Homework2_432001358

February 1, 2022

0.1 22 SPRING CSCE 629 600: ANALYSIS OF ALGORITHMS - Homework 2

0.1.1 Name: Rohan Chaudhury

0.1.2 UIN: 432001358

0.1.3 Question (1) Textbook page 422, Exercise 16.1-3.

Not just any greedy approach to the activity-selection problem produces a maximum-size set of mutually compatible activities. Give an example to show that the approach of selecting the activity of least duration from among those that are compatible with previously selected activities does not work. Do the same for the approaches of always selecting the compatible activity that overlaps the fewest other remaining activities and always selecting the compatible remaining activity with the earliest start time.

0.1.4 Answer:

1. Let us consider the following example for the process of selecting the activity of the least duration from those compatible with activities which were previously selected:

We are considering 4 activities with start time, end time, and duration as shown below:

Activity number	Start time	End time	Duration
1	1	6	5
2	7	14	7
3	13	17	4
4	16	22	6

The activity with the least duration is Activity 3. So, selecting the activity of least duration would yield a result of selecting Activity 3 and Activity 1. So a total of 2 activities are selected in this case. Activity 2 and Acivity 4 are overlapping with Activity 3.

However, if we select activities based on their end times (whichever activity ends first) we would get an optimal solution of selecting a total of 3 activities (Activity 1, Activity 2, and Activity 4). Hence, selecting activities based on the least duration does not give the optimal solution to the activity-selection problem.

2. Let us consider the following example for the process of selecting the activity that overlaps the fewest from those compatible with activities which were previously selected:

We are considering 8 activities with start time, end time, and number of overlaps as shown below:

Activity number	Start time	End time	Number of overlaps
1	1	6	4
2	4	8	6
3	4	8	6
4	4	8	6
5	4	8	6
6	7	14	5
7	13	17	2
8	16	20	5
9	18	26	6
10	18	26	6
11	18	26	6
12	18	26	6
13	24	28	4

The activity with the least number of overlaps is Activity 7. So, selecting the activity with the least number of overlaps would yield a result of selecting Activity 7, Activity 1, and Activity 13. So a total of 3 activities are selected in this case. Activity 6 and Acivity 8 are overlapping with Activity 7.

However, if we select activities based on their end times (whichever activity ends first) we would get an optimal solution of selecting a total of 4 activities (Activity 1, Activity 6, Activity 8, and Activity 13). Hence, selecting activities with the least number of overlaps does not give the optimal solution to the activity-selection problem.

3. Let us consider the following example for the process of selecting the activity with the earliest start time from those compatible with activities which were previously selected:

We are considering 4 activities with start time and end time as shown below:

Activity number	Start time	End time
1	1	15
2	2	6
3	7	10
4	16	22

The activity with the earliest start time is Activity 1. So, selecting the activity with the earliest start time would yield a result of selecting Activity 1 and Activity 4. So a total of 2 activities are selected in this case. Activity 2 and Acivity 3 are overlapping with Activity 1.

However, if we select activities based on their end times (whichever activity ends first) we would get an optimal solution of selecting a total of 3 activities (Activity 2, Activity 3, and Activity 4). Hence, selecting activities with the earliest start time does not give the optimal solution to the activity-selection problem.

0.1.5 Question (2) Textbook page 446, Problem 16-1.

16-1 Coin changing: Consider the problem of making change for n cents using the fewest number of coins. Assume that each coin's value is an integer.

a. Describe a greedy algorithm to make change consisting of quarters, dimes, nickels, and pennies. Prove that your algorithm yields an optimal solution.

b. Suppose that the available coins are in the denominations that are powers of c, i.e., the denominations are $c^0, c^1, ...c^k$ for some integers c > 1 and $k >= 1$. Show that the greedy algorithm always yields an optimal solution.

c. Give a set of coin denominations for which the greedy algorithm does not yield an optimal solution. Your set should include a penny so that there is a solution for every value of n.

d. Give an O(nk)-time algorithm that makes change for any set of k different coin denominations, assuming that one of the coins is a penny.

0.1.6 Answer (a):

Main idea of the algorithm

The main idea of the algorithm is to use the highest denomination as many number of times to make the amount n (the change that we have to make) till the amount that is left is less than that highest denomination but greater than 0 (i.e., left amount= n-temp_num*highest denomination where temp_num is the integer quotient obtained when highest denomination divides n). Then we repeat this process using the second highest denomination on the amount left after the above first step, then using third highest denomination and so on using the rest of the denominations in a descending order till the amount that is left is reduced to 0. For every denomination we add the temp_num to another variable num_coins. This accumulated value num_coins gives us the optimal solution for the problem of making change for n cents using the fewest number of coins

Pseudocode for the greedy algorithm

```
1 #let us consider an array named denominations containing the denominations quarters(25), dimes(10), nickels(5), and pennies(1) sorted in descending order. We have to make change of n cents.
2 denominations=[25, 10, 5, 1]
3 # considering denominations array to be 0-indexed. We have to return num_coins
4 k=length(denominations), num_coins=0, temp_num=0
5 for i = 0 to k-1:
6    if (denominations[i]<=n):
7        temp_num=n/denominations[i]
8        n=n-temp_num * denominations[i]
```

```
9       num_coins=num_coins+temp_num
10 return num_coins
```

Proof of correctness:

In each step of the algorithm we are taking maximum number of multiples of the largest denomination which is lesser than the change amount left to make our total change. The amount that is left after each step is lesser than the largest denomination used in that step but greater than 0. So in each step we are greedily choosing the optimal number of coins for making the change using the largest denomination in that step. So, at the end the total number of coins used in all the steps is the minimum which gives us our optimal solution for the problem of making change for n cents using the fewest number of coins.

