Homework 4

1. [10 points] 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):
   - Find median takes O(1) time
   - Insert takes O(log n) time

   Do the following:

   1. Describe how your data structure will work.
   2. Give algorithms that implement the Find-Median() and Insert() functions.

Data Structure Design

- A Max-Heap (Left Heap) to store the smaller half of the numbers.
- A Min-Heap (Right Heap) to store the larger half of the numbers.

Algorithm for Insert(x) in O(log n)

1. Insert into the correct heap:
   - If x is less than or equal to the maximum of the left heap (max-heap), insert it into the left heap.
   - Otherwise, insert it into the right heap (min-heap).
2. Rebalance the heaps to maintain size constraints:
   - If one heap has more than one extra element than the other, move the top element from the larger heap to the smaller heap.

Algorithm for Find-Median() in O(1)

1. If both heaps have the same size, return the average of their top elements.
2. If one heap has more elements, return its top element (max-heap if odd count).

Rubric (10pts):

3 pts: Data structure.

4 pts: Insert().

3 pts: Find-Median().

2. [10 points] In a busy grocery store, there are k self-checkout lanes, each with customers waiting in line. The store organizes these lanes so that within each line, customers are arranged in increasing order of their time of arrival—nobody skips ahead, and everyone follows the rules. The store manager wants to merge all k lines into a single checkout queue, ensuring that customers are still ordered by their arrival time. However, since the store is crowded, they need to do this as efficiently as possible. How can the manager merge these k sorted lines into one single sorted line in the fastest possible way, ensuring an optimal time complexity of O(n log k), where n is the total number of customers across all lines? Can you describe an algorithm that achieves this, justify its correctness, and explain why it runs in O(n log k) time?

Algorithm:

1. Initialize a Min-Heap with the first element from each of the k-sorted lists. The heap stores tuples

of the form (value, list_index, element_index) to keep track of which list and position the element came from.
2. Extract the minimum element from the heap and append it to the result list.
3. Insert the next element from the same list (from which the minimum was extracted) into the heap.
4. Repeat until all elements from all lists have been processed.

Time Complexity Analysis:

- Heap Initialization: We insert the first element of each list into the heap, which takes $O(k)$ time.
- Heap Operations: Since all lists have n total elements, we perform n insertions and deletions from the heap. Each insertion and deletion in a min-heap takes $O(\log k)$ time (since the heap contains at most k elements at any moment). Thus, the heap operations contribute to $O(n \log k)$ time.
- Final Complexity: $O(n \log k)$.

Rubric (10pts):

4 pts: Algorithm.

3 pts: Correctness argument.

3 pts: Time Complexity Analysis.

3. [5 points]A town wants to build roads connecting all neighborhoods at the lowest total cost, with each road having a unique construction cost. The mayor needs to know: Can there be more than one way to build this minimum-cost road network, or is the solution always unique? Prove why the answer must be yes or no.

Proof by contradiction:

1. If T1, T2 are Distinct, there must be at least one edge that's in T1 but not in T2 (and vice versa) since one MST cannot contain the other. Let e1 be the minimum weight edge among all such edges that are in exactly one but not both.
2. T2 + e1 contains a cycle, say C. C must contain e1 since C wasn't present in T2. Further, the edges in C other than e1 cannot all be in T1 since otherwise, T1 would contain the cycle C. Hence, there's an edge in C, say e2, which is in T2, but not in T1. However, since e1 has the minimum weight among edges that are in exactly one, but not both, $w(e1) < w(e2)$. The strict inequality is because we have all edge weights distinct.
3. Note that $T = T2 \cup \{e1\} \setminus \{e2\}$ is a spanning tree. The total weight of T is smaller than the total weight of T2, but this is a contradiction since we have supposed that T2 is a minimum spanning tree.

Rubric (5pts):

5 pts: Correctness.

4. [10 points] Suppose you are given a connected undirected graph where every edge weight is either 1 or 2. You want to find an MST.
   - Show that Prim's algorithm will always find an MST in $O(n \log n)$ time for this graph.
   - Can we do better than $O(n \log n)$? Justify your answer.

Solution:

[First part] Prim's algorithm works by:

1. Initializing a priority queue (min-heap) with the starting node.
2. Extracting the minimum-weight edge that connects a visited node to an unvisited node.
3. Adding the new node to the MST and updating the heap with its adjacent edges.
4. Repeating until all n nodes are included in the MST.

Time Complexity Analysis:

- The priority queue stores at most n elements at any time.
- Each node is inserted into the heap once, and each insertion takes $O(\log n)$ time.
- Each edge (at most $O(n)$ edges in a sparse graph, up to $O(n^2)$ in a dense graph) is processed at most once.
- Extracting the minimum from the heap takes $O(\log n)$, and we do this n times.
- Thus, the overall complexity of Prim's algorithm with a priority queue (min-heap) is: $O(n \log n)$ which holds regardless of the edge weights.

Since edge weights are restricted to 1 or 2, Prim's algorithm still runs in $O(n\log n)$ using a min-heap implementation.

[Second part] Yes, we can achieve $O(n)$ time complexity using a modified Breadth-First Search (BFS) approach, leveraging the special structure of edge weights.

Alternative Approach: Using a 0-1 BFS Strategy

Since the only edge weights are 1 or 2, we can:

- Use a double-ended queue (deque) instead of a priority queue.
- Process weight-1 edges before weight-2 edges, mimicking Dijkstra's algorithm optimized for small integer weights.

