Name: Kevina Wong
ID: 109179049

CSCI 3104, Algorithms **Profs. Chen & Grochow**
Problem Set 5 – Due Thurs Feb 20 11:55pm **Spring 2020, CU-Boulder**

Advice 1: For every problem in this class, you must justify your answer: show how you arrived at it and why it is correct. If there are assumptions you need to make along the way, state those clearly.

Advice 2: Informal reasoning is typically insufficient for full credit. Instead, write a logical argument, in the style of a mathematical proof.

Instructions for submitting your solutions:

- All submissions must be easily legible.
- You should submit your work through the **class Canvas page** only.
- You may not need a full page for your solutions; pagebreaks are there to help Gradescope automatically find where each problem is. Even if you do not attempt every problem, please allot at least as many pages per problem (or subproblem) as are allotted in this template.

Quicklinks: 1a 1b 2a 2b 3a 3b 3c

---

1. *In this question we consider the change-making problem (as covered in recitation), of making change for n cents using the smallest number of coins. Suppose we have coins with denominations of $v_1 > v_2 > \cdots > v_r$ for r coins types, where each coin's value v_i is a positive integer. Your goal is to obtain a set of counts $\{d_i\}$, one for each coin type, such that $\sum_{i=1}^{r} d_i = k$ and where k is minimized, and such that the sum of the values $\sum_{i=1}^{r} d_i v_i = n$.*

   (a) *A greedy algorithm for making change is the **cashier's algorithm**. Consider the following pseudocode—meant to implement the cashier's algorithm—where n is the amount of money to make change for and v is a vector of the coin denominations:*

```
change(n,v,r) :
   d[1 .. r] = 1       // initial histogram of coin types in solution
   while n > 0 {
      k = r
      while ( k > 0 and v[k] > n ) { k++ }
      if k==0 { return 'no solution' }
      else { n = n - v[k] }
   }
   return d
```

Name: Kevina Wong
ID: 109179049

This code has bugs. Identify the bugs and explain why each would cause the algorithm to fail.

Answer:
Note that for each issue, I am assuming that the issue(s) prior were fixed as explained.

- **Issue 1:** The list, d, that holds the number of times we use each coin is initialized as a list filled with ones. This would cause the algorithm to fail because it would return one more occurrence of each coin than it is supposed to. To fix this bug, d could be initialized as a set of all zeros, not ones.
- **Issue 2:** The conditions of the first while loop on line 2 of the pesudocode would cause the algorithm to fail. This is because it is checking to see if $n > 0$, but it should be checking if n is greater than the smallest coin value. This is because if n was a value that was larger than zero, but smaller than the smallest coin value in v, it would cause an infinite loop to occur. Therefore, we can fix this bug by changing the while loop condition to $n >= v[r]$.
- **Issue 3:** The variable k is initialized as r. However, this would cause the algorithm to fail because the algorithm would start checking if it could use the smallest coin value first rather than the largest coin value. This would, then, lead to the algorithm outputting a non-optimal solution. ie.) if we had v[100,50,25,10,5,1] (Dollar..Penny), the algorithm would return n amount of pennies. To fix this bug, k should be initialized as 1, so the algorithm checks if larger coin values can be used first.
- **Issue 4:** The conditions of the second while loop are incorrect. Instead of $k > 0$, we should be checking if $r > 0$ to keep consistent with the change that we made in issue 3. Furthermore, the if statement should also be changed to keep consistent with the change and be moved outside of this while loop. The if statement checks if $r == 0$ which would never return true within the while loop. Thus, to fix this, it should be moved outside. Also, we shouldn't be checking $v[k] > n$. This is because in issue 3, we changed k to initially be 1. Therefore in this while loop, we would be checking if the largest coin value is larger than n. Having this condition would cause the algorithm to be incorrect because it assumes that there cannot be change if the n value is smaller than the largest coin value; however, n can be made up of small coin values. This can be fixed by checking for $n >= v[r]$. This checks to make sure that n is larger or equal to the smallest coin value because if n is smaller than the smallest possible coin, change cannot be made at that point. Lastly,

the actions within the else statement should be placed inside the while loop, not after the while loop happened.

