1. [10 points]

Consider the activity-selection problem. Give an example (set of activities with start and end times) to show that the approach of selecting the activity of ***least duration*** from among those that are compatible with previously selected activities, does not give an optimal solution of mutually compatible activities. Note: The idea is to make up your own set of activities with start and end times (not the specific example shown in the book).

activity #	1	2	3	4	5	6	7	8	9	10
start time	1	2	3	6	1	5	4	6	3	7
end time	2	3	5	7	7	8	9	10	10	11

Given this made-up data of activity start and end times, if we simply select the shortest activities that are compatible with the ones we've already selected, it will go like this:

1) First we select **a1**, so now we need an activity that starts after 2. Our options are a3 (duration 2), a4 (duration 1), a6 (duration 3), a7 (duration 5), a8 (duration 4), a9 (duration 7), and a10 (duration 4). The shortest activity is thus a4.

2) From **a4**, we need an activity that starts after 7. There are no options, so our final set is **{ a1, a4 }**.

It is quite easy to show that this strategy did not give us an optimal solution. Our set { a1, a4 } only managed to fit 2 activities out of 10 into our schedule, which is not a lot. If we don't restrict ourselves to selecting the shortest activities, we can manage to build a bigger set:

1) Select **a1**, so now we need an activity that starts after 2. Our options are a3, a4, a6, a7, a8, a9, and a10, just as before except this time we don't care about duration. Let's select a3 for some arbitrary reason, perhaps because its a fun activity or because it doesn't end very late.

2) From **a3**, we need an activity that starts after 5. Our options are a4, a8, and a10. Let's select a4 because it doesn't end very late compared to the other two, and we want to get home early.

3) From **a4**, we need an activity that starts after 7. There are no options, so our final set is **{ a1, a3, a4 }**.

We didn't get a very good solution since our reasons for picking times were arbitrary, but even so we got a set that is bigger than the set we originally got by choosing the shortest duration activities. This shows that this original strategy could not possibly be optimal, since we found a better solution.

2. [10 points]

Consider the problem of making change for n cents using the ***fewest number of coins***. Assume that each coin value is an integer. You have available coin denominations of **1 cent, 5 cents, 10 cents, 25 cents**, as in the US coins of penny, nickel, dime, quarter. Assume there is infinite availability of each of these four coin denomination types (so that you can reuse each coin type as many times as you like).

a. Describe a greedy algorithm to make change using the fewest number of coins. (you do not need to write pseudocode, but rather just explain your greedy strategy).
b. Explain how your algorithm would work for n=30 cents.

Since we are describing a greedy algorithm, we know we won't be able to consider all possible solutions and (maybe) we won't be able to get an optimal solution. Despite this limited information, we still need to make a decision at each step until we have reached the amount of change we need, so we will simply choose the coin that will give us the most value without going over the amount of change we need (since we don't want to give the customer more change than we owe them).

If I was going to implement this algorithm, I would declare a variable total = 0 and start a while loop which terminates when total == the amount of change we want to make. Inside the while loop, I would do a for loop over { 1, 5, 10, 25 } (which represents the penny, nickel, dime and quarter) and I would choose the maximum of these four such that its value + total is not greater than the amount of change we want to make. If it doesn't go over our change value, total += this value. If none of { 1, 5, 10, 25 } satisfy this constraint, no solution can be found and the function will return an error.

For n = 30 cents, our algorithm will go as follows:

1) total is 0, total != 30 so we execute our while loop. We choose 25 cent quarter since, of the four coins, it has the highest value which satisfies (its value + total) <= 30:

(25 + 0) <= 30 --> 25 <= 30 is true, so we do:
total = total + 25 --> total = 0 + 25 = 25

2) total is 25, total != 30 so we execute our while loop. We choose 5 cent nickel since, of the four coins, it has the highest value which satisfies (its value + total) <= 30:

(5 + 25) <= 30 --> 25 <= 25 is true, so we do:
total = total + 5 --> total = 25 + 5 = 30

3) total is 30, total == 30 so our while loop terminates and we are done. We can see that we selected one quarter and one nickel in order to make our change.

3. [10 points]

Although the greedy algorithm is optimal for the US coins, it is not necessarily optimal for any set of coins.

a. Give an example of a different set of made up coin denominations with different values, for which the greedy algorithm does not yield an optimal solution. Your set should include a penny (1 cent) so that there is a solution for every value of n. As before, assume there is infinite availability of each of your coin denomination types. Show specifically a counter example in which the greedy algorithm yields a change solution that is not optimal, that is, does not include the fewest number of coins.
b. Name another algorithmic approach that will find an optimal solution to your example in (a).

