COT 6405 Analysis of Algorithms
Homework 2

Michael Novo

October 2, 2015

Note: I am new to Latex. I apologize if some things are not quite right.
Discussed with: Nestor Hernandez, Gregory Reis

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, $\phi = \dfrac{1+\sqrt{5}}{2}$

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

   Stategy 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, $\frac{n!}{r_1!r_2!...r_k!}$ 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.

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

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 is in S, then its children are dominated and we add 1 to S. However, if v is not in S then at least one of its children must be in S for it to be dominated. We can then break the problem down by making $DS(v)$ be the minimum dominating set of a tree rooted at v, and $DS'(v)$ be the minimum dominating set when v does not have to be dominated. We can then write a formula to calculate the dominating set:

$DS(v)$ = min (1 + sum(DS'(x)) for each child(v),
$min_{y \in child(v)}$ (1 + sum(DS'(z)) for each child(y) + sum(DS(k) for k $\neq$ y $\in$ child(v)))

The first term in the minimum function signifies that we are selecting the root to be in S, we can then consider each child without the requirement of it needing to be dominated since it is already dominated by the selection of v. The second term is the case when we do not select v to be in S, instead choosing one of its children to be. The next $DS(z)$ call will be looking at the solution for the grandchildren of v. We then must look at each child of v with the criteria of it needing to be dominated.

We then write the recursion: $DS'(v)$ = min (1 + sum(DS'(x)) for each child(v), sum(DS(x)) for each child(v))

From here, we note that the running time should be $O(b|V|)$, where b is an upper bound on the tree branching factor. In the worse case, we would have $O(|V|^2)$

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 mod Quarter.value
3: n' = n - q*Quarter.value
4: d = n' mod Dime.value
5: n'' = n' - d*Dime.value
6: nic = n'' mod Nickel.value
7: p = n'' - nic*Nickel.value
8: return q Quarters, d Dimes, 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:
$g = a + b + c + d$
$n = a$*Quarters + b*Dimes + c*Nickels + d*Pennies

where a, b, c, and d are natural numbers. Then, $Opt(n) < g$ 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, $Opt(n) < g$ 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 $n \; mod \; c^k$, 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 $a_k c^k, a_{k-1}c^{k-1}, ..., a_1 c^1$ and that $g = a_1 + a_2 + \ldots + a^k$. Assume that g is not optimal and that $OPT(n)$ gives us the optimal solution. Then, $OPT(n) < g$ must hold. If this is the case then for the optimal solution one or more of $a_1 + a_2 + \ldots + a^k$ 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 $OPT(n)$ exceeds that of g, which is a contradiction.

4. *Processing Parts on Machine*

We can solve this problem by adopting a greedy strategy. We first note that since the processing time must be done serially, the ordering of the processing times has not bearing on our final result if we did not have post-processing time. Then, we notice that the ending time for our parts really depends on the scheduling of parts based on post-processing time. The idea is to begin with the part that has the longest post-processing time so that it gets the biggest "head start" that it can. We can sort the jobs by their corresponding q_i values in $O(nlogn)$ time.

To prove that this method works, let us prove by induction. First assume that the algorithm works for the case when $n = k$. Then, we need to show that it also works for $n = k + 1$. If we look at two parts, J_i and J_l and know the processing times $q_i \geq q_l$, then we know that J_i followed by J_l must finish before the J_l followed by J_i. Let the post-processing times be p_i and q_l. Then, we have:

For J_i followed by J_l:
$p_i + \max(q_i, p_l + q_l) = p_i + q_i$ if $q_i > p_l + q_l$
$p_i + p_l + q_l$ otherwise

For J_l followed by J_i:
$p_l + \max(q_l, p_i + q_i) = p_l + q_l$ if $q_l > p_i + q_i$
$p_l + p_i + q_i$ otherwise
Because we know that $q_i \geq q_l$, we see that the sequence that gives us the shorter duration is indeed J_i followed by J_l.

5. *Graph Travelling Max Profit*

We can proceed to solve this problem with a dynamic programming approach. First we notice, that the naive strategy will be enumerating all the distinct possiblities of tours which would take $n!$ time. Instead, we use the dynamic programming approach that exploits the optimal substructure seen in the problem.

OPT(n) = p(0,i) - c(0,i) - c(i,0) for all i when we do not have any elements in our subset.// min for all vertices for (p(0,j) - c(0,j) + OPT(i-j))

There are 2^{n-1} distinct subsets of n-1 vertices, and it take n to compute the value for all vertices, so we get $2^{n-1} * n^2$ time.