# Homework 3

1. Consider a collection of $n$ ropes which have lengths $L_1, L_2, \ldots, L_n$, respectively. Two ropes of length $L$ and $L'$ can be connected to form a single rope of length $L + L'$, and doing so has a cost of $L + L'$. We want to connect the ropes, two at a time, until all ropes are connected to form one long rope. Design an efficient algorithm for finding an order in which to connect all the ropes with minimum total cost. You do not need to prove that your algorithm is correct. (20 points)

   Ans:
   Explanation:
   The best approach to get the minimum total cost is by adding the 2 shortest lengths at every point, which in turn would give us the lowest possible cost (for that iteration) after every iteration. For this, we use min heap. We pop the first 2 ropes from the heap (2 shortest ropes) and add the 2 lengths to get the cost, and put the sum back in the heap. At every step, we store the 2 ropes that are being used in an empty list as a tuple to get the order of the rope. We do this till we have 1 rope remaining in the list. This 1 rope would be the final rope with the minimum total cost.

   Algorithm:
   Form a min heap
   Add all the lengths of ropes in the min heap.
   Initialize total_cost to store the total cost.
   Initialize rope_order as an empty list to store the order of ropes.
   Initiate a while loop till the heap has only 1 value left.
   In the while loop:
   Pop 2 ropes that have the smallest length from the heap.
   Add them and store it in r_combined variable.
   Add r_combined to the value of total_cost, and store it in total_cost.
   Send r_combined back to the heap.
   Append the 2 ropes as a tuple in rope_order
   End Loop
   Print total_cost to give the minimum total cost.
   Print rope_order to give the order of ropes connected.

2. Suppose you want to drive from USC to Santa Monica. Your gas tank, when full, holds enough gas to drive *p* miles. Suppose there are *n* gas stations along the route at distances $d_1 \leq d_2 \leq \ldots \leq d_n$ from USC. Assume that the distance between any neighboring gas stations, and the distance between USC and the first gas station, as well as the distance between the last gas station and Santa Monica, are all at most *p* miles. Assume you start from USC with the tank full. Your goal is to make as few gas stops as possible along the way. Design an efficient algorithm for determining the minimum number of gas stations you must stop at to drive from USC to Santa Monica. Prove that your algorithm is correct. Analyze the time complexity of your algorithm. (25 points)

   Ans:
   Explanation:
   In this question, we start from USC with a full gas tank. At every gas station, we have to check whether we can reach the next gas station with the amount of gas left in our tank or not. If we can, we proceed to the next gas station. If not, we fill our tank on the gas station we are at that moment. This continues till we reach Santa Monica.

   Algorithm:
   Initialize variables current_station to get the current gas station at which the we are
   Initialize gas_stops as a list to get all the gas stops where we stop to fill the tank.
   Initialize gas_left to track the level of gas left in the tank. Start with gas_left = p.
   Initiate a for loop for distance till we don't have any more distance left to travel (till we reach Santa Monica)
   If we can reach the next station, then travel to the next station. Subtract the distance of gas station from gas_left.
   If not, stop and refuel your tank and reset gas_left to p. Append current_station to gas_stops.
   End Loop
   Print gas_stops.

   Proof of Correctness:
   Suppose this algorithm returns a list A = ($a_1$, $a_2$, $a_3$, …, $a_n$). Let's assume that the optimal solution is O = ($o_1$, $o_2$, $o_3$, ..., $o_n$). Since the algorithm has chosen $a_1$ as the gas station to fill our tank, it is not possible to travel to next gas station $a_2$ as our tank would finish before reaching it. Hence, the gas station in the optimal solution would be $o_1 <= a_1$.
   If this is true, then we can swap $o_1$ with $a_1$ as it is the latest station that can be travelled. This won't change the total number of gas stations.
   This argument can be considered true for all the stations. Hence, we can generalize this by saying that for any $a_r$ gas station, $o_r <= a_r$. At every point, this algorithm would have

its gas station ahead of the gas station of the optimal solution. Hence, this solution can be considered as an optimal solution.

Time complexity:
The time complexity of this algorithm is O(n) since we travel through every gas station at least once.

3. Suppose you are given two sequences $A$ and $B$ of $n$ positive integers. Let $a_i$ be the $i$-th number in $A$, and let $b_i$ be the $i$-th number in $B$. You are allowed to permute the numbers in $A$ and $B$ (rearrange the numbers within each sequence, but not swap numbers between sequences), then you receive a score of $\prod_{1 \le i \le n} a_i^{bi}$. Design an efficient algorithm for permuting the numbers to maximize the resulting score. Prove that your algorithm maximizes the score and analyze your algorithm's running time (25 points).

   Ans:
   Explanation:
   In order to get the maximum possible score, we have to pair the largest number in A with the largest number in B, 2nd largest number in A with 2nd largest number in B and so on. For this, we have to sort both the sequences in a descending order. Then, we perform $\prod_{1 \le i \le n} a_i^{bi}$ while iterating through 1 to n. We add the value to a variable inside the loop, and then print that variable after the termination of the loop.

   Algorithm:
   Sort both the sequences A and B in descending order.
   Initialize max_score to store the product. We start with max_score being 1.
   Initiate for loop from 1 to n (number of elements in the sequences).
   Multiple $a_i^{bi}$ with max_score and store it in max_score.
   End Loop
   Print max_score.

