20191011

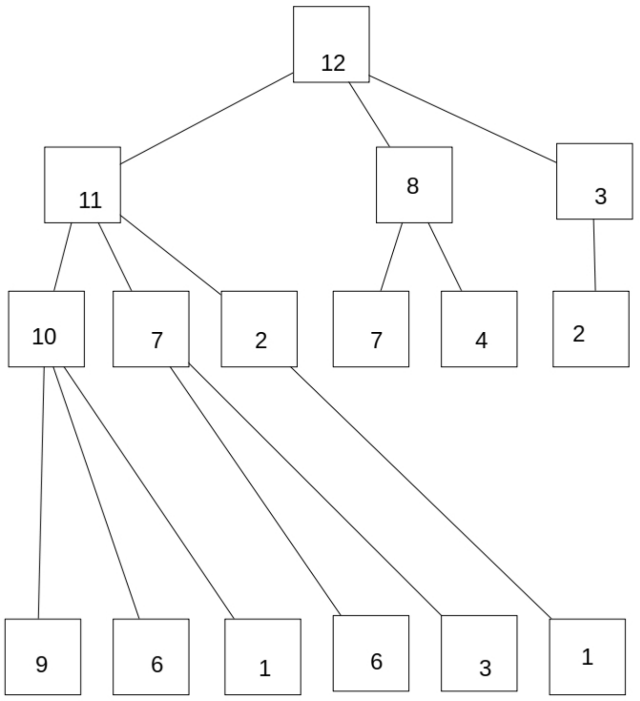
Dynamic Programming

Change Making Revisited

Recall that our Greedy algorithm for change making fails when the set of coin values is {1,4,9} and the target value is 12. We will now develop an efficient algorithm that will find an optimal solution for this (and in fact, any) set of coin values.

(In class we did an example with a larger set of coin values. To keep the discussion short I am going to use just the three-value set {1,4,9})

We can visualize the process of solving this problem using a decision tree. Each decision represents the choice of another coin. If we start with the target value 12, we can choose either a 1, a 4 or a 9. This reduces the target to 11, 8 or 3. The 11 can be reduced to either 10, 7 or 2. The 8 can be reduced to 7 or 4. The 3 can be reduced only to 2. We can continue this, getting a decision tree that looks like this (incomplete):



Each branch of the decision tree ends when the value is reduced to 0. An optimal solution is one with the shortest path from 12 to 0. If we build the entire tree we will find an optimal solution.

This partial tree is enough to permit some important observations:

- there are a limited number of subproblems
- subproblems repeat

These two facts suggest a practical approach to finding an optimal solution. We can solve all the subproblems (there aren't very many) ... and the first time we solve one of the subproblems we can store the result, so we don't need to recompute it when/if we encounter the subproblem again.

Our first step is to look for a recurrence relation that relates the solution to a problem of size n to one or more problems of size $< n$. Once found, we will use this recurrence relation to build up our optimal solution from a base case (or cases).

We can define Min_Coins(n) to be the minimum number of coins need to make a target value of n, when the coin set is

$$\{c_1, c_2, \ldots c_k\}.$$

Our base cases are

Min_Coins(0) = 0

Min_Coins(x) = ∞ if x < 0 (this odd-looking rule makes things easier later)

and the recursive part is

```
Min_Coins(n) = 1 + min{Min_Coins($n - c_1$),
                       Min_Coins($n - c_2$),
                       ...
                       Min_Coins($n - c_k$)
                   }
```

In our example the coin set is $\{1, 4, 9\}$ so the recursive part is

```
Min_Coins(n) = 1 + min{Min_Coins($n - 1$),
                       Min_Coins($n - 4$),
                       Min_Coins($n - 9$)
                       }
```

Now we can see the point in returning "infinity" as the number of coins for negative values. Such a returned value will automatically lose the "min" selection.

For the actual implementation we could certainly code a version of the decision tree and explore it, but a simpler approach is to create an array A with index values from 0 to n.

```
def getval(A,i):
    if i < 0:
        return infinity
    else:
        return A[i]

def Min__Coins(n):
    Set A[0] = 0
    for i = 1 to n:
        A[i] = 1 + min(getval(A,i-1),getval(A,i-4), getval(A,i-9))
    return A[n]
```

Note that this returns the ***size*** of the optimal solution – it doesn't actually tell us which coins to use. We can recover this information easily if we have access to the array A, by working backwards from the end of the solution.

Here is the array A as it starts (the top row is the index):

0	1	2	3	4	5	6	7	8	9	10	11	12
0												

After the first four iterations, A looks like this:

0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	2	3	1								

After four more, this:

0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	2	3	1	2	3	4	2				

And after the final four, this:

0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	2	3	1	2	3	4	2	1	2	3	3

So we know that the optimal solution for $n = 12$ uses 3 coins ... but which coins?

0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	2	3	1	2	3	4	2	1	2	3	3

The 3 for $n = 12$ comes from one of the three highlighted values. In fact it is defined as 1 + the minimum of these values. With the table filled in, we can easily (in O(1) time) find the minimum of these three values – it is the 2 under 8. This tells us that our solution for $n = 12$ is built on the solution for $n = 8$ ... and the action that gets from $n = 8$ to $n = 12$ is "choose a coin with value 4"

0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	2	3	1	2	3	4	2	1	2	3	3

Now we see that the 2 for $n = 8$ comes from either the 1 for $n = 4$ or the 4 for $n = 7$. Of course it comes from the 1 for $n = 4$, and again we see that the decision that gets us from $n = 4$ to $n = 8$ is "choose a coin with value 4"

Finally, we see that the 1 for $n = 4$ comes from the 0 for $n = 0$, with the same necessary decision: "choose a coin with value 4"

Thus in three steps, each with time in O(1), we have reconstructed the details of the optimal solution: choose three "foursies"

In general, the number of steps in this trace-back procedure will be in O(n)

We see that Min_Coins(n) depends on the solution of k smaller problems, where k is the size of the coin set. If k is fixed (it usually is, as in our example) then the value of Min_Coins(i) can be computed in constant time for each value of i ... assuming the smaller problems have already been solved. We can guarantee this by solving all problems from 1 up to n in order - this guarantees that we will have all the required subproblems already solved when we get to solving for n ... which means the algorithm runs in O(n) time to fill the table. As we have seen, we need O(n) to reconstruct the details of the solution. Thus the entire algorithm runs in O(n) time.

Note that the Greedy algorithm for change making runs in O(1) time, as discussed previously in these notes. So the Greedy algorithm will be faster on sets of coin values for which it works ... but the Dynamic Programming solution will work for **all** sets of coin values.