- **Issue 5:** The variable k is incrementing at every iteration of the while loop. This causes the algorithm to be incorrect because it moves to the next coin value after one iteration even if the largest coin value can still be used. This causes the function to return a solution that is not the most optimal. For example, if we had v[100...1] (Dollar...Penny), and we wanted change for 50 cents, the function would return a quarter, then a dime, then a nickel, and so on rather than just two quarters. This can be fixed by controlling when k in incremented using a while loop or an if statement of some sort to state that k increments when the value of $v[k] > n$.
- **Issue 6:** The list, d, was not updating with each iteration of the while loop. This would cause the algorithm to be incorrect because it would have simply returned the list that was initialized in line 1 with no changes. This can be fixed by adding a line of code after updating n in line 8 of the pseudocode.

Here is a new and updated pseudocode with the bug fixes and comments that link each change to its corresponding issue listed above:

```
change(n, v, r):
    d[1...r] = 0 //Issue 1
    while (n >= v[r]): //Issue 2
        k=1 //Issue 3
        if r == 0: //Issue 4
            return "no solution"
        while (r > 0 and n >= v[r]): //Issue 4
            n = n - v[k]
            while n < 0: //New pseudocode that would fix Issue 5
                n = n + v[k] //Issue 4 & 5
                k++ //Issue 5
                n = n - v[k] //Issue 5
           d[k] = d[k]+1 //Issue 6
   return d
```

Name: Kevina Wong
ID: 109179049

(b) *Identify a set of Euro coin denominations (a subset of the denominations here) for which the greedy algorithm does not yield an optimal solution for making change. Justify your answer in terms of optimal substructure and the greedy-choice property. (The set should include the 1 Euro cent so that there is a solution for every value of n.) Include an example where the cashier's algorithm with your choice of denominations yields a set of coins that is larger than it needs to be, and also show the smaller set of coins adding up to the same value.*

Answer:
A subset of Euro coin denominations for which the greedy algorithm does not yield an optimal solution for making change is the subset A=[100,50,20,1]. The greedy algorithm would not yield an optimal solution for this set of coin denominations because it does not satisfy the optimal substructure property or the greed-choice property.

The Optimal Substructure Property

The optimal substructure property says that if the n were to be broken down into smaller subproblems, it would still return the most optimal solution when recombined into one solution. However, that is not the case with this algorithm and coin denominations because say, for example, $n = 22$. When n is broken down to its smallest subproblems, it would resort to returning 22 ones rather than 1 twenty and 2 ones. Therefore, because the optimal substructure property does not hold, we can conclude that the set of coin denominations for the greedy algorithm will not always yield an optimal solution.

The Greedy-Choice Property

The greedy choice property says that an optimal solution can be constructed without referencing future or past decisions. However, for this set of coin denominations, the algorithm will return a non-optimal solution and does not refer to future or past decisions. For example when $n = 180$, it will make the greedy choice of taking 1 hundred, 1 fifty, 1 twenty, and 10 ones. However, this is a non-optimal solution. For it to have made the optimal solution, it would have needed to refer to a future decision of taking 1 hundred, skipping the fifty, and taking 4 twenties. Therefore, because the optimal substructure property and the greedy-choice property does not hold, we can conclude that this set of Euro coin denominations for the greedy algorithm does not yield an optimal solution.

Example: Let the possible coins be subset A = [100,50,20,1], like the subset mentioned above. Let $n = 180$. In this example, the Cashier's algorithm yields a set of coins that is larger than it needs to be because it uses 13 coins, and the optimal solution uses 5 coins while they both sum up to the same value of $n = 180$. The table below shows the coins that would be used in the Cashier's Algorithm and the Optimal Solution when $n = 180$:

A	Coins used in Cashier's Algorithm	Coins used in Optimal Solution
100	1	1
50	1	0
20	1	4
1	10	0
Sum:	**180**	**180**
Total Coins:	**13**	**5**

Example 2: Let the possible coins be subset A = [100,50,20,1], like the subset mentioned above. Let $n = 60$. In this example, the Cashier's algorithm yields a set of coins that is larger than it needs to be because it uses 11 coins, and the optimal solution uses 3 coins while they both sum up to the same value of $n = 60$. The table below shows the coins that would be used in the Cashier's Algorithm and the Optimal Solution when $n = 60$:

