value. Note that by construction of the game he is guaranteed to get 0 points from the first (topmost) card picked. His final score is the sum of scores gained from every individual card.

Given an array A that is the permutation of the deck of cards, with the first element corresponding to the topmost card in the deck, design a divide and conquer algorithm to output the total score from that permutation of cards. Also prove its correctness and analyze its runtime.

Example: if $n = 3$ and $A = [2, 1, 3]$, Shalin's total score is 2. He gets 0 points from drawing 2, 0 points from drawing 1, and 2 points from drawing 3.

Exam continues on next page ⟹

Solution: Here, we can employ a divide and conquer strategy. For each card, we want to calculate the number of cards to the left of it that have a value smaller than it. If we have an array A consisting of the values of 1 to n, we can split it up in half and calculate each half separately. Let's say the halves are B and C respectively. So, for each card in B, we already know the number of cards in front of that card that have a lower number. However, for each card in C, we only know the number of cards in front of that card in C that have a lower number; we still have to add the number of cards in B (the first half) that have a lower number. The way we do this is through merging (same merging as in merge sort). Whenever we add a card in B to the merged array, we do nothing. Whenever we add a card in C to the merged array, we have to increment the number of points by the number of cards in B that are smaller than it; this is equal to the number of cards in B that are in the merged array. We can store this number as a counter and increment it every time we add a card in B to the merged array. This gives a runtime of $T(n) = 2T(n/2) + O(n)$, which is $O(n \log n)$ by the Master Theorem.

Exam continues on next page ⟹

9 Maximize the Number (13 points)

Jonathan has an integer n and would like to maximize this integer by performing some sequence of swaps, where two adjacent digits are swapped. For example, if we swap the 1 and the 7 in 710, the resulting number would be 170. However, Jonathan can only perform at most k swaps. Please help Jonathan determine the maximum integer possible after performing at most k swaps.

1. (3 points) If $k = +\infty$, describe an algorithm that runs in $O(\log n)$ time that can determine the largest we can make n. **You do not need to prove its correctness, but you do need to analyze its runtime.**

2. (1 point) If $n = 1263$ and we are allowed 1 swap, what is the largest we can make n?

3. (1 point) If $n = 1263$ and we are allowed 2 swaps, what is the largest we can make n?

Exam continues on next page ⟹

4. (8 points) Describe an algorithm that runs in polynomial time (in terms of $\log n$ and k) that can compute the maximum integer possible after performing at most k swaps. **You do not need to prove its correctness nor analyze the runtime.**

Exam continues on next page ⟹

Solution:

1. If we have unlimited swaps, we can basically choose any permutation of the digits of n. The optimal permutation is one where the digits are sorted in descending order, and we can use an exchange argument to prove this.
   Let the digits outputted by our number be $A = a_1a_2 \dots a_k$ and let the maximum permutation of the digits of n be $B = b_1b_2 \dots b_k$. If our algorithm is wrong, then there exists an index i such that $a_i \neq b_i$. However, since our algorithm selects the maximum digit remaining, we know that $a_i > b_i$. But this contradicts the fact that $B > A$! Therefore, our algorithm is correct.
   For runtime, we can count the number of times each digit appears and output 9 however many times 9 appears, followed by 8 however many times 8 appears, etc. This is counting sort, which runs in $O(\log n)$ time.

2. Here, it is optimal to swap the 2 and the 1, yielding an answer of 2163.

3. Here, it is optimal to move the 6 to the front, yielding an answer of 6123.

4. In general, notice that it is optimal to move the largest digit to the front if we can. We can prove this by the exchange argument. Assume that we have enough swaps to move the largest digit to the front, but the maximum number that is attainable does not start with that digit. However, if we simply move the largest digit to the front, then we have found a greater number that is attainable, thus contradicting the maximality of the maximum number. Therefore, it is optimal to move the largest digit to the front.
   Alternatively, you can also say that it is optimal to move the maximum digit in the first $k + 1$ digits to the front, using a similar argument to above. If the maximum digit appears multiple times, it is optimal to move the left-most occurrence to the front. Here k is the number of swaps we have left. After we move the largest digit to the front, we can focus on the rest of the number and repeat the process.
   Once we reach $k = 0$ then we are done.

Exam continues on next page ⟹

10 Wilson's Werewolves (15 points)

n friends are playing a game. k of them are werewolves, while the other $n-k$ of them are villagers. The game is set in a cave with some number of rooms, and rooms may be connected by corridors. At the start of the game, there is a set of broken rooms in the cave that require repairs, and each of the players spawns in a random room. There is no limit on the number of players that can spawn in each room, and it is possible for a player to spawn in a broken room. Everyone then rushes to the closest broken room (determined by number of corridors to get to the room) to either help fix the cave (villager), or pretend to help fix the cave (werewolf). **We want to determine how many villagers are in danger because they end up in the same broken room as a werewolf.**

Assume that the cave can be represented as an unweighted, undirected graph $G = (V, E)$, where the vertices represent rooms and the edges represent corridors connecting rooms. The set of broken rooms is $B \subseteq V$. Further assume that all the players take the shortest path to the closest broken room, and there are no ties. Finally, assume that every room has the capacity to hold any number of players.

Design an efficient algorithm to solve this problem and provide its runtime in terms of any of $n, k, |V|, |E|, |B|$; **proof of correctness is not required.**

Exam continues on next page ⟹

Solution:

Algorithm:

Method 1:

Create an auxiliary node s and create edges between s and each broken room $b \in B$. Then, run BFS from source node s to generate the shortest paths between each player and the closest broken room. From here, we already know which player ends up in which broken room since the BFS will generate a tree. We can traverse the tree using DFS, keeping track of the current broken room, and any villagers or werewolves we encounter will end up in that broken room. Thus, we go through each broken room, and if there is a werewolf that ends up there, we add the number of villagers to our answer.

Method 2 (synopsis):

Essentially the same as Method 1 except instead of creating an auxiliary node s, we can just add all of the broken rooms $b \in B$ to the queue at the start of running BFS. The rest of the algorithm is the same after that.

Runtime: The initial BFS is $O(|V| + |E|)$, and the final summation is $O(|B|) = O(|V|)$. Thus, the overall runtime is

$$O(|V| + |E|)$$

Blank scratch page.
This page **will not be graded**.