

## Homework Assignment #3

Due: March 26

Student: Shuo Yang

1. (a) Suppose we have 6 activities, for activity  $i$ ,  $s_i$  is the start time,  $f_i$  is the finish time and  $d_i$  is the duration. And we sort them in order of increasing duration:

$i$	1	2	3	4	5	6
$s_i$	8	3	1	4	4	0
$f_i$	9	5	4	8	9	6
$d_i$	1	2	3	4	5	6

The greedy procedure when the activities are considered in order of increasing duration would be: 1) select activity 1 since it is the local best; 2) select activity 2 since it is compatible with activity 1; 3) no other activities can be selected since they are not compatible with either activity 1 or 2. So the procedure yields the solution  $\{1, 2\}$ , but the optimal solution should be  $\{3, 4, 1\}$ . Therefore the greedy procedure is not correct.

- (b) With the same input, now let us sort them in order of increasing start-time:

$i$	1	2	3	4	5	6
$s_i$	0	1	3	4	4	8
$f_i$	6	4	5	8	9	9

The greedy procedure when the activities are considered in order of increasing duration is:

**Function** *GreedyActivitySelection*( $S, F, n$ )

```

 $A := \{1\}$  // activity with the earliest start time
 $j := 1$  // indicates the activity with greatest start time in  $A$ 
for  $i := 2$  to  $n$ 
    if  $S[i] \geq F[j]$ 
         $A := A \cup \{i\}$ 
         $j := i$ 
return  $A$ 

```

Running the above procedure with the same input yields the solution  $\{1, 6\}$  which is certainly not an optimal solution. Therefore the greedy procedure is not correct.

- (c) Consider the following 9 activities ordered by the number of overlaps which is denoted by  $o_i$ :

$i$	1	2	3	4	5	6	7	8	9
$s_i$	4	0	7	1	2	3	5	6	6
$f_i$	6	3	10	4	4	5	7	8	9
$o_i$	2	2	2	3	3	3	3	3	3

When considering the order of increasing number of overlaps, the greedy procedure would first pick activity 1, then pick activity 2, and finally pick activity 3. Thus the solution derived by the greedy procedure is  $\{1, 2, 3\}$ , but the optimal solution should be  $\{2, 6, 7, 3\}$ . Therefore the greedy procedure is not correct.

2. Label  $n$  gas stations as  $g_1, g_2, \dots, g_n$  and let  $d_i$  be the distance between gas stations  $g_i$  and  $g_{i+1}$ , specially,  $d_0$  denotes the distance between  $g_1$  and city  $A$ , and  $d_n$  denotes the distance between city  $B$  and  $g_n$ . Let  $D$  be the total distance between city  $A$  and  $B$ , where  $D = \sum_{i=0}^n d_i$ .

### Greedy algorithm

The greedy strategy is to refuel at the gas station whose distance from the current stop is closest to  $m$  but less than  $m$ .

**Function** *GreedyTripRefuel()*

```

     $A := \emptyset$  // the set of refueling stops
     $i := 0$  // indicates the farthest gas station one can make.
     $T_1 := 0$  // distance traveled before making a refuel
     $T_2 := 0$  // total distance traveled
    while  $T_2 < D$ 
        if  $(T_2 - T_1) \leq m$ 
             $T_2 := T_2 + d_i$ 
             $i := i + 1$ 
        else
             $i := i - 1$  // cannot make it to station  $g_i$ , so must refuel at station  $g_{i-1}$ 
             $A := A \cup \{g_i\}$ 
             $T_2 := T_2 - d_i$ 
             $T_1 := T_2$ 
    return  $A$ 
```

The algorithm runs in  $O(n)$  time since there are at most  $n$  gas stations along the way.

### Correctness

**Lemma:** Suppose  $A$  is a subset of an optimal solution where the latest refuel stop is station  $g_k$ . Let  $g_i$  be the gas station after  $g_k$  whose distance from  $g_k$  is closest to  $m$  but less than  $m$ . Then  $A \cup \{g_i\}$  is also a subset of an optimal solution.