Time complexity

The algorithm uses a single for loop which runs k times (where k is the length of the denominations array, so k is the total number of denominations provided in the problem). All the other operations are constant time operations. The single for loop would run k number of times, hence the worst case time complexity would be $O(k)$.

0.1.7 Answer (b):

Given the denominations as $c^0, c^1, ...c^k$ for some integers c > 1 and $k >= 1$, the greedy algorithm would pick the highest denomination c^k and use it as many number of times to make the amount n (the change that we have to make) till the amount that is left is less than c^k but greater than 0. Then the algorithm would repeat this process for the rest of the denominations c^i in a descending order till the amount that is left is reduced to 0. If the number of coins for each denomination c^i be represented as a_0^i using the greedy solution then we can note that for every i<k, $a_0^i < c$ since if any a_0^i becomes greater than c then we can use c^{i+1} denomination once instead of using c^i denomination c times to make the change. This decreases the number of coins a_0^i by c and increases a_0^{i+1} by 1.

Now, let us assume that their is an optimal solution to this problem given by $[a_1^0, a_1^1, a_1^2, ..., a_1^k]$ where a_1^i is the number of coins of denomination c^i used to obtain the optimal solution. Now let us assume that for some $i < k$, we have $a_1^i > c$. If that is so, then there are are atleast c number of c^i denomination coins in this solution which if replaced by one c^{i+1} will reduce the number of c^i denomination coins from a_1^i to $a_1^i - c$ and increase the number of c^{i+1} denomination coins from a_1^{i+1} to $a_1^{i+1} + 1$. This reduces the number of coins required by c-1 amount and hence the assumed solution is not an optimal one. On the other hand, the strategy discussed to prove that this is not the optimal solution is the same one that we use in the greedy solution where for every i<k, $a_0^i < c$, since in each step of the algorithm we are taking maximum number of multiples of the largest denomination which is lesser than the change amount left to make our total change. Thus, for the assumed denominations, greedy algorithm always gives an optimal solution.

0.1.8 Answer (c):

Let us assume the following denominations: (1,4,6) required to make a change of 8. The greedy solution would result in choosing '6' once and '1' twice which is 3 coins in total whereas the optimal solution would be to choose '4' twice which is 2 coins in total. Hence for the above shown set of coin denominations, the greedy solution doesn't give an optimal solution.

0.1.9 Answer (d):

Main idea of the algorithm:

The main idea of the algorithm is to use dynamic programming and build the optimal solution in a bottom up fashion. We will consider an array ‘count’ of length n+1 where all the values of the array are initialized to a large number, say ‘infinity’. This array will store the optimal number of coins required to make the change j where j varies from 0 to n. Now we set count[0]=0 as number of coins required to make a change of 0 is 0. Then we vary i over k denominations and use each denomination to check whether it makes up change j (we now vary j from that denomination value to n) and update count[j] to count[j-denominations[i]] + 1 if it is lesser than count[j]. We use this for every denomination for all values of i from 1 to n which effectively stores the minimum value amongst all of them in count[j]. At the end we will return count[n] which is our required optimal solution.

Pseudocode for the dynamic programming algorithm:

```
1 #let us consider an array named denominations containing the given k different coin denominations. We have to make change of n cents.
2 denominations=[array containing given k different coin denominations]
3 # considering all arrays to be 0-indexed.
4 # Let us consider an array ‘count’ of length n+1 where all the values of the array are initialized to a large number, say ‘infinity’
5 count=[‘infinty’]*(n+1)
6 # this initializes count to [infinity, infinity, ... n+1 times]
7 # now we set count[0]=0 as number of coins required to make a change of 0 is 0
8 count[0]=0
4 k=length(denominations)
5 for i = 0 to k-1:
6     for j = denominations[i] to n
7         count[j]=minimum(count[j], count[j-denominations[i]]+1)
10 #since one of the denominations is a penny, we will always have a solution
11 return count[n]
```

Proof of correctness:

As we are varying i over all the possible denominations to check the optimal solution for each value of change for 0 to n, we are trying all the possible ways to make up change j where j varies from 0 to n (change[0]=0) and storing the minimum of them in count[j]. At the end, count[n] will store the optimum solution since we have tried all the possible ways to come up with change n and stored the minimum of it in count[n].

Time complexity:

The variable i in the for loop varies from 0 to k-1 and variable j in the for loop varies from denominations[i] to n. In the worst case j varies from 1 to n when denominations[i]=1. So, the worst case time complexity would be: $O(n * k)$