1 Overview

By the end of this topic we hope you will understand the goals of algorithm analysis. We also aim to equip you with the tools you need to analyse the more common types of algorithms.

2 Motivation

An algorithm is a set of instructions to solve a problem. Almost all coding you'll do (in work, school, play. play?) will require you to design an algorithm to solve some problem. In fact, you use algorithms in almost everything you do. For example, we use algorithms to go grocery shopping, to serve up tennis balls, to sort our laundry, and to drive a car. There are many different ways to do each of these things, some of which are better than others.

For example, a grocery shopping algorithm may be as follows: first, make a list. Bring your list to Wegman's, find the first item on your list and put it in your cart. Then find the second item and put it in your cart, and so on, until you've finished your list. This would accomplish our goal of getting everything we need, but at a cost of making a list and consulting it at regular intervals. Another way to grocery shop would be to make a list, put that list in your pocket, go to Wegman's... wander the ailes aimlessly without ever looking at your list but hopefully picking up what you need as you go along. This is my strategy. The problem with this though is that you usually forget something and have to go back later in the week (or immediately if you've forgotten your candy). But the advantage is that you don't waste time looking at a list. A third strategy is to screw the list altogether and just go to the shop. This has the definite advantage of saving valuable time otherwise wasted making a list, but at the cost of maybe not getting what you want. Everyone is different.

The above example brings up a few issues. There were three algorithms above, and I'm sure we can all name people we know who do each of the three. What makes one better than the other? Only one of them actually is 'correct' in the sense that it guarantees us having everything we want every time. But the other two are faster. And sometimes it's ok if we don't get everything we wanted, and at other times it is more crucial to get everything. For example, on an average day it is ok if I've forgotten to pick up milk for my corn pops ... maybe I'll just have the cereal dry, or have candy for breakfast instead. But if I'm having a bunch of people over for dinner and I've forgotten to pick up the main ingredient of my meal, well, I may have a few unhappy hungry friends (which ain't pretty letmetellyou).

So when you are designing an algorithm, not only do you usually want to design a correct algorithm, but also hopefully a *good* one. How do you know an algorithm is good?

3 How do we know an algorithm is good?

There are many things to consider when designing or choosing an algorithm:

i. **Correctness**: Does the algorithm solve our problem? For some people this part is easy — you can be very inventive in your solution at the cost of being slow and inefficient (I tend to be this way :)

   - In our grocery shopping example, this corresponds to getting each item on our list in order, even if it means going down the same aisle 5 times throughout our grocery shop. But it accomplishes what we came for.
   - As another example, back around 94-95 Intel had a slight calculation/roundoff error on one of its chips. This error, when compounded over many calculations, created huge roundoff error. Now picture a bank using this chip to trade millions of dollars of bonds every few minute. Ouch! (On the other hand, using the chip to calculate your grades probably doesn't make much of a difference as long as it rounds up, right?)

ii. **Speed/Space**: How fast does the algorithm run? How much space (memory) does it use?

- In grocery shopping, this is the issue of taking the time to make the list and how long you spend at the grocery store doing your shopping. If I took extra time making my list so that the items were ordered aisle by aisle then I would be much faster in the store, for example.
- Slow processor example. Can't you think of anything more interesting?! Puhleeeze.

iii. **Ease of Programming**: Is it going to take hours, days, or years to implement? Do we have this time? An easy-to-program algorithm is faster to implement — which saves manpower, money, and frustration. Furthermore, it makes the code easier to read and understand. And finally, elegant code is sleek, impressive, and appreciated!

- Some would say that it's easiest to shop if you have a list telling you what to get. But others would argue that making the list itself is not an easy process. Dilemma! Everyone has their own idea of 'easy'.

iv. **Suitability for the Application**: How often will this algorithm be used in our application? Does the algorithm *need* to be fast? Does it *need* to be accurate? Does it *need* to use little space?

