# Lecture Notes, CSE 232, Fall 2014 Semester

Dr. Brett Olsen

## Week 3 - Dynamic Programming

So in the class outline I said that I'd be talking about dynamic programming today. That's still the case, but I want to cover it in the context of alternative approaches. So let's talk about four quite general approaches that we can take to solve problems and when each of them are going to be appropriate.

**Complete Search** The first approach, which we discussed two weeks ago, is called complete search or brute force. To use this approach, we need several things:

- We need to be able to comprehensively enumerate all possible answers

  Say the answer is an integer between 1 and 10,000 (10,000 possible answers) or a permutation of 5 elements (5!=120 possible answers) or a subset of 10 elements (2^10=1024 possible answers). In each case, we can enumerate all of the possible answers.

- We need some way of quickly testing whether each possible answer is correct

Once you've enumerated your answers and written a testing method, the actual code to do the search will be quite straightforward: loop through each possible answer and check if it's correct. If it is, you're done.

```
In []: def solve_problem():
           for answer in all_possible_answers:
               if is_valid(answer):
                   return answer
           return False
```

So, for nearly every problem, these first two points will hold - we can identify all possible ways of solving the problem, and we can test whether any answer is correct. The limit of the feasibility of this approach is the number of possible solutions we must test. In order to use a complete search method, this number must be "small".

The particular number that counts as a "small" number of possible answers is going to depend on the application and the speed of your test, but for a programming competition, a complete search approach is probably worth thinking about if there are less than 10 million or so possibilities to check. This number is important because in the worst case, where the correct answer is the last one you check, this approach will take $O(N)$ time, where $N$ is the total number of possible answers.

When the answer is a positive integer, then that means complete search can work if the maximum value is 10 million.

When the answer is a permutation, then that means it can work if you have no more than 10 elements (10! ~ 0.4 million, 11! ~ 40 million).

When the answer is a subset, it can work for sets smaller than 23 (2^23 ~ 8 million).

And so forth for other types of problems.

If you can feasibly use a complete search method, it's usually a good idea. These solutions are generally much simpler to implement and less bug-prone than solutions using other methods. The other place to use complete search is to test more complicated algorithms. Because it is so easy to make sure these solutions are correct, if you're having problems getting correct answers with a more efficient algorithm, try coding a quick complete search with some simple test cases to check where the bugs are in your other algorithm.

**Binary Search** The second approach, is called binary search. For this method to work, we need:

- An upper and lower bound on the correct answer
- A way to test not only if an answer is correct but also if it is less than or greater than the correct answer

The key to binary search is to use trial answers to narrow the search space until we've homed in on the correct answer.

Let's take Bullseye, the problem on the first week's homework, as an example. In that problem, we wanted to find out how many rings we could draw with a given internal radius $r$ and amount of paint $t$, and we worked out a formula for how much paint it would take to draw $n$ rings: $T(n, r) = n(2n + 2r - 1)$. Our lower bound is 1, as guaranteed by the problem statement, and our upper bound will depend on the input limits. For the Small input limits, we worked out that it is 22.

So suppose we get an input to work with of $r = 1$ and $t = 550$.

We'll start off with a guess of n=11.

We can use the formula to calculate how much paint it would take to draw 11 rings:

```
In [15]: def T(n):
             return n * (2 * n + 1)
         print T(11)
```

```
253
```

We get 253 units of paint, which is less than the total amount of paint we have. That means that 11 must be smaller than or equal to the correct answer. So we can replace our old lower bound of 1 with a new lower bound of 11. In this way, we've cut the total number of possible answers in half. Next, we repeat this, picking a new guess of (11 + 22)/ 2, or 16:

```
In [16]: print T(16)
```

```
528
```

This is still less than our total amount of paint, so our new bounds are $(16, 22)$. The next guess is 19:

```
In [17]: print T(19)
```

```
741
```

19 is too big, so our new bounds are (16, 18). Next we guess 17:

```
In [18]: print T(17)
```

```
595
```

So now we know that we can paint 16 rings but we can't paint 17 rings, so the correct answer is 17. Because we're dividing the search space in half at each step, we can find the correct answer in only 4 tests rather than 17, which is what we would have had to do with a complete search. Because we never have to explicitly test most of the possible answers, and so this gives us an $O(logN)$ solution, much better than an $O(N)$ complete search.

This type of binary search on the correct answer will be useful on a wide range of problems.

**Greedy Methods** For greedy methods to work, we need the problem to have two properties. It must:

- Exhibit "optimal substructure"
- Have the "greedy property"

Let's talk about what those mean.

In order to have optimal substructure, first you need to be able to break up your original problem into smaller subproblems that can be solved independently. And then the correct solution to the large problem must be able to be built from the correct solutions to these subproblems.

Let's look at an example: the coin change problem.