*Proof.* Since  $A$  is a subset of an optimal solution, let  $A^*$  be an optimal solution with  $A \subseteq A^*$ . Let  $g_j$  be the gas station in  $A^* - A$  that is next to  $g_k$ . We have two cases:

- case-1:  $i = j$ . Then  $A \cup \{g_i\} \subseteq A^*$ , the lemma holds.
- case-2:  $i \neq j$ . Since the distance from  $g_k$  to  $g_i$  is closest to  $m$  but less than  $m$ , this distance must be greater than the distance from  $g_k$  to  $g_j$ . Thus we can replace  $g_j$  with  $g_i$  and still yields an optimal solution since  $g_i$  is closer to the future stops than  $g_j$ .

□

**Theorem:** *GreedyTripRefuel()* finds an optimal solution.

*Proof.* Initially,  $A$  is an empty set, which is a subset of an optimal solution and  $g_k$  in this case would be city  $A$ . By the lemma,  $A \cup g_i$  is a subset of an optimal solution where  $g_i$  is the gas station whose distance from city  $A$  is closest to  $m$  but less than  $m$ .

By induction on the number of iterations, when the function terminates,  $A$  is a subset of an optimal solution.

Since from the starting point, the greedy algorithm makes the least possible stops, there is no better solution with fewer stops than the one produced by the greedy algorithm. Therefore  $A$  is optimal. □

### 3. (a) Greedy algorithm

Let the set of  $n$  tasks be  $\{a_i | 1 \leq i \leq n\}$ . The greedy strategy is to schedule the shortest task first.

- i. Merge sort the  $n$  tasks in the order of increasing execution time such that  $t_1 \leq t_2 \leq \dots \leq t_n$ .
- ii. Schedule the tasks in the order of  $\{a_1, a_2, \dots, a_n\}$ .

Step (a) takes  $O(n \log n)$  time and step (b) takes  $O(n)$  time, thus the overall runtime is  $O(n \log n)$ .

#### Correctness

First, we define containment relationship.

We define  $\{a_1, a_2, \dots, a_m\}$  as a schedule sequence such that  $a_i$  is scheduled after  $a_{i-1}$  where  $1 < i \leq m$ . For a schedule sequence  $\{a_1, a_2, \dots, a_n\}$ , we define  $\{a_1, a_2, \dots, a_k\}$  as a prefix-subsequence of it where  $k \leq n$ .

**Lemma:** Suppose  $A$  is a prefix-subsequence of an optimal schedule sequence. Let  $a_i$  be the task that has the shortest execution time among those tasks that are not in  $A$ . Let  $A'$  be the prefix-subsequence by appending  $a_i$  to the end of  $A$ . Then  $A'$  is also a prefix-subsequence of an optimal schedule sequence.

*Proof.* Since  $A$  is a prefix-subsequence of an optimal schedule sequence, let  $A^*$  be the optimal schedule sequence that contains  $A$  as a prefix-subsequence. Let  $a_i$  be the task that has the shortest execution time among those tasks in  $A^* - A$ . By appending  $a_i$  to the end of the  $A$ , we get another prefix-subsequence  $A'$ . There are two cases:

- i.  $A'$  is a prefix-subsequence of  $A^*$ , the lemma holds.
- ii.  $A'$  is not a prefix-subsequence of  $A^*$ , Let  $a_j$  be the task that appears first in the subsequence  $A^* - A$ . In this case we must have:  $t_i = t_j$ . We prove this by contradiction:

