

COT 6405 Analysis of Algorithms  
Homework 2

Michael Novo

October 2, 2015

### 1. Strategy 1

As we try to solve the problem manually, we realize there is a trend apparent. Namely, the solution for a given  $n$  can be gathered by altering the solutions for  $n-1$  and  $n-2$ . Specifically, adding one to the solution of  $n-1$  and adding two the solution of  $n-2$ , we can reach the solution for  $n$ . We notice that the specific trend corresponds to *naive-Fibonacci*( $n+1$ ) Hence, the recurrence should be:

$$T(n) = T(n-1) + T(n-2)$$

This is the same recurrence found upon analyzing *naive-Fibonacci* in class. To provide a closed form for this solution, we know that we can use:

$$T(n) = \frac{1}{\sqrt{5}}(\phi^{n+1} - (1-\phi)^{n+1})$$

where the golden ratio,  $\phi = \frac{1+\sqrt{5}}{2}$

To prove that this strategy works, let us use a proof by induction. First, assume that our strategy works for  $n$  and let's prove that it also works for  $n+1$ . If we are assuming that the solution works for  $n$ , then the solution to  $n-1$  and  $n-2$  must also have been correct by our scheme, since they are necessary to produce the correct answer for  $n$ . To arrive at the solution for  $n+1$  we must use the solution to  $n-1$ , which we know is correct, and the solution to  $n$ , which we have assumed is correct. Thus, our solution works for all  $n$ .

### Strategy 2

The problem stipulates that the patient can only take one or two pills on each day. Therefore, each distinct way for taking pills on a given day will consist of a permutation of ones along with twos. The permutation may also consist of solely ones or solely twos.

One way of enumerating all permutations is to start out with  $n$  ones as a first way of taking  $n$  pills. There is only one way to enumerate this possibility. Next, we consider combining two of the ones in the previous way. In so doing, we have decreased the number of days the patient takes pills. We then consider the expression that tells us the number of permutations when repetition of elements is allowed,  $\frac{n!}{r_1!r_2!\dots r_k!}$  where  $n$  is the number of elements and  $r_i$  is the number of repetitions.

Surely, we are enumerating permutations, but how many times? It turns out that using this scheme we have a longest sequence corresponding to all ones and a shortest length sequence corresponding the maximum number of twos we can use for a given  $n$ . Taking into account both even and odd  $n$ , the maximum number of twos is  $\lfloor \frac{n}{2} \rfloor$ . We then realize that we are subtracting  $k$  ones each time we combine ones into a two. We also increase the number of two repetitions by one and decrease the number of one repetitions by two.

$$T(n) = \sum_{k=0}^{\lfloor \frac{n}{2} \rfloor} \frac{(n-k)!}{k!(n-2k)!}$$

2. *The Dominating Set Problem*: We can solve this problem using dynamic programming. To do so, we notice that the problem has overlapping subproblems. The subproblem is looking at a tree rooted at  $u$ , if  $u$  has children that are in  $S$ , then do not choose it to be in  $S$ , else choose  $u$  to be in  $S$ .
3. (a) The greedy algorithm that minimizes the number of coins returned when producing change for  $n$  cents using quarters, dimes, nickels and pennies follows.

---

**Algorithm 1** Make Minimum Change for  $n$  Cents

---

```
1: procedure MAKE MIN CHANGE
2:  $q = n \bmod \text{Quarter.value}$ 
3:  $n' = n - q * \text{Quarter.value}$ 
4:  $d = n' \bmod \text{Dime.value}$ 
5:  $n'' = n' - d * \text{Dime.value}$ 
6:  $\text{nic} = n'' \bmod \text{Nickel.value}$ 
7:  $p = n'' - \text{nic} * \text{Nickel.value}$ 
8: return  $q$  Quarters,  $d$  Dimes,  $\text{nic}$  Nickels,  $p$  Pennies
9: end procedure
```

---

In a generalized form of the algorithm, we would first sort the coins by their respective values. We then proceed by finding the maximum number of coins we can use at the highest denomination, then the maximum number of coins at the second highest denomination, until we reach the lowest denomination. For any given  $n$  cents, we find the maximum amount of coins of the highest denomination by using the modulus operation.

In order to prove that our greedy algorithm yields an optimal solution, assume that the algorithm does not yield an optimal solution. That is, there exists a better solution than what is produced by our greedy algorithm. Suppose our solution yields  $g$  as the number of coins, where:

$$g = a + b + c + d$$
$$n = a * \text{Quarters} + b * \text{Dimes} + c * \text{Nickels} + d * \text{Pennies}$$

where  $a$ ,  $b$ ,  $c$ , and  $d$  are natural numbers. Then,  $\text{Opt}(n) < g$  must hold for the optimal solution. In comparison to our greedy solution one or more of  $a$ ,  $b$ ,  $c$ , and  $d$  in the optimal solution must be less than the values found in the greedy solution. Suppose we have smaller value for  $a$  in the optimal solution. Then, to still reach  $n$  cents, we must have at least three more lower denomination coins. Similar cases follow if we try to decrease the values of  $b$  or  $c$ . That is, decreasing the number of higher denomination coins will only serve to increase the number of lower denomination coins by at least two. Additionally, we cannot decrease the value of  $d$  without failing to return  $n$  cents. If this is the case,  $\text{Opt}(n) < g$  does not hold, which is a contradiction.

(b) If the available coins are in denominations that are powers of  $c$ , we will still proceed to return change by returning the maximum number of highest denomination coins, followed by the next highest until we reach  $c^0$ . If we consider that we only have two coins of denomination  $c$ ,  $c^0$  and  $c^k$ , for some  $k > 0$ , we can see that the optimal solution would consist of returning the maximum number of the higher denomination  $c^k$  coins, which we obtain by performing the operation  $n \bmod c^k$ , followed by returning the  $c^0$  coin for the remaining balance. If we did not follow this procedure and wanted a solution that was not returned by our algorithm, then we would have to use more coins. In particular, we would have to use  $(c^k.\text{value}) * c^0$  coins, which is not optimal. We can observe the same result when considering more than the two denominations of  $c$ . That is, any other solution will result in the conversion of a higher denomination coin into more than one lower denomination coins, which yields a less than optimal solution.

To formalize the proof, assume that once again we have a solution that was produced by our algorithm that is of the form  $a_k c^k, a_{k-1} c^{k-1}, \dots, a_1 c^1$  and that  $g = a_1 + a_2 + \dots + a^k$ . Assume that  $g$  is not optimal and that  $\text{OPT}(n)$  gives us the optimal solution. Then,  $\text{OPT}(n) < g$  must hold. If this is the case then for the optimal solution one or more of  $a_1 + a_2 + \dots + a^k$  must be smaller. But, in order to make of  $a^i$  smaller, we must use two or more lower denomination coins to achieve the same amount of change. In so doing, the value of  $\text{OPT}(n)$  exceeds that of  $g$ , which is a contradiction.

#### 4. The Scheduling Problem

## 5. *DP Travelling Max Profit*