NORTHWESTERN UNIVERSITY

EECS 336 – Homework 5

Weihan Chu

May 12, 2016

1 QUESTION 1

My algorithm is that the car goes as as far as it can then refill. Which means it will refill only when the car can not get to the next gas station.
For example: the car's gas can travel a miles far from now and second station is b miles far, and the third station is c miles far.
case1: if $a \geq b+c$, then the car will not refill at the second station .
case2: if $b+c > a > b$, then the car will refill at the second station.
this example illustrates my algorithm that how the car refill only when the car can not get to the next gas station.

```
problem1 (n) {
    now = m
    for (i = 2; i <= n; i++) {
        if(now < d[i] - d[i - 1]){
            now = m
            res.add(i - 1)}
        else
            now -= (d[i] - d[i - 1])}
    return res}
```

The time complexity of this greedy algorithm is $O(n)$, because it compare its current gas with the distance to make a decision whether it stops at next station. Since there are n stations, so the time complexity is $O(n)$. To prove this, we can see that there is only one layer circle in this algorithm, and this circle need to do n comparisons. Every comparison takes $O(1)$ time

complexity. So this algorithm takes $O(n)$ time complexity.
To prove its correctness, for example, there is 4 stations A,B,C,D, our car begins at A and follow the way to B,C,D. if the car has enough gas to get to C, but not enough gas to D. Then the car has two choices, it can stop at B to refill or it can stop at C to refill. Compare the two choices. If the car refills at B, then the car can travels m miles from B before it need to refill again. If the car refills at B, then the car can travels m miles from C before it need to refill again. So if the car refills at C, it can travels longer. Which means that refills at C is a better choice. So this can prove this algorithm is a optimal solution.

2 Question 2

I think that all three greedy algorithms are not optimal.
We assume that c_1 is the first knapsack's capacity, and c_2 is the second knapsack's capacity. n is the number of different items we have. $w = w_1, w_2, ..., w_n$ is the queue which contains the weight of the items. $p = p_1, p_2, ..., p_n$ is the queue which contains the value of the items.
(a):increasing order of weight:
For example, if we have $c_1 = 20, c_2 = 5, n = 2, w = [10, 20], p = [5, 100]$, if we use the greedy algorithm, we can get that we will put w_1 and 10 of w_2 into c_1, which means $x_1 = [1, 0.5], x_2 = [0, 0]$so we get the final total value is 5+50=55, but actually, we can make $x_1 = [0, 1], x_2 = [0.5, 0]$ so the total value is 102.5. So this greedy algorithm is not right.
(b): decreasing order of value:
For example, if we have $c_1 = 100, c_2 = 10$, $n = 2, w = [100, 10, 10, 10], p = [20, 15, 15, 15]$.
If we use the greedy algorithm of decreasing order of value, we assume that x_1 is the solution for first knapsack and x_2 is the solution for the second knapsack. We will find the that our solution is. $x = x_1 + x_2$. And $x_1 = [1, 0, 0, 0]$ and $x_2 = [0, 1, 0, 0]$, which means we will put the item with weight 100 into the first knapsack and weight 10 into the second knapsack. Then we get the total value is 35. But actually we can make $x_1 = [0, 1, 1, 1]$ and $x_2 = [0, 0, 0, 0]$,so the actual total value we can get is 45. So this greedy algorithm is not right.
(c): decreasing order of value to weight ratio:
For example,if we have $c_1 = 1, c_2 = 2, w = [1, 4], v = [2, 6]$, if we use the greedy algorithm, we can get that $c_1 = [0, 0], c_2 = [1, 0.25]$, so the total value is 2+0.25*6=3.5. But we can make $x_1 = [1, 0], x_2 = [0, 0.5]$, then the total value is 2+0.5*6=5, so this greedy algorithm is not right.

