CS 42—Optimization, Part 1

Thursday, November 1, 2018

Summary

Today, we'll learn about a couple of optimization techniques: **memoization** and **dynamic programming**. We use these techniques *after* we've developed a clear recursive algorithm; but discover that our algorithm performs redundant work, causing it to be significantly inefficient (i.e., our solution is intractable, even though the problem itself is tractable).

Today, we'll get our first glance at these techniques. Next week, we'll get more practice, plus talk about when we would want to use memoization instead of dynamic programming, and vice-versa.

As usual, we'll pick up some more Python programming features along the way.

Recall: Slicing a sequence

seq[start : end : step]

Some examples:

```
>>> values = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  # or: list(range(9))
>>> values[3:]
[3, 4, 5, 6, 7, 8, 9]
>>> values[3:5]
[3, 4]
>>> values[-1]
9
>>> values[::2]
[0, 2, 4, 6, 8]
```

Recall: List comprehensions

```
# map
data = list(map(lambda n: n * 2, values))

# equivalent list comprehension (preferred)
data = [n * 2 for n in values]

# filter
data = list(filter(lambda value: value > 5, values))

# equivalent list comprehension (preferred)
data = [value for value in values if value > 5]
```

Next time: algorithmic optimization techniques

Python dictionaries

A **dictionary** is a collection of **key-value pairs**. We often say that the dictionary "maps" each key to its value. In other languages, a dictionary might be called a "map" (not to be confused with the higher-order function!) or a "hash table".

Python statements to create and use dictionaries

```
# empty dictionary
>>> d = {}

# add a key-value pair to a dictionary
# NOTE: dictionary keys must be immutable values
>>> d[0] = 0
>>> d[1] = 2
>>> d[2] = 4

# look up a value, given a key
>>> d[1]
2
>>> d[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 3

# dictionary comprehensions!
>>> d = {i : 2 *i for i in values}
>>> d[3]
6
```

Memoization

Memoization is an optimization technique that avoids doing redundant work. Memoization often uses a dictionary to store the results of subcomputations. Before computing a subcomputation, a memoized algorithm first checks whether that subcomputation has already been computed. If so, the algorithm returns the stored result.

Here's how memoization works:

1. Design and implement a straightforward algorithm (often as a recursive function *f*).
2. Determine that *f* performs redundant work and can be optimized.
    1. Create an empty "cache", which we'll use as a map: input $\mapsto$ *f*(input).
    2. Given an input I, use the input as a key in the cache, to see if we've already computed the result.
        A. If we've already computed the result, just return the result from the cache.
        B. If we haven't already computed the result:
            a. compute *f*(I)
            b. store the result in the cache
            c. return the result

Note that memoization assumes that *f* does not have side effects.

Dynamic programming with tabulation

Dynamic-programming with tabulation is an optimization technique that avoids doing redundant work. Dynamic programming uses a table to store the results of subcomputations, and computes smaller versions of the problem before larger versions of the problem.

Here's how dynamic programing works:

1. Design and implement a straightforward algorithm (often as a recursive function *f*).
2. Determine that *f* performs redundant work and can be optimized.
3. Using the intuition you gained from implementing the recursive version, design the dynamic-programming (DP) table, by asking the following questions:
    1. **What is the meaning of a cell?** What kind of information is stored in a cell? (i.e., what *is* a subcomputation)?
    2. **How many cells will there be**, for an input of size N? (i.e., how many unique recursive calls / subproblem solutions are needed to solve the full problem of size N?)
    3. **Which cell contains the answer** to the full problem of size N? (i.e., we'll eventually return the value of which cell?)
    4. **Which cells are easy to fill in**, and how do we do so? (i.e., which cells correspond to base cases, and what are their values?)
    5. **How does the value of a single cell depend on the value of other cells?** (i.e., how do the recursive cases work?)
    6. **In what order should we fill in the cells**, so that we can be sure to compute the results for each subproblem *before* it's needed?
4. Write the code
    1. Consider directly returning values for the base case(s).
    2. To compute the "recursive cases", create a table of the appropriate size.
    3. Write code to fill in the base-case cells.
    4. Write code (usually a loop) to fill in the remaining values. The loop should go in the order you determined above, and each iteration of the loop fills in the value of one cell.
    5. Return the value in the result cell.

Next time: more of the same!