CIS 675 (Fall 2018) Disclosure Sheet

Name: Wentan Bai
HW # 4

Yes **No** Did you consult with anyone on parts of this assignment, including other students, TAs, or the instructor?

Yes **No** Did you consult an outside source (such as an Internet forum or a book other than the course textbook) on parts of this assignment?

If you answered **Yes** to one or more questions, please give the details here:

I consulted question 1 and 4 with teaching assistant, Arash Sahebolamri, to confirm my understanding. I also consult all questions and extra question with another student, Wentian Bai. For all questions, we discussed our ideas and I finish my algorithms independently.

By submitting this sheet through my Blackboard account, I assert that the information on this sheet is true.

Question 1:

My algorithm is :

- Sort the locations of the paintings in increasing order. The locations of sorted paintings are l_1, ..., l_k.
- Place one guard at $l_1 + 1$ location, where 1 represent the 1 unit of distance.
- move along the hallway until there is a painting l_m , which is not protected. Place one guard at $l_m + 1$ location.
- Repeat the third step until all paintings are protected.

Sort locations of the paintings in increasing order

guard $\leftarrow l_1 + 1$
num $\leftarrow 1$
    For i = 2 to k :
        If guard $< l_i$
            guard $\leftarrow l_i + 1$
            num $\leftarrow num + 1$

---

running time:

Sorting uses $O(nlogn)$ time. We iterate through sorted paintings once using $O(n)$ time. The final time is $O(nlogn) + O(n)$ = $O(nlogn)$

Question 2:

My goal is to maximize the amount of payment. Thus, my idea is trying to schedule the high-paying jobs in the time slot just before their deadline. My algorithm is:

- Sort all jobs by their payments in decreasing order. The sorted jobs are $J_1, J_2, ..., J_m$, and for each job J_i, $p_i \leq p_{i+1}$.
- Schedule J_1 in a time slot from time $d_1 - 1$ to time d_1.
- Iterate through the list of remaining jobs in order, and at each step i, check whether we can schedule current job J_i from time $d_i - 1$ to time d_i.
  - If this time slot is already scheduled for a job, traverse the timeline forward from time $d_i - 1$ and find whether there is a empty slot to schedule current job.
  - If there is not a empty slot after traversing, we will not schedule this job.
- After going through list, the timeline is expected result.

Sort all jobs by their payments in decreasing order.
timeline[m] // m-size array
timeline[d_1] $\leftarrow J_1$
For i = 2 to m :
    If timeline[d_i] **is** empty
        timeline[d_i] $\leftarrow J_i$
    Else :
        For k = $d_i - 1$ to 1
            If timeline[k] **is** empty
                timeline[k] $\leftarrow J_i$
Return timeline

---

running time:

The sorting all jobs uses $O(n \log n)$. When we iterate through sorted jobs, for some jobs, we may need to traverse the timeline forward which uses $O(n)$ time. Therefore, whole iteration need $O(n^2)$ time.

The final time is $O(n \log n) + O(n^2) = O(n^2)$

Question 3:

I need to consider all possible ways from the first rental shop to the last rental shop. Based on dynamic programming, I will record the minimum costs of traversed shop to avoid re-computations. My algorithm is:

- Initialize C[1] to 0, which means that there is no cost from first shop to first shop. For other C[2]...C[n], initialize to $c_{12}...c_{1n}$, which we suppose the cheapest cost from first shop to i_{th} shop is to pick up a canoe at the first rental shop and directly travel to last shop without dropping.
- Using a nested loop to find the cheapest way for each rental shop :
  - For the outer loop, each C[i] represents the cheapest recorded way to get to rental shop i.
  - Since we have already the cheapest way from first shop to the i_{th} shop, then we use a nested loop to check whether current $C[i] + c_{ij}$ is cheaper than recorded cost to get to rental shop j.
  - If current $C[i] + c_{ij}$ is cheaper, update to current cost. Otherwise keep the previous record.
- After iteration, the C[n] records the cheapest cost from first rental shop to last rentail shop.

C[n] // n-size array
C[1] $\leftarrow 0$
For i = 2 to n :
    C[i] $\leftarrow c_{1i}$

For i = 2 to n :
    For j = i + 1 to n :
        If C[j] $>$ C[i] + c_{ij}
            C[j] = C[i] + c_{ij}

Return C[n]

---

running time:

The initialization uses $O(n)$ time. The whole nested loop needs $O(n^2)$ time.
The final time is $O(n) + O(n^2) = O(n^2)$

Question 4:

There may be many sets of coins with total value exactly equal to V, but here my goal is to find one. My algorithm is:

- Build an array and initialize every element of this array to an empty set. Each set will records only one possible combination of coins.
- Using a nested loop to find a possible set :
  - For the outer loop, at each step i, we only choose the coin with value v_i.
  - In the inner loop, at each step j, check whether we can construct a new set by set[j - v_i] $\cup$ v_i exactly equal to value j.
  - If the new set equals to our expected value V, just return it and end loop.

```
n = numbers of coins
set[V+1]
For i = 0 to V :
    set[i] ← ∅  // every element of this array is an empty set.

For i = 0 to n :
    For j = v_i to V :
        if set[j] is ∅
            set[j] = set[j - v_i] ∪ v_i
        if set[V] is not ∅
            Return set[V]
```

---

running time:

The initialization uses $O(n)$ time. The whole nested loop needs $O(n^2)$ time.

