Homework 2 - Robert Duque

Monday, October 3, 2022 12:28 PM

PSID: 1582182

I understand and agree to abide by the provisions in the University of Houston Undergraduate Academic Honesty Policy.
This is my own work.

P0 (15pt): Give an algorithm to print out all possible subsets of an n-element set, such as $\{1, 2, 3 \ldots, n\}$.

I will work off of the solution given by an algorithm I have found on Chegg, the link is https://www.chegg.com/homework-help/questions-and-answers/po-15pt-give-algorithm-print-possible-subsets-n-element-set-1-2-3-n--q68250358.

We will use a function that will take a vector array and return a vector of vectors of arrays that will hold the result in a 2D vector. We will call a function called subsets that will take parameters: the original vector, the results 2D vector, the current subset vector, and the index of the original vector. In this function, we will use .push_back for the results vector to push back the current subset passed into the function and we will make a for loop with the iterator index starting at the index passed through the function. For each iteration we will do the following:

Push_back the current iterator index of the original vector
Call the helper function with the same arguments but add 1 to the index argument
Pop the last result of the subset to initiate backtracking.

Finally, we return the resulting 2D array from the first function and there should be a 2D vector of all possible subsets of the original vector.

Pseudocode:

```
Subsets(A, res, subset, index)
    Push back subset to res
    For(I in A[index:])
        Push back A[i] to subsets
        Call subsets w/ par. (A, res, subset, i+1)
        Pop back from subset
    Return

Initialize the following variables:
    Array -> vector of ints
    Subset -> vector of ints
    Index = 0
    Res -> vector of vector of ints
Call subsets(array, res, subset, index)
```

The runtime of this algorithm will be explained using an example input for the input array. Let's say we have the input array of size 5 that is {1, 2, 3, 4, 5}. When we call the subsets on this array, we will first iterate through the first push_back of each index and call the function again which will be a runtime complexity of O(n). Each index will have a for loop that will iterate until an arbitrary for loop index reaches then end of the original array size. This will result in a runtime complexity of O(n^2) for the final algorithm.

P1 (15pt): A *derangement* is a permutation p of $\{1, \ldots, n\}$ such that no item is in its proper position; i.e. $p_i \neq i$ for all $1 \leqslant i \leqslant n$. For example, $p = \{3, 1, 2\}$ is a derangement of $\{1,2,3\}$, whereas $p = \{3, 2, 1\}$ is not.

Write an efficient backtracking program that prints out all derangements of $\{1, \ldots, n\}$.

First, we will initialize a vector of pairs of integers that will use the first int as the index of the original array and the second int will be used to store the value that is in the respective index of the original array, we will call this pair of vectors positions. We will also initialize an empty array to keep the current deranged array and we will call it deranged. We will implement a deranged function that will take the empty deranged array, the size of the original array, current index (which will be 0 for first call, we will call this index curInd), the vector for positions, and a vector that will keep track of what values have been placed already called placed.

In the function, we will make an if statement that says if the current index is equal to the size of the array, then we will print out the deranged array that we currently have. If not full, we will make an iterator for a vector of pairs that will go through the positions vector. Everything below will be in a for loop that takes the iterator as the parameter and will terminate if the iterator equals the end of the positions vector.

If the pointer to the first element of the iterator does not equal the curInd and the pointer to the second element (value) of the iterator is not already in the placed vector, we will push the value of the iterators second pointer value into the curInd of the deranged array. We will then insert the second pointer value of the iterator into the filled array and recursively call the deranged function but adding one to the current index. After the recursive call, we will erase the element from the placed vector which will initialize backtracking of the algorithm.

Pseudocode:

```
Deranged(derangedArray, n, curInd, positions, placed)
    If curInd == n
        Print out derangedArray
        Return
    Else
    IT -> vector pair iterator
    For IT in positions
        If IT->first != curInd AND IT->second is not in placed
            DerangedArray[curInd] = IT->second
            Push back IT->second into placed
            Deranged(derangedArray, n, curInd + 1, positions, placed)
            Pop IT->second from placed
    Return

Initialize the following variables:
    Size -> input number
    Array -> array of ints of size
    Positions -> vector pairs that hold values of Array and their index
    Placed -> vector of ints
    derangedArray -> int array of size size
Deranged(derangedArray, size, 0, positions, placed)
```

This algorithm will have a runtime complexity of O(n!). Let's take an example of 4 different elements in a vector that consists of {1, 2, 3, 4}. In each derangement, we cannot have 1 in the first position, 2 in the second position, 3 in the third position, and 4 in the fourth position. If this is the case, 1 has 3 different options to choose from that isn't it's own index. If 1 took up the position where 2 was, that gives 2 3 options as well but this happens only in one iteration, all the other options will be 2 for input 2. So if we put this in big O time complexity notation, we will have O(n-1) for the first

option, O(n-1) + O(n-2) for the second option, O(n-1) + O(n-2) + O(n-3) for the third option and O(n-1) + O(n-2) + O(n-3) (the extra options are for the possibility that the previous entries took up the original space of the current index). Which will result in a runtime complexity of O(n!).

