This homework is due on gradescope Friday May 18th at 11:59pm. Remember to justify your work even if the problem does not explicitly say so. Writing your solutions in \LaTeX is recommend though not required.

**Question 1** (Well Spaced Points, 25 points). *Given a sorted list $T$ of $n$ real numbers, you want to find a well spaced set of $k$ of them. In particular, you would like to find a subset $S \subseteq T$ with $|S| = k$ and with $\min_{x,y \in S, x \neq y}|x - y|$ as large as possible. Give an algorithm to find the best such $S$. For full credit, your algorithm should run in time $O(n^2)$ or better.* [Hint: First devise an algorithm to see if you can achieve a given minimal separation, and use this to hone in on the optimal.] [Note: The runtime bound given here is not optimal. It is possible to achieve $O(n \log(n))$ runtime.]

**Solution:**

**Algorithm:**

We first develop an helper algorithm that given $T$, $k$, and $d$, returns $S$, a subset of $T$ s.t. $|S| = k$ and $\min_{x,y \in S, x \neq y}|x - y| \geq d$ if there exists one, or $\emptyset$ if there does not. Our helper algorithm starts from the first element in $T$ then greedily picks the smallest element in $T$ that is at least $d$ away from the greatest element selected.

\begin{algorithm}
1. initiate: $S = \{T[0]\}, i = 1$
2. while $|S| < k$ and $i < |T|$ :
   (a) if $T[i] - S[-1] \geq d : S += T[i]$
   (b) $i += 1$
3. return $S$ if $|S| = k$ else $\emptyset$
\end{algorithm}

Then we write an overall algorithm that traverses all possible maximum $d$’s utilizing the helper algorithm and returns the maximum one.

\begin{algorithm}
1. if $k > |T|$ or $|T| = 0$ : return $\emptyset$
2. initiate: $l = 0, r = 1, d = 0$
3. while $r < |T|$ :
   (a) if $T[r] - T[l] \leq d$ or algorithm_11($T, k, T[r] - T[l]$) $\neq \emptyset$ :
      i. $r += 1$
      ii. $d = \max(d, T[r] - T[l])$
   (b) else : $l += 1$
4. return algorithm_11($T, k, d$)
\end{algorithm}
Runtime: For algorithm11, step 1 takes $O(1)$, step 2 takes $O(n)$, and step 3 takes at most $O(n)$, so the whole algorithm takes $O(n)$. In algorithm1, step 1, 2, 4 take $O(n)$ in combine, and step 3 contains at most $2n$ iterations with each having at most one call of algorithm11 so takes $O(n^2)$. Thus the total runtime is $O(n^2)$.

Proof of correctness:

Correctness of algorithm11:

We first prove the helper algorithm is correct. We want to show by exchange argument if we do not break step 2 when it finds $k$ elements, the greedy algorithm will be able to find the largest valid solution:

For any arbitrary subsequence $A$ sorted in increasing order with minimum separation of $d$, we want to prove $A$ can always be exchanged to another valid solution $A'$ with at least one more front element agreeing to the greedy.

Denoting $a_i$ as the first element in $A$ that is not in $G$, there are two cases:

- $i = 0$: replacing $a_i$ by $G_0 = T_0$ (which must exist given step 1 in algorithm1) will not violate the minimum separation because for any $a_j, j > 0$, $a_j - T_0 \geq a_j - a_i \geq d$.
- $i \neq 0$: Since all elements before in $A$ $a_i$ are in $G$ and $a_i$ exists, there must exist at least one element in $T$ at least $d$ away from the elements in $A$ before $a_i$ so $g_i$ must exist. Since $g_i \leq a_i$, $g_i$ maintains the minimum gap between elements after $a_i$. Therefore, replacing $a_i$ by $g_i$ will preserve the minimum gap.

Replacing $a_i$ by $g_i$ in $A$ will generate a equal-size valid solution $A'$ with at least one more element in the front agreeing with the greedy policy. Since for every $A$ different from the greedy we can always make more of its front elements to agree with the greedy policy, we will always be able to exchange $A$ to a subsequence of the greedy solution.

