CSDS 310 HW 4

1

We started activity selection problem in class. Prove that the following greedy choices do not lead to optimal solutions for the activity selection problem:

a

Select the activity with the earliest starting time.

✓ **Answer**

Counterexample:
$[[1, 15], [2, 3], [4, 5], [6, 7]]$

This would choose $[1, 15]$, which is one activity, while clearly the optimum is the other three.

b

Select the activity with least duration.

✓ **Answer**

Counterexample:
$[[0, 10], [9, 11], [10, 20], [19, 21], [20, 30]]$

This would choose $[[9, 11], [19, 21]]$ when the optimal is the other three (longer) activities.

c

Select the activity that overlaps the fewest other remaining activities

✓ **Answer**

Counterexample:
$[[0, 10], [9, 11], [9, 11], [9, 11], [10, 20], [10, 20], [10, 20], [19, 21], [19, 21], [19, 21], [20, 30]]$

Number of overlaps:
$[3, 6, 6, 6, 7, 7, 7, 6, 6, 6, 3]$

This strategy would pick one of the $[9, 11]$ and one of the $[19, 21]$, which is less optimal than selecting the three other activities.

2

You are given two sets A and B, each containing n positive integers. You can choose to reorder each set however you like. After reordering let a_i be the ith element of set A and b_i the ith element of set B. You then receive a payout of $\prod_{i=0}^{n} a_i^{b_i}$.

Give an algorithm that maximizes your profit.

✓ **Answer**

It is evident that we would like as many multiples of larger numbers as possible, as all values in A will be producted in the end anyway.

We should sort A and B in both either increasing order or decreasing order, as long as A and B are in the same order.

Pseudocode

```
procedure maxPay(A, B)
        return A.sort(), B.sort()
```

Proof

Our greedy choice is to pick to do the activity that is due the soonest first, claiming that this leads to the minimum max late activity. This leads to a selection of activities by the order of their deadline.

Assume that this ordering of activities S is suboptimal.

There must be an optimal ordering that we call S^*

We can now define an order of swaps that will convert our solution S into the optimal solution S^*

Let s_i^* be the ith activity in the solution ordering S^* and correspondingly s_i for S.

Let $p_{s_i^*}$ be the position of s_i^* in the set S.

Now we define a set of swaps W_i that will move $s_{p_{s_i^*}}$ to the position i in our solution set S. First, swap $p_{s_i^*}$ forward one position. Repeat until it is in the i position.

We are abot to get from S to S^* by the swap order W which is defined as the list of swaps $W_1, W_2, \ldots, W_n$.

For any swap j and $j-1$ we can guarantee that $d_j > d_{j-1}$ as we maintain the sorted status of the array for all items right of i in the set of swaps W_i and right of $p_{s_i^*}$ during the swapping process.

Let $T_k = \sum_{i=1}^{k} s_i$, which represents the total time taken since the beginning until the completion of activity s_i.

We now define the completion times of the items j and $j-1$ in the swap as $T_j = T_{j-2} + s_{j-1} + s_j$ and $T_{j-1} = T_{j-2} + s_{j-1}$. We can denote the delay of item j and $j-1$ as
$\Delta_j = T_{j-2} + s_{j-1} + s_j - d_j$ and
$\Delta_{j-1} = T_{j-2} + s_{j-1} - d_{j-1}$.

After swapping j and $j-1$, our delays will be
$\Delta_j^* = T_{j-2} + s_j - d_j$
$\Delta_{j-1}^* = T_{j-2} + s_j + s_{j-1} - d_{j-1}$

We can denote Δ_j^* and Δ_{j-1}^* in terms of Δ_j and Δ_{j-1} now.

$\Delta_j + d_j = \Delta_{j-1}^* + d_{j-1}$
$\Delta_j = \Delta_{j-1}^* + d_{j-1} - d_j$
$\Delta_j < \Delta_{j-1}^*$ since $d_j > d_{j-1}$

$\Delta_{j-1} = \Delta_{j-1}^* - s_j$
$\Delta_{j-1} < \Delta_{j-1}^*$

Thus, the delay after any swap in the swap order W_i will lead to a less optimal solution. As such, our original solution S will be more optimal than S^* regardless of how you choose S^* to be, assuming $S^* \neq S$.

3

We have n activities. Each activity requires t_i time to complete and has deadline d_i. We would like to schedule the activities to minimize the maximum delay in completing any activity; that is, we would like to assign starting times s_i to all activities so that $\max_{1 \leq i \leq n} \Delta_i$ is minimized, where $\Delta_i = f_i - d_i$ is the delay for activity i and $f_i = s_i + t_i$ is the finishing time for activity i. Note that we can only perform one activity at a given time (if activity i starts at time s_i, the next scheduled activity has to start at time f_i).

For example, if $t = \langle 10, 5, 6, 2 \rangle$ and $d = \langle 11, 6, 12, 20 \rangle$, then the optimal solution is to schedule the activities in the order $\langle 2, 1, 3, 4 \rangle$ to obtain starting/finishing times $s/f = \langle 5/15, 0/5, 15/21, 21/23 \rangle$ and achieve a maximum delay of 9 (for the third activity).

Give an algorithm that minimizes the maximum delay.

✓ **Answer**

The next activity we choose to do will be the one that has the soonest deadline. Essentially, we will sort the indices of the deadline array by the values of the deadlines. These will be the order of the indices that we will perform the tasks in.