Algorithm:

1. Start at an arbitrary node and initialize an empty deque.
2. Insert the starting node into the deque.
3. Process nodes in BFS order:
    - If traversing an edge of weight 1, push the new node to the front of the deque.
    - If traversing an edge of weight 2, push the new node to the back of the deque.
4. Continue until all nodes are processed.

Time Complexity Analysis:

- Each node is visited once in $O(n)$ time.
- Each edge is processed at most once using deque operations, which take constant time.
- The total complexity is $O(n)$, better than the usual $O(n \log n)$.

Conclusion:

By taking advantage of the fact that edge weights are only 1 or 2, we avoid heap operations and achieve linear time. Prim's algorithm is general-purpose and runs in $O(n \log n)$, but we can do better with a specialized approach for this specific case. Thus, we can indeed do better than $O(n \log n)$ and achieve $O(n)$ complexity for this problem.

Rubric (10pts):

3 pts: Time Complexity Analysis.

4 pts: Correctness of Algorithm.

3 pts: Time Complexity of BFS.

5. Design an algorithm to find the maximum bandwidth path between a given source vertex s and a destination vertex t in a weighted graph G = (V, E), where each edge weight represents bandwidth capacity. The bandwidth of a path is defined as the minimum edge weight along that path (i.e., the bottleneck). Explain how to modify Dijkstra's algorithm to do this.

We will modify Dijkstra's algorithm to find a maximum weight path, where path weight is calculated as the minimum of all edge weights on the path. Dijkstra normally computes path weights as an incremental sum, fortunately, the minimum operation can also be done incrementally. Dijkstra's correctness relies on the **incremental and non-decreasing** nature of the sum operation for path weight, min shares these properties (non-increasing in this case since we looking for maximum weight) so Dijkstra will be correct for it as well.

Changes:

1. Instead of initializing each vertex to inf cost, initialize to 0 bandwidth.
2. Replace the min-priority queue with a max-priority queue (or, equivalently, do calculations with negative bandwidth in a min-priority queue).
3. Instead of updating a vertex if we find shorter path (decreaseKey), we update it if we find a higher bandwidth path (increaseKey, which replaced decreaseKey as we are using a max-priority queue).

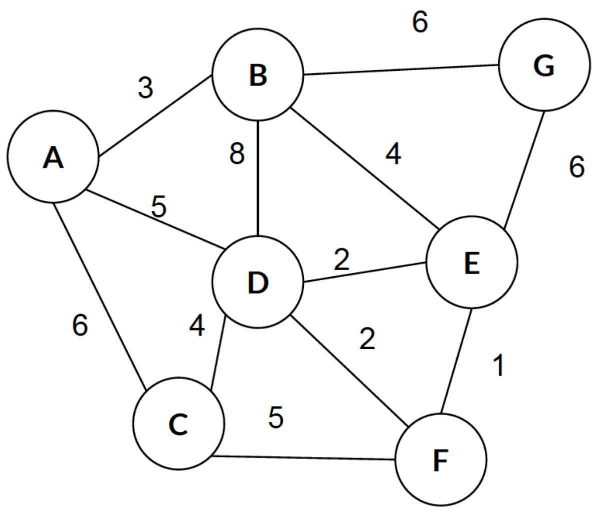
To change weight calculation from sum to minimum: When calculating cost for a path from a vertex, instead of adding the cost of the edge, we find the minimum of the edge and the vertex's best bandwidth (i.e. current cost).

Rubric (5pts):

5 pts: Correctness.

Ungraded Problems

6. Considering the following graph G in Fig 1:
   a. In graph G, if we use the Kruskals Algorithm to find the MST, what is the third edge added to the solution?
   b. In graph G, if we use the Prims Algorithm to find MST starting at A, what is the second edge added to the solution?
   c. What is the cost of the MST in the graph?



Solution.

a. A-B
b. B-E
c. 20

7. Given a connected graph G = (V, E) with positive edge weights. In V , s and t are two nodes for shortest path computation, prove or disprove with explanations:
   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., w to w2, then the shortest path between s and t will be the same as before but with different costs.

Solution.

a. False. Counter example: (s, a) with weight 1, (a, t) with weight 2, and (s, t) with weight 3. There are two shortest paths from s to t though the edge weights are unique.
b. False. Counterexample: suppose the shortest path s → t consists of two edges, each with cost 1, and there is also an edge e = (s, t) in G with cost(e)=3. If now we increase the cost of each edge by

2, e will become the shortest path (with a total cost of 5).

c. False.
   i. Only true when we have the assumption that after decreasing all edge weights are still positive (however we don’t have this in the problem). For any two nodes s, t, assume that P1, . . . , Pk are all the paths from s to t. If e belongs to Pi then the path cost decreases by k, otherwise the path cost is unchanged. Hence all paths from s to t will decrease by at most k. As the shortest path is among them, then the shortest path cost will decrease by at most k.
   ii. When 1) there is a cycle in the graph, 2) there is a path from s to t that goes through that cycle, 3) and after decreasing, the sum of edge weights of that cycle becomes negative, then the shortest path from s to t will go over the cycle for infinite times, ending up with infinite path cost, hence not “decrease by at most k”.

d. False. Counterexample: 1) suppose the original graph G is composed of V = {A, B, C, D} and E : (A → B) = 100, (A → C) = 51, (B → D) = 1, (C → D) = 51, then the shortest path from A to D is A → B → D with length 101. 2) After squaring this path length becomes 1002 + 12 = 10001. However, A → C → D has path length 512 + 512 = 5202 < 10001 Thus A → C → D becomes the new shortest path from A to D.