- If you're only going to use the algorithm once every 5 years, it's probably a-OK if it takes an extra 5 seconds to run.
- For example, if I'm only missing one item at home, I probably don't need a list to go to the store, since I probably won't forget why I went in the first place (although it *has* been known to happen!)

4 Why do we want to quantify how good an algorithm is?

i. to compare algorithms

ii. to get a lower bound on time — tells you when to quit trying

iii. to tell us if there even IS a solution and to make big money! See
http://www.claymath.org/prizeproblems/index.htm
to win 1 million buckaroos if you solve the P-NP problem. (we won't cover this in 211, shucks!)

5 How do we quantify space and time?

Recall that an important criteria to a good algorithm is its speed and space requirement. I.e. how long does it take me to grocery shop or serve a tennis ball? In general, it would make sense to

$$\text{minimize} \left\{ \begin{array}{l} \text{time} \\ \text{space} \\ \text{number of processors} \\ \text{number of random bits} \end{array} \right\}$$

So how do we measure these things?

5.1 Measuring Time

To measure time we count the number of operations as a function of the size of the input, n. Strictly, the *size* of the input is the number of bits needed to specify the instance of the problem, but realistically, we relate the size to a parameter of the problem. There are 3 general ways to measure time:

worst case : max of inputs I of size n {time taken by algorithm A on input I}
For example, in the worst case I may have to pick up every single item in the grocery store! How well would my different algorithms do on this input?

best case : min of inputs I of size n {time taken by algorithm A on input I}
For example, what if I didn't need *anything* at all from the grocery store? Which algorithm would be best then?

average case : given a probability distribution on inputs,

$$\sum_{\text{inputs I of size n}} (\text{no. of operations used by A on I}) \times Pr[\text{input } I]$$

For example, how many items do I get on average, and how would my algorithms perform on this average case?

5.2 Measuring Space

- use only the worst-case: max of inputs I of size n {space taken by algorithm A on input I}

6 What are some analysis techniques?

Given an algorithm, how do we calculate the space and time requirements?

i. Asymptotic Notation (Big-O) — used for most algorithms

ii. Worst case/ Average case — used, for example, to analyse sorting & searching

iii. Recurrences — used, for example, to analyse mergesort, binary search

iv. NP-Completeness — covered in a later algorithms course

v. Approximation algorithms — covered in a later algorithms course

Often we use a combination of the above techniques to analyse algorithms, but in this course we will focus on the first three.

6.1 Asymptotic Notation (a.k.a. Order Notation or "Big-O" Notation)

Asymptotic notation is a method of measuring the time of an algorithm as a function of its input size. It is used primarily for the comparison of algorithms. We just want a general, asymptotic estimate for how long the algorithm is going to take. We only care about the *biggest*, fastest-growing term in the expression for execution time.

We interrupt this algorithm broadcast to inform you that your instructor Alexa has just been abducted by aliens and replaced by an evil stunt double. We have taken her to our faraway galaxy of Zxoloq. As ransom, we demand a Snickers bar, a canoe-shaped rock, and Adam Sandler in return for her safe release.

6.1.1 How is it done?

Given an algorithm, we determine its order by

- counting the fundamental operations
  * arithmetic/logical operations count as 1 operation
  * assigment counts as 1 operation (but calculating the RHS is done separately)
  * loops count as (no. of iterations)×(no. of operations per iteration)
  * method invocation counts the number of operations executed by the method before it returns

- ignoring unnecessary details such as
  * additive terms and constant factors (such as assignment statements, etc)
  * constants that depend on implementation or model of computation

6.1.2 No honestly, how is it *really* done? (a.k.a. Tell Me More)

Right, so that was a nice overview, but you're probably still wondering exactly how you would do this stuff. So I'll go over some particular structures and their costs:

Cost of a loop : You need to know

1. number of iterations
2. cost of loop control (i.e. checking if $i \leq 10$) $\leftarrow$ we can usually ignore this, as it's usually small
3. cost of body of loop

