Homework 6

aryanrao

1. **Pseudocode:**

```
GreedyColoring(m1, m2, ..., mn):
   Sort the numbers m1, m2, ..., mn in increasing order

   Initialize an empty list usedPens to keep track of the green pens used

   while there are still uncolored points do:
      Let m be the leftmost uncolored point that has not been covered by
any previous pen
      Use a magical green pen at the coordinate m
      Add this pen to usedPens
      Mark all the points in the range [m - 5, m + 5] as colored (if not
already colored)

   return usedPens
```

Algorithm:

1. Initialize an empty list to keep track of the green pens used.
2. While there are still uncolored points, do the following: a. Select the leftmost uncolored point (m) that has not been covered by any previous pen. b. Use a magical green pen at the coordinate m. c. Add this pen to the list of used pens. d. Mark all the points in the range [m - 5, m + 5] as colored (if not already colored).

Proving the correctness of the algorithm:

Let's use the exchange argument to prove the correctness of this algorithm. The exchange argument is a common technique used to prove the optimality of a greedy algorithm.

Assume that there exists an optimal solution (opt) that uses fewer pens than our greedy solution.

Let opt_pens be the set of magical pens used in the optimal solution opt. Now, consider the first point m' in our greedy solution that differs between our solution and opt. This point must exist since opt is different from our greedy solution.

Let pen_greedy be the magical green pen we used to color m' in our greedy solution.

Since m' is the leftmost uncolored point, and opt is an optimal solution, there must exist a pen_opt in opt_pens that colors m'.
Now, consider the earliest point m" (can be the same as m') that pen_greedy covers in our solution. The pen_opt in the opt_pens must also color m" since it colors all points in [m" - 5, m" + 5].

Now, consider the range [m' - 5, m' + 5]. Since pen_greedy colors all points in this range, it also colors m". However, since opt is an optimal solution, there must be a pen_opt' in opt_pens that colors all points in this range.

Since both pen_opt and pen_opt' color the same range and pen_opt' colors m", we can swap pen_opt' and pen_greedy without affecting the coverage of points. After this exchange, our greedy solution still colors all points, and we use one fewer pen (pen_opt') than before.

By repeating this process and making exchanges, we can convert the entire greedy solution into the optimal solution opt while using one fewer pen in each exchange. But this contradicts the assumption that opt is an optimal solution that uses fewer pens than our greedy solution.

Hence, there cannot exist an optimal solution that uses fewer pens than our greedy solution, and our greedy solution is optimal.

Runtime Complexity:

The runtime complexity of this algorithm depends on the number of points and the number of pens used.
Let n be the number of points on the number line, and let k be the number of pens used in the solution.

1. Finding the leftmost uncolored point can be done in O(n) time since the points are sorted.
2. Coloring all the points in a range [m - 5, m + 5] takes O(1) time for each pen.
3. We might use up to n pens (one for each point), so the overall runtime complexity is O(n).

In summary, the algorithm runs in O(n) time complexity.

2. Pseudocode:

```
 GreedyCakeScheduling(cakes):
# Calculate the completion time for each cake
for each cake Ci in cakes:
   Ci.completion_time = Ci.baking_time + Ci.decoration_time

# Sort the cakes in increasing order of completion time
Sort(cakes, by=completion_time)

# Initialize the schedule list
schedule = empty list

# Initialize the time at which the oven is available
oven_available_time = 0

# Schedule the cakes
for each cake Ci in cakes:
   # Determine the time when the oven will be available for this cake
   available_time = max(oven_available_time, Ci.completion_time -
Ci.baking_time)

   # Update the oven_available_time for the next iteration
   oven_available_time = available_time + Ci.baking_time

   # Add the cake to the schedule
   schedule.append((Ci, available_time))

return schedule
```

Algorithm:

1. Create a list of tasks, where each task Ci is represented by its index i, baking time Bi, and decoration time Di.
2. Calculate the completion time for each task, which is the sum of its baking time Bi and decoration time Di.
3. Sort the tasks in increasing order of completion time.
4. Initialize a list of scheduled tasks, **schedule**, as an empty list to keep track of the order in which we schedule the cakes.
5. Iterate over the sorted tasks, and for each task Ci, do the following: a. If the oven is not currently busy (no cake is baking), schedule this task in the oven for baking. b. If the oven is busy, calculate the time at which the oven will be available again (time of completion of the cake

currently baking) and schedule this task in the oven to start baking at that time. c. Add this task Ci to the **schedule** list.

