CS5800: Algorithms — Spring '21 — Virgil Pavlu

Homework 4
Submit via Gradescope

Name: Zerun Tian

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.*

Suppose that in a 0-1 knapsack problem, the order of the items when sorted by increasing weight is the same as their order when sorted by decreasing value. Give an efficient algorithm to find an optimal solution to this variant of the knapsack problem, and argue that your algorithm is correct.

Solution:

In this variant of the knapsack problem, we can sort the items by weight in increasing order after which they are in decreasing order by value. After this preprocessing, we always pick the first (lightest) item at each iteration until our knapsack is full or its remaining capacity is too small to hold any other item.

```
1:  function SolveKnapsackVariant(items, W)      ▷ W indicates max capacity of the knapsack
2:      items.sort(by: weight)                   ▷ heapsort the items by increasing weight in-place
3:      total = 0                                                          ▷ accumulated value
4:      selected = []
5:      i = 0
6:      while items[i].weight ≤ W do
7:          W = W − items[i].weight
8:          selected.append(items[i])
9:          total = total + items[i].value
10:         i = i + 1
11:     return total, selected
```

We will show that at every step, the total value of the *selected* array is optimal.

- Base case: before the while loop, we set the *total* to 0 which is optimal to a subproblem where no item is processed yet. At the first iteration of the loop, if the first item weighs more than the maximum capacity, W, it will not be included in the knapsack and the loop ends. Trivially, this is optimal. If it weighs less than W, it will be included in the knapsack. This is optimal too because if we were to replace it with any other item, we would end up with a lower total value and a reduced available capacity.

- Induction hypothesis: assume that after k greedy steps (k iterations of the loop), the current *selected* is part of the optimal solution.

- Induction step: at the greedy step $k+1$, if the item at $k+1$ weighs more than the capacity available W, the loop terminates. Since nothing is added to our knapsack at $k+1$, and we assumed the solution at k is optimal, the solution at $k+1$ is optimal. On the other hand, if the $k+1$ item weighs less than W, it will be included to our knapsack. This is optimal too because none of the remaining items will reduce as less capacity while gaining as much value as the $k+1$ item will do. With the first k selections being optimal, the solution at $k+1$ is optimal.

In the end, if all items can fit in the knapsack, the solution is optimal. On the other hand, when the knapsack is full, or the remaining capacity cannot hold any remaining item, the solution that contains all previous items is optimal. It is not worthwhile to replace any previous items to fit an item that weighs more and values less.

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

Give an efficient method by which he can determine which water stops he should make. Prove that your strategy yields an optimal solution, and give its running time.

Solution:

To minimize the number of water stops, professor Gekko should skate as far as possible before he has to refill water within m miles. The first stop he should make is the furtherest water station within m miles from the starting point. After refilling, he can regard the water station as the new starting point, and repeat the same process until he reaches destination. This greedy approach can be proved optimal using the greedy stays ahead argument.

Suppose the greedy approach gives us a solution that SOL = $\{a_1, a_2, a_3, ..., a_k\}$ in order in which they are added. And, there exists an optimal solution OPT = $\{b_1, b_2, b_3, ..., b_j\}$, arranged by their distances from the starting position. We design a function d that gives us the distance of a water station from the starting point. We compare SOL with OPT and show that for $i \leq k$, $d(a_i) \geq d(b_i)$ using induction.

- Base case: Water station a_1 is the furtherest within m miles of the staring point. Station b_1 has to be closer to the staring point or simply is a_1, otherwise professor would run out of water. Thus, $d(a_1) \geq d(b_1)$.

- Induction hypothesis: assume it is true that $d(a_t) \geq d(b_t)$ for the t-th stop.

- Induction step: after a_t, greedy could have picked b_{t+1} because it is within m miles from a_t, which is implied by our hypothesis $d(b_t) \leq d(a_t)$ and that b_{t+1} is within m miles from b_t. However, greedy picked a_{t+1}. This implies that $d(a_{t+1}) \geq d(b_{t+1})$.

By induction, we know that for $i \leq k$, $d(a_i) \geq d(b_i)$. Now, let's say that $k > j$, meaning the number of stops of the greedy solution is greater than that of the optimal solution. At station a_{k-1}, the destination is still more than m miles away. Suppose the optimal solution has $j = k - 1$ stops. Since we know that $d(a_{k-1}) \geq d(b_{k-1})$, had the professor lastly stopped at b_{k-1}, he definitely would have run out of water on his way to destination. In this case, the optimal solution is invalid, which presents a contradiction. Then, we know that k has to be less than or equal to m. Therefore, the greedy solution is optimal.

The running time of this greedy method is $O(n)$ where n is the number of water stops. This is because greedy finds the solution simply by visiting each water stop from the starting location to the destination. Specifically, it keeps track of the distance between the currently visiting stop and the last stop in the current solution while maintaining a pointer to the previous stop. Once the distance exceeds m, it just adds the previous water stop to the solution, and starts anew from there.

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

Describe an efficient algorithm that, given a set $\{x_1, x_2, ..., x_n\}$ of points on the real line, determines the smallest set of unit-length closed intervals that contains all of the given points. Argue that your algorithm is correct.

Solution:

```
1: function FindMinIntervals(X)          ▷ X represents the set of points {x_1, x_2, ..., x_n}
2:     X.sort()                          ▷ heapsort the points in-place in ascending order
3:     intervals = [[X[0], X[0] + 1]]    ▷ the first interval starts from the left-most point
4:     for x in X do
5:         end = intervals[intervals.length − 1][1]
6:         if x > end then               ▷ when x is not covered by any existing interval
7:             intervals.push([x, x + 1])
8:     return intervals
```

First, we want to sort the set of points to enable our greedy strategy. The first interval is selected to start from the left-most point. Then, we ignore the points that can be covered by an existing interval. In the meantime, we are selecting the left most point that has not been covered by any interval to be the starting point of a new interval. The above pseudocode assumes 0-indexing.

