Homework 2: Greedy

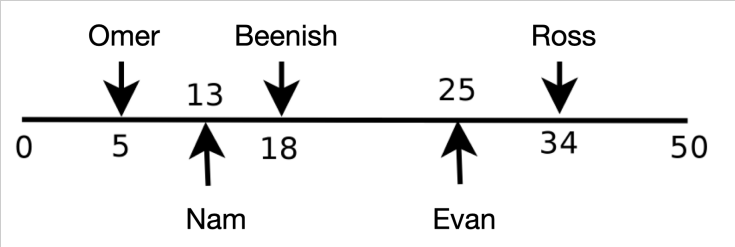
Due: September 25, 2025

This homework must be typed in LaTeX and submitted via Gradescope.

Please ensure that your solutions are complete, concise, and communicated clearly. Use full sentences and plan your presentation before your write. Except where indicated, consider every problem as asking for a proof.

Problem 1. The power lines along a country road have been modified to carry broadband Internet. Wi-Fi towers are being built along the road to provide the community with internet. To find the minimum number of towers required so that each house is sufficiently close to at least one tower, we model the problem as follows:

a) The entire course staff has taken up residence on Algorithm Street. The diagram below shows where they all live on the street.



Omer lives 5 miles down the road, Nam lives 13 miles down the road, and so on. Wi-Fi towers have an effective radius of 5 miles. Determine the minimum number of Wi-Fi towers needed such that each staff member has internet, and give the locations for these towers as well.

b) We're given a line segment ℓ, a set of non-negative numbers N that represents the locations of customers on ℓ, and a distance d. We wish to find a set of Wi-Fi towers of minimal size on ℓ such that each location in N is at most d away from some tower. Give an efficient greedy algorithm that returns a minimum size set of points. Prove its correctness and justify its runtime.

Now we generalize our model to account for houses that are not by the side of the road.

c) We're given a line segment ℓ, a set of pairs N representing the locations of customers, and a distance d. For each pair $(x, y) \in N$, let $x \in [0, \infty)$ be the distance along ℓ and $y \in [-d, d]$ be the distance above or below ℓ. We wish to find a set of Wi-Fi towers of minimal size on ℓ such that each location in N is at most distance d from some tower (here, we are using Euclidean distance).

Modify your algorithm from part (b) to solve this variation of the problem. You do not need to prove its correctness, but please explain how your proof from part (b) would (or would not) need to change based on your modifications.

Can we generalize further?

d) Does the correctness of your algorithm depend on the fact that ℓ is a line segment and not some curve? If so, give an example that illustrates the problem with your algorithm when ℓ is a curve. If not, explain how your algorithm could handle a curve. You shouldn't be writing another algorithm, or modifying your existing algorithm, just explain your reasoning.

Solution. a) We position 3 Wi-Fi towers within the ranges [8, 10], [20, 23], and [29, 39] such that

no two are placed within the same interval.

b) We will use a greedy algorithm to place the towers. The idea is to place each tower at the farthest possible location that covers the most uncovered customers.

Sort the customer locations in N in increasing order. Let $N = \{n_1, n_2, \ldots, n_k\}$ where $n_1 \leq n_2 \leq \cdots \leq n_k$. Begin with the first customer in the sorted list. Since this customer is uncovered, place a tower at the farthest point that can still cover this customer. Specifically, place the tower at $n_1 + d$ (since the tower covers up to a distance d). After placing a tower, skip all customers that are within d units of the current tower. Then repeat the process for the next uncovered customer. Terminate when all customers are covered (end of list reached).

Below is the pseudocode:

Algorithm 1 Minimum Number of Wifi Towers

```
Input: A set of nonnegative numbers N representing locations of customers and
       nonnegative distance d
Output: Minimum number of wifi towers guaranteeing full coverage
function FINDLOCATIONS(N, d)
    SORT(N)
    towers ← 0
    i ← 0
    n ← LENGTH(N)
    while i < n do
        towers += 1
        tower_location ← N[i] + d
        while i < n and N[i] ≤ tower_location +d do
            i+ = 1
    return towers
```

Proof of correctness: We argue correctness using the greedy exchange argument.

