Problem 1.2-6: How can we modify almost any algorithm to have a good best-case running time?

To improve the best case, all we have to do it to be able to solve one instance of each size efficiently. We could modify our algorithm to first test whether the input is the special instance we know how to solve, and then output the canned answer.

For sorting, we can check if the values are already ordered, and if so output them. For the traveling salesman, we can check if the points lie on a line, and if so output the points in that order.

The supercomputer people pull this trick on the linpack benchmarks!

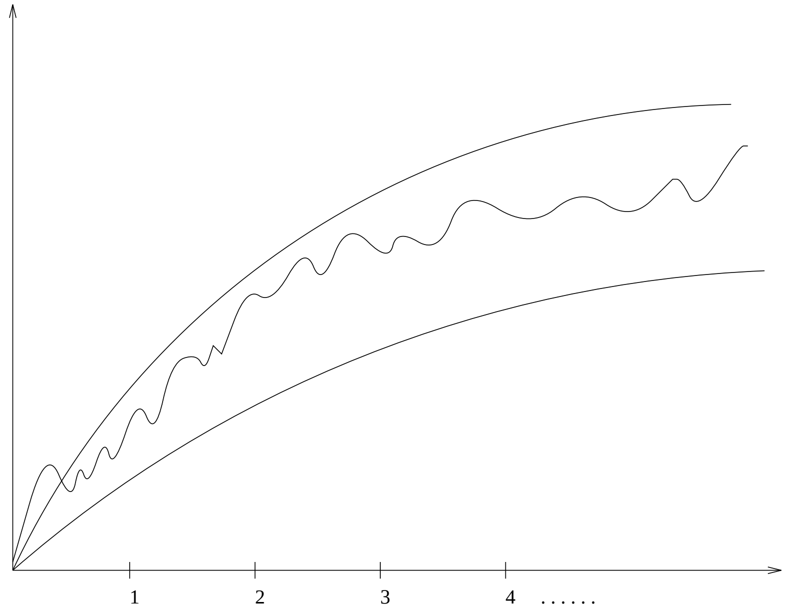
Because it is so easy to cheat with the best case running time, we usually don't rely too much about it.

Because it is usually very hard to compute the average running time, since we must somehow average over all the instances, we usually strive to analyze the worst case running time.

The worst case is usually fairly easy to analyze and often close to the average or real running time.

Exact Analysis is Hard!

We have agreed that the best, worst, and average case complexity of an algorithm is a numerical function of the size of the instances.



However, it is difficult to work with exactly because it is typically very complicated!

Thus it is usually cleaner and easier to talk about *upper and lower bounds* of the function.

This is where the dreaded big O notation comes in!

Since running our algorithm on a machine which is twice as fast will effect the running times by a multiplicative constant of 2 - we are going to have to ignore constant factors anyway.

Names of Bounding Functions

Now that we have clearly defined the complexity functions we are talking about, we can talk about upper and lower bounds on it:

- $g(n) = O(f(n))$ means $C \times f(n)$ is an *upper bound* on $g(n)$.

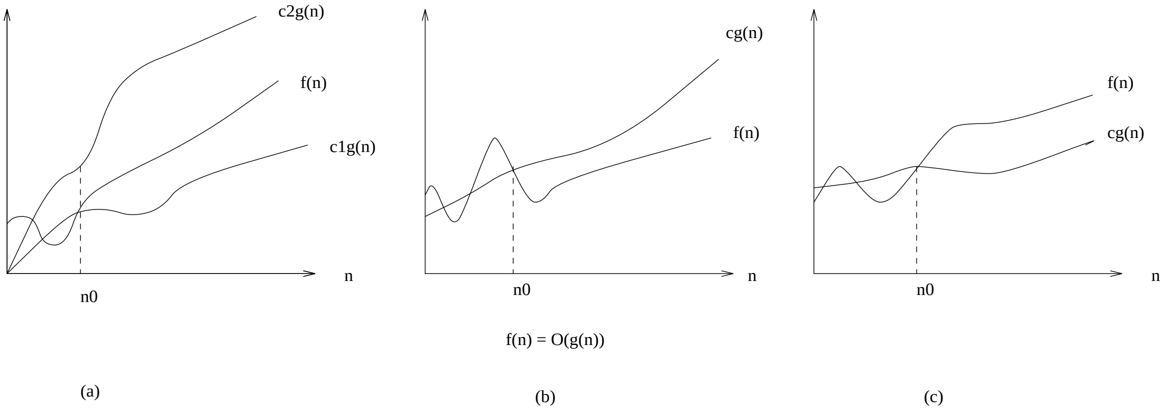
- $g(n) = \Omega(f(n))$ means $C \times f(n)$ is a *lower bound* on $g(n)$.

- $g(n) = \Theta(f(n))$ means $C_1 \times f(n)$ is an upper bound on $g(n)$ and $C_2 \times f(n)$ is a lower bound on $g(n)$.

Got it? C, C_1, and C_2 are all constants independent of n.