Now, let's prove this works by the greedy stays ahead argument. Suppose the greedy solution yields a SOL = $\{a_1, a_2, a_3, ..., a_k\}$ in order in which the intervals are added. And, there exists an optimal solution that OPT = $\{b_1, b_2, b_3, ..., b_m\}$, sorted by the upper bounds of intervals.

We create a function l such that $l(a_i)$ or $l(b_i)$ returns the lower bound of each interval, and a function u that returns the upper bound. Now, let's show that for $i \leq k$, $u(a_i) \geq u(b_i)$ using induction.

- Base case: the interval a_1 begins at the left-most point and ends at a unit-length away from where it begins. To cover the left-most point, the first optimal interval, b_1, has to start no more than one unit-length to the left of the point. It is trivial to see that $u(a_1) \geq u(b_1)$.

- Induction hypothesis: assume it is true that $u(a_t) \geq u(b_t)$ for the t-th interval.

- Induction step: by our hypothesis, we know that the upper bound of b_t is less than or equal to that of a_t. If there are points between $u(b_t)$ (exclusion) and $l(a_{t+1})$ (exclusion) for which b_{t+1} has to cover, it has to start from a position that is less than $l(a_{t+1})$. Thus, $u(b_{t+1}) < u(a_{t+1})$. On the other hand, if there is no point between $u(b_t)$ and $l(a_{t+1})$, $l(b_{t+1)}$ cannot be greater than $l(a_{t+1})$ because we know $l(a_{t+1})$ corresponds to a point that b_{t+1} has to cover. Thus, $u(a_{t+1}) \geq u(b_{t+1})$.

Let's say that OPT is indeed a smaller set i.e $k > m$. The greedy solution did not stop after adding interval a_{k-1} because there are still points to be covered after $u(a_{k-1})$. Let's set $m = k - 1$, then $u(a_{k-1}) \geq u(b_{k-1})$. There are still points after $u(a_{k-1})$, which is after $u(b_{k-1})$, so the optimal solution did not cover all the points. When $k > m$, OPT is not a solution. Therefore, we know that $k \leq m$, which implies the optimality of the greedy solution.

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

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.

Solution:

```
1:  function MakeChange(n, D)          ▷ D represents available denominations
2:      D.sort(reversed=True)          ▷ sort D in descending order
3:      coins = []
4:      for d in D do
5:          while n ≥ d do
6:              coins.push(d)
7:              n = n − d
8:          if n == 0 then
9:              break
10:     return coins
```

We first sort the denominations in descending order. We try to make changes using the largest denomination as many times as possible until we have to pick a smaller denomination. Repeat this procedure until n becomes 0.

We prove its optimality using an exchange argument. Suppose there is an optimal solution OPT=$\{b_1, b_2, b_3, ..., b_m\}$. Counting occurrences based on denominations, we find c_1 number of quarters, c_2 number of dimes, and so on so forth. We use $\{c_1, c_2, c_3, c_4\}$ to denote the counts of all denominations in the optimal solution. Let's say that the optimal solution does not include the greedy choice. One such case might be c_1 is one less than what it could have been when the greedy choice is made. We can build a new solution by adding a quarter and getting ride of 2 dimes and 1 nickel. The total value is still the same but of different counts, $\{c_1 + 1, c_2 - 2, c_3 - 1, c4\}$. We just built a solution that used 2 less coins than the optimal solution. This contradiction shows the greedy algorithm has to be optimal.

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

Solution:

The greedy algorithm designed in (a) can also be applied here to yield an optimal solution. Let's denote the count of each denomination using q_i, i.e. for denominations $c^0, c^1, \cdots, c^k$, the corresponding counts are $q_0, q_1, \cdots, q_k$. By greedily selecting the largest denomination as many times as possible, q_i can be computed using the formula $\lfloor (n \bmod c^{i+1})/c^i \rfloor$. The formula is as such because the c^i coin will only try to make up the reminder after the c^{i+1} coin is used by the design of our algorithm. For the largest denomination c^k, $q_k = \lfloor (n/c^k) \rfloor$.

Let's analyze the counts. From above, we know that $q_i < c$ for $i < k$ because $\lfloor (n \bmod c^{i+1})/c^i \rfloor$ is at most $\lfloor (c^{i+1} - 1)/c^i \rfloor = \lfloor c - 1/c^i \rfloor < c$. For q_k, it is not bounded by a particular value so long as $q_k \cdot c^k \leq n$.

Now, we are ready to show the optimality of the greedy algorithm using an exchange argument. Suppose there is an optimal solution that does not follow the greedy choice by having a q_i that is more than or equal to c for some $i < k$. We shall build a new solution that have c less coins of the denomination c^i and 1 more coin of the denomination c^{i+1} because $c^i \cdot c = c^{i+1} \cdot 1$. Since $c > 1$, we have that $1 - c < 0$, which means the total number of coins in the new solution is smaller than that of the optimal solution. Violating the greedy policy leads to an invalid optimal solution. Therefore, the greedy solution is optimal.

(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.

Solution:

Let's say the coin denominations are $\{1, 7, 10\}$. To make change for 14, the greedy approach yields $[10, 1, 1, 1, 1]$. However, the optimal solution oughts to be $[7, 7]$.

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:

I will try to solve this problem using DP in the next assignment.