HW2

1. **We have N ropes having lengths L1, L2, . . . , LN . We can connect two ropes at a time: Connecting ropes of length L and L′ gives a single rope of length L + L′ and doing so has a cost of L + L′. We want to repeatedly perform such connections to finally obtain one single rope from the given N ropes. Develop an algorithm to do so, while minimizing the total cost of connecting. No proof is required. (10 points).**

   Solution: The sorting of ropes can be achieved using a min-heap. The process involves repeatedly selecting the two ropes with the lowest cost from the heap, connecting them, and inserting the resulting combined length back into the heap. This procedure continues until there is only one string remaining in the heap.

2. **There are N tasks that need to be completed using 2 computers A and B. Each task i has 2 parts that take time: ai (first part) and bi (second part) to be completed. The first part must be completed before starting the second part. Computer A does the first part of all the tasks while computer B does the second part of all the tasks. Computer A can only do one task at a time, while computer B can do any amount of tasks at the same time. Find an O(n log n) algorithm that minimizes the time to complete all the tasks, and give a proof of why the solution obtained by the algorithm is optimal. (15 points).**

   Solution: The solution would include each task to be arranged in descending order of the time that it takes to complete part bi with no gaps between part ai of each task. The time complexity in this case would be O(nlogn) since the only steps included in the solution would be sorting the tasks.

   Proof:

   a. An optimal solution with gaps or idle time between the first half of each task can have no gaps and still remain optimal.

   b. A inversion can be defined as when a task with second part finish time bi precedes a task with second part finish time bj and bi < bj. All solutions with no inversions and no idle time would result in the same completion time.

   c. An optimal solution with no idle time and no inversions would still give an optimal solution.

   Taking point b and point c into consideration, we can conclude that our solution would result in a single solution which will also be an optimal solution.

3. **Suppose you want to drive from USC to Santa Monica. Your gas tank, when full, holds enough gas to go p miles. Suppose there are n gas stations along the route at distances d1 ≤ d2 ≤ ... ≤ dn from USC. Assume that the distance between any neighbouring 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. Give the most efficient algorithm to determine which gas stations you should stop at and prove that your algorithm yields an optimal solution (i.e., the minimum number of gas stops). Give the time complexity of your algorithm as a function of n. (15 points)**

   Solution: The solution would be to travel till the i'th station that is under 'p' miles from last refuelled station. The first refuel would be the base since we already start with a full tank. After reaching the i'th station, if the car can travel till the i+1st station safely without running out of fuel, then the car will move on otherwise the car will refuel to full tank. This solution follows the greedy strategy.

   Let's assume {g1,g2,...,gm} is our selected solution set, representing the gas stations where we refuel based on our algorithm. We'll contrast this with {h1, h2, ... , hk}, an alternative, optimal solution.

   Due to the impossibility of reaching the gas station g1 + 1 without refueling, any alternative solution must also refuel at a station on or before g1. Therefore, h1 must be less than or equal to g1. If h1 is less than g1, we can swap the two, creating a new solution {g1, h2 , ... , hk}, which is just as effective because the fuel level upon leaving g1 remains the same or greater than before.

   For an inductive step, let's assume {g1, g2, ... , gc−1, hc, ... , hk} is optimal. Our algorithm's approach means that hc cannot be greater than gc. If hc is less than gc, swapping these two results in {g1,g2,...,gc−1,gc,hc+1,...,hk}, which is also valid. The reason it's valid is the same as before - upon leaving gc, we have at least as much fuel as before, ensuring we can reach our destination. This implies that {g1,g2,...,gc,hc+1,...,hk} is also an optimal solution.

   By this induction process, we show that our solution {g1, g2 , ... , gm} is indeed optimal. Given that we only make one decision at each station, the algorithm runs in O(n) time complexity.

4. **Suppose you are given two sets A and B, each containing n positive integers. You can choose to order the numbers in each set however you like. After you order them, let ai be the i'th number in set A, and let bi n be the i'th element of set B. You then receive a payoff of . Give an algorithm to decide the ordering of the numbers so as to maximize your resultant payoff (6 points). Prove that your algorithm maximises the payoff (10 points) and state its running time (2 points).**

Solution: Certainly, we can also sort both sets in descending order and form pairs of elements based on this order. This approach will also lead to the maximized payoff, but the justification differs slightly from the previous one.

- Sort set A in descending order.
- Sort set B in descending order.
- Pair elements from the two sets based on their new order.

Similar to the previous approach, the idea here is to maximize the payoff by pairing high-value elements together. Given that exponentiation is a rapidly increasing function, a larger base raised to a larger exponent will generally produce a larger result. By sorting both sets in descending order, we're ensuring that the highest values in each set are paired together.

Let's prove this using contradiction. Suppose there is a more optimal solution. This implies that there are at least two pairs (ai,bi) and (aj,bj) where ai > aj and bi > bj. If we swap bi and bj to form new pairs (ai,bj) and (aj,bi), the new product will be (ai^bj) * (aj^bi). Given that ai > aj and bi > bj, (ai^bj) * (aj^bi) > (ai^bi) * (aj^bj). Hence, our assumption that there is a better order is wrong. Therefore, our algorithm gives the maximum payoff.

The major time complexity contributor is the sort operation, which is O(n log n) for each set. Therefore, the overall running time of the algorithm is O(n log n).

