Aman Patel

November 9, 2019

CSCI-B 403

Assignment 4

1. Describe and prove an efficient algorithm that makes change for n cents, using the least number of quarters, dimes, nickels and pennies.
   1. In order to use the least number of coins, the largest coins need to be used as many times as possible before moving onto smaller coins. For example, if , a quarter can be removed (). Then At this point, removing a quarter would be more efficient than removing a dime. After the quarter is removed, . A dime cannot be removed, but a nickel can. is now equal to and two pennies can be removed to reach This function returns a list in the form In this case, the returned list would be . Using this approach, the maximum number of dimes will not exceed 2, nickels will not exceed 1, and pennies will not exceed 4. This is because and
2. Design an algorithm that takes as input two lists of players and the heights of the players and checks if it is possible to place players to take the photo subject to the placement constraint.
   1. In order to place the players, every player in the back row must have at least one person on the other team that is shorter than them. This can be checked by sorting both arrays in increasing order and iterating through each array, making sure that every value in array A is greater than their corresponding values in array B. The time complexity of this approach is , as the sorting takes and the comparisons take . These are added and simplified to
3. Given a positive integer n, describe an algorithm that breaks the n into the sum of at least two positive integers and maximize the product of those integers and returns the maximum product you can get.
   1. The basis of the algorithm can be discovered by analyzing the best possible combinations from 4 to 6. For , the best product is . For , the best product is . For , the best product is . Notice that the maximum products are all greater than or equal to their n values. This pattern can be observed , thus, these values will not be used when finding the maximum product. That leaves 3, 2, and 1. Using 1 does not increase the product, so it will only be used as a base case when . Therefore, only 2 and 3 need to be used to form the maximum products.
   2. The algorithm will work as follows. The function takes in two arguments, an integer and an accumulator list (*acc*). The helper function *product* takes in a list and returns the product of the values in the list. The following conditions will be added to the main function:
      1. If , return 1.
      2. If and is empty, return 1.
      3. If and is not empty, return
      4. If and is empty, return 2.
      5. If and is not empty, return
      6. If , return
      7. Else, call
4. Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight without alerting the police.
   1. Greedy approach
      1. Compare values in 0th and 1st positions
      2. If 0th is greater, add its value to
      3. If 1st is greater, add its value to
   2. Non-Greedy approach
      1. This approach takes three houses into account at a time. A helper function is used that takes the current index as an argument. The array is iterated starting from the left. First check if the current position is outside the length of the array. If so, return 0. If not, the following will be applied: . This compares the sum of the recurred max value for the house in position 2 and the value of position 0 with the recurred max value in position 1.
5. Describe an algorithm that returns the starting gas station's index if you can travel around the circuit once in the clockwise direction and returns -1 otherwise.
   1. The algorithm would begin with the initialization of two variables to 0: the starting index to be returned (*start*) and the current amount of gas in the tank (*tank*). The sums of the two given arrays are calculated and compared. If the sum of *cost* is greater than the sum of *gas,* -1 is returned. This is because it is impossible to reach every station if the net amount of gas is less than 0. For example, assume *gas* = [1 2 3] and *cost* = [2 4 1]. Adjacent gas stations can be merged and treated as one. Thus, *gas* can be rewritten as [3 3] and *cost* as [6 1]. Repeating the process gives *gas* = [6] and *cost* = [7]. This trip is not possible, as the cost to get to the next station is larger than the maximum amount of gas. In this case, the returned value would be -1. If the preceding condition is false, the given arrays are iterated through simultaneously. The value of *tank* is updated to the sum of the current value of *tank*, the amount of gas at the current station (gas[i]), and the cost to get to the next station (-1 \* cost[i]). If the updated value of *tank* is less than 0, the current gas station is not eligible and *start* and *tank* are reset to 0. Otherwise, the loop continues, and no values are reset. After the loop is complete, *start* is returned.