Suppose  $t_i \neq t_j$ , since  $a_i$  is the task with the shortest execution time in  $A^* - A$ , we must have:  $t_j > t_i$ . Let the last task in  $A$  be  $a_m$ . Without loss of generality, assume  $A^* - A$  is ordered as  $a_j, a_{x_1}, \dots, a_{x_s}, a_i, a_{y_1}, \dots, a_{y_r}$  such that there are  $s$  tasks scheduled after  $a_j$  and before  $a_i$ , and there are  $r$  tasks scheduled after  $a_i$ . Let  $c_{x_k}$  denotes the completion time for the tasks scheduled in between  $a_j$  and  $a_i$ , and let  $c_{y_k}$  the completion time for the tasks scheduled after  $a_i$ . We have:

$$c_j = c_m + t_j \tag{1}$$

$$c_{x_k} = c_j + \sum_{1 \leq i \leq k}^{k \leq s} t_{x_i} \tag{2}$$

$$c_i = c_j + \sum_{1 \leq i \leq s} t_{x_i} + t_i \tag{3}$$

$$c_{y_k} = c_i + \sum_{1 \leq i \leq k}^{k \leq r} t_{y_i} \tag{4}$$

Now suppose we exchange the positions of  $a_i$  and  $a_j$  to produce another subsequence  $(A^* - A)'$

$a_i, a_{x_1}, \dots, a_{x_s}, a_j, a_{y_1}, \dots, a_{y_r}$  and together with  $A$ , this produces another schedule sequence  $A^{*'}.$  In this case, let  $c'_j$  be the completion time for  $a_j$ ,  $c'_i$  be the completion time for  $a_i$ ,  $c'_{x_k}$  be the completion time for the tasks scheduled in between  $a_j$  and  $a_i$ , and let  $c'_{y_k}$  be

the completion time for the tasks scheduled after  $a_j$ . We have:

$$c'_i = c_m + t_i \quad (5)$$

$$c'_{x_k} = c'_i + \sum_{1 \leq i \leq k}^{k \leq s} t_{x_i} \quad (6)$$

$$c'_j = c'_i + \sum_{1 \leq i \leq s} t_{x_i} + t_j \quad (7)$$

$$c'_{y_k} = c'_j + \sum_{1 \leq i \leq k}^{k \leq r} t_{y_i} \quad (8)$$

Now we compare completion times for the two subsequences:

$$c_j - c'_i = (c_m + t_j) - (c_m + t_i) \quad (9)$$

$$= t_j - t_i \quad (10)$$

$$> 0 \quad (11)$$

$$c_{x_k} - c'_{x_k} = (c_j + \sum_{1 \leq i \leq k}^{k \leq s} t_{x_i}) - (c'_i + \sum_{1 \leq i \leq k}^{k \leq s} t_{x_i}) \quad (12)$$

$$= c_j - c'_i \quad (13)$$

$$> 0 \quad (14)$$

$$c_i - c'_j = (c_j + \sum_{1 \leq i \leq s} t_{x_i} + t_i) - (c'_i + \sum_{1 \leq i \leq s} t_{x_i} + t_j) \quad (15)$$

$$= c_j - c'_i + t_i - t_j \quad (16)$$

$$= t_j - t'_i + t_i - t_j \quad (17)$$

$$= 0 \quad (18)$$

$$c_{y_k} - c'_{y_k} = (c_i + \sum_{1 \leq i \leq k}^{k \leq r} t_{y_i}) - (c'_j + \sum_{1 \leq i \leq k}^{k \leq r} t_{y_i}) \quad (19)$$

$$= c_i - c'_j \quad (20)$$

$$= 0 \quad (21)$$

Therefore, the sum of the completion time for  $A^* - A$  is absolutely greater than the sum of the completion time for  $(A^* - A)'$ . So the average completion time of  $A^*$  must be greater than that of  $A^{*'}$ , we have found a better solution than  $A^*$ . This contradicts the fact that  $A^*$  is an optimal solution. Thus we must have  $t_i = t_j$ .

So we can get another optimal schedule sequence by exchanging the order of  $a_i$  and  $a_j$  because since  $t_i = t_j$ , the exchange will not affect the overall average completion time. And this new optimal schedule sequence has  $A'$  as a prefix-subsequence. The lemma still holds.  $\square$

**Theorem:** The greedy algorithm finds an optimal schedule sequence.

*Proof.* Initially,  $A$  is a empty sequence, which is always a prefix-subsequence of any schedule sequence. Suppose all  $n$  tasks  $a_1, a_2, \dots, a_n$  have been sorted in the order of increasing execution time. By the lemma,  $\{a_1\}$  is a prefix-subsequence of an optimal schedule sequence.