You can get a rough bound for the cost of a loop with

$$\text{(no. of iterations)} \times \text{(cost of most expensive iteration)}$$

But a better bound would be

$$\sum_{\text{iterations } i} \text{(cost of iteration } i\text{)}$$

For example, suppose the cost of iteration i is i, and suppose the number of iterations is n. Then

$$\begin{aligned}
\text{rough bound} &= \text{(no. of iterations)} \times \text{(cost of most expensive iteration)} \\
&= n \cdot n \\
&= n^2 \\
&= O(n^2) \\
\text{better bound} &= \sum_{i=1}^{n} i \\
&= \frac{n(n+1)}{2} \\
&= O(n^2)
\end{aligned}$$

They're the same!

As another example, suppose the cost of iteration i is $1/2^i$, and suppose that the number of iterations is n. Then

$$\begin{aligned}
\text{rough bound} &= \text{(no. of iterations)} \times \text{(cost of most expensive iteration)} \\
&= n \cdot 1 \\
&= n \\
&= O(n) \\
\text{better bound} &= \sum_{i=1}^{n} \frac{1}{2^i} \\
&= \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots + \frac{1}{2^n} \\
&\leq \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots
\end{aligned}$$

$$
\begin{aligned}
&= \frac{1/2}{1-1/2} \\
&= 1 \\
&= O(1)
\end{aligned}
$$

They're *not* the same! Interesting...

Cost of a nested loop

- solve loops one at a time
- count the inner loops as you would count any other statement in the outer loop

Example: What is the running time of the following code snippet?

```
for i = 1 to n do
    for j = 1 to m do
        a = 5;
    end
end
```

Well, the inner loop (on `j`) runs m times, and its inside is only a constant $O(1)$ statement, so the inner loop is $O(m)$. The outer loop runs n times. The inside is $O(m)$. Therefore the outer loop is $O(n \cdot m)$.

Cost of recursion

- How do we analyse a recursive call?

- You need to know how many times the method is called recursively, and how much work it does on each call.

For example, what is the running time of preorder or postorder traversal? What about the `List toString()` method, or the `BinaryTree's size()` method? If we have time, we'll talk about this later.

6.1.3 Simplified Summary

The major steps in determining the complexity of an algorithm is:

1. count the number of operations as a function of the input size n.
2. drop all except the leading term and ignore all constant multiples.

6.1.4 Formal Definition

Now that we hopefully have the gist of asymptotic notation, let's get to the math of it. Don't worry too much about the definition, but you *do* need to understand what it means.

We say that $g \in O(f(n))$ if there exists a constant $c > 0$ and $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$, $g(n) \leq c \cdot f(n)$.

In English: Sooner or later, as the input size n gets large, $g(n)$ will always run *slower* than some constant factor of $f(n)$. Here is a graphical representation:

Example 1

Suppose $f(n) = n$ and $g(n) = 5$. If we take $n_0 = 5$ and $c = 1$ then it is true that for all $n \geq n_0$, $g(n) \leq c \cdot f(n)$. Therefore $5 \in O(n)$. In fact, $5 \in O(1)$ (can you prove this?)

Example 2

Suppose $f(n) = n^2 - 4$ and $g(n) = n$. If we take $n_0 = 2$ and $c = 1$ then it is true that for all $n \geq n_0$, $g(n) \leq c \cdot f(n)$. Therefore $n \in O(n^2)$.

Example 3

Suppose $f(n) = n$ and $g(n) = \log n$. If we take $n_0 = 1$ and $c = 1$ then it is true that for all $n \geq n_0$, $g(n) \leq c \cdot f(n)$. Therefore $\log n \in O(n)$.

Excercise

Can you find the n_0 and c to determine whether $f \in O(g)$, where $f(n) = 5^n$ and $g(n) = n$?