Chapter 1

Asymptotic Analysis, Big-O and other pains

1.1 Introduction

When dealing with computer science algorithms, it's not just good enough to create an algorithm that can solve problems, it's also important to ask yourself how efficient it is in terms of two things: time and space.

1.2 Big-O Definition

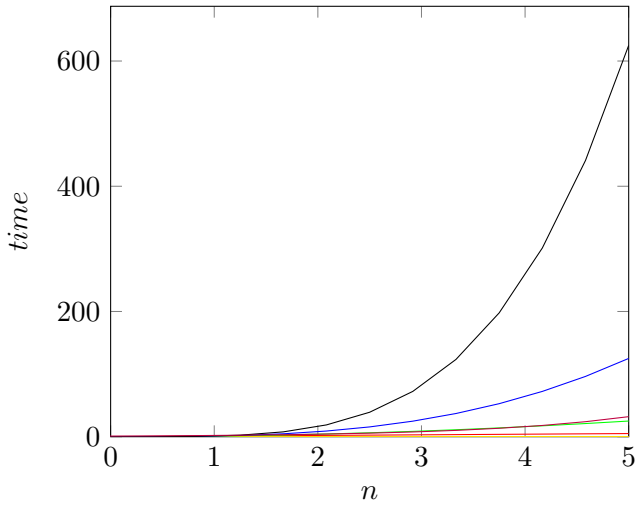
A function $f(n)$ is of order $g(n)$ if and only if for some constants c and $n_0 > 0$ $f(n)$ is less than or equal to $c * g(n)$ for all x greater than or equal to n_0.

$$f(n) \leq c * g(n) \forall\ n > n_0 > 0$$

Common orders of growth organized from fastest to slowest are:

$O(1)$	constant
$O(log(n))$	logarithmic
$O(n)$	linear
$O(n * log(n))$	n log n
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(n^c)$ where c is constant	polynomial
$O(c^n)$ where c is constant	exponential

A simple way to visualize these things are to graph them.



1.3 A Simple Example

Show that $f(n) = n + 2$ is $O(n)$

Find constants c and n_0 such that satisfy the defintion

$$\begin{aligned} n &\leq n \; \forall \; n > 0 \\ 2 &\leq n \; \forall \; n > 2 \\ \therefore n + 2 &\leq n + n \; \forall \; n > 2 \\ \implies n + 2 &\leq 2n \; \forall \; n > 2 \\ \implies f(n) &= O(n) \end{aligned}$$

1.4 Limits and Basic Calculus

Do you remember basic concept behind limits? For example in $\lim_{x\to\infty} \frac{f(x)}{g(x)}$ if f grows faster than g, the answer is infinity. If it is the other way, the answer is 0. If they have the same growth rates, it comes down to a non-zero constant. Let's take a look at our previous example: Show that $f(n) = n + 2$ is $O(n)$

What's $\lim_{n\to\infty} \frac{n+2}{n}$? It's 1! This shows that they have the same growth rate. Let's do this again but take functions with different growth rates:

What's $\lim_{n\to\infty} \frac{n^2}{n}$? It's ∞! This shows that n^2 has a faster growth rate than n.

For this reason, there it's important to remember L'Hopital's rule.

$$lim_{n\to\infty} \frac{f(n)}{g(n)} = lim_{n\to\infty} \frac{f'(n)}{g'(n)}$$

Just take the derivatives of f and g and see if it's easier to find the limit.

1.5 Applying This to Code

Determine the Big-O of this function used to find the element in a sorted array.

```
public static int find_in_array(int[] array, int item) {
    for (int i = 0; i < array.length; i++) {
        if (array[i] == item) {
            return i;
        }
    }
    return -1;
}
```

First, we need to determine how many "steps" this function will take in its worst case. First, we note that n in this case is the length of the array. We see that the function performs several steps n times and then returns -1 when it determines the item is not in the array. Therefore, it could be written as $f(n) = k * n$ where k is the number of steps executed inside the loop, which is a constant number. Leaving the mathematical proof to you, this will end up being $O(n)$.