Let G be the greedy solution in sorted order, and let g_i be the position of the i-th tower placed by the greedy algorithm. Similarly, let O be any optimal solution in sorted order, and let o_i be the position of the i-th tower placed by the optimal solution.

Suppose $G \neq O$; that is, G and O differ in their tower placements. Let t be the smallest index such that $g_t \neq o_t$. Up to index $t-1$, the greedy solution and the optimal solution place towers at the same positions: $g_i = o_i$ for all $i < t$. Let $a \in N$ be the leftmost customer not covered by the first $t-1$ towers (i.e., by $g_1, \ldots, g_{t-1}$). Since G and O agree on the first $t-1$ towers, a is the same for both solutions.

The greedy algorithm places the t-th tower at $g_t = a + d$. The coverage interval of this tower is $[a, a + 2d]$.

The optimal solution places the t-th tower at $o_t \neq g_t$. Since o_t must cover customer a, it must satisfy $o_t \in [a - d, a + d]$.

Replace o_t with g_t in O to form a new solution O' without increasing the number of towers or decreasing coverage. All customers to the left of a are already covered by the first $t-1$ towers in both G and O. Since $g_t \geq o_t$, g_t covers all the customers to the right of a that o_t covers.

We have now decreased the number of differences between G and O by performing the exchange. By iterating this exchange we can turn O into G without impacting the quality of the solution.

Therefore, G must be optimal.

Since any optimal solution O can be transformed into the greedy solution G through a series of exchanges that do not increase the number of towers or reduce coverage, the greedy solution G must be optimal.

Runtime Justification: Sorting the list of customer locations takes $O(n \log n)$, where $n = |N|$. Placing the towers and moving through the list takes $O(n)$, since we go through the list of customers exactly once.

Thus, the overall time complexity of the algorithm is $O(n \log n)$.

c) **Algorithm description**: For each customer (x_i, y_i), calculate $h_i = \sqrt{d^2 - y_i^2}$. Then determine the interval $[l_i, r_i] = [x_i - h_i, x_i + h_i]$. This interval represents all possible positions along ℓ where a tower can be placed to cover customer i. Then sort the intervals $[l_i, r_i]$ in increasing order of their right endpoints r_i. Create an empty list S to store tower positions. While there are intervals not yet covered, select the interval with the earliest right endpoint. Let $[l_i, r_i]$ be the interval with the smallest right endpoint r_i. Place a tower at position $s = r_i$ and add s to S. Remove all intervals $[l_j, r_j]$ where $l_j \leq s$. These are the intervals that are covered by the tower at s. Finally, return the size of S.

Below is the pseudocode:

Algorithm 2 Generalized Minimum Number of Wifi Towers

Input: A set of pairs N representing locations of customers where $(x, y) \in N$ has $x \geq 0$ and $y \in [-d, d]$ and nonnegative distance d
Output: Minimum number of wifi towers guaranteeing full coverage

```
function findGeneralizedLocations(N, d)
    Initialize list Intervals
    for (x_i, y_i) ∈ N do
        h_i ← sqrt(d^2 − y_i^2)
        l_i ← x_i − h_i
        r_i ← x_i + h_i
        Add interval [l_i, r_i] to Intervals
    Sort Intervals in increasing order of right endpoints r_i
    Initialize empty list S
    while Intervals is not empty do
        Let [l_i, r_i] be the first interval in Intervals
        Place a tower at position s = r_i
        Add s to S
        Remove all intervals [l_j, r_j] from Intervals where l_j ≤ s
    return |S|
```

Modification of the Proof from Part (b): The proof from part (b) needs to be adjusted for part (c) because customers are no longer points on the line but have coverage intervals along the line segment due to their positions above or below it. In part (c), each customer defines an interval of feasible tower positions where they can be covered, turning the problem into an interval covering problem. Therefore, the exchange argument must account for these intervals rather than fixed points. The modified proof demonstrates that placing towers at the right endpoints of the earliest finishing intervals (as per the greedy algorithm) remains optimal.