3 Question 3

First, there are several conditions considering the amount of $|C|$
case1: if $|C| \in 5n+1, n = 0,1,2....\infty$, then we are fine
case2: if $|C| \in 5n+2, n = 0,1,2....\infty$, then we need to add four new character a_1, a_2, a_3, a_4 to C with frequency 0
case3: if $|C| \in 5n+3, n = 0,1,2....\infty$, then we need to add four new character a_1, a_2, a_3 to C with frequency 0
case4: if $|C| \in 5n+4, n = 0,1,2....\infty$, then we need to add four new character a_1, a_2 to C with frequency 0
case5: if $|C| \in 5n+5, n = 0,1,2....\infty$, then we need to add four new character a_1 to C with frequency 0
Follow are the main steps:

```
$n = |C|$
$Q = C$
for $i = 1$ to $\left\lfloor \frac{n}{5} \right\rfloor$
do allocate a new node z
$z.l_0 = u = EXTRACT - MIN(q)$
$z.l_1 = v = EXTRACT - MIN(q)$
$z.l_2 = w = EXTRACT - MIN(q)$
$z.l_3 = x = EXTRACT - MIN(q)$
$z.l_4 = y = EXTRACT - MIN(q)$
$z.l_5 = z = EXTRACT - MIN(q)$
z.freq=u.freq+v.freq+w.freq+x.freq+y.freq+z.freq
INSERT(Q,z)
return EXTRACT-MIN(Q)
```

And when we encode, we will give the $z.l_0$ with symbol 0, $z.l_1$ with symbol 1, $z.l_2$ with symbol 2, $z.l_3$ with symbol 3, $z.l_4$ with symbol 4, $z.l_5$ with symbol 5. This method will ensure that no codeword would be the prefix of other codeword.
If $|C| = n$, every time through one iteration we remove 6 nodes from the queue and then add 1 node back. So we have 5 different cases to add different number of nodes in the beginning to ensure that we can get a single node at the last.
As for the time complexity, we implement this method with a heap. So every time we use the EXTRACT-MIN, it takes $O(lgn)$ time and insert is also $O(lgn)$. In this method, we make $6(n\left\lfloor 5 \right\rfloor)$ calls to the EXTRACT-MIN , and $(n\left\lfloor 5 \right\rfloor)$ calls to insert. So the total time complexity is $O(nlgn)$
To prove this algorithm is right, we show that this exhibits the greedy-choice and optimal-substructure.
For the greedy-choice, is u,v,w,x,y,z are 6 characters in C having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords for them have the same length and differ only in the last bit. If we modify to make another optimal prefix code that they are sibling leaves of the maximum depth in the new tree. If we can construct such a tree then the codewords for x and y will have the same length and differ only in the last bit.

Suppose that T does not represent an optimal prefix code for c and there exists an optimal tree T' such that $B(T') < B(T)$. T' has u,v,w,x,y,z as siblings.if T" is the Tree T' with common parent of u,v,w,x,y,z replaced by a leaves. we can get that B(T")=B(T), which contradicts the assumption that T' represents an optimal prefix code for C'. Thus T must represent an optimal prefix code for C.
So this algorithm is right.

4 Question 4

For this problem, we can sort set A and set B both in monotonically decreasing order or both in monotonically increasing order. This method is an optimal solution. To prove this ,we can consider a small part of the set. For example,i and j, where $i < j$, and $a_i > a_j, b_i > b_j$. We can get that $a_i^{b_i - b_j} \geq a_j^{b_i - b_j}$ because $a_i, a_j, b_i - b_j$ are both non-negative numbers. Then we multiply both sides by $a_i^{b_j} * a_j^{b_j}$, we can get that:

$$a_i^{b_i} * a_j^{b_j} > a_j^{b_i} * a_i^{b_j}$$

so if we sort them both in decreasing or increasing order, then we will get the answer.
Since the sorting is $O(nlgn)$, and the multiply is $O(n)$,so the final time complexity is $O(nlgn)$