This is a nice function and all, but remember that I mentioned this is a sorted array; we can do better with binary search.

```
// i dont guarantee the correctness of this function
public static int binary_search(array int[] array, int item) {
    int start = 0;
    int end = array.length;
    while (start < end) {
        int mid = (start + end) / 2;
        if (array[mid] > item) {
            end = mid;
        } else if (array[mid] < item) {
            start = mid;
        } else {
            return mid;
        }
    }
    return -1;
}
```

This is a little harder to determine. What we have to observe is that the size of what has to be searched is halved each time. When you see something like this, it suggests $log(n)$ is in play. Specifically, this method is $O(log(n))$.

Now at this point, you might be wondering what the base of log is; it really doesn't matter. Remember the change of base formula: by multiplying $log_{b1}(y)$ by $1/log_{b1}(b2)$ we get $log_{b2}(y)$, $1/log_{b1}(b2)$ is a constant which eventually gets melded into the constant c in the defintion.

1.6 Recursion

So how will we go about applying this to a recursive function or formula? Let's look at this one for taking a number to a power thats greater than or equal to 0:

```
public static int pow(int base, int power) {
    if (power == 0) return 1;
    else if (power == 1) return base;
    else return pow(base * base, power - 1);
}
```

Note that this time n is power. We can set up our function defintion to match the recursive function:

$$f(n) = \begin{cases} 1, & \text{if } 1 \geq x \geq 0 \\ f(n-1)+2 & \text{otherwise} \end{cases}$$

Note that I add 2 to the recursive case because of the multiplication and subtraction, it won't affect the final answer so don't worry about it too much. What we do is expand it a bunch of times until we see a pattern, I like to keep a counter.

$$\begin{aligned} f(n) = & f(n-1)+2 && i=1 \\ & (f((n-1)-1)+2)+2 && i=2 \\ & f(n-2)+4 && i=2 \\ & (f((n-2)-1)+2)+4 && i=3 \\ & f(n-3)+6 && i=3 \end{aligned}$$

Now, we can see the numbers that are changing depending on i, so we can replace those numbers with i: $f(n-i)+2i$.

We still need to get rid of the function though. We know that it terminates at either $f(0)$ or $f(1)$ so we can get rid of both the function and i when $i = n$. So put that in, which gives $f(n-n) + 2n = f(0) + 2n = 2n + 1$. Now it's a simple job of proving that this is $O(n)$.

As an aside, you can do get the power more efficiently if you take advantage of the fact that $x^n = x^{2^{n/2}}$. This makes the runtime logarithmic.

1.7 Omega, Theta, omega, theta

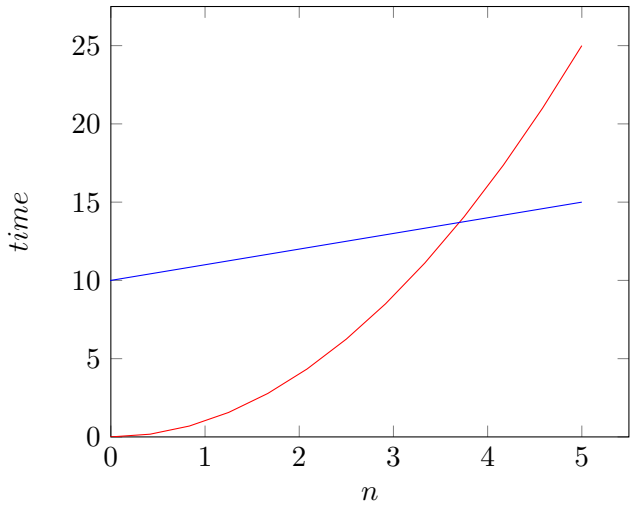
There exist other notations to describe the runtimes of a function in addition to big-O:

O	big-O	$f(n) \leq cg(n)$
o	little-o	$f(n) \leq \epsilon g(n)$
Ω	big-Omega	$cg(n) \leq f(n)$
ω	little-omega	$\epsilon g(n) \leq f(n)$
Θ	Theta	$c_1 g(n) \leq f(n) \leq c_2 g(n)$

Here, ϵ (epsilon) means an arbitrarily small constant that is greater than 0. In other words, for little-o and little-omega, f and g cannot have the same growth rates.

1.8 A Warning

While big-O is great and all, do not blindly and design your functions based on what has the best worst-case runtime. Understanding what the n_0 means is important.



Here we have two functions, $f_1(n) = x^2$ and $f_2(n) = 10 + x$. We know by big-O $10 + x$ is better than x^2. However, if you know your inputs are only up to size 3, it's better to use the function that runs x^2 steps. So if you care about speed, always make sure to benchmark your stuff too.