A Quick Guide to Big Oh

Professor Don Colton

Brigham Young University Hawaii

1 What Are We Trying To Do?

The amount of collected data in the world gets bigger every day. Credit card transactions. Log files. Web hits. Customer lists. Bigger. More.

One programming challenge is to build systems that do not fall to their knees under the weight of higher speeds and bigger transaction counts. We want systems that are robust, systems that degrade gracefully rather than collapse and melt down.

"Big Oh" analysis is that branch of computer science that measures program performance by simply looking at the program itself. There is a lot one can tell by looking at the code. We cannot easily measure the actual speed in seconds, but we **can** tell whether doubling the input will double the running time, or whether things will be worse or better.

There are some algorithms that are more work to program, but give a faster program. Programmers must choose the best approach to their task, given what they know of the future loads their program must support.

2 Introduction

"Big Oh" is the popular name for running-time analysis of algorithms. It is generally acknowledged that although you can buy more memory or a faster CPU chip, these things will not save you if you are running an inefficient algorithm. Computer Science students learn this material (and more) in their introductory courses. It is helpful for IS students to also have a grasp of the basic terminology and to have the ability to measure (in Big Oh fashion) the running time of various programs.

The words "Big Oh" have reference to "on the Order of," or "Order of magnitude." Specifically it is applied to running times of programs. As the input grows, what happens to the running time of the program?

The phrase "Big Oh" itself is used loosely here. Precisely it means that the algorithm runs at least that fast. Theta (Θ) is a more precise term used in Computer Science, but we will use somewhat less precise but much more familiar terminology, making "tight big oh" technically equivalent to "theta."

3 n Times As Much Input

We want to know what happens to the running time of our program if we get n times as much input data. For each algorithm, just by looking at the program code, we can come to some reliable conclusions. We imagine n to be really large. Thousands. Millions. Billions.

3.1 Typical Algorithms

In this section we talk about some typical algorithms and tell what their big oh running time would be.

3.2 Constant-Time Algorithms

An example of a constant-time algorithm would be one to pick the first number from a list. It does not matter how long the list is. In one step we can pick the first number and then stop.

If we have n times as much input, the running time does not change. Or, it "changes" by a factor of one. When the running time does not change, we say the algorithm is $\Theta(1)$, "theta one," "big oh one," or constant.

A real-life example would be selecting a new employee by taking the first application on the pile. It would not matter how many applications were on the pile.

(On my Big-Oh quizzes, simple statements run in $\Theta(1)$ time. That is why they are called simple.)

Constant-time algorithms are the best possible algorithms, unless that time is very long.

3.3 Linear-Time Algorithms

Let's say we want to find the biggest number in a list, when the list is in unpredictable order. To find the biggest number, we must look at each entry in the list.

If we have n times as much input, the running time is n times longer. Such an algorithm is called $\Theta(n)$, "theta n," "order n," or "linear."

Linear algorithms are very common in IS programming, and are generally accepted as being efficient, unless there is a known way to do the job faster.

3.4 Logarithmic-Time Algorithms

Logarithmic algorithms have running times that grow more slowly than the size of the input. Double the input and the running time only gets a little longer. It does not double.

The classic example of a log-time algorithm is binary search. Take the (in)famous "guess my number" game. In this game, I think of a number and you must guess it. On each turn, you make a guess and I tell you whether you are too high, too low, or just right. If my number is between 1 and 100, your first guess may be 50. By guessing 50, you cut in half the number of remaining possibilities. Say my number is 78, but you don't know that. You say 50. I say higher. You say 75. I say higher. You say 88. I say lower. You say 81. I say lower. Each step you narrow the possibilities by half (roughly).

In this game, if we were to double the initial range, making it between 1 and 200, would it take you twice as many guesses? No. One extra guess at the front would determine whether it was above or below 100. From there, we are back to the same original challenge.

If we have n times as much input (meaning n times as many numbers to search), it will take us $\log_2 n$ steps before we get down to the original input size. Algorithms that run in log time are said to have a running time of $\Theta(\lg n)$, "theta log n," "big oh log n," or simply "log n."

3.5 Root-n-Time Algorithms

Root-n algorithms run in time proportional to the square root of n (the input size). An example would be finding whether a number n is prime. To be prime, a number must not be the mathematical result of multiplying too smaller numbers. To find if a number is prime, we can test all the smaller numbers to see if they divide exactly into n. But there is a trick. If n is 101, we can stop when we have tested 10, because if 11 goes in, then $101 = 11 * a$, and a must be smaller than 11. But since we have tested all the numbers smaller than 11, we can quit. Without even trying it, we know 11 could not work. Such an algorithm would have running time $\Theta(\sqrt{n})$, or "root n."

3.6 Exponential-Time Algorithms

If you are just preparing for the quiz, you can skip this section. There are no exponential-time algorithms on the quiz.

Just as logarithmic algorithms are not much affected by a doubling of the input, there are other algorithms that may work well up to a point, but then the running time seems to explode.

The classical example of an exponential algorithm is the "Traveling Salesman Problem" (TSP). This problem is much studied in theoretical computer science. The task is simple. Given n cities, a traveling salesman must visit each exactly once before returning home. The goal is to do it the fastest possible way (or cheapest or shortest). Under the most general assumptions, the only way known to reliably solve the problem is to look at every possible route and then pick the best one. There is no known way to eliminate a meaningful proportion of the routes without checking each one.

