Enlai Yii
2064210

Homework 3

Exercise 1

(1)

Let Max-Dirt(i, j) denote the path that cleans the max number of dirty cells in the m x n grid.

(2)

Max-Dirt(i, j) = max(Max-Dirt(i - 1, j), Max-Dirt(i, j - 1)) + floor[i][j]

Let floor be an array m x n matrix where floor[i][j] is 1 if there is dirt at cell (i, j). Otherwise 0 if there is no dirt at cell (i, j)

The base cases are:

Max-Dirt(0, 0) = floor[0][0]

Max-Dirt(i, j) = -∞ if i < 0 or j < 0

(3)

There are nm subproblems and the time to solve each problem, given the recursive formulation, will take constant time.

Running time = O(# of subproblems * time per problem) = O(nm * O(1)) = O(nm)

(4)

```
cache = {(0,0) : floor[0][0]}

function Max-Dirt(i, j):
    if i < 0 or j < 0:
        return -∞

    if (i, j) not in cache:
        cache[(i, j)] = max(Max-Dirt(i-1, j), Max-Dirt(i, j-1)) +
floor[i][j]

    return cache[(i, j)]
```

(5)

A backtracking algorithm where we can start at the bottom right corner in the m x n grid. We can work our way to the upper left cell. We can make 1 if the machine goes up, 0 if the machine goes to the left, into an array to keep track of the path. As the machine works its way to the start of the

grid, the machine will clean up dirt and mark it as 1 if it cleaned up dirt or 0 if nothing was cleaned in a separate variable, which is tied to the array path. Once the machine makes its way to the start, the maximum dirt cleaned will be found, and the tied path will output the path that achieves the maximum dirt cleaned.

Exercise 2

(a)

Let Min-Penalty(i, j) denote the minimum penalty from traveling from hotel i to hotel j

(b)

Min-Penalty(i, j) = min(min i < k < j (Min-Penalty(1, k) + Min-Penalty(k, j)), |300 - (aj - ai)|)

Min-Penalty from ai to an and substitute 0 and n must travel to some hotel ak

(c)

Running time = $O(n^3)$

(d)

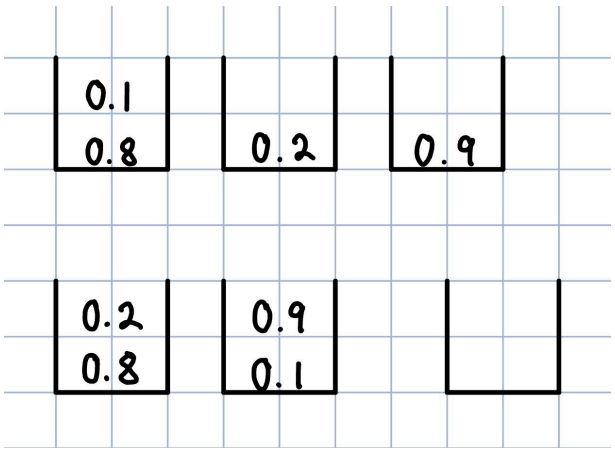
```
cache = {}

function Min-Penalty(i, j):
    if (i, j) not in cache:
        direct = |300 - (ai - aj)|
        indirect = min(min i<k<j (Min-Penalty(1,k) + Min-Penalty(k,j)),
|300 - (aj - ai)|)
        cache[(i, j)] = min(direct, indirect)

    return cache[(i, j)]
```

The way to find the optimal sequence of hotels is to store the information we obtain as k, then append the value to the array *hotels*, the array will store all of the stops