By induction on the number of iterations, when the algorithm terminates,  $A$  is still a prefix-subsequence of an optimal schedule sequence. And when the algorithm terminates,  $A$  would contain all  $n$  tasks, no other schedule sequence can properly contain  $A$ , thus  $A$  is the optimal schedule sequence.  $\square$

(b) **Algorithm**

The algorithm will use the shortest-remaining-time policy to find an optimal schedule. It uses a min-heap to organize the released but not completed tasks where heap ordering is based on their remaining execution times. A scheduling decision is made when a task is completed or when a new task is released.

Assume that the heap has the following operations:

- $heap.insert(a, r)$ : insert task  $a$  with the remaining time  $r$ .
- $heap.pop()$ : remove the task  $a$  with the smallest remaining time  $r$  and return the pair  $(a, r)$ .
- $heap.findmin()$ : find the task  $a$  with the smallest remaining time  $r$  and return the pair  $(a, r)$ .
- $heap.notempty()$ : return true if the heap is not empty, otherwise return false.

The routine  $InsertHeap(heap, A, R, T, r)$  is used to insert tasks with release time of  $r$  into the min-heap where  $A$  is the set of tasks,  $R$  is the set of release time and  $T$  is the set of execution time.

**Function**  $InsertHeap(heap, A, R, T, r)$

```

for  $i := 1$  to  $n$ 
    if  $r_i == r$ 
         $heap.insert(a_i, t_i)$ 
         $A := A - \{a_i\}$ 
         $R := R - \{r_i\}$ 

```

$min(R)$  is used to find the minimum release time from  $R$ .

The scheduling algorithm is implemented as follows:

**Function** *ScheduleTasks*(*heap*, *A*, *R*, *T*, *r*)

$S := \emptyset$  // set of scheduled tasks

$minr_1 := \min(R)$  // start time for the next scheduled task *a*

*InsertHeap*(*heap*, *A*, *R*, *T*,  $minr_1$ )

$(a, r) := heap.pop()$

$S := S \cup \{a\}$  // schedule task *a*

**while**  $R \neq \emptyset$

$minr_2 := \min(R)$  // next release time

**if**  $minr_2 - minr_1 \geq r$  // task *a* can finish before the next batch of released tasks

$minr_1 := minr_1 + r$  // update the start time for the next scheduled task

$(a, r) = heap.pop()$  // schedule the next task on heap before next batch of release

$S := S \cup \{a\}$

**else** // task *a* runs for  $(minr_2 - minr_1)$  time, then the next batch of release comes

*InsertHeap*(*heap*, *A*, *R*, *T*,  $minr_2$ )

$(a', r') := heap.findmin()$

$r := r - (minr_2 - minr_1)$  // update the remaining time for task *a*

$minr_1 = minr_2$  // update  $minr_1$

**if**  $r' \geq r$  // continue to run task *a*

**continue**

**else** // current running task *a* is preempted

*heap.insert*(*a*, *r*) // put task *a* back to the heap

$(a, r) := heap.pop()$  // schedule the task with the smallest remaining time

$S := S \cup \{a\}$

**while** *heap.notempty*() // no more tasks to release

$(a, r) := heap.pop()$

$S := S \cup \{a\}$

#### 4. (a) Greedy algorithm

The greedy strategy is to always choose the largest denomination that is not greater than the remaining amount.

The function *GreedyCoinChange*(*n*) implements this strategy.

**Function** *GreedyCoinChange*(*n*)

$A := \emptyset$  // the set of choose coins

*denominations* := {25, 10, 5, 1} // set of available denominations

**while**  $n > 0$

$x :=$  the largest value in *denominations* that is  $\leq n$

$A := A \cup \{x\}$  // pick *x*

$n := n - x$

**return** *A*

The runtime of the algorithm is  $O(n)$ .

