COMS 311: Homework 5
Due: Nov 17, 11:59pm
Total Points: 40

Late submission policy. If you submit by Nov 18, 11:59PM (and after the specified deadline), there will be 10% penalty. That is, if your score is x points, then your final score for this homework after the penalty will be $0.9 \times x$. Similarly, if you submit by Nov 19, 11:59PM (after Nov 18, 11:59PM), there will be 20% penalty.

Submission after Nov 19, 11:59PM will not be graded without (prior) explicit permission from the instructors.

Submission format. Your submission file should be in pdf format. Name your submission file: `<Your-net-id>-311-hw5.pdf`. For instance, if your netid is `asterix`, then your submission file will be named `asterix-311-hw5.pdf`.

You are required to type-set your solution. You can use word, latex and any other type-setting tool to type your solution; you need to make sure you export/save it in pdf before submission.

For questions where you are asked to write an algorithm, you are required to write pseudocode. Do not write code in any specific language such as Java, Python, C/C++, etc. You are also required to present the runtime of your algorithm. Part of the grade will depend on the efficiency of your algorithm.

Algorithm Design Problems

- **When asked to design/write an algorithm, write pseudo code, not code of any specific programming language and not library operations of any specific programming language library.**
- For any algorithm design problem, present some justification for correctness of the algorithm you are presenting. For instance, checking whether there exists a subarray whose sum is 0 depends on computing prefix sums. This is because if there are two prefix sums that are equal then there exists a subarray whose sum is 0.
- For any algorithm design problem, you will present the runtime of your algorithm. The derivation does not need to be detailed but sufficient for explaining the final runtime.
- For all algorithm design problems, part of the grade depends on its runtime efficiency.

Learning outcomes.

1. Divide and Conquer
2. Greedy Algorithms

---

1. You are given two sorted arrays each of size n and all the $2n$ elements in the arrays are distinct. Write an algorithm to find the median of $2n$ elements.

```
findMedian(array A, array B) {
    // BASE CASES
    if(A.size == 1) {
        return A[0]
    }
    if(B.size == 1) {
        return B[0]
    }

    // RECURSIVE CALLS
    if(A[size/2] > B[size/2]) {
        // Get subarray of left elements in A
        A = subarray(A, 0, size/2)
        // Get subarray of right elements in B
        B = subarray(B, size/2, size - 1)
    }
    else {
        // Get subarray of right elements in A
        A = subarray(A, size/2, size - 1)
        // Get subarray of left elements in B
        B = subarray(B, 0, size/2)
    }

    return findMedian(A, B)
}
```

Justification: We can find the median of each sub-array and recursively call the function until we get a sub-array of size 1 (which means one of the medians of the 2n elements is inside). When I say "one of the medians", I mean that the professor said we can have two medians since the number of elements if even, therefore, there is not a true median. As for determining the sub-arrays, if the median of array A is larger than the median of array B, then either value could be the median. However, the median could also be less than the median of A or greater than the median of B. The reverse can be said for if the median of B is larger than the median of A.

Runtime: O(log n) because each time we call the function, we are splitting the arrays in half giving us log behavior

2. The chief engineer is in charge of deciding the mountainous road-network that will be kept open (cleared of debris and maintained regularly) during the winter months. Each road connects different small towns in the mountains, and all towns are connected to each other either directly or indirectly. Each road is associated with a value indicating the level of danger in maintaining that road during winter months. There are many subsets of roads such that the roads in such subsets keep the towns connected directly or indirectly. Each subset is assigned a cost of maintenance, which is directly proportional to the highest danger level of the road present in that subset. The engineer wants to select the smallest subset that keeps the towns connected directly or indirectly and that also has the lowest cost of maintenance. Develop an algorithm to find such a subset. Justify the correctness of your algorithm and derive its runtime.

```
findLowestCostOfMaintenance(graph G = (V, E, wt)) {
    // Empty set for the returned solution
    set S = {}
    for all v: d(v) = infinity, v.parent = undefined
    // Start with an arbitrary s, and initialize to 0
    d(s) = 0, s.parent = 0
    for all v: insert (v, d(v)) into a minheap H using d(v) as the key
               v.inH = true

    while(H is not empty) {
        u = extractMin(H)
        u.inH = false
        for(all (u, w) in E AND w.inH == true) {
            if(d(w) > wt(u, w)) {
                d(w) = wt(u, w)
                w.parent = u
                updateHeap(H, w, d(w))
            }
        }
        // Add the edge if the vertex is not the starting vertex
        if(u.parent != 0) {
            S.add((u.parent, u)
        }
    }

    return S
}
```

Justification: Using Prim's algorithm, we can find the smallest subset with the lowest cost of maintenance. In this implementation, the d-value of each vertex is the danger level of each road (the edge). Therefore, since Prim selects the lowest weight between

the two graphs, that adds the lowest level of danger to the subset. By doing this, we ignore the higher danger levels and achieve the lowest cost of maintenance while finding the smallest subset.