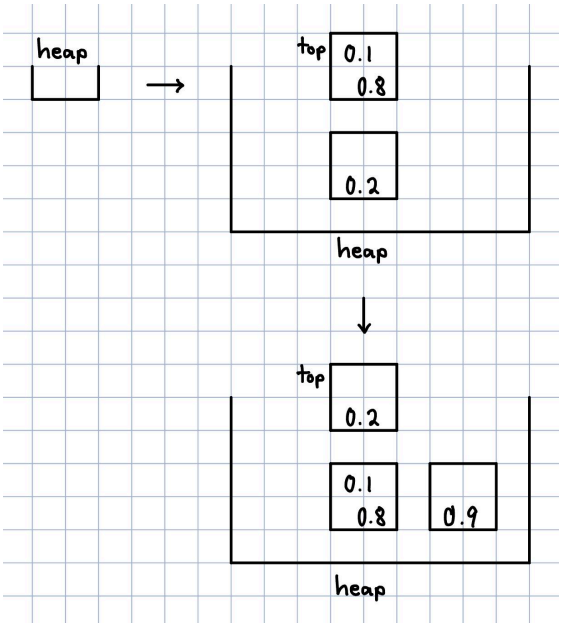
Exercise 3

From Khalid's Office Hour

(1)

Ex: 0.8 0.1 0.2 0.9		Greedy Solution Optimal Solution; which isn't found using greedy

(2)

0.8 0.1 0.2 0.9 n size		First, create a heap to hold all of the bins created. The top of the heap will be the bin that still has room (less than 1). Once 0.2 is next in the set, we have to create a new bin because if it is added to the bin (0.8, 0.1), then it will not satisfy the conditions and go over the limit of 1. So, we have to create a new bin, push the new bin into the heap and add the object (0.2) into the new bin. The top of the heap will be the new bin. Finally, 0.9 (the last object) will create a new bin since if it is added to the bin (0.2), it'll be over the limit. After the new bin is created and 0.9 is added, the bin (0.9) won't be the top of the heap because bin (0.2) has the most amount of space available.

```
bin = heap.top()                          O(1)

if this number fits inside the bin:       O(1)
    bin.push_back(number)                 O(1)

else                                      O(1)
    create a new bin                      O(n)
    heap.push(new bin)                    O(log n)
    bin.push_back(number)                 O(1)
```

The run time of the algorithm is O(n log n).

Exercise 4
(1)
The best possible way for 3 friends to buy all the items is to buy the most expensive items first, then move on towards the cheapest items until all of the items are bought.

	Items	Total
Friend 1	10 + 2(5) +3(2)	26
Friend 2	7 + 2(3)	13
Friend 3	6 + 2(2)	10
	Total Cost	49

The minimum cost needed to buy all of the items between 3 friends is 49.

(2)
Before starting the algorithm, we should sort the cost of all the n items in descending order and store the sorted costs into an array called prices (c1, c2, …, cn). For the algorithm, to calculate the optimal (minimum) cost, we must create a variable called OPCost and add the first k costs from prices to OPCost. Like the example above, we add the first 3 expensive costs: 10 + 7 + 6 into Total Cost (or OPCost for the algorithm) and we get 23. Next, we calculate the next k costs from prices and multiply each cost by 2 and so on (3, 4, …, n) until we are out of items to buy. As we finish the example, 2(5) + 2(3) + 2(2) for the next 3 costs into OPCost (which is now 43). Finally, the last item to add costs 2, so it's going to be 3(2) into OPCost and the final value of OPCost will be 49.

(3)

```
# Sort in descending order
prices[c1, c2, ..., cn]
prices.sort(reversed=True)                    O(n log n)

function Minimum-Cost(n, k):
    # OPCost is the total cost
    OPCost = 0
    # knum is the number of k costs
```

```
knum = 0
# mult is the multiple for each cost
mult = 1
# iterate through the whole list prices
for index = 0 . . . n-1:                                    O(n)
    # check if k costs is k
    if knum = k:                                            O(1)
        # increment mult by 1 and reset knum to 0
        mult += 1
        knum = 0
    # add the cost of each item and multiple by mult
    OPCost += mult*prices[index]                            O(1)
    # incrmenet knum
    knum += 1
# return the total cost
return OPCost
```

The running time for the implemented algorithm will be $O(n^2 \log n)$.

(4)
The given greedy algorithm will always output the optimal solution because if you try to add up all of the costs with any other strategy or algorithm, the total cost will never be lower than the lowest possible cost (as in OPCost).

Costs of items

C1	C2	C3	C4	C5	C6	C7
3	6	10	7	2	5	2

The OPCost from the algorithm will be:
After being sorted, the order based on the algorithm will be
OPCost = C3, C4, C2, C6, C1, C5, C7. This will give you a total of 49.

