1.5em 0pt

CS218, Spring 2025 **Due: 11:59pm, Friday 04/25, 2025**
Assignment #2 [1]
Name: Lakhan Kumar Sunilkumar **Student ID: 862481700**

1 Deadlines, again! (3.2pts)

1. (0.2pts)

   One known and feasible greedy approach for scheduling a particular assignments is to always choosing the task with the earliest deadline first. Even if it looks like a possible solution at first, it does not always says that it is the most feasible solution

   To show this, we can try a counter example which involves three assignments:

Assignment	**Points** (p_i)	**Deadline** (d_i)
A1	15	2
A2	35	4
A3	50	9

   Using the "earliest deadline first" strategy, we begin the process by selecting A1 on day 1 since it has the closest deadline. On day 2, we may either choose between A2 and A3. For example if we pick A2. That means A3, which is the most valuable assignment, is left out since there are no remaining days to schedule it. This gives us a total of $15 + 35 = 50$ points.

   However, if we try to use an effective greedy approach which prioritizes the tasks by considering their point value and assigning each to the latest available day on or before its deadline we can come up with a better solution. In this case:

   - A3 (50 points) is assigned to day 2.
   - A2 (35 points) is assigned to day 1.
   - A1 (15 points) is skipped due to limited days.

   This results in a total of $50 + 35 = 85$ points, which is clearly much better.

   This example shows that by simply prioritizing based on earliest deadlines, it can lead to poor a choices, especially when a lower-value tasks is blocking the opportunity to complete a better ones. A more strategic greedy approach based on point maximization yields significantly better outcomes.

2. (0.2pts)

   Another natural greedy strategy is to always work on the assignment with the most highest points value first. While this method appears feasible since it can be seen to maximize the rewards early it doesn't always give us the optimal outcome, especially when deadlines can have a restriction in scheduling.

   Consider the following three assignments:

Assignment	**Points** (p_i)	**Deadline** (d_i)
A_1	80	1
A_2	40	2
A_3	30	2

   Using the "highest point first" approach, we should start with A_1 on day 1 and then choose A_2 on day 2. A_3 is then left out. The total score in this case is $80 + 40 = 120$. However, this ignores the tight deadline of A_1. A good solution would be to schedule A_2 on day 2 and A_3 on day 1 so ensuring both are completed before their deadline of day 2, which results in $40 + 30 = 70$ points.

[1]Some problems are adapted from existing problems from online sources. Thanks to the original authors.

While the total is least in this specific case, there could be a few scenarios (especially when deadlines are tighter) where choosing the highest-scoring task first may avoid completing more overall tasks, leading to a lower final score. This shows that a more niche strategy balancing point value with available time before deadlines is a must for the best outcome.

3. (0.5pts)

   To show that any good solution must have the assignment with the highest number of points, let's assume the opposite, that the solution does not include it. Let us consider $A_{\max}$ be the assignment with the highest score called $p_{\max}$, and by chance, we have chosen a different set of assignments with lower total points.

   Since each assignment takes exactly one day and can be completed on or before its deadline, if there is any available day which is $\leq d_{\max}$, we could swithc in $A_{\max}$ in place of a lower-value assignment. This would increase the total score, disputing any kind of assumption which says that the solution was already optimal.

   Therefore, if it's possible to schedule $A_{\max}$ within its deadline, any optimal solution must have it in it. Leaving the most valuable assignment would always result in a lower total score.

4. (0.5pts)

   To solve this problem optimally, I will be using a greedy strategy which is prioritizes assignments with the highest point value while also considering the other deadlines. Since we know that each assignment takes exactly one day, and can only work on one assignment per day, we need to schedule them carefully so that we don't waste early days on low-value tasks that block more valuable ones.

Algorithm 1 Greedy Assignment Scheduler

```
Function GreedyScheduler(assignments):
    Sort the list in decreasing order, for assignments by $p_i$
    Start days[1..n] as false foreach assignment $(p, d)$ in sorted list do
        for day from $\min(d, n)$ up to to 1 do
            if days[day] equal false then
                Mark days[day] as true Add $(\text{day}, p)$ to schedule break
            end
        end
    end
    return schedule
```

So the idea is to first sort all assignments in descreasing order of their point values. For each assignment, we try to schedule it on the latest available day that does not exceed its deadline. This ensures that high-value assignments are given the best chance of being included, while leaving earlier days open for other tasks.

This algorithm is taken from a job scheduling problem with deadlines and profits. It works because it maximizes the use of available days without allowing less valuable tasks to take up valuable time slots. Since all point values are distinct, the ordering is unambiguous.