Our coin denominations will be { 1, 3, 20, 25 }. The amount of change we want to make is 40 cents. The greedy approach, since it always chooses as many of the biggest coins as possible, will do this:

1) select the 25 cent coin since it has the highest value which is <= 40

2) select 3 cent coin 5 times, making our total 25 + 3*5 = 25 + 15 = 40, the loop terminates because we have reached our goal

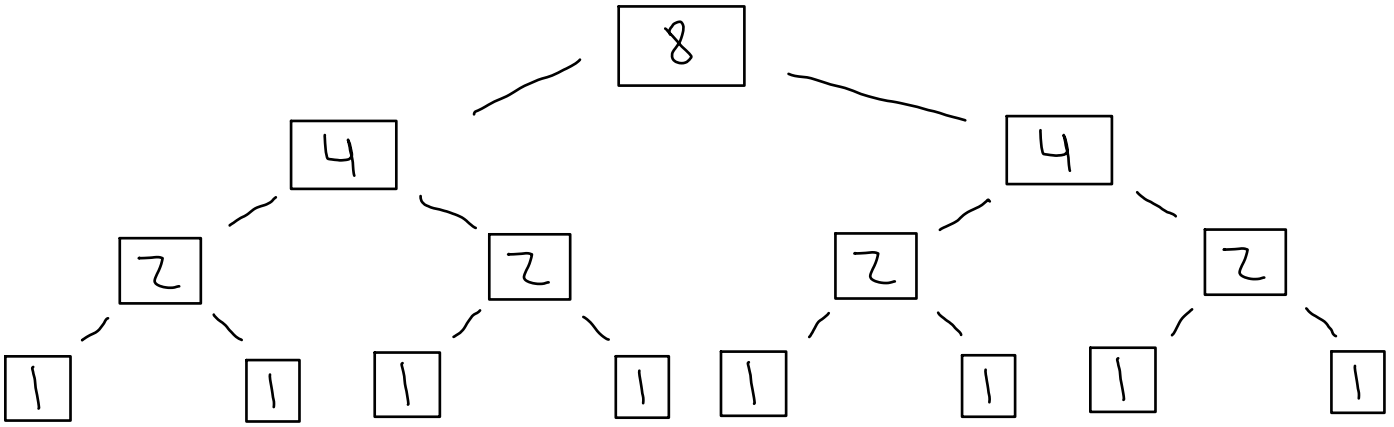
With these denominations and our goal being to make 40 cents, the greedy algorithm chose one 25 cent coin and five 3 cent coi ns, totaling to 6 coins. This is obviously not optimal, since we could have just chosen two 20 cent coins to get to 40 using just 2 coins instead of 6, but ou r greedy algorithm ignores this simple solution in favor of always choosing the biggest coins.

A dynamic programming approach would find the optimal solution in this scenario, since it would find every possible combinati on of coins that total to 40 cents and would choose the solution that requires the least number of coins. It would also save the optimal solutions to subproblem s in a table, in this case the smallest number of coins that sum to a certain value, in order to compute the current optimal subproblem with less computation (it can just lookup the solution to a subproblem in the table rather than recompute it). A brute force approach would also find an optimal solution to this problem , but it would be much slower since the same optimal solutions to subproblems would be calculated multiple times, so dynamic programming in definitely the way to go in this situation.

4. [10 points]

Consider sorting n unique numbers using mergesort. Explain why dynamic programming will not speed up your algorithm. Illustrate this by considering the sub-problems for an array of size 8.

The mergesort algorithm will split the array in half as much as possible until we have a bunch of arrays of size 1, and then it will merge all of these arrays into sorted subarrays until we have a sorted array of our original size. Dynamic programming is effective when there are many repe ating subproblems, but in this case we have unique subproblems. When we split an array in half and sort both sides, the solution to the left side (a sorted subar ray) will not necessarily be the solution to our right side (in fact it often won't, unless we have identical subarrays on both sides, which is unlikely). Considering an array of size 8, merge sort will do this:



So it is clear that dynamic programming doesn't help us here since it is unlikely that the solution to the subarray 4 on the left will be the same solution to the subarray 4 on the right, and so on for the smaller subarrays. We would have to have an original array of size n which, when cut in half, results in two identical subarrays of size n/2, and this is usually not the case. In most cases, splitting an array in half results in unique data on both sides, which would result in unique solutions and thus repeating subproblems would be unlikely.