5 Question 5

5.1 Part a

The algorithm I use in this problem is as follows:
a. if n=0, so this is easily can know that the answer is 0 coins.
b. if n>0, we just find the largest coin, for example a_1, whose value is less or equal to n. Then we mark this coin as one element in the optimal solution. then we use $n - a_1$ to recursive use this method to solve this problem until the value becomes 0.
First we need to prove that this problem has a optimal substructure. To prove that, suppose that we want to get the optimal solution to n cents. And we assume that there is a coin which is a cents is in this optimal solution. We also assume that the solution to this n cents problem is k coins. So we can get that the optimal solution for the n-a cents is k-1 coins. If there is the

solution for the n-a cents that we can have less than k-1 coins, then we can use less coins to get the n cents problem, which contradicts our solutions optimality. So this problem has a optimal substructure.
The greedy-choice property is that a global optimal solution can be arrived by making locally optimal greedy choice. We need to prove that this problem has this property. Assume that a is the largest coin value the $a <= n$, if we can prove that the optimal solution for this problem includes a, then we are done. There are four major cases:
1.if $1 \le n \le 5$, then a=1, and the optimal solution contains penny.
2.if $5 < n \le 10$, then a=5, if the optimal solution doesn't contain nickel. then we need to use five pennies to replace. So the coins total amount will increase. So the optimal solution contains the nickel
3.if $10 < n \le 25$, then a=10, if the optimal solution doesn't contain dime, so it only has nickels and pennies. Then there are nickels and pennies which will add up to 10 cents. But we can use a dime to replace and reduce the total amount of the coins. So the optimal solution contains the dime. 4.if $25 < n$, the condition is same as above.
Thus, we conclude that the optimal solution includes the greedy choice. And we can combine them to get the optimal solution to final optimal solution. So we can solve this problem by greedy.
For the time complexity, since there are only four kinds of coins and every time we will choose a coin and this takes $O(k)$, where k is the number of coins used in this solution. Since we only need to do constant number of this step. So the final time complexity is $O(1)$.

5.2 Part B

The algorithm works like this:

```
partb(n) {
    res[k]
    for (i = k, i >= 0; i--){
        if(n >= c^i){
            res[i] += n / c_i
            n = n % c^i
        }
    }
    return res
}
```

To prove the correctness, we consider that in the greedy algorithm the largest coins used is c^i and the number is m. If there exists another optimal solution which works better than this greedy algorithm. It uses coins of denominations of $c^0, c^1, ...c^{i-1}$, and a_j. So we have $\sum_{j=0}^{i-1} a_j c^j = n \ge c^i$. we then can get that $\sum_{j=0}^{i-1} a_j c^j \le \sum_{j=0}^{i-1}(c-1)c^j = (c-1)\frac{c^i-1}{c-1} = c^i$, which contradicts previous conclusion. So this algorithm is right.
The time complexity for greedy algorithm is $O(k)$. Because we have k kinds of coins. And we need $O(1)$ time to do with each kind. So the total complexity is $O(k)$

5.3 Part C

For example, we have three different kinds of coins: one cent, three cents and four cents(1,3,4). If $n = 6$ cents, the result of greedy solution is $(4,1,1)$, where in fact the optimal solution is $(3,3)$, so in this condition the greedy method doesn't get the optimal solution.

5.4 Part D

Since that we need to apply this algorithm into any k kinds of different coins, so we couldn't use greedy algorithm here. We need to use dynamic programming.
Suppose that the coins are $d_1, d_2, d_3......d_k$ and the minimum number of coins for the j cents is $c[j]$, one of the coins is penny. So we can get that:
case1: if $j \leq 0$ then $c[j] = 0$, this is the base case.
case2: if $j > 1$, then $c[j] = 1 + min_{(1 \leq i \leq k)}(c[j - d_i])$
this will create a table $c[1...n]$,if first find the optimal solution for c[1], then c[2], then c[3], until c[n]. every time it needs to do k times comparisons. So the time complexity is $O(nk)$ totally.