Pseudocode

```
procedure minDeadline(t, d)
        return [
                x[1]
                for x
                in sorted(
                        zip(
                                d,
                                range(len(d))
                        ),
                        x ⇒ x[0]
                )
        ]
```

Proof

Our greedy choice is to pick to do the activity that is due the soonest first, claiming that this leads to the minimum max late activity. This leads to a selection of activities by the order of their deadline.

Assume that this ordering of activities S is suboptimal.

There must be an optimal ordering that we call S^*

We can now define an order of swaps that will convert our solution S into the optimal solution S^*

Let s_i^* be the ith activity in the solution ordering S^* and correspondingly s_i for S.

Let $p_{s_i^*}$ be the position of s_i^* in the set S.

Now we define a set of swaps W_i that will move $s_{p_{s_i^*}}$ to the position i in our solution set S. First, swap $p_{s_i^*}$ forward one position. Repeat until it is in the i position.

We are abot to get from S to S^* by the swap order W which is defined as the list of swaps $W_1, W_2, \ldots, W_n$.

For any swap j and $j-1$ we can guarantee that $d_j > d_{j-1}$ as we maintain the sorted status of the array for all items right of i in the set of swaps W_i and right of $p_{s_i^*}$ during the swapping process.

Let $T_k = \sum_{i=1}^{k} s_i$, which represents the total time taken since the beginning until the completion of activity s_i.

We now define the completion times of the items j and $j-1$ in the swap as $T_j = T_{j-2} + s_{j-1} + s_j$ and $T_{j-1} = T_{j-2} + s_{j-1}$. We can denote the delay of item j and $j-1$ as
$\Delta_j = T_{j-2} + s_{j-1} + s_j - d_j$ and
$\Delta_{j-1} = T_{j-2} + s_{j-1} - d_{j-1}$.

After swapping j and $j-1$, our delays will be
$\Delta_j^* = T_{j-2} + s_j - d_j$
$\Delta_{j-1}^* = T_{j-2} + s_j + s_{j-1} - d_{j-1}$

We can denote Δ_j^* and Δ_{j-1}^* in terms of Δ_j and Δ_{j-1} now.

$\Delta_j + d_j = \Delta_{j-1}^* + d_{j-1}$
$\Delta_j = \Delta_{j-1}^* + d_{j-1} - d_j$
$\Delta_j < \Delta_{j-1}^*$ since $d_j > d_{j-1}$

$\Delta_{j-1} = \Delta_{j-1}^* - s_j$
$\Delta_{j-1} < \Delta_{j-1}^*$

Thus, the delay after any swap in the swap order W_i will lead to a less optimal solution. As such, our original solution S will be more optimal than S^* regardless of how you choose S^* to be, assuming $S^* \neq S$.

4

We have infinite supply of integer coin denominations of $c_1 = 1 < c_2 < \cdots < c_k$ to make change for a given an integer amount n. For this purpose, we would like to find the minimum number of coins that add up to n. An obvious greedy choice for this problem is to use the largest coin that has value less than or equal to n (e.g., if $c_k \leq n$, then return c_k, and solve the problem for $n - c_k$).

a

Prove that, if the coin denominations are arbitrary, this greedy choice is not guaranteed to lead to an optimal solution. (Just prove the greedy choice, no pseudocode or run time)

✓ **Answer**

Counterexample:
$n = 10$
$c = [1, 5, 8]$

Using the largest possible coin possible, we are left with the solution of $[8, 1, 1]$ of 3 coins.

This is clearly not optimal as we are able to represent 10 as $[5, 5]$, with 2 coins.

b

Prove that, if the coin denominations are powers of 2, i.e., $c_i = 2^{i-1} : 1 \leq i \leq k$, then this greedy choice is guaranteed to lead to an optimal solution. (Just prove the greedy choice, no pseudocode or run time)

✓ **Answer**

This is equivalent to the binary counting problem.

I assert that given any target amount n, the largest possible coin that is less than n must be part of the solution S for the minimal number of coins to represent n.

Assume that this statement is not true.

If the largest coin is c_k, where $c_k \leq n$, c_k is not part of S.

By the definition of the available coins, $\sum_{i=1}^{k-1} c_i = c_k - 1 = c'_k$

Thus, $c'_k < n$.

c'_k is the maximum possible amount possible without repeating coins and without using the max coin c_k. Since $c'_k < n$, we know that at least one coin has to be repeated if c_k is not part of the solution.

For any coin c_j that is repeated, we are able to construct S' where the number of coins is one less, where we remove two c_j and add one c_{j+1}, as $2c_j = c_{j+1}$. Thus $S' = S - 1$ and so S cannot be an optimal solution.

Now, say S'_j is S with all possible c_j coins merged into c_{j+1} coins and all smaller coins merged upwards as well. We know there will be no repeats of coins smaller than c_{j+1}.

Let $c''_j = \sum_{i=j}^{k-1} a_i c_i$ with these coins merged where a_i is the amount of coins present for that coin type. Also let $c'''_j = \sum_{i=1}^{j-1} p_i c_i$ where p_i is whether the coin c_i is present when all coins below c_j have been merged.

$c''_j + c'''_j = n > c'_k$

As $j \to k-1$, $c''_{k-1} + c'''_{k-1} > c'_k$,
We know $c'''_{k-1} < c'_{k-1} < n$ by their definitions. Thus $c''_{k-1} > n - c'_{k-1}$.

However, since $n > c'_k$, $n - c'_{k-1} > c_{k-1}$
$\implies c''_{k-1} > c_{k-1}$
Since c''_k is a sum of only c_k, we must merge at least two c_{k-1} coins into a c_k coin.

As such, a solution containing c_k must always be more optimal than a solution not containing it. Thus our greedy solution picking c_k is correct.