CS5800: Algorithms — Spring '21 — Virgil Pavlu

Homework 4
Submit via Gradescope

Name: Xuran Feng
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:
Although this is a 0-1 knapsack problem, greedy algorithm can still be applied. The algorithm simply pick the item with largest value and minimum weight from all available items and put it into the knapsack, the loop continues when either the knapsack is full or there is no more available item. The final knapsack is the optimal solution.

```
GREEDY-FOR-0/1-VARIANT
let current-load be 0, total-value be 0 and knapsack max load to be W
let value[] contains value of each item from 1...k
let weight[] contains weight of each item from 1...k
Sort value[] in descending order and weight[] in ascending order
for i=1...k
    current-load = current-load+weight[i]
    if(current-load>W)
        end for loop
    total-value = total-value+value[i]
return total-value
```

Prove correctness:
Base case happens when there are no available items and this is trivially valid. Induction hypothesis assumes we already deal with $n-1$ items and now the knapsack has the optimal solution. Since everytime we choose the item with greatest value and smallest weight (most valuable) from rest of items, greedy requires the nth item must be next optimal candidate that will be picked and put in the knapsack. After choosing the nth item, we still get a optimal solution for dealing with n items. The loop continues and this invariant that everytime we choose the most valuable item is maintained all the time.

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

Solution:
The greedy idea can be applied. The professor will maximize the distance within m miles where he can still find a place to refill his water and repeat this until he reaches destination. For example, the professor will skate as far as possible within m mile from the starting point, find a water stop $W1$ and refill his water, then he will skate as far as possible within m mile from this water stop $W1$, find a water stop $W2$ and refill his water, then he picks up from $W2$... until he reaches the destination. To prove this is correct, we can prove by using argument exchange technique. Let O be an optimal set of water stops $\{o_1, o_2, o_3...\}$ and our solution set is S $\{s_1, s_2, s_3...\}$. Because greedy algorithm allows us to choose the furthest distance from starting point, if$o_1 < s_1$, then we replace o_1 with s_1, o_2 with s_2... because there will be less journey distance to be covered. By replacing O with S, we can get an at-least-good solution. The total running time $\theta(klogk)$ is dominated by sorting the k given water stops in ascending order since pseudocode below shows there is only one discrete while loop.

```
TRIP-PLAN
let distance be the total miles the professor skates
```

let start-point be 0
let A[] be the distance of all water stops relative to start-point
Sort A[]
let Set be the record of all candidate water stops
while(distance<whole-trip-distance)
    old-start-point = start-point
    start-point = start-point+MAX of A[] whose elements are <= m miles
    add start-point to Set
    update A[] relative to start-point
    distance = distance+start-point − old-start-point
return Set

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

Solution:
Start from the leftmost point, find the furthest point that can be covered by unit-length, update the result set; then find the neareat point that is not covered, start from this point and find the furthest point again...repeat this process until there is no more points that are not covered.
FIND-MINIMUM-SET
let A[] be the given points
Sort A[]
let start-point be 0
let Set be the record of all unit-length
while(true)
    index = index of A whose cell value is the furthest point that gets covered from start-point
    add start-point to Set
    if index>= A.length
        end while loop
    start-point = A[index+1]
return Set

Prove correctness:
The argument-exchange + contradiction technique can be applied here. If the greedy algorithm produces a solution set $S = \{s_1, s_2, ..., s_k\}$ and the optimal solution set is $O = \{o_1, o_2, ..., o_m\}$ where all elements are not all same as those in S. Since greedy algorithm always try to cover the furthest point, if o_1 is less than s_1, it can always be replaced by s_1 to gain more because there will be less points left to be covered, by using induction, we can find that all elements in O can finally be replaced by greedy algorithm because greedy always achieves current step optimality.

4. ***(15 points)*** *Exercise 16.2-6.*

Solution:
Can combine with quickselect algorithm to find median quality. First, divide value by its amount to get quality for all items. Next use quickselect to find median quality, compute the sum S1 of all items whose quality is higher than this median. If S1 is greater than knapsack load W, then do recursion on items of upper half with new knapsack load W; else if [S1+item weight equals to median] is greater than knapsack load, then find correct portion of median item plus upper half items

will be returned as answer; else, do recursion of lower half with new knapsack load 'W-S1-item weight equals to median'. The recursion equation is $T(n) = T(\frac{n}{2}) + O(n)$ because quickselect, sum and comparison all have $O(n)$ time complexity. According to master theorem, the running time of T(n) is $O(n)$.

```
let Set store knapsack results
let A[] store the quality of each item
FRACTIONAL-KNAPSACK-On(A[begin:end],W)
median=QUICK-SELECT(A[begin:end], rank median)
S1=0
for i=begin to i=end
    if(A[i]>A[median])
        S1 = S1+weight of i
S2 = S1+weight of item whose quality equals to A[median]
if(S1>W)
    FRACTIONAL-KNAPSACK-On(A[median+1:end],W)
else if(S2>W)
    add all upper half and correct proportion of median item to Set
else FRACTIONAL-KNAPSACK-On(A[begin:median-1],W-S2)
return Set

QUICK-SELECT(A[begin:end], rank r)
pivot=PARTITION(A)
if(pivot==r)
    return pivot
else if(pivot<r)
    QUICK-SELECT(A[begin:pivot-1],r)
else
    QUICK-SELECT(A[pivot+1:end], r-pivot)
```

