# Homework 6: Solution outlines by your TA

## Problem 1 [Ex 16.2-5] Placing Unit Intervals

Input: Set of real numbers $\{x_1, x_2, \ldots, x_n\}$.

Output: Smallest set of unit–length closed intervals that contains all of the points.

Greedy Solution:

1. Sort the set so that $x_1 \leq x_2 \leq \ldots \leq x_n$.
2. Let $x_{\min}$ be the smallest number in the set.
3. Place the closed interval at $[x, x+1]$.
4. Remove from the sorted set all the numbers that do not fall in the above interval.
5. Repeat steps (2)–(4) until the set is empty.

Running time:

Step(1) takes $O(n \lg n)$ time using any of the standard sorting techniques.
Step(2) just has to return the first number in the set. Therefore, Steps(2) & (3) take $O(1)$ time.
Step(4): Since the set is sorted, this step takes time proportional to the number of elements that fall in the particular interval; this will take $O(1)$ time per element in the set, and thus $O(n)$ time on the whole.
Therefore the running time of the algorithm is $O(n \lg n) + O(n) = O(n \lg n)$.

PROOF of correctness

Greedy Choice property: We claim that (one of) the optimal solution will have the first unit–interval starting at the smallest number $x_{\min}$ in the set.

Proof:

The smallest number $x_{\min}$ should be covered by atleast one of the unit–intervals. Say the interval covering $x_{\min}$ starts before $x_{\min}$, i.e. it is given by some $[y, y+1]$, where $y < x_{\min}$. Since there are no numbers less that $x_{\min}$, we can move the interval from $[y, y+1]$ to $[x_{\min}, x_{\min}+1]$ without increasing the number of intervals. Therefore our claim is correct.

Optimal Sub–structure property: Once the first interval is chosen, we claim that the remaining problem has an optimal sub–structure.

Proof:

Since the elements that are contained in the first interval are already covered, the position for the next interval can be completely determined by the rest of the elements in the set. The rest of the points form a new set with fewer elements and the same problem of being covered with the fewest number of unit–intervals. Therefore this has an optimal sub–structure property.

## Problem 2 [Prb 16-1] Coin Changing

The problem is to give change for $n$¢ with smallest number of coins.

(a) Denomination set D = $\{25¢, 10¢, 5¢, 1¢\}$. The greedy algorithm first gives as many quaters as possible, then give as many dimes as possible, then as many nickels as possible and remaining pennies.

1. # quarters, $q \leftarrow \lfloor \frac{n}{25} \rfloor$.
2. $n \leftarrow n - 25q$
3. # dimes, $d \leftarrow \lfloor \frac{n}{10} \rfloor$.
4. $n \leftarrow n - 10d$
5. # nickels, $k \leftarrow \lfloor \frac{n}{5} \rfloor$.
6. $n \leftarrow n - 5k$
7. # pennies, $p \leftarrow n$

Proof of correctness:
We justify the following claims.

- There can not be more than 4 pennies in the optimal solution, since any solution with 5 pennies can be replaced by one additional nickel and 5 fewer pennies, which will result in a better solution (with 4 coins less).
- Similarly, there cannot be more than 1 nickel in the optimal solution, since an solution with 2 nickels can be replaced by one additional dime and 2 fewer nickels, which will result in a better solution (with 1 coin less).
- Along the same lines, there cannot be more than 2 dimes because 3 dimes can be replaced by a quarter and a nickel resulting in a better solution with 1 coin less.

Note that the algorithm ensures that there are no more than the required number of coins of each denomination. The optimal solution has $q = \lfloor \frac{n}{25} \rfloor$ quarters, which is the maximum possible quarters in it. There cannot be more than $q$ quarters since $(q+1)25 > n$.

Caution: It is always true that in order to replace one coin of a higher denomination we need more than 1 coin of lower denominations. But the same may not be true when we have to replace more than 1 coins of the higher denomination. This will be evident in part (c) of the problem.

(b) Denomination set $D = \{c^k¢, c^{k-1}¢, \ldots, c^1¢, c^0 = 1¢\}, c > 1$.
As in the part (a), the greedy algorithm works by giving as many coins of the higher denomination as possible before giving coins of the smaller denominations.

Proof of Correctness: The proof is similar to the one in part(a). The important observation is that there cannot be more than $c - 1$ coins of any denomination except the $c^k$ (the highest

denomination), as any $(c-1)$ coins of $c^i$¢ can be replaced by 1 $c^{i+1}$¢ coin resulting in a better solution.

Greedy choice property: The optimal solution contains the maximum possible coins of $c^k$¢. If there are fewer, there would have to be atleast $c$ more coins of $c^{k-1}$¢ for each $c^k$¢ coin less.