5. (0.3pts)

   To show the working of greedy algorithm, we shall be sorting the assignments in descending order of points and let's schedule each one on the latest available day that does not come after it's deadline. This shows that all high-value tasks are given high priority while making it flexible for the earlier days.

   Consider this example where given the input assignments and their deadlines, and we use the algorithm to produce the following schedule:

Day	**Assignment**	**Deadline**	**Points**
1	C	2	7
2	B	2	8
3	F	3	12
4	G	4	4
5	D	5	15
11	E	7	1
12	H	5	5
13	A	13	10

By following this strategy, we finally completed all 8 assignments without missing a single deadlines. The total number of points earned is $15 + 12 + 10 + 8 + 7 + 4 + 5 + 1 = 62$. This shows how effective the greedy approach is in maximizing the total score.

6. (0.9pts)

   The greedy choice that I have considered for the algorithm is to always make sure to select the assignment with the highest point value that is available and should schedule it on the latest possible day that is still found within its deadline. Now let's reorder the assignments in descending order of their point values, then we focus on the most valuable tasks first and schedule it without not consdering other filling earlier days that might be more useful for tighter-deadline tasks.

   Just to tell that this greedy choice is "safe," we try ti see that if we skip the maximum valued assignment and instead consider the lower value ones, we are actually blocking ourselves out of earning a higher score. Since each assignment takes exactly one day and deadlines allow us to work on the last possible day up to d_i, assigning a high-value task to the latest available slot helps preserve flexibility for future assignments while ensuring we lock in its high point value.

   We could try this for the argument that there exists an optimal solution that does not include the assignment with the highest remaining point value, say A_{max}. If A_{max}'s deadline can allow it to be scheduled at any earlier day which is vacant, we can remove a lower-value assignment from that solution and insert $A_{\max}$ in its place instead. This will definetly increases the total score, which contradicts the assumption that the solution was already optimal.

   Therefore, always choosing the highest point assignment that fits within the available deadline slots is a "safe" choice, and no better solution can be thought of by skipping it. The algorithm ensures that we never miss a high-value opportunity when it's possible to schedule it, which is what makes this greedy approach both effective and optimal for the problem at hand.

   So, the greedy decision is used to process high-value assignments first, and then assign them as late as feasible, balances value and scheduling flexibility, and is provably safe in contributing to an optimal solution.

7. (0.6pts)

   So an optimal substructure is a fundamental property in algorithm design, particularly in greedy and dynamic programming approaches. A problem is said to exhibit optimal substructure if an optimal solution to the entire problem can be generated by combining all optimal solutions to its subproblems. In other words, solving smaller parts optimally helps to build an overall optimal solution.

   So when we consider for this assignment scheduling problem, optimal substructure means that if we have an optimal schedule for the first k assignments, we can make this schedule longer by adding the $(k + 1)^{th}$ assignment by assuming that we are allowed to follow the same greedy rule and can still maintain optimality. Each decision we make which is choosing the best task that fits, leads to a smaller version of the same problem which was this earlier, choosing the next best assignment for the remaining days and assignments.

We can assume that we have an optimal schedule for n assignments. If we remove one assignment (say, the one with the latest scheduled day), the remaining $n-1$ assignments should still form an optimal schedule for that reduced set. Otherwise, we could replace the sub-solution with a better one, leading to a better overall schedule, which contradicts the assumption of optimality.

This plan is good because our greedy choice is always selecting the highest point assignment that fits in the latest available slot and it also does not affect the feasibility of scheduling other assignments. We can see that each assignment is placed independently based on availability and deadline constraints, without blocking higher-value options that could have been scheduled earlier.

References

- Prof. Yan Gu Notes on Greedy Algorithms
- `https://www.geeksforgeeks.org/scheduling-in-greedy-algorithms/`

2 Don't be late! (1.8pts)

Solution.

1. (0.7pts)

   To analyze Yan's situation, lets define a function $f(s)$, where s is considered as the time the elevator would take to arrive when Yan starts waiting. The parameter W shows the maximum amount of time Yan is ready to wait before losing hope and start tkaing the stairs. If the elevator arrives within this gap, he will use it. Otherwise, he will start using the stairs after waiting W seconds.

   The function $f(s)$ is defined as follows:

$$f(s) = \begin{cases} E, & \text{if } s \leq W \\ W + S, & \text{if } s > W \end{cases}$$

   This function mentioned above reflects the two possible cases of Yan's behavior. So, in the first case, where $s \leq W$, the elevator arrives before he gives up, so the total time would actually be the timme taken for the elevator to arrive, E. In the second case, where the elevator seems to be taking longer than W, Yan will wait W seconds and then he will take the stairs, which actually takes an additional S seconds, giving a total of $W + S$.