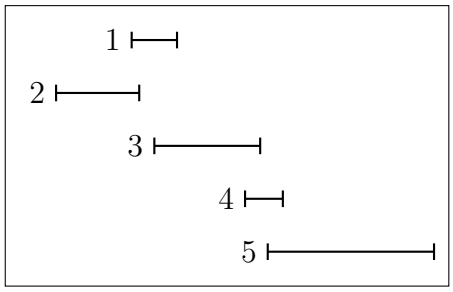
A	Coins used in Cashier's Algorithm	Coins used in Optimal Solution
100	0	0
50	1	0
20	0	3
1	10	0
Sum:	**60**	**60**
Total Coins:	**11**	**3**

Name: Kevina Wong
ID: 109179049

2. *In this question we consider the interval scheduling problem, as covered in class.*

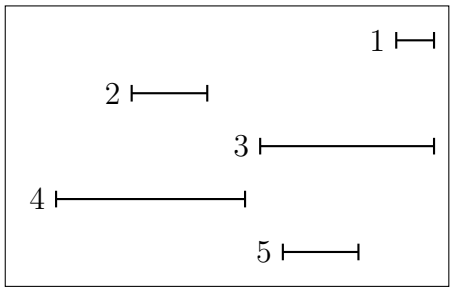
   (a) *Consider a greedy algorithm which always selects the shortest appointment first. Draw an example with at least 5 appointments where this algorithm fails. List the order in which the algorithm selects the intervals, and also write down a larger subset of non-overlapping intervals than the subset output by the greedy algorithm.*

   A comment on level of justification (applies to all of problems 2 and 3): *to help us understand your thinking, it is worth writing a little about the order in which the algorithm selects the intervals. For example "The algorithm takes intervals in the order [1,3,4]: first the algorithm takes interval 1 because that is the shortest. Interval 1 conflicts with intervals 2 and 5, so they are removed. The next shortest is 3, which conflicts with interval 6, and the only remaining interval is 4."*



   The shortest-appointment-first algorithm takes in the intervals in the order [4,1]. First, the algorithm takes interval 4 because it is the shortest interval. Because interval 4 conflicts with intervals 3 and 5, those two intervals are removed. Then the next interval the algorithm takes is interval 1. This is because interval 1 is the next shortest interval. This conflicts with interval 2 and 3. There are no intervals left, so the only two appointments on the schedule are [4,1]. This, however, is a case where the greedy algorithm fails because it yields two possible appointments when the most optimal solution is three possible appointments consisting of appointments [2,3,5] because these three are all non-overlapping. This optimal subset is larger than the non-optimal subset that was produced by our greedy algorithm.

(b) *Consider a greedy algorithm which always selects the longest appointment first. Draw an example with at least 5 appointments where this algorithm fails. Show the order in which the algorithm selects the intervals, and also show a larger subset of non-overlapping intervals than the subset output by the greedy algorithm. The same comments apply here as for 2a in terms of level of explanation.*

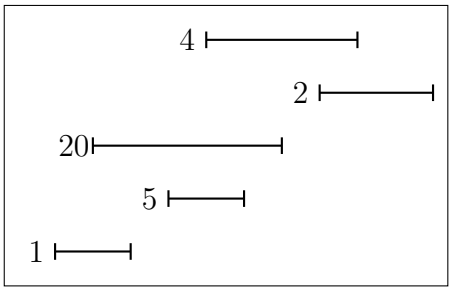


The longest-appointment-first algorithm takes intervals in the order [4,3]. This is because the algorithm takes the longest appointment first, which in this case is 4. Since appointment 4 conflicts with appointment 2, interval 2 is removed. The next longest path is interval 3. Since interval 3 conflicts with intervals 1 and 5, both of those are removed as well. There are no intervals left, so the greedy algorithm yields a non-optimal schedule of only two appointments in the schedule. This is a case where the greedy algorithm fails because the subset of appointments that it yields is not the largest subset possible. The larger, most optimal subset would be a schedule with the appintments [2,5,1], which all do not overlap. Thus, tihs optimal subset is larger than the non-optimal subset that was produced by the longest-appointment greedy algorithm.