Runtime: O((V+E)log(V)) because we are essentially implementing Prim's algorithm here. The runtime is derived from looping through every vertex and edge, but each time we do that, we extract and update the heap which is log(V).

3. You are writing an algorithm for Cheapofle that helps its customers to find cheap flights between two cities (with possibly some stopovers). The information about flights and routes is given in the form of database relations: $(from, to, cost)$, where $from$ is the city from where the flight starts, to is the destination for that flight and $cost$ is the amount to pay for that flight. Write an algorithm to compute the minimum cost for customers who want to travel from a given source to a given destination with at most k intermediate stops. (if there is no solution for the customer's needs you can return "no plan available").

```
// V = 'from', E = ('from', 'to'), wt = 'cost'
findCheapestFlight(graph G = (V, E, wt), k, start s, end t) {
    // Initialize the cheapest cost to infinity
    // This is so the check works on the first time
    cheapestCost = infinity
    for all v: d(v) = infinity, v.parent = undefined, v.stop = -1
    // Start with given city (s), and initialize to 0
    d(s) = 0, s.parent = 0, s.stop = 0
    for all v: insert (v, d(v)) into a minheap H using d(v) as the key
               v.inH = true

    while(H is not empty) {
        u = extractMin(H)
        u.inH = false
        for(all (u, w) in E AND w.inH == true) {
            if(d(w) > d(u) + wt(u, w)) {
                d(w) = d(u) + wt(u, w)
                w.parent = u
                w.stop = u.stop + 1
                updateHeap(H, w, d(w))
                // Check if we found a cheapest flight path <= k stops
                // t = end destination as described above
                if(u.stop <= k AND w == t) {
                    // d(w) = cost of cheapest flight path
                    cheapestCost = d(w)
                }
            }
```

```
        }
    }

    if(cheapestCost < infinity) {
        return cheapestCost
    }
    else {
        return "no plan available"
    }
}
```

Justification: For this problem, we are simply implementing Dijkstra's algorithm with some additional checks for the problem specifications (is it cheapest? is it within k stops?). We want to use Dijkstra's because the problem is asking for the cheapest flight path between two cities, which is the scenario we want to use Dijkstra's. I have the check for the cheapest flight within the for loop that is checking all edges of the current vertex. This just checks to make sure that the number of intermediate stops ¡= 4 and that the cost to get there is cheaper than the current cheapest path.

Runtime: O((V+E)log(V)) because we are just implementing Dijkstra's algorithm here. The runtime is derived from looping through every vertex and edge, but each time we do that, we extract and update the heap which is log(V).

4. **Extra Credit** You are given n jobs numbered $1, 2, \ldots, n$ to complete and each job i comes with a difficulty d_i. Each job takes exactly one week to complete irrespective of its difficulty. If you complete job i during week $j(\leq n)$, then you earn a profit of $d_i(n-j)$. Your objective is to maximize the sum of the profit. Consider greedy strategy that completes that jobs in the decreasing order of difficulty. Prove/disprove that such a strategy produces an optimal solution.

   First, let's assume there exists an optimal order that does not satisfy the greedy strategy suggested in the problem. This means that there exists a job difficulty d_i that appears before d_j but $d_i < d_j$ (inversion pair). This means there exists an inversion pair that appear consecutively between d_i and d_j. Otherwise, this would result in $d_i > d_{i+1} > d_{i+2}...d_{j-1} > d_j$, which means $d_i > d_j$ and that is a contradiction to our assumption that d_i and d_j are inversion pairs.

   Now, since we know there is a consecutive inversion pair (let's call them d_i and d_{i+1}), we can use the exchange argument below.

   O (optimal solution) $= d_2(n-j) + d_3(n-j-1) = d_2n - d_2j + d_3n - d_3j - d_3$
   Exchange $= d_3(n-j) + d_2(n-j-1) = d_3n - d_3j + d_2n - d_2j - d_2$

   This is saying: $d_2 \leq d_3$, which shows how exchanging these results in another optimal solution since it is not less than the original optimal solution. Therefore, removing all the inversions results in an optimal solution where it is in ascending order thus proving the greedy strategy.