   This function clearly shows the decision-making process and outcomes of Yan's planning. It will also evaluate the competitive ratio in later parts of the problem, helping us determine the most efficient value of W to minimize the worst-case performance compared to the offline optimal strategy.

2. (0.5pts)

   Consider the function $g(s)$ is defined as:

$$g(s) = \min(s + E, S)$$

   The function $g(s)$ demonstrates the **optimal time** Yan would spend to reach the first floor, assuming he knows and has the full knowledge of when the elevator will arrive. This is a best-case, hindsight-aware strategy. As we know that the elevator takes s seconds to arrive, Yan can simply compare whether taking the elevator or using the stairs would get him downstairs faster.

   Here, $s + E$ is the total time if Yan decides to wait for the elevator and then uses it to go down, while S is the fixed time required to take the stairs. Since he knows the s in advance, he will be choosing whichever option seems to be faster. This function serves as the base level for calculating the competitive ratio that Yan actually uses.

3. (0.6pts)

   To minimize the worst-case competitive ratio, let us consider a scenario where the elevator does not arrive on time for within the waiting time W. In this scene, Yan will be taking around $W+S$ seconds. But for the optimal offline strategy, as knowing that the elevator will never arrive, it would be good for Tan to take the stairs immediately, which takes S seconds. Thus, the competitive ratio in the worst case is:

$$\frac{W+S}{S}$$

   For us to find the best W, we try to balance this worst-case with the best case when the elevator comes on just in time. We try to achieve the competitive ratio in both the cases to be as little as possible. The optimal value is found by keeping the worst-case competitive ratio as to the ratio when the elevator arrives just at W, giving us:

$$\frac{W+S}{S} = \frac{E}{W+E}$$

   Solving this equation for W mentioned above will give the best possible waiting time. It would be good for Yan's to follow this strategy, that is to wait exactly this optimal W, take the elevator if it arrives by then, or otherwise take the stairs. This shows that no matter when the elevator comes (or doesn't), Yan's method is always pretty close to the best possible solution, even if he knew everything in advance. It never does way worse, and there's a guaranteed limit on how far off it can be.

3 Homework 2 Basic

1. **A. Seating Arrangements: Codeforce Submission ID: 316016144**

   Explanation:

   So A person is only happy when they sit next to a family member in a pair or sit alone in a row. Now We go through all possible numbers of family pairs (x) that can be seated in r rows. For each x, we check the leftover people and the remaining rows. Then, check how many singles can be seated alone rather than sitting with a different family member which indeed breaks the happiness condition. The final result is the maximum total number of happy people that can be covered among all possibilities and combinations.

   Code Structure

```
Function MinimizeMovesToSort(n, arr):
    Initialize a map to track the length of consecutive sequences
    set longest_sequence_length = 0

    For each value v in arr:
        If (v - 1) exists in map:
            seq_length = length of sequence ending at (v - 1) + 1
        Else:
            seq_length = 1

        Update the map for v with seq_length
        Update longest_sequence_length to maximum of itself and seq_length

    minimum_moves_required = n - longest_sequence_length

    Print minimum_moves_required
```

Time Complexity:

For each test case, we loop up to $\mathcal{O}(r)$ (maximum number of rows), where $r \leq 500$. Preprocessing steps like summing array elements take $\mathcal{O}(n)$, where $n \leq 100$. Thus, the complexity per test case is $\mathcal{O}(r)$. Overall, the time complexity for t test cases is $\mathcal{O}(t \times r)$.

Reference:

`https://www.geeksforgeeks.org/puzzle-sitting-arrangement/`

2. **B. Artistic Swimming: Codeforce Submission ID: 316013960**

Explanation:

The plan is to form the max number of swimming trios groups consisting of 1 male and 2 females, and the condition is that the male is always taller than both selected females. First, we should sort both male and female height arrays. Then, using a greedy two-pointer approach, we iterate over each male and try to find the smallest pair of vacant females who are both shorter than the male that is selected. If found, we make and increment the count of the trio and move the female pointer forward by 2 as 2 females are selected. This shows that pairing is done for shortest compatible females first.

