This is a cheat sheet from the Coding for Interviews Udemy course (psst, first 9 readers). You can find this and other cheat sheets at the full course:

https://www.udemy.com/programming-code-interview/

Dynamic Programming

"Dynamic programming"... sounds scary! How come? Probably because it's a general method to approach solving a problem, not so much a specific data structure or algorithm that's easy to find how to wrap your head around.

Practice problems are a great way to learn dynamic programming, so once you get a sense for the general approach, jump in to those!

What is Dynamic Programming?

Dynamic programming is considered a very powerful (some say a *sledgehammer*) problem solving technique.

To build a dynamic-programming-style algorithm, your code will solve smaller versions of a problem and **store the intermediate results**—remembering its past work. You then **build on top** of these intermediate results, normally in a simple and straightforward step, to get the next set of intermediate results.

If you're thinking the DP sub-problem breakdown sounds a lot like the **bottom-up breakdowns** for recursion, that's because *it is*! While you can technically solve dynamic programming problems using either a top-down or bottom-up approach, typically "dynamic programming" refers to the bottom-up problem approach.

*Note: the term dynamic programming **language** is different from dynamic programming. Dynamic in that context means that many things are evaluated at runtime rather than compilation time.*

Memoization (DP for top-down problems)

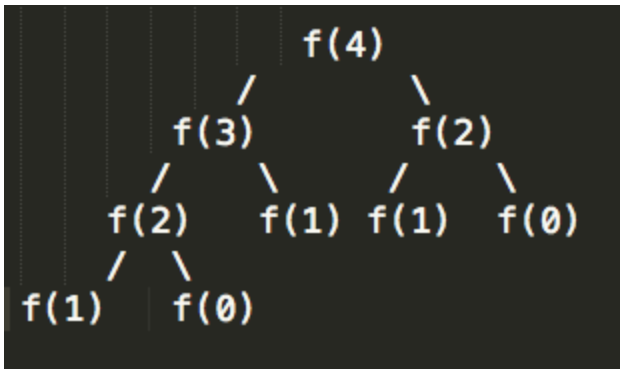
Consider a simple example of our fibonacci function—the same one we covered in our recursion review.

```
def fibonacci(n):
    if n < 2: return n
    else: return fibonacci(n-1) + fibonacci(n-2)
```

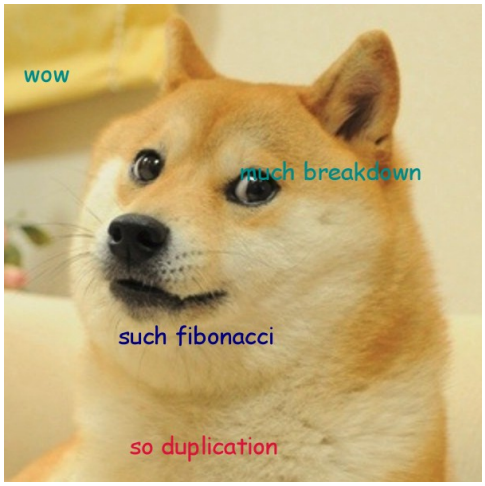
This is a cheat sheet from the Coding for Interviews Udemy course (psst, first 9 readers). You can find this and other cheat sheets at the full course:

https://www.udemy.com/programming-code-interview/

Let's consider how this gets evaluated for input 4:



Such duplication!



For larger numbers, this duplicate evaluation really adds up. It becomes an O(2^n) algorithm—we double our number of calls every step.

One solution to this top-down specific issue is to cache input/output pairs from your function. This is known as **memoization**.

In this example of a **memoized top-down fibonacci**, we store the output for each input in a fibonacci cache:

```
__fibonacci_cache = {}

def fib(n):
    if n in __fibonacci_cache:
```

This is a cheat sheet from the Coding for Interviews Udemy course (psst, first 9 readers). You can find this and other cheat sheets at the full course:

https://www.udemy.com/programming-code-interview/

```
        return __fibonacci_cache[n]
    else:
        __fibonacci_cache[n] = n if n < 2 else fib(n-2) + fib(n-1)
        return __fibonacci_cache[n]
```

In certain programming languages there are libraries that will let you mark a function to be **memoizable**—and it will automatically store input/output pairs.

Top-down or bottom-up?

Working through a problem top-down with dynamic programming often ends up being **memoization**.

Using dynamic programming to build up a solution using a bottom-up approach often involves computing a set of intermediate solution values (usually in an *array* [], *dictionary* {} or *n-dimensional array* [][]) and using them to continue building up more and more complete solutions.

How do you recognize a dynamic programming problem?

DP problems can sometimes be recognized by properties of the expected solution: often you're looking for an **optimal** solution and the problem can be naturally broken down into optimal **partial solutions**. Often times you're searching for an **optimal permutation** of the inputs, or an **optimal choice** of certain ingredients. Other times you're enumerating **all possible valid solutions** for a problem.

One thing to watch out for—the **number of sub-problems solved** in a DP solution will be **polynomial** with respect to the input size.

Conclusion

Bottom-up dynamic programming is building up a set of optimal solutions to subproblems and using them to find optimal solution to your own problem. Top-down dynamic programming is often known as memoization.

Try some practice problems! The best way to learn dynamic programming is by attempting and, if that fails, implementing solutions you find to common dynamic

This is a cheat sheet from the Coding for Interviews Udemy course (psst, first 9 readers). You can find this and other cheat sheets at the full course:

https://www.udemy.com/programming-code-interview/

programming problems. Once you get the pattern, it becomes much more straightforward to approach these problems.

Resources

1. CMU lecture on dynamic programming, including a solution to the longest common subsequence problem
2. Virginia Tech lecture slides on dynamic programming
3. Minimum edit distance problem statement from Stanford NLP group
4. Minimum edit distance on StackExchange
5. Minimum edit distance solutions from WikiBooks