Correctness of algorithm1:

Then we want to prove the correctness of the main algorithm. We only need to show that the algorithm finds the correct $d$, which equals to $min\{d|\text{algorithm11}(T,k,d) \neq \emptyset\}$, after step 3. For any subset $S$, the minimum gap must equal to the distance between a pair of elements in $T$ so $D = \{d|\text{algorithm11}(T,k,d) \neq \emptyset\}$ can be seen as a union of all $D_i = \{T_j - T_i|j > i \land \text{algorithm11}(T,k,d) \neq \emptyset\}$, the set of gaps whose left boundary is $T_i$. Therefore, we only need to prove algorithm1 finds $max_{i=0}^{n-2}(max(D_i))$.

In algorithm1, we explore $D_i$’s in increasing order. Step 3(a) ensures we will explore every element in $D_i$ larger than $max_{j=0}^{i-1}(max(D_i))$, so after step 3, $d = max_{i=0}^{n-2}(max(D_i))$.

Alternative Algorithm:

Another way to approach this question is to use binary search in the top level algorithm. This algorithm first gets $D$, the set of all possible $|T_i - T_j|$. Then it will keep picking a random element $d$ from $D$ and run algorithm11($T, k, d$), if it returns a non-empty set, the algorithm will throw away the elements in $S$ that is smaller than $d$; otherwise the algorithm will throw away those larger than $d$. At the end, there will be only one element remained in $S$, and that is the largest minimum gap possible. Finally the algorithm will use the helper algorithm to get the well spaced subset.

Since this algorithm will keep throwing away the gaps either too large to be minimum gaps or too small to be the greatest minimum gaps, it will leave the correct greatest minimum gap to us. Similar to normal binary search, there would be $O(|D|) = O(n^2)$ comparisons involved. In addition, there would be $O(log(|D|)) = O(log(n))$ calls of the helper algorithm so the calls of the helper algorithm take $O(nlog(n))$ in total. Therefore, the alternative algorithm takes $O(n^2)$.
Question 2 (Roadtrip, 35 points). Thomas is once again travelling down a highway. This highway has rest stops $1, 2, \ldots, n$ with Thomas starting at $1$ and trying to reach $n$. Once again, he does not have enough fuel to make the trip in one go, however, this time, he brought an extra gas can with him and can store fuel picked up at various stops. Stop $i$ is located $x_i$ miles down the road with $x_1 = 0$ and $x_i < x_{i+1}$. When Thomas stops at rest stop $i$, he can obtain enough fuel to travel an additional $f_i$ miles. He starts with enough fuel to travel $f_1$ miles.

So, for example, if $f_1 = 7$ he might travel to stop $2$ with $x_2 = 5$, leaving himself with $7 - 5 = 2$ units of fuel left. He could then pick up $f_2$ (say in this case $f_2 = 3$) units of fuel, and would then have $2 + 3 = 5$ units and could travel to another stop as far as $5 + 5 = 10$ miles from the start.

Devise an efficient algorithm to compute a schedule for Thomas that gets him to his destination in the smallest possible number of stops. For full credit your runtime should be $O(n \log(n))$. [Hint: you will want to use a greedy algorithm, but you may not want to find the stops Thomas makes in the same order in which he makes them.]

Solution:

Algorithm: The algorithm starts from picking stop $1$ into the basket. Until the sum of the fuel available in the stops in the basket can support Thomas to reach to the end, it will greedily add the “richest” rest stop reachable with the current in-basket gas to the basket. Finally the stops in the basket would form the shortest valid path where Thomas can reach to the end.

\begin{algorithm} [Input: $x$, $f$, Output: $S$):
\begin{enumerate}
  \item if $n < 1$ : return $\emptyset$
  \item initiate: $S = \{1\}$; $pq = \{1\}$, priority queue sorted by $f_i$; range = $0$; $i = 2$
  \item while $i < n$ and not $pq$.empty() :
    \begin{enumerate}
      \item front = $pq$.deleteMin()
      \item $S +=$ front
      \item range += $f_{front}$
      \item while $x_i \leq$ range:
        \begin{enumerate}
          \item $pq$.insert($i$)
          \item $i += 1$
        \end{enumerate}
    \end{enumerate}
  \item return $S$
\end{enumerate}
\end{algorithm}

Runtime: Step 1, 2, 3, 5 takes at most $O(n)$ in combine. Step 4(e) executes at most $n$ iterations in the whole algorithm so takes $O(n)$ in total. Excluding step 4(e), step 4 has at most $n$ iterations, with each taking $O(\log(n))$, so takes $O(n \log(n))$ in total. Thus the whole algorithm take $O(n \log(n))$

Proof of correctness:

We will prove with exchange argument. We want to show that any arbitrary schedule $S$ different from the greedy $G$ can be exchanged to a solution $S'$ that has at least one more shared element with the greedy without increasing the total stops.