Optimal Substructure property: The remaining value for which change has to be given needs to be given using a subset of the coins $\{c^{k-1}¢, \ldots, c^1¢, c^0 = 1¢\}$ and the optimal solution will contain optimal change for this sub–problem.

One student made a very interesting observation: This problem can be solved by looking at $n$¢ in a different base. When is represented using base–$c$, the different digits in $(n)_c$ will give the number of coins of each denomination in the solution. Another good reason to write numbers the way they are written :)

(c) $D = \{4¢, 3¢, 1¢\}$.
Change for 6¢ by the greedy algorithm will yeild $4¢ + 1¢ + 1¢$ which is 3 coins where as the optimal solution has just 2 coins ($3¢ + 3¢$).

(d) Given a denomination set $D = \{d_k¢ > d_{k-1}¢ > \ldots > d_2¢ > d_1¢\}, d_1 = 1¢$. we need to give a $O(nk)$ time algorithm that gives optimal change for $n$¢. (We do this using dynamic programming).

We define, $C(i)$ to be the optimal number of coins for changing $i$¢. Then we have the following recursion.

$$C(i) = \begin{cases} \infty & \text{if } i < 0 \\ 0 & \text{if } i = 0 \\ \min\limits_{1 \le j \le k} \{C(i - d_j) + 1\} & \text{otherwise} \end{cases} \tag{1}$$

The answer is available in $C(n)$. The size of the memoization table is $O(n)$. To fill up each cell will take $O(k)$ comparisons. Therefore the running time is $O(nk)$.

## Problem 3 [Prb 16-2(a)] Scheduling to Minimize Average Completion Time

Input: Set of processes $\{a_i\}, 1 \le i \le n$.
Process time of $p_i$ for each process $a_i$.

If the order in which the processes are excuted is $a_1, a_2, \ldots, a_n$, the completion time for the $a_k$ will be given by

$$c_k = \sum_{i=1}^{k} p_i \tag{2}$$

Intuition: If we execute a long process first, all the processes excuted after it will have longer completion times. So it is wiser to have shorter jobs executed first.

Greedy Algorithm

1. Sort the processes in the ascending order of their processing times. so that $p_1 \leq p_2 \leq \ldots \leq p_n$.
2. Execute the processes in that order. (i.e. $\{a_1, a_2, \ldots, a_n\}$.

Step (1) takes $O(n \lg n)$ time and step (2) takes $O(n)$ time. Therefore running time of the greedy algorithm is $O(n \lg n)$.

Proof of Correctness

Greedy Choice property

Claim: The shortest process should be executed first in the optimal schedule.

Proof (by contradiction): Say $a_m$ is the process with the shortest process time $p_m$. Let the optimal process schedule be $a_1 a_2 \ldots a_m \ldots a_n$, where the first job has a larger process time that the minimum, i.e. $p_1 > p_m$.

The completion times of processes between $a_1$ and $a_m$ will be given by

$c_k = \sum_{i=1}^{k} p_i, 1 \leq i \leq m$

If we exchange the processes $a_1$ and $a_m$, let us see how the completion times change.

The new process schedule would be $a_m a_2 a_3 \ldots a_1 a_{m+1} \ldots a_n$.

All processes that are executed after $a_1$ (i.e. from $a_{m+1}$ onwards) will have the same completion time as in the original schedule.

$c'_m = p_m$ and $c'_1 = p_m + \sum_{i=2}^{m-1} p_i + p_1 = \sum_{i=1}^{m} p_i$

$$\begin{aligned} c'_m + c'_1 &= p_m + \textstyle\sum_{i=1}^{m} p_i \\ &= p_m + c_m \\ &= p_1 + c_m + (p_m - p_1) \\ &= c_1 + c_m + (p_m - p_1) > c_1 + c_m. \end{aligned} \tag{3}$$

All processes between $a_m$ and $a_1$ will have completion times, given by

$$\begin{aligned} c'_k &= p_m + \textstyle\sum_{i=2}^{k} p_i \\ &= [p_1 + \textstyle\sum_{i=2}^{k} p_i] - p_1 + p_m \\ &= c_k + (p_m - p_1) > c_k \end{aligned} \tag{4}$$

From the above two observations, we note that sum of the completion times of tasks before $a_m$ only improves when exchaged with $a_1$. And since it remains the same for the rest, we can conclude that having $a_m$ as the first task gives a better schedule. This is a contradiction. Hence the optimal schedule has the shortest task scheduled first.

Optimal Substructure: Once the shortest job is scheduled, the rest of the jobs form a sub-problem which again have to scheduled optimally in order for the big problem to have an optimal solution.