All of these definitions imply a constant n_0 *beyond which* they are satisfied. We do not care about small values of n.

O, Ω, and Θ



The value of n_0 shown is the minimum possible value; any greater value would also work.

(a) $f(n) = \Theta(g(n))$ if there exist positive constants n_0, c_1, and c_2 such that to the right of n_0, the value of $f(n)$ always lies between $c_1 \cdot g(n)$ and $c_2 \cdot g(n)$ inclusive.

(b) $f(n) = O(g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or below $c \cdot g(n)$.

(c) $f(n) = \Omega(g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or above $c \cdot g(n)$.

Asymptotic notation (O, Θ, Ω) are as well as we can practically deal with complexity functions.

What does all this mean?

$$\begin{array}{rcl}
3n^2 - 100n + 6 & = & O(n^2) \ \textit{because} \ 3n^2 > 3n^2 - 100n + 6 \\
3n^2 - 100n + 6 & = & O(n^3) \ \textit{because} \ .01n^3 > 3n^2 - 100n + 6 \\
3n^2 - 100n + 6 & \neq & O(n) \ \textit{because} \ c \cdot n < 3n^2 \quad \textit{when} \quad n > c
\end{array}$$

$$\begin{array}{rcl}
3n^2 - 100n + 6 & = & \Omega(n^2) \ \textit{because} \ 2.99n^2 < 3n^2 - 100n + 6 \\
3n^2 - 100n + 6 & \neq & \Omega(n^3) \ \textit{because} \ 3n^2 - 100n + 6 < n^3 \\
3n^2 - 100n + 6 & = & \Omega(n) \ \textit{because} \ 10^{10^{10}} n < 3n^2 - 100 + 6
\end{array}$$

$$\begin{array}{rcl}
3n^2 - 100n + 6 & = & \Theta(n^2) \ \textit{because} \ O \quad \textit{and} \quad \Omega \\
3n^2 - 100n + 6 & \neq & \Theta(n^3) \ \textit{because} \ O \quad \textit{only} \\
3n^2 - 100n + 6 & \neq & \Theta(n) \ \textit{because} \ \Omega \quad \textit{only}
\end{array}$$

Think of the equality as meaning *in the set of functions*.

Note that time complexity is every bit as well defined a function as $\sin(x)$ or you bank account as a function of time.

Testing Dominance

$f(n)$ dominates $g(n)$ if $\lim_{n\to\infty} g(n)/f(n) = 0$, which is the same as saying $g(n) = o(f(n))$.

Note the little-oh – it means "grows strictly slower than".

Knowing the dominance relation between common functions is important because we want algorithms whose time complexity is as low as possible in the hierarchy. If $f(n)$ dominates $g(n)$, f is much larger (ie. slower) than g.

- n^a dominates n^b if $a > b$ since

$$\lim_{n\to\infty} n^b/n^a = n^{b-a} \to 0$$

- $n^a + o(n^a)$ doesn't dominate n^a since

$$\lim_{n\to\infty} n^a/(n^a + o(n^a)) \to 1$$

Complexity	10	20	30	40
n	0.00001 sec	0.00002 sec	0.00003 sec	0.00004 sec
n^2	0.0001 sec	0.0004 sec	0.0009 sec	0.016 sec
n^3	0.001 sec	0.008 sec	0.027 sec	0.064 sec
n^5	0.1 sec	3.2 sec	24.3 sec	1.7 min
2^n	0.001 sec	1.0 sec	17.9 min	12.7 days
3^n	0.59 sec	58 min	6.5 years	3855 cent

Logarithms

It is important to understand deep in your bones what logarithms are and where they come from.

A logarithm is simply an inverse exponential function. Saying $b^x = y$ is equivalent to saying that $x = \log_b y$.

Exponential functions, like the amount owed on a n year mortgage at an interest rate of $c\%$ per year, are functions which grow distressingly fast, as anyone who has tried to pay off a mortgage knows.

Thus inverse exponential functions, ie. logarithms, grow refreshingly slowly.

Binary search is an example of an $O(\lg n)$ algorithm. After each comparison, we can throw away half the possible number of keys. Thus twenty comparisons suffice to find any name in the million-name Manhattan phone book!

If you have an algorithm which runs in $O(\lg n)$ time, take it, because this is blindingly fast even on very large instances.

Properties of Logarithms

Recall the definition, $c^{\log_c x} = x$.

Asymptotically, the base of the log does not matter:

$$\log_b a = \frac{\log_c a}{\log_c b}$$

Thus, $\log_2 n = (1/\log_{100} 2) \times \log_{100} n$, and note that $1/\log_{100} 2 = 6.643$ is just a constant.

Asymptotically, any polynomial function of n does not matter:

Note that

$$\log(n^{473} + n^2 + n + 96) = O(\log n)$$

since $n^{473} + n^2 + n + 96 = O(n^{473})$, and $\log n^{473} = 473 * \log n$.

Any exponential dominates *every* polynomial. This is why we will seek to avoid exponential time algorithms.