How many routes are there? n factorial. That is, n possibilities for the first visit, and $n - 1$ for the second visit, until eventually there is just one city left for the last visit.

We like to stay away from algorithms that are exponential. Instead we invent "heuristics" which are shortcuts that tend to give good results but are not guaranteed to be the absolute best. A heuristic for TSP might be: go next to the nearest unvisited city. Or, link up the closest pair of cities. Then link the next closest pair of cities. Good heuristics can be rather tricky, but the payoff is a programming solution that you can use before the salesman dies of old age.

We say that exponential algorithms run in $\Theta(e^x)$ or exponential time. There are substantial differences between exponential algorithms, but we will leave that discussion for the CS students.

4 Loops

The running time of a simple loop (nothing but simple statements inside it) depends on how many times the loop will execute. We will look at several simple cases.

4.1 Counting Up to n

The most common case is a loop whose index starts at one (or zero) and counts by ones up to some limit n. This is a $\Theta(n)$ loop, the most common type of loop.

4.2 Counting Down from n

Another common case is a loop whose index starts at n and counts by ones down to some set limit, usually one or zero. This is also a $\Theta(n)$ loop, (still) the most common type of loop.

4.3 Add/Subtract any Constant

Whether you count up (add) or down (subtract), and whether you count by ones or fives or tens, the result is still the same. Those factors do not affect the running time of the loop. It is still a $\Theta(n)$ loop.

4.4 Multiplying or Dividing

If you multiply by a constant greater than one, your running time will be $\Theta(\lg n)$. That is, your index starts at one, then doubles each time until you reach or exceed n. It does not matter whether you double each time, or multiply by three each time (or four or ten or one hundred). The running time is still log n.

Similarly, if you start at n and count down by dividing by two or three or ten at each step, stopping when you reach one (or ten or one hundred), the running time is also log n.

4.5 Unusual Limits

Watch especially for this one variation on the limit: $i*i < n$. In this case, we are running a loop where i starts at one, for instance, and steps up by a constant while $i*i < n$. This loop will not run the full n times, but will stop when i reaches $\sqrt{n}$. Thus, it becomes a root-n loop, written $\Theta(\sqrt{n})$.

If we step up or down by multiples, then the $i*i < n$ limit has no special effect. It would theoretically be $\Theta(\lg \sqrt{n})$, but mathematically this is still the same as $\Theta(\lg n)$.

5 Combinations of Algorithms

When we have a $\Theta(n)$ loop (block) buried inside another $\Theta(n)$ loop (block), the effects are multiplied. The total running time becomes $\Theta(n^2)$, or "n squared." An example would be comparing two unsorted lists to see if the same item is present in each list. We might take the first item from list one and compare it to each item in list two. That would time order n time. Then we repeat for the next item in list one. As we go through all n items in list one, we have $n \times n$ or n^2 comparisons. If we double the inputs, it takes us four times as long to complete the task.

5.1 Sequences of Statements

For a sequence of statements (including possibly whole blocks of statements), we take the worst running time among the statements.

For instance, a log n block followed by a linear block would have an overall running time that is linear. The effects of the log n loop just vanish. They are too small to worry about. There is an old saying in English: Take care of the dollars and the pennies will take care of themselves.

Simple statements run in $\Theta(1)$ time. That is why they are called simple. A series of however many simple statements still runs in $\Theta(1)$ time.

5.2 If-Else Constructs

For if-else constructs, we always assume the worst case when we are not sure what will happen. The worst case for an if-else construct is that it will do either the if side, or the else side, whichever one is worse. For practical purposes, this behaves the same as if we did both sides (see "sequences of statements" above).

5.3 Worst Running Time

In selecting the worst running time, we can follow two simple rules.

(1) If the running time includes a power of n, like n^2 or $n^{\frac{1}{2}}$ (which equals $\sqrt{n}$), then the block with the higher power of n is worse.

(2) If the powers of n are the same (or there are no powers of n), the block with more logs is worse.

For example, in comparing $n\sqrt{n}\lg n$ to $\lg^3 n$, the first has a power of n of 1.5 (one for n, a half for

$\sqrt{n}$). The second has a power of n of zero. So the first is worse.

5.4 Nested Blocks

When the blocks (typically loops) are nested, we multiply their running times to get the overall running time.

5.5 Recursive Subroutines

If you are just preparing for the quiz, you can skip this section. There are no recursive subroutines on the quiz.

There is a more elaborate analysis that goes on for recursive subroutines. These are subroutines (or functions, or procedures) that call themselves. They are typical of a divide-and-conquer programming approach, where a function foo divides its input into smaller sets and calls itself, foo, on each of those sets.

This topic is covered in Computer Science courses.

6 And the Winner Is ...

In the long run, a program with better running time is a better program. Some programs are not meant to live a long life. They run once or a few times and are permanently retired. For these programs, it does not matter much which algorithm you use (except exponential, which may not even finish once in your lifetime).

For any program that will run possibly many times, maybe for years and years, it is generally worth the extra effort to use the best possible algorithm. A simple algorithm is fast to program and takes longer to run. A more complex algorithm costs more to program, but then it runs faster forever.

It is important for every programmer to be able to do simple kinds of running-time analysis, such as those presented here. It is important to be able to identify a better algorithm by seeing that it has a faster running time.

Beyond that, for more information one should consult a basic book on Computer Science, or take a course in analysis of algorithms, where other aspects of Big Oh analysis are more fully explored.