Name: Kevina Wong
ID: 109179049

3. *In this question we'll consider weighted problems.*

   (a) *Consider the* weighted *interval scheduling problem. In this problem, the input is a list of n intervals-with-weights, each of which is specified by* $(start_i, end_i, wt_i)$*. The goal is now to find a subset of the given intervals in which no two overlap and to maximize* the sum of the weights*, rather than the total number of intervals in your subset. That is, if your list has length n, the goal is to find* $S \subseteq \{1, \ldots, n\}$ *such that for any* $i, j \in S$*, interval i and interval j do not overlap, and maximizing* $\sum_{i \in S} wt_i$*. Consider the greedy algorithm for interval scheduling from class, which selects the job with the earliest end time first. Give an example of weighted interval scheduling with at least 5 intervals where this greedy algorithm fails. Show the order in which the algorithm selects the intervals, and also show a higher-weight subset of non-overlapping intervals than the subset output by the greedy algorithm. Same comments apply as on problem 2.*



The earliest-end-time-first algorithm take intervals with the weights [1,5,4]. This is because the algorithm first takes in the interval with weight 1 because it has the earliest end time out of all five appointments. This interval has a conflict with the interval with a weight of 20, so that one is eliminated. The next interval would be the appointment with a weight of 5. This is because this appointment has the next earliest end time. Then, the algorithm would take in the appointment with weight 4. This appointment has a conflict with the appointment with weight 2, so that one is removed, as well. Therefore, we are left with the schedule of appointments with the weights [1,5,4]. However this is not the most optimal solution because the sum of these weights is 10. The most optimal appointments, which are [20,2] would have a summed weight of 22, which is a higher-weight subset than what the algorithm yielded.

(b) *Consider the Knapsack problem: the input is a list of n items, each with a value and weight* (val_i, wt_i)*, and a threshold weight* W*. All values and weights are strictly positive. The goal is to select a subset* S *of the items such that* $\sum_{i \in S} wt_i \leq W$ *and maximizing* $\sum_{i \in S} val_i$*. (Note that,* unlike *change-making, here there is only one of each item, whereas in change-making you in principle have an unlimited number of each type of coin.) Consider a greedy algorithm for this problem which makes greedy choices as follows: among the remaining items, choose the item of maximum value such that it will not make the total weight exceed the threshold* W*. Give an example of knapsack with at least 5 items where this greedy algorithm fails. Show the order in which the algorithm selects the items, and also show a higher-value subset whose weight does not exceed the threshold. Same comments apply as on problem 2.*

Answer:

W threshold = 20

Value	Weight	Greedy Algorithm Selection	Optimal Solution Selection
5	20	1	0
4	5	0	1
3	5	0	1
2	5	0	1
1	5	0	1
Total Weight		20	20
Total Value		5	10

For this algorithm, its first choice would be to choose the highest value, which is value 5. Using the greedy algorithm, this would be the only item in the subset because the item with value 5 has a weight of 20, which means that the threshold weight has already been reached, and this is all that we are able to get. However, this algorithm fails because there is a more optimal subset, which is the subset [4,3,2,1]. This is because our total value for this subset is 10, which is a higher-value than what the greedy algorithm yielded. Furthermore, the sum of all of its weight just reached the threshold but does not exceed it.

(c) *Now consider the following algorithm for the knapsack problem. Call the* relative value *of item i the ratio val_i/wt_i. Consider the greedy algorithm which, among the remaining items, chooses the item of maximum* relative value *such that it will not make the total weight exceed the threshold W. Give an example of knapsack where this greedy algorithm fails. Show the order in which the algorithm selects the items, and also show a higher-value subset whose weight does not exceed the threshold. Same comments apply as on problem 2.*

W threshold = 20

Value	Weight	Ratio	Greedy Algorithm Selection	Optimal Solution Selection
8	10	0.8	0	1
8	10	0.8	0	1
10	12	0.833	1	0
Total Weight			12	20
Total Value			10	16

For this algorithm, its first choice would be to choose the highest ratio, which is the item with value 13. Since it has a weight of 12, and there are no other items with a weight of 8 or less, this is the only item that the greedy algorithm could hold in its subset in order to stay below or equal to the threshold weight. The greedy algorithm would return a subset of value 10 and weight 12. However, this is not the most optimal solution because there is another subset that exists that does not exceed the threshold weight. The most optimal solution is to take the two items with value 8 because then we would have a subset with a higher-value of 16. Furthermore, it does not exceed the threshold as the weight is 20.