Suppose that a schedule $S$ uses all of the first $i$ stops used by $G$ but not the $i + 1^{st}$ stop of $G$. Then it must use some stop $j$, not one of the first $i$, so that $x_j \leq \sum_{a=1}^{i} f_{sa}$. We note that by the greedy procedure $x_j$ must have less fuel than the $i + 1^{st}$ stop and since either stop is clearly reachable, replacing $x_j$ by this other stop will give us another valid schedule $S'$ that shares one more common stops with $G$.

Therefore, any arbitrary valid schedule can be exchanged to another same-length schedule that contains all stops of $G$ so $G$ contains the stops in a shortest valid path.
**Question 3** (Other Minimum Spanning Tree Greedy Algorithms, 40 points). For each of the following proposed greedy algorithms for the Minimum Spanning Tree problem either prove that it always finds the optimal answer or find a counter-example.

(a) While $G$ is not a tree, find a cycle in $G$ and remove the heaviest edge on the cycle. [10 points]

(b) While $G$ is not a single vertex, find a cycle in $G$ (that is not a self-loop) and add the lightest edge in the cycle to your tree and associate its endpoints. [10 points]

(c) Start with $T$ containing no edges. While $T$ doesn’t connect $G$, find two connected components of $T$ and add the lightest edge of $G$ connecting them to $T$. [10 points]

(d) Start with $T$ containing no edges. While $T$ doesn’t connect $G$, find a partition of the vertices of $G$ into two parts so that no edge of $T$ connects the halves. Add the lightest edge of $G$ connecting the halves to $T$. [10 points]

**Solution:**

(a) The algorithm is correct:

Denote the greedy solution as $G = \{g_1, g_2, ..., g_{|V|-1}\}$. For any arbitrary different solution $A = \{a_1, a_2, ..., a_{|V|-1}\}$, we want to show that we can exchange $A$ to $G$ without increasing the total edge weight.

Denote $a_i$ as the last element in $A \setminus G$ thrown away by the algorithm. Then we know in the original graph there exists a cycle $C$ consisting of only $a_i$ and some edges in $G$ that are not heavier than $a_i$. Removing $a_i$ from $A$ we will partition the vertices into two sets, $T_1$ and $T_2$. There must be one edge $e \neq a_i$ in $C$ that has exact one end point in $T_1$, and $e$ must not be in $A$ (otherwise $A$ would contain cycle before $a_i$ was removed). Then replacing $a_i$ by $e \in G \setminus A$ in $A$ would give us a new solution $A'$ that has one more common edge with $G$ and at most the same total edge weight as that of $A$. Keep exchanging edges in this way, we will be able to transfer $A$ to $G$ without increasing the total edge weight.

Therefore, algorithm(a) will always generate the optimal.

(b) The algorithm is wrong:

As we can see from the graph above, if the we start picking edge from a cycle (the upper triangle) whose lightest edge is the heaviest one in other cycle, we will get a wrong answer.
(c) The algorithm is wrong:

As we can see from the same graph above, if we start from the two endpoints of a heaviest edge, then we will get a wrong answer.

(d) The algorithm is correct:

Denote the greedy solution as \( G = \{g_1, g_2, \ldots, g_{|V|-1}\} \). For any arbitrary different solution \( A = \{a_1, a_2, \ldots, a_{|V|-1}\} \), we want to show that we can exchange \( A \) to \( G \) without increasing the total edge weight.

Pick an edge \( e \in G \setminus A \). Then we know in the original graph there exists a partition of vertices, \( T_1 \) and \( T_2 \) where \( e \) is lightest edge connecting them. Adding \( e \) to \( A \) will create a cycle \( C \). There must be at least one edge \( a_i \neq e \) in \( C \) that has exact one end point in \( T_1 \), and \( a_i \) must not be in \( G \) (otherwise \( G \) would contain cycle). Then replacing \( a_i \) by \( e \) in \( A \) would give us a new solution \( A' \) that has one more common edge with \( G \) and at most the same total edge weight as that of \( A \). Keep exchanging edges in this way, we will be able to transfer \( A \) to \( G \) without increasing the total edge weight.

Therefore, algorithm(d) will always generate the optimal.