#### Correctness

First, we make the following claims:

- Claim-1: There cannot be more than 4 pennies in an optimal solution because 5 pennies can be replaced by 1 nickle which yields a better solution with 4 coins less.
- Claim-2: There cannot be more than 1 nickle in an optimal solution because 2 nickles can be replaced by 1 dime which yields a better solution with 1 coin less.
- Claim-3: There cannot be more than 2 dimes in an optimal solution because 3 dimes can be replaced by 1 quarter and 1 nickle which yield a better solution with 1 coin less.
- Claim-4: There cannot be 2 dimes and 1 nickle both in an optimal solution because 2 dimes and 1 nickle can be replaced by 1 quarter which yield a better solution with 2 coins less.

**Lemma:** Suppose  $A$  is a subset of an optimal solution  $A^*$ . Let  $m$  be the amount of the remaining cents, that is, total cents in  $A^* - A$ . Let  $x$  be the largest value among  $\{25, 10, 5, 1\}$  that is not greater than  $m$ . Then  $A' = A \cup \{x\}$  is also a subset of an optimal solution.

*Proof.* We prove by cases. There are total 4 cases.

- Case-1:  $x = 25$ .  
In this case, we have  $m \geq 25$ . If  $x \in A^* - A$ , the lemma holds. If  $x \notin A^* - A$ , then  $A^* - A$  cannot contain quarters. We have two sub-cases: 1)  $25 \leq m < 30$ , then the best possible solution is to pick 2 dimes and 1 nickle, and  $m - 25$  pennies. But this violates the *Claim-4*; 2)  $m \geq 30$ , then the best possible solution must contain at least 3 dimes, but this violates the *Claim-3*. Since in both sub-cases, the best possible solutions violate some claims, we know that they must not be subset of an optimal solution. Therefore  $x$  must be in  $A^* - A$ , the lemma holds.
- Case-2:  $x = 10$ .  
In this case, we have  $10 \leq m < 25$ . If  $x \in A^* - A$ , the lemma holds. If  $x \notin A^* - A$ , then  $A^* - A$  cannot contain quarters and dimes. The best possible solution is to pick at least 2 nickles, but this violates the *Claim-2*. So  $x$  must be in  $A^* - A$ , the lemma holds.
- Case-3:  $x = 5$ .  
In this case, we have  $5 \leq m < 10$ . If  $x \in A^* - A$ , the lemma holds. If  $x \notin A^* - A$ , then  $A^* - A$  cannot contain quarters, dimes and nickles. The best possible solution is to pick  $m$  pennies, but this violates the *Claim-1*. So  $x$  must be in  $A^* - A$ , the lemma holds.
- Case-4:  $x = 1$ .  
In this case, we have  $1 \leq m < 5$  and  $x$  must be in  $A^* - A$  because pennies are the only choices, thus the lemma holds.

Since for all 4 cases, the lemma holds, it must be true.  $\square$

**Theorem:** Greedy algorithm *GreedyCoinChange* finds an optimal solution.

*Proof.* Initially,  $A$  is an empty set which is always a subset of any optimal solutions. By the lemma,  $\{x\}$  is a subset of an optimal solution where  $x$  is the largest denomination that is not greater than  $n$ . By induction on the number of iterations, when the algorithm terminates,  $A$  is still a subset of an optimal solution. When the algorithm terminates,  $A$  would contain coins with the total amount equals  $n$  and satisfy the 4 claims, thus no other solutions can properly contain  $A$ , thus  $A$  is optimal.  $\square$

- (b) *Proof.* prove by giving an counter example. Suppose the denomination system is the set:  $\{1, 3, 4\}$  and  $n = 6$ . The greedy algorithm *GreedyCoinChange* yields the solution  $\{4, 1, 1\}$ . But the optimal solution is  $\{3, 3\}$ . This proves that greedy algorithm from Part (a) does not make optimal change for all systems of coins.  $\square$

- (c) **Lemma:** There cannot be more than  $(a - 1)$  coins with the value  $a^i$  in an optimal solution where  $0 \leq i < k$ .

