CS 42—Optimization, Part 3; Summations

Thursday, November 8, 2018

Summary

Today, we'll close out our discussion of optimization by contrasting memoization and tabulation. We'll consider questions such as: "When should we optimize?", and "When is it better to use memoization over tabulation?"

We'll also examine a new analysis technique—**summations**—that helps us analyze loops.

Thinking critically about optimization

We always want to maximize the "goodness" of our code, for several definitions of "good". Many definitions of "good" can be placed into one of two categories: properties of the code (e.g., its readability or maintainability) or properties of the running program (e.g., its correctness or its use of resources).

We often start by making sure the code is correct and clear. Then, we evaluate whether the program use significantly more resources than necessary. If so, we try to optimize. To optimize code, we often must reduce its clarity (which makes it harder to debug and / or modify in the future) and risk affecting its correctness. To be worth it, we therefore want our optimizations to be significant: to move programs from "intractable" [e.g., exponential such as 2^n] to "tractable" [e.g., polynomial such as $O(n)$ or $O(n^2)$]; or from "tractable" to "no problem" [e.g., $O(1)$, or $O(\log n)$].

For the kinds of problems we see in 42, we usually start with a recursive solution. Then, we analyze the solution, and we determine whether the solution performs redundant work (i.e., computes the results for the same subproblem multiple times). If the analysis yields anything other than "no problem" and / or if the solution performs redundant work, we should consider optimizing, usually by finding a way to trade space for time. Dynamic programming does just that, by saving the results of subcomputations, e.g., via memoization or tabulation.

Memoization *vs* tabulation

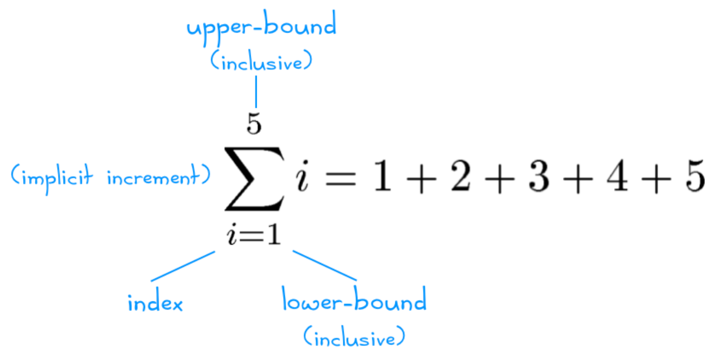
Memoization and tabulation are complementary techniques, with different effects on our code:

	memoization	**tabulation**
preserves clarity of code?	**yes** minimal changes to code	**no** significant code rewrites
easily enables further time optimizations?	**yes** computes *necessary* subproblems	**no and yes** computes *all* subproblems tables can be faster than calls
easily enables further space optimizations?	**no** must store *all* computed subproblems	**yes** possible to store only *some* computed subproblems

In general, we should try memoization before we try tabulation.

Summations

A **summation** is a sequence of values to be summed.



$$\sum_{i=1}^{5} i = 1 + 2 + 3 + 4 + 5$$

Common, simple summations

$$\sum_{i=1}^{N} 1 = \underbrace{1 + 1 + \cdots + 1 + 1}_{N} = N$$

$$\sum_{i=1}^{N} i = 1 + 2 + \cdots + (N-1) + N = \frac{N(N+1)}{2}$$

Converting loops to summations

In Computer Science, we use summations to describe a theoretical model for the cost of a loop.

There is a (mostly) straightforward procedure for translating loops to summations:

1. Choose a cost metric: what are we counting?
2. Work from the "inside out". For each instance of a (possibly nested) loop:
    1. Start with the body of the inner-most loop. Write down the cost of executing the body once.
    2. Write a summation for the loop. First, figure out how many times the loop runs. Then translate the index, lower bound, and upper bound into a summation that corresponds to the code. Watch out for loops that don't increment the index by 1!
    3. Keep working outwards, creating summations for loops, until there are no more nested loops.

Examples

For a cost metric, we'll use the number of times that `quack` is called.

Code	Cost
`platypus.quack()`	1

Code	Cost

`for j in range(N):`
`    platypus.quack()`
	$\sum_{j=0}^{N-1} 1$

Code	Cost

`for i in range(N):`
`  for j in range(N):`
`    platypus.quack()`
	$\sum_{i=0}^{N-1} \sum_{j=0}^{N-1} 1$

Finding a closed form for summations

We can "unroll" the summation, just like we did with recurrence relations. For example:

$$\sum_{i=1}^{N} \sum_{j=i}^{N} 1 \quad \Longrightarrow$$

i	j	cost	total cost for i
1	1	1	N
	2	1	
	...	1	
	N	1	
2	2	1	N-1
	3	1	
	...	1	
	N	1	
...	...	...	...
N	N	1	1

$$\Longrightarrow \quad \sum_{k=1}^{N} k = \frac{N(N+1)}{2}$$

Next time: object-oriented programming