algorithm

CS5800: Algorithms — Spring '21 — Virgil Pavlu

Homework 4
Submit via Gradescope

Name: Shri Datta Madhira (NUID: 001557772)
Collaborators:

Instructions:

- Make sure to put your name on the first page. If you are using the LaTeX template we provided, then you can make sure it appears by filling in the `yourname` command.
- Please review the grading policy outlined in the course information page.
- You must also write down with whom you worked on the assignment. If this changes from problem to problem, then you should write down this information separately with each problem.
- Problem numbers (like Exercise 3.1-1) are corresponding to CLRS 3^{rd} edition. While the 2^{nd} edition has similar problems with similar numbers, the actual exercises and their solutions are different, so make sure you are using the 3^{rd} edition.

1. ***(15 points)*** *Exercise 16.2-3. Use induction to argue correctness.*

Solution: This is a 0-1 Knapsack problem, which means that we either take the item or we do not.

In this problem, they gave that increasing order of weight equals the decreasing order of value of the items. That means, the highest value item is the least weight.

As we are trying to put as much valuable stuff in the knapsack as possible, the best idea is to take the items with the highest value at each step until your bag is full or the remaining items cannot fit in the bag.

Algorithm 1: Algorithm for knapsack when sorted by increasing weight is the same as their order when sorted by decreasing value

```
Function KNAPSACK-SAME-ORDER(W, w, v):
    V = 0, i = 0
    w.sort()
    v.sort(reverse=True)
    While W > 0 OR i < v.length (or) w.length
        temp = W
        If w[i] < temp :
            temp -= w[i]
            V += (v[i]*w[i])
            w[i] = 0
        W = temp
        i += 1
    Return V
```

If there is an alternative solution that has an item that is more valuable than the one taken by the algorithm, then, we can replace that item with the item in our knapsack because, the item in the alternative solution will be lighter than the item we have. So, we will have more space to fit more items, or at least be as good as alternative solution, which contradicts the theory that the alternative solution is better than our greedy solution. This implies our algorithm returns optimal solutions.

2. ***(15 points)*** *Exercise 16.2-4.*

Solution:

1. Fill the water bottle at the start, say A.
2. Go until the farthest possible refuelling station S.
3. Make S the new starting point A.
4. Repeat until he reaches the destination with minimum number of refills.

Algorithm 2: Algorithm to determine which water stops professor Gecko should make.

```
Function WATER-REFILLING(waterStations, m):
    n = waterStations.length - 1
    currPos = 0,
    startPoint = waterStations[0],
    dest = waterStations[1],
    While currPos ≤ n
        tempPos = currPos
        While (tempPos ≤ n) AND (waterStations[tempPos+1]-waterStations[tempPos] ≤ m)
            tempPos = tempPos + 1
        If tempPos ≤ n :
            numRefills += 1
            currPos = tempPos
    Return numRefills
```

This strategy yields the optimal solution because, we are making sure we stop at the farthest possible refilling station, which means that, we make a stop only when it is absolutely necessary. This strategy takes the professor to the destination with the minimum amount of stops made.

The above algorithm runs in linear time. We are iterating through the entire array. So, the time taken for the algorithm to complete is $\Theta(waterStations.length)$.

3. ***(15 points)*** *Exercise 16.2-5.*

Solution: The algorithm to find a unit-length closed intervals that contain all of the given points.

1. Sort the items in the set $x_1, x_2, \cdots, x_n$ in the way they appear on the real line. Say this set is called 'S'.
2. Take the interval $[S[1], S[1]+1]$ which is of unit length.
3. See what all elements of the set 'S' will come under that interval and store the interval in a separate output set/array.
4. The next interval will be from $[S[i], S[i]+1]$, considering the first interval covered all elements until S[i-1].
5. If the length of the length(S) is still greater than 0, then repeat the above step.

