Program Analysis

1. TIME ANALYIS

Today we talk about analyzing the cost of a program—that is, how much time and space it uses. Cost analysis may not seem too relevant to you right now, as you have been mostly struggling with graphics in Java and interfaces.

Bust cost analysis is something that you'll keep in the back of your mind while you program (and while you take other computer sicence classes), and we'll keep coming back to it when we talk about data structures. All decisions when designing a solution for a problem are taken by analyzing the cost.

We are interested in two costs: time and space. The time is the time taken to run the algorithm on some input. The space is the memory used by the algorithm when running on some input.

We could measure the time and space empirically, that is, simply run th eprogram and measure (there are ways to see how much memory a program uses in Java). Both time and space depend on the machine you run the program on, on the size of the input, and on the particular input.

We'd like to have a way to analyze programs without having to implement them. We'd like a model of a computer. Here is a the model that is used to analyse the cost: we assume we run the program on a machine with infinite memory, and all instructions cost the same. In this model, the *cost* or *running time* of an algorithm is the number of instructions.

DEFINITION 1. *The* cost *or* running time *of a program on a particular input is the number of instructions the program executes on that input.*

Usually we denote the size of the input as n, and express the running time as a function of n. Note that, to compare two algorithms, we need to compare them on the same input; otherwise the comparison is irelevant.

On an input of size n the running time may very depending of the particular input. For example, to search an array, the target may be the first element or the last or not found. We are interested in the *worst-case running time*:

DEFINITION 2. *The* worst-case running time *of an algorithm on an input of size n is the maximum number of instructions that the algorithm executes for* any *input of size n.*

The *worst-case running time* is a function of n. We could compute an exact expression for the worst case running time, but instead we are only interested in expressing the *rate of growth* of the time: that is, how fast it grows when n goes to ∞. For a linear search in an array of size n, the running time grows linearly with n. So its order of growth is *linear*.

Formally we express order of growth using big-oh notation: we say that $f(n)$ is $O(g(n))$ if there exists a constant $c > 0$, and n_0 such that $f(n) \leq c \cdot g(n)$ for any $n > n_0$. In other words, f is below g modulo a constant. We say that g is an upper bound for f.

Examples:

—$2n + 4$ is $O(n)$
—$3n + 7$ is $O(n)$
—$2n^2 + 5n + 9$ is $O(n^2)$

In other words, any linear expression in n is $O(n)$. We don't care about the constant in front of the linear term because...they are not accurate anyways (not all instructions are the same, in the real world). So why bother.

So, when we want to know the running time of a program, we are interested in finding the order of growth of the worst-case number of instructions that the program will execute an any input of size n.

Usually the order will fall into one of these classes:

$$O(1), O(\lg n)O(n)O(n \lg n)O(n^2)O(n^3)O(2^n)$$

For large values of n, there is a big difference in speed between an algorithm which is $O(n)$ and one which is $O(n^2)$. Time matters!!!

Examples:

(1) printing an array of size n
(2) summing up all the elements in an array
(3) computing the average of an array
(4) computing the max element in an array
(5) printing an array
(6) searching for a target in an array (linear search)
(7) searching for a target in a sorted array (binary search)
(8) printing an addition table
(9) reversing a string
(10) the two-sum problem: given an array of n elements, find how many distinct pairs add up to zero.
(11) the three-sum problem: given an array of n elements, count how many distinct triples add up to zero.

2. SPACE ANALYSIS

Analyzing the space complexity of a program means estimating how much space (that is, memory), it uses. Of course, the space depends on the size of the input and on the input itself.

The space used by a program is the sum of:

(1) the space for the program itself
(2) the space for its data
(3) the space for the function stack during execution (each method call is put on this stack before its finished; thus nested calls result in all methods on stack).

We'll ignore (1). Typically by space complexity we mean the space of its data, though sometimes we'll talk specifically about the size of the stack.

To estimate the data complexity, its useful to be aware of the sizes of the primitive types: a boolean value takes 1B, one char takes 2B, an int or float take 4B, a long or double take 8B, and so on. Using this we can estimate the size of a complex object as the sum of the space for its instance variables.

We are interested in the worst-case space usage for an input of size n. As for time, we don't want to compute the space precisely, but we are happy with its order of growth.

3. CONCLUSION

How can you analyze the cost if your program?: find the $O()$ of its worst-case running time.

What if the program is not fast enough? Understand why. Buy a faster computer. Come up with a better algorithm... It's all about algorithms. This world revolves around algorithms. Well, almost.

Reading

—online notes by Sedgewick, Wayne at Princeton

—Bailey Chapter 2 (w/o recursion)