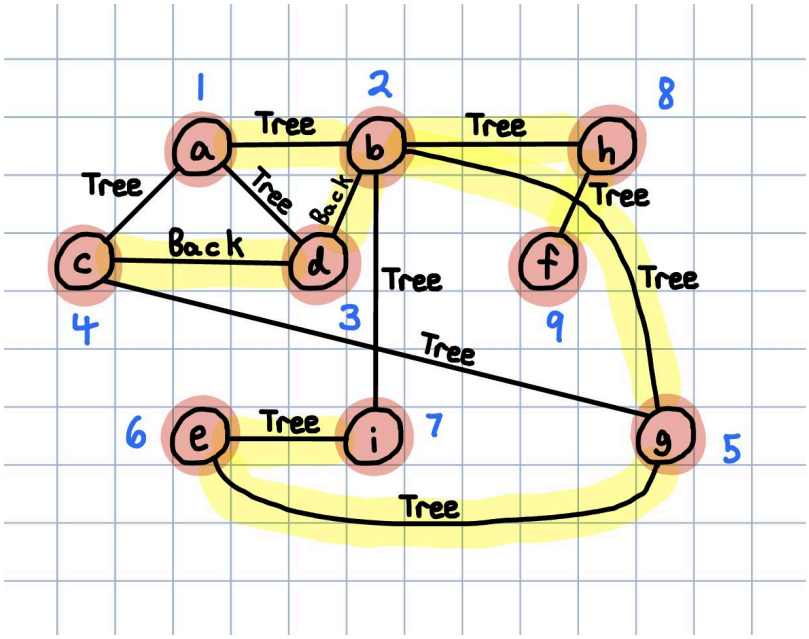
How about trying the same thing in a different order?
Cost = C3', C4', C7', C6', C1', C5', C2', and that total will be 54.

If we compare both total costs, OPCost < Cost. So, OPCost is always less than Cost.
C1 + C2 + C3 + C4 + C5 + C6 + C7 - (C1' + C2' + C3' + C4' + C5' + C6' + C7')
= (C6 - C6') + (C7 - C7') = -5 + 2 < 0

So, OPCost - Cost >= 0.

Exercise 5

(a)



(b)

The cut vertices of G:

b, h

(c)

The biconnected components of G:

b, h

a, b, c, d, e, g, i

h, i

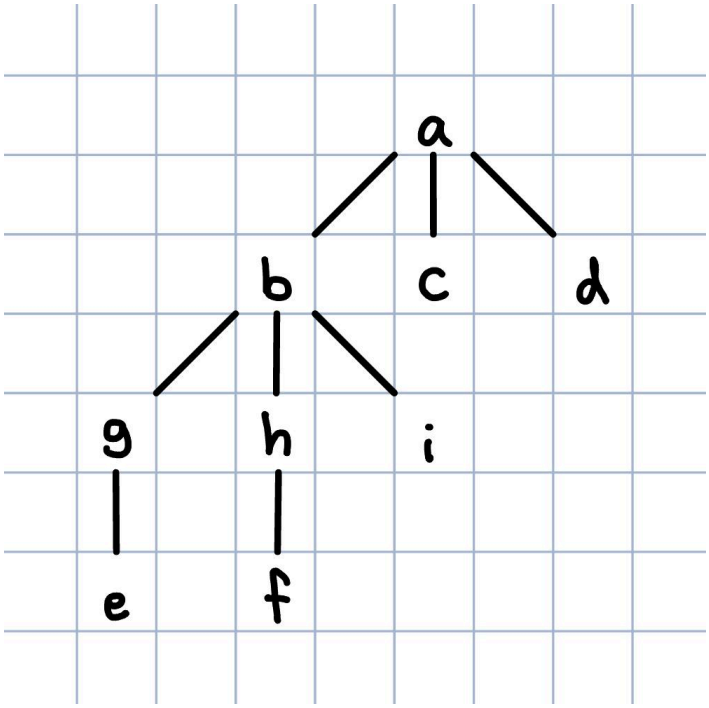
(d)

The cut edges of G:

b-h

h-f

(e)


a
b
c
d
g
h
i
e
f