*Proof.* Prove by contradiction. Suppose there is an optimal solution that contains more than  $(a - 1)$  coins of the value  $a^i$  for some  $i$  where  $0 \leq i < k$ . But  $a$  coins of the value  $a^i$  can be replaced with 1 coin with the value of  $a^{i+1}$ . This yields a better solution with  $a - 1$  coins less since  $a > 1$ . This is a contradiction. Therefore the lemma must always hold.  $\square$

**Theorem:** The greedy algorithm in part-a makes optimal change for the system of coins with denominations  $\{a^0, a^1, \dots, a^k\}$  where  $a > 1$  and  $k \geq 1$ .

*Proof.* Prove by contradiction.

Let  $A$  be the solution found by the greedy algorithm. Suppose  $A$  is not optimal, then let  $A'$  be an optimal solution. Define  $C = \{c_i | 0 \leq i \leq k\}$  as the set of coin counts for  $A$  where  $c_i$  is the number of coins with the value of  $a^i$  in  $A$ . And define  $C' = \{c'_i | 0 \leq i \leq k\}$  as the set of coin counts for  $A'$  where  $c'_i$  is the number of coins with the value of  $a^i$  in  $A'$ .

Let  $j$  be the first highest index for which  $c_j \neq c'_j$ . Then we must have:  $c_j > c'_j$  because our algorithm makes greedy choices locally and picks as much  $a^j$  as possible and this is the first value on which they disagree. Also  $j > 0$  because if only  $c_0$  and  $c'_0$  are different, then the total amount in  $A$  and  $A'$  would be different.

Since  $c_j > c'_j$  and  $c'_j \geq 0$ , the partial amount  $B = \sum_{i=0}^{j-1} c'_i a^i$  in  $A'$  must satisfy the condition that  $B \geq a^j$ .

Also, by the lemma, the largest possible value for the partial amount  $B$  is:

$$\sum_{i=0}^{j-1} c'_i a^i = (a - 1)a^0 + (a - 1)a^1 + \dots + (a - 1)a^{j-1} \quad (22)$$

$$= (a - 1) \sum_{i=0}^{j-1} a^i \quad (23)$$

$$= (a - 1) \frac{1 - a^j}{1 - a} \quad (24)$$

$$= a^j - 1 \quad (25)$$

$$< a^j \quad (26)$$

This contradicts the fact that  $B \geq a^j$ . Thus  $A$  must be an optimal solution.  $\square$

- (d) **Recursive structure of an optimal change**

Give a set of denomination  $D = \{d_1, d_2, \dots, d_k\}$  where  $d_1 = 1$  and an amount of cents  $n$ , an optimal change must start with one of coins in  $D$ , say it is  $d_i$  where  $1 \leq i \leq k$ . Then the optimal change must be  $1 +$  (optimal change for  $n - d_i$ ).

#### Recurrence equation

Define  $C(i)$  be the optimal number of changes made for amount of  $i$  cents. The goal value is  $C(n)$ .

$$C(i) := \begin{cases} \min_{1 \leq j \leq k} (C(i - d_j)) + 1 & i > 0 \\ 0 & i = 0 \\ \infty & i < 0 \end{cases} \quad (27)$$

#### Evaluate the recurrence equation



We fill the table up from  $C(0)$  until  $C(n)$ .

The size of table is  $O(n)$  and to fill each cell, we need  $O(k)$  comparison, so the total run time of filling the table is  $O(kn)$ .

**Recover the optimal solution**

For the optimal number of coins, we simply look up the value of  $C(n)$ .

For the number of each denomination in the optimal solution, we can trace back to see how many coins are needed to make change for  $(n - d_i)$  cents for each  $i$  where  $1 \leq i \leq k$ , and choose the smallest number. Repeat this process until the remaining amount is  $\leq 0$ .