5. ***(15 points)*** *Exercise 16.3-3 (Extra Credit). First prove by induction that* $\sum_{i=0}^{n-1} F(i) = F(n+1) - 1$

Solution:

6. ***(20 points)*** *Problem 16-1, (a), (b) and (c).*

Solution:

(a)

The greedy algorithm always picks the largest denomination coin within rest cents; then repeat this step until rest cents drops to zero.

```
COIN-CHANGE
let counter be 0
while(n>0)
    max-denomination=penny
    if n>= quarter
        max-denomination=quarter
```

```
    else if n>= dime
        max-denomination=dime
    else if n>= nickel
        max-denomination=nickel
    n = n−max-denomination
    counter = counter+1
return counter
```

Prove correctness:
The denomination set is {1,5,10,25}. We can use argument exchange technique to prove. Suppose our greedy solution set is $S = \{s_1, s_2, s_3, s_4\}$ and the optimal solution is $O = \{o_1, o_2, o_3, o_4\}$ that $o_i = s_i$ is not valid for all i. If taking out one 25, we must add at least two 10s and one 5 to make up the total amount difference; if taking out one 10, we must add at least two 5s; if taking out one 5, we must add at least five 1s. This means we can always adjust some o_i to create a better solution set, so O definitely is not an optimal solution and this is a contradiction. Since greedy solution is the final version of above adjusting process, we can show the correctness of greedy.
(b)
We can prove this by using argument exchange +contradiction technique. The denomination set now is $\{c^0, c^1, ..., c^k\}$, we assume our greedy solution set is S which contains $\{s_0, s_1, ..., s_k\}$ and s_i denotes the number of ith denomination coin; the optimal solution set is O with$\{o_0, o_1, ..., o_m\}$. m should be less or equal than k and we first prove m must equal to k. To prove m must equal to k, we can replace s_i with o_i since $o_i <= s_i$ by using induction poof. After replacing, if $k > m$, S becomes $\{o_0, o_1, ..., o_m, ..., s_k\}$, but the loop should stop when o_m is dealt with because $\{o_0, o_1, ..., o_m\}$ is already a solution. Since S still contains $\{s_{m+1}, ..., s_k\}$, there is a contradiction and thus $k = m$. Now we have $k = m$, we need to prove $o_i = s_i$.
Since S and O have same numbe of items but different set elements, then S and O must have at least two different elements where the differences can offset. For example, if we take out d from s_2 to make $s_2 = o_2$, we add at least $cd \cdot\ c^1$ to offset the $d \cdot c^2$ amount deduction. This means there is only O that all $o_i = s_i$ since $cd > d$ will definitely increase total number of coins. So S, the greedy algorithm is the optimal solution.
(c)
denomination set = {1,2,8,10}. For example, if $n = 16$ and according to greedy, one 10 and three 2s will be picked, however two 8s is the optimal solution. Greedy now doesn't yield an optimal solution.

7. *(15 points)* *Exercise 15.4-5. Hint: try to solve this problem using a greedy approach -it may not work; if it doesn't work, it means you must use DP and you can leave it for HW5.*

Solution:
Greedy+Binary Search algorithm can ensure a $O(nlog)$ time complexity, but need another $O(n)$ space to store temporary results; Dynamic programming can ensure a $O(n^2)$ time complexity and $O(n)$ space complexity to store the subproblem answers.

GREEDY-LIS
let A[] be the given n inputs
let List store temporary result that all elements will be strictly increasing

```
List.add(first index)
let Prev[] store the index which the last element of List gets transfered from
for i=2...i=n
    position=BINARY-SEARCH-POSTION(List, A[i])
    if position> List.length
        insert i at the end of List
    else
        update i into the postion of List
    Prev[i]=the second last element of List if it exists
//roll back from last element of List to adjust
length = List.length
prev-index = second last element of List
while(length>0)
    List.set(length) = prev-index
    prev-index = Prev[prev-index]
    length = length-1
return elements of A[] corresponding to List

BINARY-SEARCH-POSITION(List, element)
let left be 1, let right be List length
while(left<right)
    middle = (left+right)/2
    if List.get(middle) < element
        right=middle
    else
        left=middle+1
return left
```

Analysis:
The intuition is smaller element index should always be added to the List, so a strictly increasing List can be updated to have at least same size or larger size. However, elements in List are probably not in the same relative order as in A[] because everytime we update an element instead of inserting an element at the end of List is to ensure the List has at least same size as before, while inserting an element is to actually enlarge the List size. For example, A=[20,50,30,10,40], List = {}. We begin by inserting 20 into the List, next we find 50 is larger than last element of List A then 50 will be added to List, List={1,2}; next we find 30 is less than the last element 50, after doing binary search(List is ordered), 50 will be replaced and List={1,3}; next we find 10 is less than the last element 30, after doing binary search, 20 will be replaced by 10 and List = {4,3}, although the relative order gets reversed, the size of List remains 2; the last element 40 is larger than 30 and will be added into List = {4,3,5}. Now we get the correct size, we need to adjust order so we can get a correct subsequence by rolling back List according to Prev[].