Solution to Homework Assignment 4(CS 430)

Saptarshi Chatterjee
CWID: A20413922

February 25, 2018

1 Question

Q. Define the red depth of a node in a red-black tree as the number of red ancestors that the node has. Can the red depths of nodes in a red-black tree be maintained as fields in the nodes of the tree without affecting the asymptotic performance of any of the red-black tree operations? Show how, or argue why not.

Answer -

No, we can't. When an insertion or deletion is done on Red-Black tree, if there is a rule violation at any node we try to push it up closer to Root and Recurse. So the Red depth of the node can't be determined from the info available at x, x:left, and x:right . So augmenting any such info will deteriorate asymptotically performance of $O(\lg n)$ as per the following Theorem.

Page 346 CLRS - Theorem 14.1 (Augmenting a red-black tree) Let f be an attribute that augments a red-black tree T of n nodes, and suppose that the value of f for each node x depends on only the information in nodes x, x:left, and x:right, possibly including x:left:f and x:right:f. Then, we can maintain the values of f in all nodes of T during insertion and deletion without asymptotically affecting the $O(\lg n)$ performance of these operations.

2 Question

Q. India and Pakistan are to meet each other in the world championship of squash. The champion will be the first to win n matches in a series of 2n?1 matches. For any given match there is a fixed probability p that India will win, and hence a probability q = 1 - p that Pakistan will win. Let P_{ij} be the probability that India will win

the series given that they still need i more victories, whereas Pakistan needs j more victories for the championship. $P_{0j} = 1$, $1 \leq j \leq n$, because India needs no more victories to win. $P_{i0} = 0$, $1 \leq i \leq n$, as India cannot possibly win if Pakistan already has.

Q.(a) Explain why $P_{ij} = p \times P_{i-1,j} + q \times P_{i,j-1}$

Answer -

$$\begin{aligned} P_{i,j} &= \text{Probability that india wins the series after winning i more matches} \\ &= \text{India wins the match and wins (i-1) more matches to win the series +} \\ &\quad \text{Pakistan wins the match but India wins i matches after to win the series} \\ &= p \times P_{i-1,j} + q \times P_{i,j-1} \end{aligned}$$

Q.(b) What is the value of P_{00}?

Answer -

$P_{00} = 0$
P_{00} is the case that suggests Both India and Pakistan have already won . Which is impossible , so probability of such event is 0.

Q.(c) Devise and analyze an unmemoized dynamic programming algorithm that calculates P_{nn}, the probability that India will win the series.

Answer -

Python Code -

```
def IndWinSeries(totWinIndNeed, totWinPakNeed, p):

    if totWinIndNeed == 0 and totWinPakNeed == 0:
        return 0

    if (totWinIndNeed == 0):
        return 1

    if (totWinPakNeed == 0):
        return 0

    indWinsNow = p * IndWinSeries(totWinIndNeed - 1, totWinPakNeed, p)
    pakWinsNow = (1 - p) * IndWinSeries(totWinIndNeed, totWinPakNeed - 1, p)

    return indWinsNow + pakWinsNow

print(IndWinSeries(3,2,.5)) #0.3125
print(IndWinSeries(2,2,.5)) #0.5
print(IndWinSeries(0,2,.5)) #0.5
```

Complexity -

$$
\begin{aligned}
T(n) &= \sum_{i}^{n} \sum_{1}^{1} O(time - to - create - i \times j - Matrix) \\
&= 1^2 + 2^2 + 3^2 + ... + n^2 \\
&= \frac{n(n+1)(2n+1)}{6} \\
&= O(n^3)
\end{aligned}
$$

Q.(d) Devise and analyze a memoized $O(n^2)$ time dynamic programming algorithm that calculates P_{nn}.

Answer -

Following is the dynamic Programming version . It stores the previous computation results in a $n \times n$ matrix . And then multiplies that with probability which take O(1) time. So essentially complexity is

$$
\begin{aligned}
T(n) &= O(time - to - create - n \times n - Matrix) + O(1) \\
&= O(n^2)
\end{aligned}
$$

```
def IndWinSeries(totWinIndNeed, totWinPakNeed, p, memo):

    if (totWinIndNeed,totWinIndNeed) in memo:
        return memo[(totWinIndNeed,totWinIndNeed)]

    if totWinIndNeed == 0 and totWinPakNeed == 0:
        return 0

    if (totWinIndNeed == 0):
        return 1

    if (totWinPakNeed == 0):
        return 0

    indWinsNow = IndWinSeries(totWinIndNeed - 1, totWinPakNeed, p, memo)
    memo[(totWinIndNeed - 1,totWinPakNeed)] = indWinsNow
    pakWinsNow = IndWinSeries(totWinIndNeed, totWinPakNeed - 1, p, memo)
    memo[(totWinIndNeed, totWinPakNeed -1)] = pakWinsNow

    return p * indWinsNow + (1-p) * pakWinsNow

memo = {}
print(IndWinSeries(3,2,.5,memo)) #0.3125
```

3 Problem 16 -1 on pages 446 - 447, Consider the problem of making change for n cents using the fewest number of coins. Assume that each coin?s value is an integer.

Q.a. Describe a greedy algorithm to make change consisting of quarters, dimes, nickels, and pennies. Prove that your algorithm yields an optimal solution.