Federal Sentencing Guidelines

2F1.1. Fraud and Deceit; Forgery; Offenses Involving Altered or Counterfeit Instruments other than Counterfeit Bearer Obligations of the United States.

(a) Base offense Level: 6

(b) Specific offense Characteristics

(1) If the loss exceeded $2,000, increase the offense level as follows:

Loss(Apply the Greatest)	Increase in Level
(A) $2,000 or less	no increase
(B) More than $2,000	add 1
(C) More than $5,000	add 2
(D) More than $10,000	add 3
(E) More than $20,000	add 4
(F) More than $40,000	add 5
(G) More than $70,000	add 6
(H) More than $120,000	add 7
(I) More than $200,000	add 8
(J) More than $350,000	add 9
(K) More than $500,000	add 10
(L) More than $800,000	add 11
(M) More than $1,500,000	add 12
(N) More than $2,500,000	add 13
(O) More than $5,000,000	add 14
(P) More than $10,000,000	add 15
(Q) More than $20,000,000	add 16
(R) More than $40,000,000	add 17
(Q) More than $80,000,000	add 18

The federal sentencing guidelines are designed to help judges be consistent in assigning punishment. The time-to-serve is a roughly linear function of the total *level*.

However, notice that the increase in level as a function of the amount of money you steal grows *logarithmically* in the amount of money stolen.

This very slow growth means it pays to commit one crime stealing a lot of money, rather than many small crimes adding up to the same amount of money, because the time to serve if you get caught is much less.

The Moral: *“if you are gonna do the crime, make it worth the time!”*

Working with the Asymptotic Notation

Suppose $f(n) = O(n^2)$ and $g(n) = O(n^2)$.

What do we know about $g'(n) = f(n) + g(n)$? Adding the bounding constants shows $g'(n) = O(n^2)$.

What do we know about $g''(n) = f(n) - g(n)$? Since the bounding constants don't necessary cancel, $g''(n) = O(n^2)$

We know nothing about the lower bounds on $g' + g''$ because we know nothing about lower bounds on f, g.

Suppose $f(n) = \Omega(n^2)$ and $g(n) = \Omega(n^2)$.

What do we know about $g'(n) = f(n) + g(n)$? Adding the lower bounding constants shows $g'(n) = \Omega(n^2)$.

What do we know about $g''(n) = f(n) - g(n)$? We know nothing about the lower bound of this!

The Complexity of Songs

Suppose we want to sing a song which lasts for n units of time. Since n can be large, we want to memorize songs which require only a small amount of brain space, i.e. memory.

Let $S(n)$ be the *space complexity* of a song which lasts for n units of time.

The amount of space we need to store a song can be measured in either the words or characters needed to memorize it. Note that the number of characters is $\Theta(words)$ since every word in a song is at most 34 letters long – Supercalifragilisticexpialidocious!

What bounds can we establish on $S(n)$?

- $S(n) = O(n)$, since in the worst case we must explicitly memorize every word we sing – "The Star-Spangled Banner"

- $S(n) = \Omega(1)$, since we must know something about our song to sing it.

The Refrain

Most popular songs have a refrain, which is a block of text which gets repeated after each stanza in the song:

> Bye, bye Miss American pie
> Drove my chevy to the levy but the levy was dry
> Them good old boys were drinking whiskey and rye
> Singing this will be the day that I die.

Refrains made a song easier to remember, since you memorize it once yet sing it $O(n)$ times. But do they reduce the space complexity?

Not according to the big oh. If

$$n = \text{ repetitions } \times (\text{verse-size} + \text{refrain-size})$$

Then the space complexity is still $O(n)$ since it is only halved (if the verse-size = refrain-size):

$$S(n) = \text{ repetitions } \times \text{verse-size} + \text{refrain-size}$$

The k Days of Christmas

To reduce $S(n)$, we must structure the song differently.

Consider "The k Days of Christmas". All one must memorize is:

> On the kth Day of Christmas, my true love
>   gave to me, $gift_k$
> $\vdots$
> On the First Day of Christmas, my true love
>   gave to me, a partridge in a pear tree

But the time it takes to sing it is

$$\sum_{i=1}^{k} i = k(k+1)/2 = \Theta(k^2)$$

If $n = O(k^2)$, then $k = O(\sqrt{n})$, so $S(n) = O(\sqrt{n})$.

100 Bottles of Beer

What do kids sing on really long car trips?

> n bottles of beer on the wall,
> n bottles of beer.
> You take one down and pass it around
> $n-1$ bottles of beer on the ball.

All you must remember in this song is this template of size $\Theta(1)$, and the current value of n. The storage size for n depends on its value, but $\log_2 n$ bits suffice.

This for this song, $S(n) = O(\lg n)$.

Is there a song which eliminates even the need to count?

> That's the way, uh-huh, uh-huh
> I like it, uh-huh, huh

Reference: D. Knuth, 'The Complexity of Songs', *Comm. ACM*, April 1984, pp.18-24