Proving the correctness of the algorithm using the Interval Scheduling technique:

Let's assume that there exists an optimal schedule **opt_schedule** that finishes all the cakes in a shorter overall time than our greedy schedule **schedule**.
Consider the first task Ci in our greedy schedule that differs between **schedule** and **opt_schedule**. This task must exist since **opt_schedule** is different from our greedy schedule.
Now, let's consider the first time of completion of a cake in our greedy schedule **schedule**, say t_greedy.
In the optimal schedule **opt_schedule**, there must exist a cake Cj that is scheduled to complete at or before time t_greedy. This is because **opt_schedule** is an optimal schedule, and there must be some cake that finishes earlier or at the same time as the first cake in our greedy schedule.
Let's consider two cases:

1. If Ci and Cj are the same cake: If Ci and Cj are the same cake, they have the same baking time Bi and decoration time Di. In this case, our greedy schedule **schedule** and the optimal schedule **opt_schedule** are identical up to this point. Since the schedules are identical up to this point and the subsequent tasks are the same in both schedules, the overall time for both schedules will be the same. This contradicts the assumption that **opt_schedule** finishes in a shorter overall time.

2. If Ci and Cj are different cakes: Since Cj is scheduled to complete at or before t_greedy in the optimal schedule **opt_schedule**, we can swap the scheduling of Ci and Cj in the **schedule** without affecting the completion time of any other cakes in the schedule. By doing this, we will have scheduled Ci earlier, and the time for **schedule** will be the same or shorter than that of **opt_schedule**. This contradicts the assumption that **opt_schedule** finishes in a shorter overall time.

In both cases, we have reached a contradiction, proving that our greedy schedule **schedule** is at least as good as the optimal schedule **opt_schedule**

Runtime complexity:

1. Calculating the completion time for each cake takes O(n) time, where n is the number of cakes. This is because it involves a simple addition of the baking time and decoration time for each cake, which is a constant-time operation for each cake.
2. Sorting the cakes based on completion time takes O(n log n) time using efficient sorting algorithms like merge sort or quicksort.
3. Iterating over the sorted tasks takes O(n) time.
4. Scheduling each cake takes constant time O(1).

Therefore, the overall runtime complexity of the algorithm is dominated by the sorting step, resulting in O(n log n) time complexity.

3. **Pseudocode:**

```
function isGreedyOptimal(greedy_solution G, optimal_solution O):
   n = length(G)
   for k = 1 to n do
      di = G[k] // Job di at position k in the greedy solution
      l = positionOf(di, O) // Find the corresponding position l in the optimal solution where di is placed

      // Calculate profits in the greedy solution
      profit_greedy_di = di * (n - k)
      dj = G[l]
      profit_greedy_dj = dj * (n - l)

      // Calculate profits in the optimal solution
      profit_optimal_di = di * (n - l)
      dj = O[k]
      profit_optimal_dj = dj * (n - k)

      // Check if di > dj and if the inequalities hold
      if di > dj and profit_greedy_di > profit_greedy_dj and profit_optimal_di > profit_optimal_dj:
         // The inequalities hold, so swapping jobs between positions k and l will not increase total profit
         continue
      else:
         // The inequalities do not hold, so the greedy strategy is not optimal
         return false

   // If the loop completes without finding any violations, the greedy strategy is optimal
   return true
```

Proof by Exchange Argument:

Assume that there exists an arbitrary position k in the greedy solution G, where d_i is placed at position k. Let l be the corresponding position in the optimal solution O where job di is placed.

We can see the following observations:

In the greedy solution G:

The profit contributed by d_i at position k is $d_i * (n - k)$.
The profit contributed by d_j at position l is $d_j * (n - l)$.
Since $d_i > d_j$ (due to the decreasing order of difficulties in G), we have $d_i * (n - k) > d_j * (n - l)$.
In the optimal solution O:

The profit contributed by d_i at position l is $d_i * (n - l)$.
The profit contributed by d_j at position k is $d_j * (n - k)$.
Since $d_i > d_j$ (due to the decreasing order of difficulties in G), we have $d_i * (n - l) > d_j * (n - k)$.
Now, let's consider the swap of job d_i and job d_j between positions k and l:

In the greedy solution G after the swap:

The profit contributed by d_i at position l is $d_i * (n - l)$.
The profit contributed by d_j at position k is $d_j * (n - k)$.
In the optimal solution O after the swap:

The profit contributed by d_i at position k is $d_i * (n - k)$.
The profit contributed by d_j at position l is $d_j * (n - l)$.
Since $d_i * (n - l) > d_j * (n - k)$ and $d_i * (n - k) > d_j * (n - l)$ (as derived from the observations above), we can conclude that swapping job d_i with job d_j between positions k and l will not increase the total profit in either solution.

Thus, for any arbitrary position k in the greedy solution G and its corresponding position l in the optimal solution O, swapping the jobs at these positions will not increase the total profit. Therefore, the greedy strategy is optimal, and completing jobs in decreasing order of difficulty maximizes the sum of profits.