The final time is $O(n) + O(n^2) = O(n^2)$

Question 5:

(a):

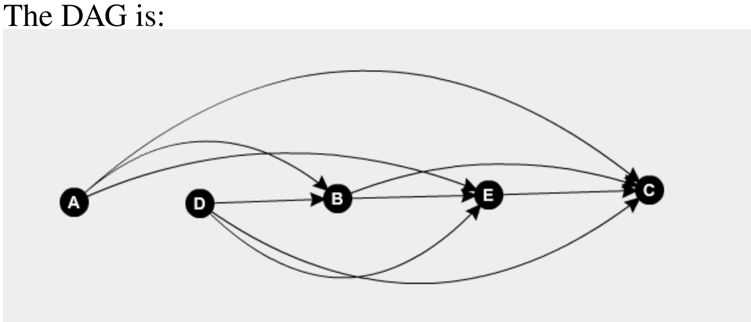
By hint, sort given boxes by their base area, $W \times L$, in decreasing order.

After sorting, we can draw a DAG: Establish a nodes i for each box box_i, and add direct edge (i, j) whenever it is possible for box_i and box_j whose area is in an decreasing order, that is, whenever $i < j$ and $W_i \times L_i > W_j \times L_j$.

For example, given 5 boxes :

Box	**Width**	**Length**	**Height**
A	9	9	1
B	7	5	2
C	1	2	4
D	10	8	2
E	2	3	2

The DAG is:



Since the constraints, after sorting boxes, every two adjacent boxes can be either stacking or not. And for all $i < j$, box_j can not be on top of box_i. Thus, when we draw a grap, there must be no cycle, which means the graph is DAG.

Question 5:

(b):

My algorithm is:

- sort given boxes by their base area, $W \times L$, in decreasing order.
- Create an array S with size of the number of boxes to record the heights of the tallest possible stack for each box.
- Using a nested loop:
  - In the inner loop, since array S records heights of the tallest possible stack of larger boxes, so we traverse S forward to find the maximun S[j] + H_i(height of current box).
  - After ending inner loop, we check whether there is a stack is taller than current box. If yes, record height of stack. Otherwise, only record the height of current box.
- Since, there may not be only one smallest box. Thus, at last we iterate through array S and find the maximum one to return.

```
Sort given boxes by their base area, W × L, in decreasing order.
n = numbers of boxes
S[n]  // n-size array
For i = 1 to n :
    S[i] ← 0

For i = 1 to n :
    next = 0
    For j = i - 1 to 1 :
        if W_j ≥ W_i and L_j ≥ L_i and (H_i + S[j]) > next :
            next = H_i + S[j]
    S[i] = max( H_i, next)

result = S[1]
For i = 2 to n :
    if S[i] > result
        result = S[i]
result result
```

---

running time:

Sorting need $O(nlogn)$ time. The initialization uses $O(n)$ time. Therefore, whole iteration need $O(n^2)$ time.

The final time is $O(nlogn) + O(n) + O(n^2) = O(n^2)$

Extra:

This question is similar as maximum contiguous subsequence. Based on description, there are two subseqences which I need to consider:

- $x_1 - x_2 + x_3 - x_4 + ... \pm x_n$
- $x_2 - x_3 + x_4 - x_5 + ... \pm x_n$

Find both sums of maximum contiguous subsequence of above subsequences respectively. The greater one is result.

```
sum1[n]
sum1[1] ← x_1
For i = 2 to n :
    If i mod 2 == 0 :
        sum1[i] ← max(x_i, sum1[x_{i-1}] - x_i)
    Else
        sum1[i] ← max(x_i, sum1[x_{i-1}] + x_i)

sum2[n-1]
sum2[1] ← x_2
For i = 3 to n:
    If i mod 2 == 0 :
        sum2[i] ← max(x_i, sum2[x_{i-1}] + x_i)
    Else
        sum2[i] ← max(x_i, sum2[x_{i-1}] - x_i)

firstMax ← maximum value in sum1
secondMax ← maximum value in sum2
return max(firstMax, secondMax)
```

---

running time:

Traversing through two subsequences uses $O(n)$ time respectively. Find maximum value in two arrays uses $O(n)$ time respectively. Thus my running time is $O(n) + O(n) + O(n) + O(n)$ = $O(n)$.