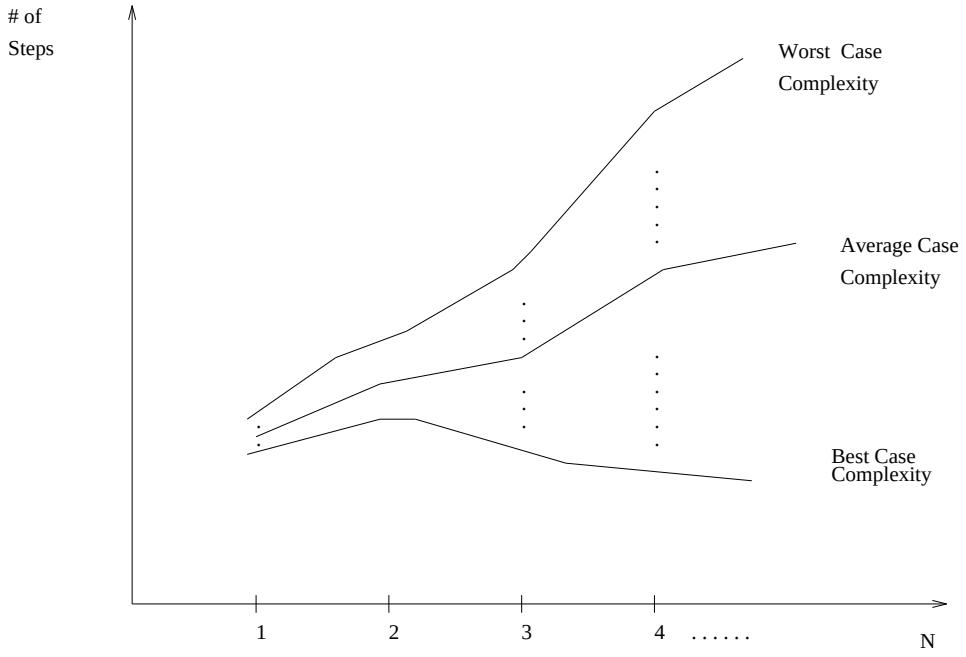
Best, Worst, and Average-Case

The *worst case complexity* of the algorithm is the function defined by the maximum number of steps taken on any instance of size n.



The *best case complexity* of the algorithm is the function defined by the minimum number of steps taken on any instance of size n.

The *average-case complexity* of the algorithm is the function defined by an average number of steps taken on any instance of size n.

Each of these complexities defines a numerical function – time vs. size!

Insertion Sort

One way to sort an array of n elements is to start with a_n empty list, then successively insert new elements in the proper position:

$$a_1 \leq a_2 \leq \ldots \leq a_k \mid a_{k+1} \ldots a_n$$

At each stage, the inserted element leaves a sorted list, and after *n* insertions contains exactly the right elements. Thus the algorithm must be correct.

But how *efficient* is it?

Note that the run time changes with the permutation instance! (even for a fixed size problem)

How does insertion sort do on sorted permutations?

How about unsorted permutations?

Exact Analysis of Insertion Sort

Count the number of times each line of pseudocode will be executed.

Line	InsertionSort(A)	#Inst.	#Exec.
1	for j:=2 to len. of A do	c1	n
2	key:=A[j]	c2	n-1
3	/* put A[j] into A[1..j-1] */	c3=0	/
4	i:=j-1	c4	n-1
5	while $i > 0 \& A[1] > key$ do	c5	tj
6	A[i+1]:= A[i]	c6	
7	i := i-1	c7	
8	A[i+1]:=key	c8	n-1

The **for** statement is executed $(n-1)+1$ times (why?)

Within the **for** statement, " key:=A[j]" is executed n-1 times.

Steps 5, 6, 7 are harder to count.

Let $t_j = 1+$ the number of elements that have to be slide right to insert the *j*th item.

Step 5 is executed $t_2 + t_3 + ... + t_n$ times.

Step 6 is $t_{2-1} + t_{3-1} + ... + t_{n-1}$.

Add up the executed instructions for all pseudocode lines to get the run-time of the algorithm:

$c_1 * n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8$

What are the $t_j's$? They depend on the particular input.

Best Case

If it's already sorted, all t_j's are 1.

Hence, the best case time is

$$c_1 n + (c_2 + c_4 + c_5 + c_8)(n-1) = Cn + D$$

where C and D are constants.

Worst Case

If the input is sorted in *descending* order, we will have to slide *all* of the already-sorted elements, so $t_j = j$, and step 5 is executed

$$\sum_{j=2}^{n} j = (n^2 + n)/2 - 1$$

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!