Algorithm 3: Algorithm to find the unit-length closed intervals.

```
Function UNIT-LENGTH-INTERVALS(S):
    I = set()
    S.sort()
    S = S - [S[1], S[1]+1]
    I = I + [S[1], S[1]+1]
    Return I
```

The question asked for a set of unit-length closed intervals of all the elements in the set 'S'.

In the algorithm, we are making a greedy choice and taking the value in the first index position in the set and checking what values in the set that will come under unit-length of it (S[1]+1). We don't need to worry about values less than the first index because we already sorted the points.

Since, in every sub-problem we have a fixed greedy choice on where the leftmost value is, this is optimal for every sub-problem. As it is optimal for all sub-problems, it is optimal for the entire problem.

4. ***(20 points)*** *Problem 16-1, (a), (b) and (c).*

Solution:

(a)

Algorithm 4: Greedy Algorithm for Coin Change

```
Function GREEDY-COIN-CHANGE(n):
    count-q, count-d, count-n, count-p = 0
    temp = n
    While temp > 0
        If temp ≥ 25 :
            count-q += floor(temp/25)
            temp = temp mod 25
        If temp ≥ 10 :
            count-d += floor(temp/10)
            temp = temp mod 10
        If temp ≥ 5 :
            count-n += floor(temp/5)
            temp = temp mod 5
        If temp ≥ 1 :
            count-p += floor(temp/1)
            temp = temp mod 1
    Return count-q, count-d, count-n, count-p
```

The algorithm can be run in constant time, and yields optimal solution every time. Say we have n = 11. We have two possibilities - 2 nickels, 1 penny or 1 dime, 1 penny.

Now, if we see for n = 11, according to algorithm, we first go to the if condition that has dime because $11 \geq 10$. We then have 1 cent. We go to the if condition that has penny. We now have 1 dime and 1 penny. As temp = 0, we will exit the loop and return this solution.

Every other n $\leq$ or $\geq$ 11 has a similar kind of argument, given that n > 0, that proves this algorithm is optimal (return as little coin change as possible).

(b) The same algorithm as (a) can be used even if the denominations are of the form c^k. The same thinking of taking the highest possible denomination to begin the coin change is applicable here too.

So, for the first iteration, we will do,

$$count - k = count - k + floor(\frac{n}{c^k})$$

$$n = n \ mod \ 25$$

For the remaining we will do,

$$count - i = count - i + floor(\frac{n \ mod \ 25}{c^i})$$

where i = $1, 2, \cdots, k-1$.

The only thing to think about is implementing the algorithm without too many if conditions as we have 'k' denominations instead of only four.

(c) A set without any one of the denominations would be a set that makes the greedy algorithm not return an optimal solution for a value.

If we take the example in (a), n = 11 without dime denomination, we would have to return 2 nickels, 1 penny; while the optimal solution would be to return 1 dime, 1 penny. Another example would be n = 30, if there are no nickels, we have to return 1 quarter, 5 pennies; which is not the optimal solution, because the optimal will be 3 dimes.

5. ***(15 points)*** *Exercise 15.4-5. Hint: try to solve this problem using a greedy approach -it may not work; if it doesn't, try an algorithm that starts from the back of the given sequence.*

Solution:

Algorithm 5: Algorithm to find the longest increasing sub-sequence

```
Function LONGEST-SUB-SEQUENCE(array,index):
    If index ≥ len(array) :
        Return 1
    count = []
    For i = index+1 to array.length
        If array[index] < array[i] :
            count.append(1+LONGEST-SUB-SEQUENCE(array,i))
    If !count.isEmpty() :
        Return max(count)-1
    Else
        Return 1
```

In the algorithm above, we check whether the index given to the function is less than the number beside it in the array. If it is, then we will recursively call the same function with the index as currIndex+1. This will go until the array is completed and returns values from there.

The number of recursive calls is basically the length of the longest monotonically increasing sub-sequence.

This algorithm works, but if you think it is wrong, please ignore it.
I want to attempt it with Dynamic Programming assignment also. Thank You.