In this problem we are provided with $A$, an amount of money, measured in cents, and $S$, a set of coin denominations, such as $S = 1, 5, 10, 25$, denoting a penny, nickel, dime, and quarter. The problem is given $A$ and $S$ identify how to make $A$ from the fewest number of coins, all in $S$. For example, if we were given $A = 13$, we would use one 10-cent coin and 3 1-cent coins for a total of four coins. Any alternative arrangement, such as 2 5-cent coins and 3 1-cent coins, would use more than the minimum number of coins.

At the beginning, we start with $A = 11$, and we have four different options, because there are four coins we could use. One, the 25-cent coin, is too large, and so we exclude it. But we could use any of the other three: if we used a penny, we would have 10 cents remaining, if we used a nickel 6 cents, and if we used a dime 1 cent. The correct answer for $A = 11$ then would be 1 (the coin we just used) plus the minimum of the solutions for $A = 10$, $A = 6$, and $A = 1$. This is an example of optimal substructure: for any A, we get subproblems with smaller values of $A$, and the correct solution for the larger problem can be made from the correct solution for the smaller problems.

The second property, the greedy property, means that locally optimal choices are also globally optimal. Backtracking is never required. Any time we're asked to make a choice, we can always choose whatever looks best to us at that point and that will lead us to a correct answer. In other words, if a step looks like the best step at any particular point, then you can take it without worrying later that you've made the wrong decision.

If we tried to solve this using a complete search method, we'd have to branch our analysis at each point: each initial problem leads to 4 subproblems, one for each coin denomination, and then those subproblems themselves branch, leading to an exponential number of subproblems to analyze. With a greedy solution, we just ignore all but one of the subproblems, greatly reducing the number of subproblems to analyze and thus making the solution much faster. For this coin counting problem, the greedy method is to simply use the largest coin we can at any particular point. So we'll start off with $A = 11$, the largest coin we can use is 10 cents, so we take a dime and ignore our other choices. And for this particular problem, the greedy method works just fine.

Unfortunately, greedy solutions don't always work. Suppose that instead of $S = 1, 5, 10, 25$, we had $S = 1, 3, 4$. If we tried a greedy apporach on $A = 6$, we would come up with the answer of 3, for 1 4-cent coin and 2 1-cent coins. But we can do better than this, if we used 2 3-cent coins instead. The change in $S$ means that the greedy method no longer works.

So, in general, if a greedy methods, it's usually much faster than any good alternative. Unfortunately, proving that a greedy method works is not always easy. One approach is to use a complete search method on some small problems and using that output to show that the greedy method works on the same small problems, suggesting that it should work on larger problems as well. There's basically no standard way of proving the greedy property for an arbitrary problem. But, we can solve this problem in another way that's still faster than complete search.

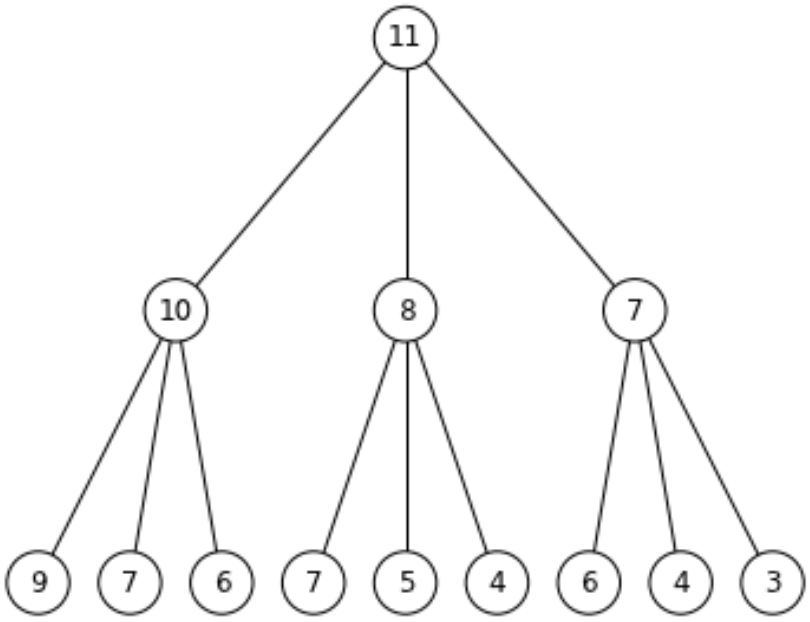
**Dynamic Programming** Dynamic programming solutions require two properties:

- Optimal substructure
- Overlapping subproblems

Let's look at some of the tree of subproblems for $A = 11$ and our new, non-greedy, $S$.

```
In [35]: import networkx as nx

         G = nx.DiGraph()
         labels = {}

         S = [1, 3, 4]
```

```
G.add_node(1); labels[1] = 11
def add_children(node, num, counter=2):
    for i, item in enumerate(S):
        new = num - item
        if new >= 0:
            G.add_node(counter)
            labels[counter] = new
            G.add_edge(node, counter)
            counter += 1

add_children(1, 11, counter)
add_children(2, 10, 8)
add_children(3, 8, 16)
add_children(4, 7, 20)

nx.draw_graphviz(G, prog='dot', node_color='white', labels=labels, node_size=700, arrows=False)
```