Code Structure

```
Sort the maleHeights array in increasing order.
Sort the femaleHeights array in increasing order.

While malePointer < size of maleHeights AND femalePointer + 1 < size of femaleHeights:
    If maleHeights[malePointer] > femaleHeights[femalePointer]
       AND maleHeights[malePointer] > femaleHeights[femalePointer + 1]:
        i. Form a trio with the current male and the two females.
        ii. Increment trioCount by 1.
        iii. Move femalePointer forward by 2 (both females used).
    Else:
        i. Move femalePointer forward by 1 (current female not usable, try next).
    Move malePointer forward by 1 (use the next male).
Return trioCount.
```

Time Complexity:

- Sorting females array: $O(n \log n)$
- Sorting males array: $O(m \log m)$
- Greedy two-pointer scan: $O(n + m)$

Overall Time Complexity: $O(n \log n + m \log m)$

References

- `https://www.geeksforgeeks.org/when-should-i-use-two-pointer-approach/`
- `https://www.geeksforgeeks.org/introduction-to-greedy-algorithm-data-structures-and-algorit`

3. **C. Number Candles: Codeforce Submission ID: 316013231**

   Explanation:

   By using Greedy Algorithm:

   The plan is to have a maximum number of candle digits by buying as much as the cheapest candles as possible. For each digit position (from left to right), replace it with the largest digit that still allows enough money to fill the remaining positions with the cheapest digit.

   Code Structure

```
Find the minimum price among all digits: min_price = min(prices).
Calculate the maximum possible number of digits: length = monies // min_price.
For each position from 0 to length - 1:
    For digit d from 9 down to 1:
        i. Get cost_d = prices[d - 1].
        ii. Calculate remaining positions: rem_pos = length - pos - 1.
        iii. If rem_money - cost_d >= rem_pos * min_price:
             a. Append digit d to result.
             b. Deduct cost_d from rem_money.
             c. Break the inner loop (move to next position).
After all positions are filled, print it.
```

   Time Complexity:

   O(n), where n: maximum number of digits ($n = \lfloor \frac{M}{\text{min\_price}} \rfloor$).

   Since the outer loop (for each digit) and the nested loop (constant 9 tries) are independent, the total complexity remains **O(n)**.

   References:

   Prof Yan Gu's slides: 05-Greedy i.e Homework Deadlines and Buying Gifts example

4 Homework 2 Challenge

1. **A. Move to the ends: Codeforce Submission ID: 317243185**

   Explanation:

   The goal is to minimize moves by preserving the longest consecutive subsequence in left-to-right order. We can move any element to the beginning or end, but elements in this sequence are should not be touched.

   So we can perfom these steps:

   - Iterate through the array once.
   - For each value v, update seq_v[v] = seq_v.get($v-1, 0$) + 1.
   - Track the maximum sequence length ('longest') during iteration.

   Finally, the minimum number of moves required is ($n-$longest), where n is the total number of elements in the array.

Code Structure

```
For each value v in array:
    seq_v[v] = seq_v.get(v - 1, 0) + 1
    longest = max(longest, seq_v[v])
Append (n - longest) to results list
```

Time Complexity:

Single pass over array elements: $O(n)$ Hash map updates and lookups per element: $O(1)$ amortized

Overall Time Complexity: $O(n)$ per test case.

References

- `https://www.geeksforgeeks.org/longest-increasing-consecutive-subsequence/`
- `https://takeuforward.org/data-structure/longest-consecutive-sequence-in-an-array/`

2. **B. Secret Message: Codeforce Submission ID: 317245888**

Explanation:

The plan of attack is to find the lexicographically smallest path from the top-left (1,1) to the bottom-right (N,M) in a given grid, moving only to adjacent cells (up, down, left, right) (DFS path) without revisiting any cell.

So for these steps:

- Start DFS traversal from the (0,0) cell with the initial prefix string.
- Then at each step, check and collect all unvisited neighbors and sort them lexicographically based on the letter it shows.
- Also prune paths early if the current prefix cannot lead us to a better solution compared to the best.
- Upon reaching (n-1, m-1), compare the current path string and update the best result if necessary.

This ensures that all paths are explored in such a way that it always looks ahead for forming the smallest possible string at each decision point.

Code Structure

```
function dfs(x, y, prefix):
    if prefix cannot beat best:
        return
    if (x, y) is bottom-right:
        update best if prefix is smaller
        return
    collect all valid, unvisited neighbors
    sort neighbors lexicographically
    for each neighbor:
        mark neighbor as visited
        call dfs(neighbor.x, neighbor.y, prefix + neighbor.letter)
        unmark neighbor as visited

function smallest_message(grid):
    initialize best as a very large string
    mark (0,0) as visited
    call dfs(0, 0, grid[0][0])
```

Time Complexity:

Exploring paths using DFS (worst case): $O(4^{N\times M})$. Sorting neighbors at each step: $O(1)$ (since at most 4 neighbors) **Overall Time Complexity:** $O(4^{N\times M})$

References

- `https://stackoverflow.com/questions/29376069/lexographically-smallest-path-in-a-nm-grid`
- `https://www.geeksforgeeks.org/print-the-lexicographically-smallest-dfs-of-the-graph-starting-f`

Side Note:

I Initially tried solving with a heap (priority queue) approach where it always expands the path with the smallest string first. However, it failed on some test cases because heap makes greedy global decisions without considering better local paths.