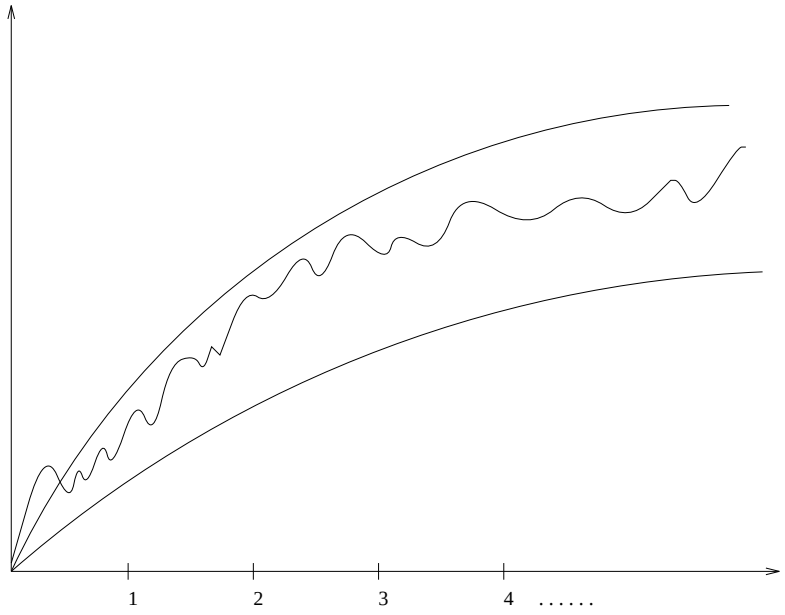
Because it is so easy to cheat with the best case running time, we usually don't rely too much about it.

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

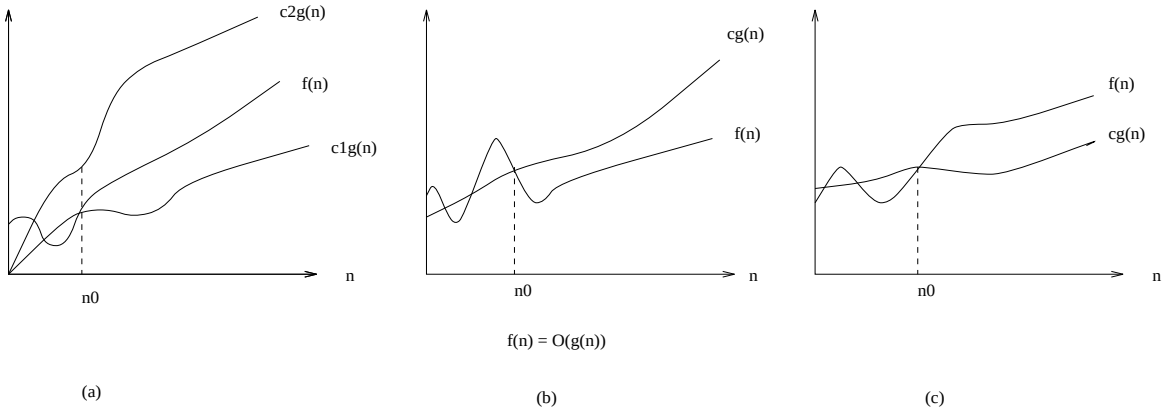
Now that we have clearly defined the complexity functions we are talking about, we can talk about upper and lower bounds on it:

- $g(n) = O(f(n))$ 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?

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

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

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

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.

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!

Show that for any real constants a and b, $b > 0$,

$$(n + a)^b = \Theta(n^b)$$

To show $f(n) = \Theta(g(n))$, we must show O and Ω. *Go back to the definition!*

- *Big O* – Must show that $(n + a)^b \leq c_1 \cdot n^b$ for all $n > n_0$. When is this true? If $c_1 = 2^b$, this is true for all $n > |a|$ since $n + a < 2n$, and raise both sides to the b.

- *Big Ω* – Must show that $(n + a)^b \geq c_2 \cdot n^b$ for all $n > n_0$. When is this true? If $c_2 = (1/2)^b$, this is true for all $n > 3|a|/2$ since $n + a > n/2$, and raise both sides to the b.

Note the need for absolute values.

(a) Is $2^{n+1} = O(2^n)$*?*

(b) Is $2^{2n} = O(2^n)$*?*

(a) Is $2^{n+1} = O(2^n)$?

Is $2^{n+1} \leq c * 2^n$?

Yes, if $c \geq 2$ for all n

(b) Is $2^{2n} = O(2^n)$?

Is $2^{2n} \leq c * 2^n$?

note $2^{2n} = 2^n * 2^n$

Is $2^n * 2^n \leq c * 2^n$?

Is $2^n \leq c$?

No! Certainly for any constant c we can find an n such that this is not true.