   Time Complexity:
   The time complexity of this algorithm is O(nlog(n)). This is because sorting the sequence takes O(nlog(n)) and the for loop takes O(n) time. Since O(nlog(n)) is the dominant step, the overall time complexity is O(nlog(n))

4. The United States Commission of Southern California Universities (USC-SCU) is researching the impact of class rank on student performance. For this research, they want to find a list of students ordered by GPA containing every student in California. However, each school only has an ordered list of its own students sorted by GPA and the commission needs an algorithm to combine all the lists. Design an efficient algorithm with running time $O(m \log n)$ for combining the lists, where $m$ is the total number of students across all colleges and $n$ is the number of colleges. (20 points)

   Ans:
   Explanation:
   We assume that the sorted list of students is ordered by increasing order of their GPA. In this question, we use min heap. Add all the first elements from the list of different colleges in the heap (since each college has its own sorted list of students in their increasing order of GPA, we get the lowest GPA scores of the college from their list). This gives us a min heap of all the lowest GPAs of individual colleges. Pop the smallest GPA from this heap and append the GPA to the final list. Add the GPA of the next student to the heap. Next student is chosen from the college that the earlier student, who was added to the final list, was. Continue this till the heap is empty. This would give us the final list of students in an increasing order of their GPA from all colleges.

   Algorithm:
   Form a min heap
   Initialize final_list to get the final sorted list in increasing order of students' GPAs.
   Initialize a for loop to iterate through the list of colleges.
   Add the first element of each college to the heap.
   End Loop
   Initialize a while loop till heap is empty.
   Pop the first student from the heap.
   Append it to the final_list.
   Add the next student of the same college to the heap.
   End Loop
   Print final_list.

   Time complexity is O(mlog(n)) as the pop and add from the min heap takes log(n) time to execute, and it is repeated till all students (m) are visited, giving us the total complexity of O(mlog(n)).

5. The array *A* below holds a max-heap. What will be the order of elements in array *A* after a new entry with value 19 is inserted into this heap? Show all your work.
*A* = {16, 14, 10, 8, 7, 9, 3, 2, 4, 1} (10 points)

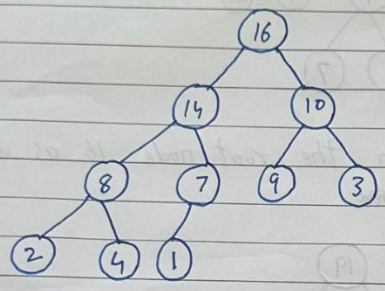
Ans:

Q-5-

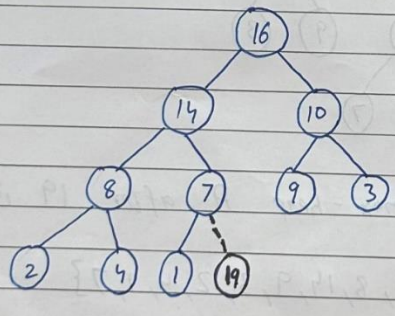
A = {16, 14, 10, 8, 7, 9, 3, 2, 4, 1}

Insert 19.

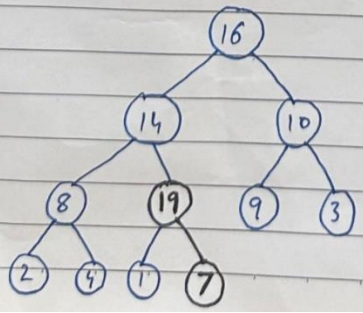
ans- The binary tree for A is :-



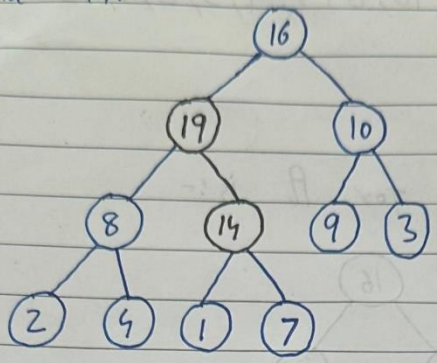

19 is inserted at the end (child node of 7)



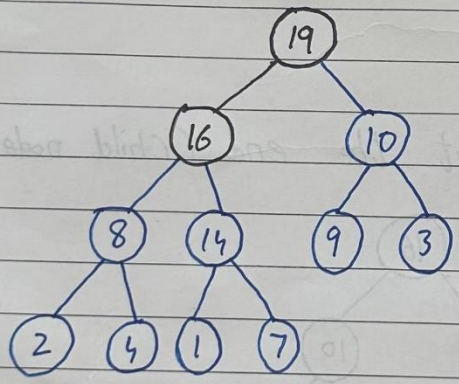

Since 19 is more than its parent node 7, we swap 19 and 7.



Since 19 is more than its new parent node 14, we swap 14 and 19.




Finally, 19 is more than the root node 16 as well, hence, 16 and 19 are swapped




This is the final max-heap A after 19 is inserted.

$\therefore A_{new} = \{19, 16, 10, 8, 14, 9, 3, 2, 4, 1, 7\}$