Dynamic Programming:

P3 (20pt): In a strange country, the currency is available in the following denominations: \$1, \$4, \$7, \$13, \$28, \$52, \$91, \$365. Find the minimum bills that add up to a given sum \$$k$.

(a) The greedy change algorithm repeatedly takes the largest bill that does not exceed the target amount. For example, to make \$122 using the greedy algorithm, we first take a \$91 bill, then a \$28 bill, and finally three \$1 bills. Give an example where this greedy algorithm uses more bills than the minimum possible. [Hint: It may be easier to write a small program than to work this out by hand.]

(b) Describe and analyze a recursive algorithm that computes, given an integer k, the minimum number of bills needed to make \$$k$. (Don't worry about making your algorithm fast; just make sure it's correct.)

(c) Describe a dynamic programming algorithm that computes, given an integer k, the minimum number of bills needed to make \$$k$. (This one needs to be fast.)

I will use a recursive function that will take the following parameters:

- Target value
- Array with denomination integers
- Size of denomination array

We will take the largest denomination that does not exceed the target amount and subtract the denomination from the current total. We will then recursively call the function with the new target total as well as add 1 to the iterator to show how many bills are used. If the target total is 0, we will return the iterator value to show the amount of bills used.

A) An example of a greedy algorithm not being an optimal choice could be \$546 target total. We will send in 546 as the initialized target total with the iterator starting at 0. The largest number that will go into 546 is 365 so the new target total is 546 - 365 = 181. We call the function again with the new target total and take 91 as the new denomination. 181 - 91 = 90, then 90 - 52 = 38, then 38 - 28 = 10, then 10 - 7 = 3. We will then use 3 1 dollar bills the make the target total 0 which is a total of 8 bills used. With that being said, 546 can be calculated by using 6 \$91 bills since 91 * 6 = 546 and we know that 6 is less than 8 so the greedy algorithm in this case does not give an optimal solution to our problem.

B) Didn't know that question B asked for the algorithm since I have it written in question A but I will write the pseudocode for this step (we will use 546 as the example:

```
Int greedy(val, denoms, size)
    If val == 0
        Return 0
    Else
        For j = size; j >= 0; j--
            If denoms[j] <= val
                Return 1 + greedy(val-denoms[j], denoms, size)
    Return -1

Denoms[] = {1,4,7,13,28,52,91,365}
Amount = greedy(546, denoms, size of denoms)
```

C) For this problem, I used the link https://www.chegg.com/homework-help/questions-and-answers/dynamic-programming-p3-20pt-strange-country-currency-available-following-denominations-1-4-q68679150 to help me understand the dynamic programming concept.

To solve this problem dynamically, we need to make a table that will hold the values of the minimum number of denominations used for value I (index of table and current total), we will call this table minTable. We will use a function that will take:

- Array of denominations
- Size of array of denominations
- Value we are concerned with

We will initialize minTable to INT_MAX that will represent infinite for all cells in minTable and we will also assign 0 to the first index of minTable. The function will then loop through range 1:value including the value, let's call the iterator i. Each iteration will then loop through the array of denominations, we will call the iterator j, and if the current value of the denominations array is smaller or equal to I, then we will make an int variable called subRes that will hold the value of the minTable at index I - denominations[j]. There will then be an if statement that will check if subRes + 1 is smaller than the minTable[i]. If the condition is met, then we will replace the value at table[i] with subRes + 1 which will then store the min amount of denominations used for the specific value. We will call the function minDen

```
Int minDen(denominations, size of denominations array, value)
        Int minTable[value+1]
        minTable[0] = 0
        For cell in minTable
                Cell = INT_MAX
        For I in range(1 to value)
                For j in range(size of denominations array)
                        If denominations[j] <= I
                                Int subRes = minTable[I - denominations[j]
                                If subRes is not INT_MAX and subRes + 1 < minTable[i]
                                        minTable[i] = subRes + 1
        Return minTable[value]

Answer = minDen(denominations, size of denominations array, initial value)
```

Programming Challenges:

Problem 1:

PSID: 1582182
Submission ID: f50123d9-ea3a-4170-952e-b0746c23778d

Problem 2:

PSID: 1582182
Submission ID: 1013a677-1aca-4a1f-9a4c-7ad59ca2681e

Problem 3:

PSID: 1582182
Submission ID: edda45b9-00ea-4e43-b7bc-a1c92a7b7d92

For the life of me I could not figure out what I was doing wrong to get 5/7 testcases right on problem 3 so I just settled for that. If you could give me some pointers on how to optimize my solution, I would greatly appreciate the feedback. Thank you!