Answer -

Step (i) - Repeatedly keep taking highest denominator coin until remaining value is not negative or 0 and keep increasing count by 1. If we get 0 stop and return count .
Step (ii) - If not move to next highest denominator and repeat Step (i)

This Algo will Always yield an optimal solution , as we are always choosing the highest denominator and thus reducing the remaining change at the fastest rate possible . As we have **pennies** in our denominator set , so reaching a 0 sum is guaranteed .

Python Code -

```
def minExchage(allCoin, change):
    #Sort array in the reverse order .
    allCoin.sort(reverse=True)
    minCount = 0
    for coin in allCoin:
        while(change >= coin):
            minCount+= 1
            change -= coin

    return minCount

print(minExchage([1,5,10,25], 9))
```

Q.b. Suppose that the available coins are in the denominations that are powers of c, i.e., the denominations are $c^0, c^1, ...c^k$ for some integers $c > 1$ and $k \geq 1$. Show that the greedy algorithm always yields an optimal solution.

Answer -

Given an optimal solution $(x_0, x_1, ..., x_k)$ where x_i indicates the number of coins of denomination c^i . We will first show that we must have $x_i < c$ for every $i < k$. Suppose that we had some $x_i \geq c$, then, we could decrease x_i by c and increase x_{i+1} by 1. This collection of coins has the same value and has c - 1 fewer coins, so the original solution must of been non-optimal. This configuration of coins is exactly the same as you would get if you kept greedily picking the largest coin possible. This is because to get a total value of V, you would pick $x_k = \lfloor Vc^{-k} \rfloor$ c and for $i < k$, $x_i \lfloor (Vmodc^{i+1})c^{-i} \rfloor$. This is the only solution that satisfies the property that there aren?t more than c of any but the largest denomination because the coin amounts are a base c representation of $Vmodc^k$.

Q.c. Give a set of coin denominations for which the greedy algorithm does not yield an optimal solution. Your set should include a penny so that there is a solution for every value of n

Answer -

If the denominations are {1, 4, 5, 7}, and the change to make is 9. The greedy solution would result in coins {7,1, 1 } but the optimal solution would be {5, 4}

Q.(d) i. As given on page 447, but use dynamic programming in its recursive formulation

Answer -

Following is a recursive DP Python code for Coin change problem

```
import sys
```

```
def minCoins(coins, change, memo):
    if change in memo:
        return memo[change]
    # base case
    if change == 0 :
        return 0

    #Init count with very High value
    count = sys.maxsize

    # Check every coin that is smaller than change
    for i in coins:
        if i <= change:
            subprob = minCoins(coins, change - i, memo) + 1
            # If subproblem size is smaller then thats the new result
            if subprob < count:
                count = subprob
    memo[change] = count
    return count


print(minCoins([9, 6, 5, 1], 93, {}))
```

Q.(d) ii. As given on page 447, but use dynamic programming in its iterative formulation

Answer -

Here is an iterative formulation of Recursive code

```
public class Solution {

    public int coinChange(int[] coins, int amount) {
        if (amount < 1) return 0;
        return coinChange(coins, amount, new int[amount]);
    }

    private int coinChange(int[] coins, int rem, int[] count) {
        if (rem < 0) return -1;
        if (rem == 0) return 0;
        if (count[rem - 1] != 0) return count[rem - 1];
        int min = Integer.MAX_VALUE;
        for (int coin : coins) {
            int res = coinChange(coins, rem - coin, count);
            if (res >= 0 && res < min)
                min = 1 + res;
            }
            count[rem - 1] = (min == Integer.MAX_VALUE) ? -1 : min;
            return count[rem - 1];
        }
}
```

Q.(d) iii. Analyze the time required.

Answer -

Time complexity : $O(S \times n)$ where S is the amount, n is denomination count. In the worst case the recursive tree of the algorithm has height of S and the algorithm solves only S subproblems because it caches pre calculated solutions in a table. Each subproblem is computed with n iterations, one by coin denomination. Therefore there is $O(S \times n)$ time complexity..

Q.(e) Suppose that, in part (d), we add the restriction that each denomination can be used just once. Modify your algorithm to determine if making change for n cents is possible.

Answer -

Same algo as recursive dynamic programming solution just that before recursively calling , remove the denominator from the list {coins.remove(i) And Check final result if its system max then no solution exists }

Here is the Algo -

```
 import sys
def minCoins(coins, change, memo):
    if change in memo:
        return memo[change]
    # base case
    if change == 0 :
        return 0

    #Init count with very High value
    count = sys.maxsize

    # Check every coin that is smaller than change
    for i in coins:
        if i <= change:
            #A coin can only be used once
            coins.remove(i)
            subprob = minCoins(coins, change - i, memo) + 1
            # If subproblem size is smaller then thats the new result
            if subprob < count:
                count = subprob
    memo[change] = count
    return count

solution = minCoins([6,5,4,2,1], 93, {})
print( solution if solution < sys.maxsize else "No Solution" ) # No solution
solution = minCoins([6,5,4,2,1], 12, {})
print( solution if solution < sys.maxsize else "No Solution" ) #3
```

References

[1] Greedy Algo for Coin Change
https : //www.geeksforgeeks.org/greedy − algorithm − to − find − minimum − number − of − coins/

[2] Dynamic Algo for coin Change
https : //www.geeksforgeeks.org/find−minimum−number−of−coins−that − make − a − change/

[3] CLSR Solution

texttthttp://sites.math.rutgers.edu/ ajl213/CLRS/Ch16.pdf

[4] Iterative
https : //leetcode.com/articles/coin − change/