Bryan Arnold

CSE 3500

Homework 4

2/20/17

Coin Change

1. First, I will show that there is at most 4 pennies at a time. This is because the value of a nickel is equal to five and pennies are equal to one, so by having 5 pennies this equates to one nickel. This would make use of using five coins when only one coin is necessary, so using a nickel would reduce the total coin count by 4. So, when the remainder of the total change value is greater than 5, use as many nickels as possible before considering pennies. Next, the number of nickels is always less than or equal to one because the value of a nickel is five and the value of a dime is ten. So, if there arises a need for two nickels, you would instead use a dime which reduces the number of coins used by one. So, whenever the total change amount remainder is greater than 10, use as many dimes as possible before using nickels. Finally, the number of dimes can never exceed two because a dime has a value of ten and a quarter has a value of 25. If two dimes are used, all the values leading up to 25 can be found using pennies, and anything after 25 can be found using a quarter and pennies. At 30 cents, a quarter and nickel would be used. By using a quarter and nickel to hit 30 instead of three dimes, the total coin count is reduced by one. The case for nickels and dimes would fall under this afterwards, so whenever the remainder is greater than 30, use as many quarters as possible first. Also, if there is one nickel, there can at most be one dime. This is because whenever an opportunity arises to replace the nickel, when the total cents has a 0 at the end, the nickel is erased. The only time nickels and dimes exist at the same time, is for 15 and this is never exceeded as quarters will begin to replace the dime and nickel combinations. Before this, a number like 5 only has one nickel and no dimes, so this satisfies the property as well.
2. It is always optimal to use as many quarters as possible because it reduces the amount of coins used. Whenever the remainder of change is greater than 30, a quarter will reduce this to 5. If you did not use quarters for this, you would have to use three dimes instead of a quarter and a nickel. Three dimes is one more coin than a quarter and nickel, so using the maximum number of quarters is optimal. Another example is when the total change is 50. Two quarters is very minimal amount, but 5 dimes is significantly more, but is the next most optimal. So, using the maximum possible amount of quarters first is always optimal.
3. The greedy algorithm is optimal due to how it deals with the remainder of change by using the maximum amount of quarters, then dimes, then nickels, then pennies needed to reach the goal. This method allows for minimal amount of coin usage,

making it optimal. So, as explained earlier with the remainders of change being used, this greedy algorithm is optimal in how to find the coin change.

Problem 4.4

Algorithm:

First, take in two sequences A and B, with A having a length of m and B having a length of n. The goal is to find out whether sequence A is a subsequence of B. To do this, I'd take two pointers p and p', to move through sequence A and B. So, they would both start at the beginning of a sequence, p starts at A and p' starts at B, and they both will compare each part of their respective sequences. If the part of the sequence that the two pointers are at are equal, so A[p] == A[p'], both pointers will increment up to the next part of their respective sequences (p = p + 1, p' = p' + 1). If the two parts of the sequence are not equal to each other, so A[p] != B[p'], only increment p' upwards, since the two parts of the sequence are not equal. This makes it so both sequences are checked for equality at each spot, ensuring whether A is a subsequence of B, and both sequences are only traversed once, so O(m+n) time. If at any point the pointer p = n, then A is a subsequence and it can be displayed, and if p'= n, then A is not a subsequence of B and can be displayed.

Problem 4.13

This problem isn't too complicated to solve. First, we are given a set of jobs that have given weight and processing time on each job. We want the order of the jobs to be put in maximum efficiency so that the total time to do each job is the smallest. To begin, suppose we have a set of jobs that we want to accomplish, denoted as S = (s1, s2,…,n). As discussed earlier, each job i has a given weight, denote this as w_i and a processing time of t_i, and a finishing time of c_i. It is given to us that the total runtime is: Runtime = $\sum_{i=1}^{n} w_i c_i$. It was also known that $c_i = t_i$. So, to minimize the total runtime, the greedy strategy, we want to do the job with the larger factor of w_i divided by t_i first because the total unit time for this job will have a larger weight, meaning the total runtime equation will be higher. So, this is the core to our algorithm as well as proves that the following algorithm is the optimal one:

First, sort the set of schedules S by taking each job's workload and dividing it by the total processing time in descending order. After that, simply process the jobs in the order they were sorted.

Algorithm:

Take in set of jobs, denoted as S. Next, sort S by workload/processing time, in descending order. Finally, process jobs in this order. So, not too complicated.

Huffman Coding

1. After sorting the characters by frequency and putting them into a rooted binary tree, the following Huffman encoding is the result:
   A: 0, C: 10, G: 110, and T: 111.

2. Given a set of characters in a given alphabet of length n, so length of S = {1, 2,…. n – 1}. The longest code word will have a length of n – 1. Now, a set of frequencies that proves this can be the following: $f_{n-i} = 2^{-i}$ for i being 1 to n – 1. The cumulative sum of all frequencies up to and including the j − 1-th smallest frequency will be less than fj. So, by construction, the tree will be a spine with a leaf joining each intermediate subtree. This means the depth is n-1, meaning that the encoding will also be of length n – 1.