The exchange moves involve replacing towers in any optimal solution with those chosen by the greedy algorithm, ensuring coverage of the same or more intervals without increasing the number of towers. Thus, while the core structure of the proof—the exchange argument—remains the same, it adapts to consider intervals instead of individual points to establish the optimality of the modified algorithm.

d) The correctness of the algorithm depends on the fact that ℓ is a line segment. Consider the unit circle. If a customer is located at $(-1, 0)$, the algorithm from part c) would attempt to place a Wi-Fi tower as far to the right of the customer as possible while staying within the allowed distance d. However, since the circle has two locations (one on the upper semicircle and one on the lower semicircle) that are equidistant from the customer, the algorithm could place the tower in either direction, potentially missing other customers or having to place additional towers. This ambiguity does not arise with a line segment because each location has only one direction that extends to the rightmost distance without looping back. Thus, the correctness of the algorithm in c) relies on ℓ being a line, and the approach may not generalize correctly for non-linear curves.

□

Problem 2. Given an array A of n distinct integers sorted in non-decreasing order, design an $O(\log n)$ algorithm to decide (i.e. output true/false) whether there exists an index i such that $A[i] = i$.

1. Provide a succinct (but clear) description of your algorithm. You may provide pseudocode.
2. Prove the correctness (optimality) of your algorithm.
3. Analyze the running time and memory utilization of your algorithm.

Solution.

1. **Algorithm description.** The array A is sorted and contains distinct integers. Define $f(i) = A[i] - i$. Observe that:
   - If $f(i) = 0$, then $A[i] = i$ and we are done.
   - If $f(i) > 0$, then for all $j > i$, we have $f(j) > 0$ (since A is strictly increasing).
   - If $f(i) < 0$, then for all $j < i$, we have $f(j) < 0$.

   Thus, $f(i)$ is strictly increasing, which means we can perform binary search to locate an index i such that $f(i) = 0$.

Algorithm 3 Check Fixed Point in Sorted Array

```
Input: Array A of n distinct sorted integers
Output: true if ∃i such that A[i] = i, else false
function FixedPoint(A)
    low ← 0, high ← n − 1
    while low ≤ high do
        mid ← ⌊(low + high)/2⌋
        if A[mid] = mid then
            return true
        else if A[mid] > mid then
            high ← mid −1
        else
            low ← mid +1
    return false
```

2. **Proof of correctness.** Since $f(i) = A[i] - i$ is strictly increasing, it crosses zero at most once.
   - If $f(\text{mid}) = 0$, then $A[\text{mid}] = \text{mid}$ and we return **true**.
   - If $f(\text{mid}) > 0$, then $f(j) > 0$ for all $j >$ mid. Therefore, if a solution exists, it must be to the left of mid, so we set high = mid -1.
   - If $f(\text{mid}) < 0$, then $f(j) < 0$ for all $j <$ mid. Therefore, if a solution exists, it must be to the right of mid, so we set low = mid $+1$.

By induction on the shrinking search interval, the binary search guarantees that if there exists an index i with $A[i] = i$, the algorithm finds it. If no such index exists, the search interval shrinks to empty and the algorithm correctly returns **false**. Thus, the algorithm is correct.

3. **Running time and memory analysis.** At each step, binary search discards half of the current interval. Therefore, the running time is:

$$O(\log n).$$

The memory utilization is constant: we only store a few integer variables (low, high, mid). Therefore, the space complexity is:

$$O(1).$$

□

Problem 3. Suppose you are given a set $S = \{a_1, a_2, ..., a_n\}$ of tasks, where task a_i requires p_i units of processing time to complete, once it has started. You have access to a computer to run these tasks one at a time. Let c_i be the **completion time** of task a_i, i.e. the time at which task a_i completes processing. Your goal is to minimize the average completion time:

$$\frac{1}{n}\sum_{i=1}^{n} c_i$$

For example, suppose there are two tasks, a_1 and a_2, with $p_1 = 3$ and $p_2 = 5$, and consider the schedule in which a_2 runs first, followed by a_1. Then, $c_2 = 5,\ c_1 = 8$, and the average completion time is 6.5.