So the root is 11, and at each node, we split into three new subproblems, at least until the numbers get small enough. That means we've got an exponential number of subproblems to solve, which will be very slow if we solve them all independently. However, take a look at the numbers that show up on each of the nodes. We know that they must be non-negative integers less than the root, 11. That means that even though there's a large number of subproblems, each of them repeats multiple times. In the tree above, we see 7 three separate times. But there's no reason for us to solve the 7 subproblem more than once - the answer won't change between calculations.

The presence of these overlapping subproblems, where we see the same number show up repeatedly, let's us solve this problem much faster. There are two standard ways of exploiting this substructure. Let's write a quick recursive solution to solve the problem with a complete search:

```
In [46]: A = 11
         S = [1, 3, 4]

         def coin_counting(A, S):
             if A == 0:
                 return 0
             return 1 + min([coin_counting(A - item, S) for item in S if item <= A])

         print ['%i: %i' % (i, coin_counting(i, S)) for i in range(1,12)]
```

```
['1: 1', '2: 2', '3: 1', '4: 1', '5: 2', '6: 2', '7: 2', '8: 2', '9: 3', '10: 3', '11: 3']
```

And we can check how fast this solution is for larger values of $A$ - on my machine, it takes about 100 us for A=11, 430 us for A=14, and 3 ms for A=18, the clear sign of an exponential algorithm.

```
In [49]: %timeit coin_counting(11, S)

10000 loops, best of 3: 101 µs per loop

In [50]: %timeit coin_counting(14, S)

1000 loops, best of 3: 431 µs per loop

In [51]: %timeit coin_counting(18, S)

100 loops, best of 3: 3.01 ms per loop
```

But we've got our answer for 11 of 3. However, as I showed, this is fairly slow because we're re-evaluating many subproblems repeatedly. One way we can avoid doing this is to use a method called memoization, which uses a dictionary to cache answers as we calculate them and return them from the cache rather than reevaluate them when asked for the same answer again, like this:

```
In [53]: A = 11
         S = [1, 3, 4]

         cache = {}
         cache[0] = 0
         def memoized_coin_counting(A, S):
             if not A in cache:
                 minval = 1 + min([memoized_coin_counting(A - item, S) for item in S if item <= A])
                 cache[A] = minval
             return cache[A]

         print ['%i: %i' % (i, memoized_coin_counting(i, S)) for i in range(1,12)]
```

```
['1: 1', '2: 2', '3: 1', '4: 1', '5: 2', '6: 2', '7: 2', '8: 2', '9: 3', '10: 3', '11: 3']
```

```
In [55]: %timeit memoized_coin_counting(50, S)

10000000 loops, best of 3: 165 ns per loop
```

This way, we can get the solution for much larger sums orders of magnitude faster.

The other way of doing dynamic programming is to iteratively loop through all possible subproblems in order of increasing size, so that by the time we reach the subproblem we want, we've already calculated the answers to all of the other subproblems.

```
In [57]: A = 11
         S = [1, 3, 4]

         def DP_coin_counting(A, S):
             answers = [0]
             for i in range(1, A+1):
                 answers.append(1 + min([answers[A-item] for item in S if item <= i]))
             return answers[-1]
```

```
In [58]: %timeit memoized_coin_counting(50, S)

10000000 loops, best of 3: 162 ns per loop
```

Dynamic programming is a widely useful approach, which you will see very often both later in this course and in real programming contests, often combined with some other interesting problem. Which approach to use depends on the problem itself. If you've already coded up a recursive solution, memoizing the solution is often very easy and gets you a DP solution very quickly. In fact, in Python, I've included a decorator to handle memoization to my solution template which lets me do this in one additional line of code by decorating my solution function. A bottom-up approach of incrementally going through all subproblems is more organized and can sometimes save you space and computation time.

The other useful thing to note about DP problems is that for this problem and many other problems, you can do a lot of computation on one case that is useful for future cases. When I solve $A = 50$, I necessarily solve for all $A < 50$ as well, meaning that if I later want to know one of those answers, if I save the dictionary or list or table that I've stored answers to those subproblems in, I don't have to do any more computation at all - I can just look up the answer.

The key thing that's required when you want to use a dynamic programming approach is correctly identifying all the subproblems you need to evaluate. In this case it's been integers less than $A$. But in other approaches, there may be two parameters that vary that define your subproblems.

**Lab Section** We're going to change things slightly this week - I know most of you haven't been able to finish the lab problems I've been setting during class, so I'm going to instead get you started on the homework problems during lab section. The problem this week is from one of the Google Code Jam rounds:

- GCJ 1C 2009 - Bribe the Prisoners