3. **C. Marathon: Codeforce Submission ID: 317253313**

Explanation:

The plan of attack is to ensure that every point on the 42,195m marathon track is covered by at least two cameras using the minimum number of placements. Each camera has a pre-determined location and coverage radius.

So first, we place each camera's coverage interval so that it stays within the valid range $[0, 42195]$ meters and ignore any cameras whose intervals do not overlap the track.

Next, we sort all the valid intervals based on their left endpoints to have an efficient left-to-right sweep of the track.

While sweeping from the start of the track, we set and handle two heaps:

- An **active** min-heap that stores the right ends of cameras that have already been chosen.
- A **candidate** max-heap that stores the right ends of cameras that are currently visible but have not yet been selected.

At every critical point particularly at the right end of a camera we perform two key operations:

- Remove any cameras from the active heap whose coverage has already expired (i.e., their right end is before the current position).
- If the number of active cameras covering the position falls below two, we greedily select the next available camera that covers the position and extends coverage as far as possible.

Finally, we move the position only to the next critical event where coverage might drop, that is, when a camera's coverage expires.

This ensures that all points along the track are continuously covered by at least two cameras using the minimal number of placements through a greedy, event-driven strategy.

Code Structure

```
function solve():
    Clamp all cameras to within [0, 42195] meters
    Sort all camera intervals by their starting point

    while pos is not beyond 42195:
        Add all cameras whose start <= pos into 'candidates'
        Remove any cameras from 'active' whose coverage ends before pos
        while less than two cameras cover pos:
            If no candidates can cover pos:
                Return -1 (coverage impossible)
            Else:
                Select the camera with the farthest right end from 'candidates'
                Add its right end to 'active'
                Increment the camera placement count

        Move pos to the smallest right end among active cameras
        Remove all cameras from 'active' that expire exactly at pos

    Return the total number of cameras placed
```

Time Complexity:

- Sorting intervals: $O(n \log n)$
- Heap operations: Each camera is pushed and popped at most once, $O(n \log n)$

Overall Time Complexity: $O(n \log n)$

References

- `https://cs.stackexchange.com/questions/151112/greedy-algorithm-for-postive-interval-covering`
- `https://www.geeksforgeeks.org/minimum-number-of-intervals-to-cover-the-target-interval/`
- `https://dilipkumar.medium.com/sweep-line-algorithm-e1db4796d638`
- `https://www.youtube.com/watch?v=YnIxejYW7cE`

4. **D. Defending Hogwarts Express: Codeforce Submission ID: 317256952**

Explanation:

The goal of the problem is to determine how many train cars are destroyed after each Death Eater attack. Each train car has:

- A fixed position on a number line.
- A defense value.

Each attack has a center position y, radius b, covering the range $[y - b, y + b]$ and an attack power A.

A train car is destroyed by an attack if:

- Its position lies within the attack range.
- Its defense value is less than or equal to the attack power.

To handle upto 10^5 cars and 10^5 attacks efficiently we should try:

(a) Sort all train cars by their positions.
(b) Build a Segment Tree on defense values to quickly find the minimum defense in any range.
(c) For each attack:
    - Use Binary Search to find indices of cars within the attack range.
    - Repeatedly find and remove the weakest car (with defense $\leq A$) using the segment tree.
    - Count how many cars were destroyed.

Once a train car is destroyed, it is permanently removed and cannot be destroyed again in future attacks.

Code Structure

```
function solve():
    Sort train cars by position

    Build a Segment Tree where each node stores:
        Minimum defense value,
        Index of that defense

    for each attack:
        Calculate the attack range [y - b, y + b]
        Use Binary Search to find the leftmost and rightmost cars in range

        while there is a train car in the range with defense <= attack power:
            Destroy the car (update its defense in Segment Tree to INF)
            Increment the destroyed count for this attack
```

Time Complexity:

- Sorting train cars by position: $O(n \log n)$
- Building Segment Tree: $O(n)$
- Each attack:
  - Binary Search to find range: $O(\log n)$
  - Destroying each affected train car: $O(\log n)$ per car

Overall Time Complexity: $O((n + m) \log n)$

References

- `https://www.geeksforgeeks.org/segment-tree-range-minimum-query/`
- `https://cp-algorithms.com/data_structures/segment_tree.html`
- `https://www.youtube.com/watch?v=ciHThtTVNto`