CSCI 570, Summer 2016 Homework 1 Solutions

On the first page of your homework, please write your name and USC ID # clearly. Please do something to indicate which is your last (family) name, such as underlining it. If your submission is multiple pages, you must *staple* them together. These are requirements of every homework assignment this semester.

Each of these questions requires a **dynamic programming** algorithm for the solution; for problems 2 and 3, (1) write out the base case and recurrence expressions; (2) explain (briefly) why their expressions are the correct things to compute; and (3) analyze the running time for the iterative version, explaining any important implementation details. You do not need to actually write out the iterative version for these two.

This homework is due on **July 18**. It is strongly suggested that you attempt problem #1 before attending lecture on **July 12** – that means either tonight or tomorrow morning/early afternoon, if you are reading this when it is passed out in lecture.

1. Suppose you owe someone v cents and must repay using only coins. You wish to use the fewest coins possible. We saw in homework 1 a strategy that worked for U.S. currency but not for every hypothetical combination of coin values. Suppose you were in a country that mints coins with the following values: one cent, 10 cents, 30 cents, and 40 cents.

   (a) Give a dynamic programming algorithm that will determine the minimum *number* of coins necessary to make change for v cents. For this part, the recurrence expression is sufficient.

   Let's define OPT(v) to be the minimum number of coins needed to make change for v cents. The decision we need to make is whether to use a 1 cent coin, a 10 cent coin, a 30 cent coin, or a 40 cent coin. No matter which decision, it's one coin.
   Accordingly, OPT(v) = 1 + min(OPT($v-1$), OPT($v-10$), OPT($v-30$), OPT($v-40$)).
   What are the base cases? OPT(0) = 0, clearly. We can then either modify the recusive algorithm above to never make a call to a negative case (such as never calling OPT($v-10$) unless $v \geq 10$), or define OPT($v < 0$) to be some value such that it will never be selected within the min – ∞ works for this (which is understood here to be shorthand for an `INT_MAX` type constant).

   (b) Write the iterative version of this algorithm and state its running time. $O(v)$ is possible.

   Declare OPT$[-39 \ldots v]$
   for $i = -39$ to -1 **do**
   OPT$[i] = \infty$
   OPT$[0] = 0$
   for $i = 1$ to v **do**
   OPT(i) = 1 + min(OPT($i-1$), OPT($i-10$), OPT($i-30$), OPT($i-40$))
   The running time for this is $O(v)$. The base cases take $O(1)$ to fill in. Each iteration of the main for loop takes $O(1)$, and there are $O(v)$ iterations.

   (c) Write the code that would follow after (b) if your goal were to output the actual coins. Don't worry about the formatting of the output.

```
i ← v
while i > 0 do
  if OPT(i) == 1 + OPT(i − 1) then
    +1 coin of type 1 cent
    i = i - 1
  else if OPT(i) == 1 + OPT(i − 10) then
    +1 coin of type 10 cents
    i = i - 10
  else if OPT(i) == 1 + OPT(i − 30) then
    +1 coin of type 30 cents
    i = i - 30
  else if OPT(i) == 1 + OPT(i − 40) then
    +1 coin of type 40 cents
    i = i - 40
```

2. When their respective sport is not in season, USC's student-athletes are very involved in their community, helping people and spreading goodwill for the school. Unfortunately, NCAA[1] regulations limit each student-athlete to at most one community service project per semester, so the athletic department is not always able to help every deserving charity. For the upcoming semester, we have S student-athletes who want to volunteer their time, and B busses to help get them between campus and the location of their volunteering[2]. There are F projects under consideration; project i requires s_i student-athletes and b_i busses to accomplish, and will generate $g_i > 0$ units of goodwill for the university.

   Use dynamic programming to produce an algorithm to determine which projects the athletic department should undertake to maximize goodwill generated. Note that each project must be undertaken entirely or not done at all – we cannot choose, for example, to do half of project i to get half of g_i goodwill. Give the running time of your algorithm. For full credit, your algorithm should have runtime $O(FBS)$.

   Define OPT(i, j, k) to be the optimal goodwill generated that considers only the first i projects and uses at most j busses and k student-athletes. As a base case, OPT[0, *, *] = 0.
   For the general case, we have a two-way decision: do we do project i or not? If $s_i > j$ or $b_i > k$, we cannot do project i and must default to "no," and our total goodwill generated is OPT$(i-1, j, k)$. If we do the project, our goodwill generated is OPT$[i-1, j-b_i, k-s_i]$. If we are able to do the project, we decide whether or not to do it by taking the max of the two possible results.
   There are $O(FBS)$ positions to fill in in the iterative version, each of which takes $O(1)$ time, for a total of $O(FBS)$.

3. Suppose you are organizing a company party. The corporation has a hierarchical ranking structure; that is, the CEO is the root node of the hierarchy tree, and the CEO's immediate subordinates are the children of the root node, and so on in this fashion. To keep the party fun for all involved, you will not invite any employee whose immediate superior is invited. Each employee j has a value v_j (a positive integer), representing how enjoyable their presence would be at the party. Produce an algorithm that will determine which employees should be invited, subject to the above constraints.

[1] Regulations mentioned in this problem are not necessarily accurate to reality and should be considered parody.
[2] The NCAA also won't permit student-athletes to use their own vehicles for these purposes. Furthermore, each bus can only be used for one project.

Let's define OPTT(u) to be the optimal total value obtained by inviting some subset of the employees in the sub-tree rooted at u, including inviting u. Similarly, we'll define OPTF(u) for the same sub-tree, with u being not invited.
The base case is any leaf, with OPTT(u) = v_u and OPTF(u) = 0.
For any non-leaf u, if we include u, we can't include any of its children; accordingly, OPTT(u) is v_u plus the sum of OPTF(w) for all children w.
It's a little trickier if we don't include u, because we can then either include or not include any child w. So, for each child w, we compute the larger of OPTF(w) and OPTT(w) and add these to get the value for OPTF(u).
The value we're looking for is in the two OPT values computed for the root.
This can be implemented in linear time using depth-first search (or any method that fills leafs before it fills in their respective parents).
Alternate solution You can do this with one set, using OPT(u) to be the optimal value obtained from a subtree rooted at u; if u is included, we take its value, plus the sum of the values of the subtrees rooted at each grandchild. If u is not included, we take the sum of the values of the subtrees rooted at the children.