5. **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 by GPA and the commission needs an algorithm to combine all the lists. Find the fastest algorithm for yielding the combined list and give its runtime in terms of m, the total number of students across all colleges, and n, the number of colleges. (12 points).**

   Solution: The solution would include merging all the school lists and sorting them in order of GPA. We will use merge sort to achieve this.

   Since all the school lists are already sorted, the merging step between an two lists can be done in O(m) time. The divide-and-conquer process repeats log n times because each step reduces the number of arrays by half. Hence, the total time complexity is O(m log n).

6. **Design a data structure that has the following properties (assume n elements in the data structure, and that the data structure properties need to be preserved at the end of each operation):**

• *Finding the median of all the n elements takes O(1) time.*
• *Inserting the element takes O(log n) time.*

Describe your data structure. Give the algorithms for Find-Median() and Insert() functions. (12 points).

Solution: Our data structure can be a combination of a max heap and a min heap. The max heap would store the lower half of the sorted n elements. In that sense, the max heap would have the maximum value from the lower half as the root node. Similarly, the min heap would store the upper half of the sorted n elements. In this sense, the min heap would store the minimum number for the upper half as the root node. The extract-min and extract-max from the min and max heaps will take O(1) time each.

Thus we would have the central elements which we can use to find the median of the n elements. Each insert operation on either heaps will take O(log n) time.

Algorithms:

```
function Find-Median() {
   if size(low) > size(high)
      return -top(low)    // The median is the top element of low
   else
      return ( -top(low) + top(high) ) / 2    // The median is the average of the tops of both heaps
}

function Insert(element) {
   if size(low) == 0 or element < -top(low)
      push(low, -element)    // Push to low if it's empty or the element is smaller than the top of low
   else
      push(high, element)    // Else, push to high
   // Rebalance the heaps
   if size(low) > size(high) + 1
      push(high, -pop(low))    // If low has more than 1 extra element, pop from low and push to high
   else if size(high) > size(low)
      push(low, -pop(high))    // If high has more elements, pop from high and push to low
}
```

7. **Given a connected graph G = (V,E) with positive edge weights. Let s and t be two given nodes for shortest path computation, prove or disprove with explanations (5 points each):**

(a) If all edge weights are unique, then there is a single shortest path between any two nodes in V .

(b) If each edge's weight is increased by k, the shortest path cost between s and t will increase by a multiple of k.

(c) If the weight of some edge e decreases by k, then the shortest path cost between s and t will decrease by at most k.

(d) If each edge's weight is replaced by its square, i.e., changed w to w2, then the shortest path between s and t will be the same as before (though possibly with a different cost.)

Solution:

(a) False. Suppose there are two nodes a and b between s and t such that
e(s, a) = 1
e(a, t) = 4
e(s, b) = 2
e(b, t) = 3
In this case all edges have unique weights but the graph has two shortest paths between s and t.

(b) False. Suppose there is a node a between s and t such that
e(s, a) = 1
e(a, t) = 1
e(s, t) = 3
Here the shortest path would be through node a.
However, if we increase all the edges by k=2 then the values would be
e(s, a) = 3
e(a, t) = 3
e(s, t) = 5
Now the shortest path would be the edge e(s, t) = 5 which is not a multiple of k.

(c) False. When edge e's weight decreases, the shortest path cost decreases by at most k if e is part of the shortest path and all edge weights remain positive. However, if a cycle becomes negatively weighted after the decrease, and a path from s to t passes through this cycle, the shortest path cost may not be bounded, contradicting the "decrease by at most k" assertion.

(d) False. Suppose there are two nodes a and b between s and t such that
e(s, a) = 1
e(a, b) = 1
e(b, t) = 1

e(s, t) = 2
Here the shorts path is the edge e(s, t)
However, after squaring each edge cost we will get
e(s, a) = 1
e(a, b) = 1
e(b, t) = 1
e(s, t) = 4
Thereby making the shortest path s→a→b→t thus contradicting the statement.

8. **Consider a directed, weighted graph G where all edge weights are positive. You are allowed to change the weight of any one edge to zero. Propose an efficient method based on Dijkstra's algorithm to find the lowest-cost path from node s to node t, given that you may set one edge weight to zero. Your algorithm must have the same running time complexity as the Dijkstra's algorithm. (15 points)**

   Solution: The task is to find the shortest path from s to all other vertices. You would first implement Dijkstra's algorithm to achieve this. Afterward, all edges are reversed, and Dijkstra's algorithm is applied again to identify the shortest paths from all vertices to t. Let's denote the shortest path from u to v as δ(u, v).

   To find the path that minimizes total weight with one edge set to zero, examine each edge (u, v) ∈ E in the original graph. Consider the path s u → v t. If we set w(u,v) to zero, the path length is δ(s, u) + δ(v, t). Identify the edge that yields the smallest path length when its weight is set to zero. The path s u → v t associated with this edge is the one we seek.

   The running time for this method is equivalent to that of Dijkstra's algorithm. The algorithm requires two instances of Dijkstra's (one for the original graph and one for the graph with reversed edges), and an additional O(E) time to go through all the edges and pinpoint the best one to set to zero.

   If Dijkstra's algorithm is implemented using a Fibonacci heap, both the original Dijkstra's and our proposed method have a time complexity of O(E + V log V). If a binary heap is used instead, the time complexity of Dijkstra's is O((E + V) log V). The proposed method's time complexity is then O(2(E+V) log V + E), which simplifies to O((E + V)log V) as E = O(E log V). Therefore, it matches the time complexity of Dijkstra's.