(a) Give an algorithm that schedules the tasks to minimize the average completion time. Each task must run non-preemptively, that is, once task a_i is started, it must run continuously for p_i units of time. Prove that your algorithm minimizes the average completion time, and prove the running time of your algorithm.

(b) Suppose now that the tasks are not available at once. Each task has a **release time** r_i before which it is not available to be processed. Suppose also that we allow **preemption**, meaning a task can be suspended and restarted later.

For example, a task a_i with processing time $p_i = 6$ may start running at time 1 and be preempted at time 4. It can then resume at time 10 but be preempted at time 11 and finally resume at time 13 and complete at time 15. Task a_i has run for a total of 6 time units, but its running time has been divided into three pieces. We say that the completion time of a_i is 15.

Give an algorithm that schedules the tasks so as to minimize the average completion time in this new scenario. Prove that your algorithm minimizes the average completion time, and state the running time of your algorithm.

Solution.

(a) Consider the following algorithm:

Algorithm 4 Non-Preemptive Task Scheduling

```
Input: A set of tasks S with their processing times
Output: A schedule (list) of tasks that minimize the average completion time
1: function NonPreemptiveTaskScheduling(S)
2:     return S sorted with respect to processing time in increasing order
```

To show this solution is optimal, we proceed by way of contradiction. Suppose, there exists a schedule $S = [a_1, \cdots, a_n]$ that has a shorter average completion time than our above greedy solution. Since this schedule is not the greedy solution we know that there exists two tasks a_i and a_j such that $i < j$ but $p_i > p_j$.

Because task a_i takes longer than a_j, we assert that the solution can be improved by first scheduling a_j and then later scheduling a_i. One can verify this by comparing the

total completion times of the supposed optimal schedule and the schedule obtained by swapping p_i and p_j.

The original schedule has a total completion time given by:

$$p_1 + (p_1 + p_2) + \cdots + (p_1 + \cdots + p_n) = \sum_{k=1}^{n}(n-k-1)p_k$$

By swapping the positions of tasks a_i and a_j, the new total completion time becomes:

$$\underbrace{(j-i)p_j - (j-i) - p_i}_{\text{replace } (j-i)\ p_i\text{'s with } p_j\text{'s}} + \sum_{k=1}^{n}(n-k-1)p_k$$

Since we assumed that $i < j$ and $p_i > p_j$, it follows that $(j-i)p_j < (j-i)p_i$. Therefore, the swapped schedule has a shorter total completion time and therefore, a shorter average completion time. This contradicts the assumption that the original schedule was optimal.

Since this algorithm simply returns a sorted list, the runtime of this algorithm is $O(n \log n)$.

(b) Consider the following algorithm:

Algorithm 5 Preemptive Task Scheduling

```
Input: A set of tasks S with their processing and release times
Output: A schedule (list) of tasks that minimize the average completion time
function PreemptiveTaskScheduling(S)
    Sort the tasks in the order of increasing release time
    currentTime ← the first (minimum) release time
    Make a priority queue, availableTasks, with all tasks whose release time
    is less than the currentTime, where the queue is ordered in ascending
    processing time
    while currentTime ≤ the final release time do
        Schedule the next available task (top of priority queue) in
        availableTasks modifying its processing time to be the difference
        between the current time and the next release time.  If the task is
        finished remove it from the queue entirely.
        currentTime ← the next task release time
        Add the newly released tasks to availableTasks
    return a list containing the scheduled tasks
```

The same concept of always doing the shortest task applies from part (a). The difference is that we have to re-determine which task is shortest whenever new tasks become available. The while loop is structured such that whenever new tasks become available, they are inserted into a priority queue. This ensures that at all times, the shortest available task is scheduled.

First, the list of n tasks is sorted by release time. This takes $O(n \log n)$ time. Then, the while loop performs the tasks. Each task gets added to the queue at most once, and removed from the queue at most once. Since adding and removing from a priority queue can be performed in $O(\log n)$, adding and removing all elements will take $O(n \log n)$ time. Therefore, the overall runtime is $O(n \log n)$.

□