Feb 9, 2011

Analysis of Algorithms and the O-notation

We care about efficiency of Algorithms. A good measure is running time which can predict the behaviour of an Algorithm even without implementing it on a specific computer.

Again we usually consider running times of the small parts of programs which are the heart of the programs and not everything (like # of multiplication, additions etc) we define certain parameters and certain measures that are the most important for the analysis (an approximation of the real world)

The main feature (as you have seen in previous classes) is to ignore the constant factors and concentrate on the behaviour of the algorithm when the size of the input goes to infinity.

The size is usually defined as a measure of the amount of space required to store the input and usually will be denoted by n.

Given a problem and a definition of size, we want to find an expression that gives the running time of the algorithm relative to the size usually in the worst case.

The best case is usually ruled out since it is not representative

The average case might be a good choice but usually very hard to measure (even the definition of average is not clear when we have different parameters), also mathematically difficult to analyze)

Also it is hard to predict what will happen in practice (say in a air-plane) and thus worst-case is safer.

The O-notation (ignore constants)
(big O)

A function $g(n)$ is $O(f(n))$ for another function $f(n)$, if there exists constants c and N, such that for all $n \geq N$, we have $g(n) \leq c\, f(n)$. (in otherwords for large enough n, the function $g(n)$ is no more than a constant times the function $f(n)$. The O notation is (like $\leq$) only from the above, though usually we try to find the tightest bound.

For example: $5n^2 + 15 = O(n^2)$, since $5n^2 + 15 \leq 16n^2$ for $n \geq 4$. Also it is in $O(n^3)$ since

since $5n^2+15 \leq n^3$ for all $n \geq 6$.

we always write $O(n)$ instead of $O(5n+4)$, also $O(\log n)$ without specifying the base of the logarithm.

$O(1)$ means constant. $O(n)$ is linear. $O(n^2)$ quadratic.

Though in general $g(n)$ is $O(f(n))$ can be not easy, it is easy for algorithms in this class.

polynomial vs. exponential:

Thm: $n^c = O(a^n)$ for all constants $c > 0$ and $a > 1$.

if we put $n = \log_a m$, we have $(\log_a m)^c = O(a^{\log_a m}) = O(n)$. $(\log_a m)^c$ is called polylog so we prefer polynomials over exponentials (e.g. 2^n) and polylog over polynomial

we can add and multiply using the following rules (but we cannot subtract or divide)

0) if $f(n) = O(s(n))$ then $cf(n) = O(s(n))$ for some constant $c > 0$

1) if $f(n) = O(s(n))$ and $g(n) = O(r(n))$ then $f(n) + g(n) = O(s(n) + r(n))$

2) if $f(n) = O(s(n))$ and $g(n) = O(r(n))$ then $f(n) \cdot g(n) = O(s(n) \cdot r(n))$

Proof is by definition (see the book).

We often try to use (tight) upper bounds on running times of algorithms. which means there is an algorithm with at most this running time. However often we need to say that there is no algorithm that can achieve a better running time. of course it is much harder since we should model every algorithm. the notation for lower bounds is Ω:

If there exists constants c and N, such that for all $n \geq N$ the number of steps $T(n)$ required to solve the problem for input size n is at least $c\,g(n)$, then we say $T(n)$ is in $\Omega(g(n))$. E.g. $n \stackrel{?}{=} \Omega(n^2 - 100)$ and $n = \Omega(n^{0.9})$

The O is corresponding to $\leq$ and Ω is corresponding to $\geq$. what about $=$? $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n)$ is $\Omega(g(n))$.

E.g. $5n \log_2 n - 10 = \Theta(n \log n)$. The constants in O and Ω need not to be the same.

Also note that if $f(n) = O(g(n))$ then $g(n)$ is $\Omega(f(n))$.

what about $<$ and $>$ instead of $\leq$ and $\geq$.

we say $f(n) = o(g(n))$ ($f(n)$ is little oh of $g(n)$) if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

For example $\frac{n}{\log n} = o(n)$, but $\frac{n}{10} \neq o(n)$.

similarly, we say that $f(n) = \omega(g(n))$ if $g(n) = o(f(n))$.

Note that

Thm: $n^c = o(a^n)$ for all constants c and $a > 1$.

Corollary: $(\log_a n)^c = o(n)$

Note that we usually ignore constants in O but sometimes in practice even the constants matters. e.g. in google. [see chap 3 and wikipedia of "Big O notation"

Time and space complexity: Also to hide logarithms, sometime we use $\tilde{O}$; e.g. $\tilde{O}(n \log n) = \tilde{O}(n)$

we analyze an algorithm by counting the number of major steps the algorithm performs.

For example in sorting we count the number of comparisons. in the celebrity problem, we count the number of questions asked. (in a sense we say since we ignore the constant factors, all other operations are only a constant factor of the number of major steps).

In this case we say time complexity or the running time of the algorithm is in $O(f(n))$.

the space complexity of an algorithm indicates the amount of temporary storage (and not the input or output) for running the algorithm. An $O(n)$ space algorithm requires a constant amount of memory per input primitive. A $O(1)$ space needs a constant amount of memory independent of the size of the input.

Give the current amount of disks that we have we mainly focus on running time and not space complexity (though it is important for google).

Now we consider some mathematical techniques for computing running time such as summation, recursion, etc. The proofs are often by Induction.

First, as you have seen the proof and applications:

(★) $S_1(n) = \sum_{i=1}^{n} i = 1+2+\cdots+n = \frac{n(n+1)}{2}$

(★) $S_2(n) = \sum_{i=1}^{n} i^2 = 1+4+\cdots n^2 = \frac{n(n+1)(2n+1)}{6}$. proof by induction in the book (also another way)

(★) $F(n) = \sum_{i=0}^{n} 2^i = 1+2+\cdots+2^n$.

Consider $2F(n) = 2+4+8+\cdots 2^n+2^{n+1}$ (compare $F(n)$ with another expression involving $F(n)$)

so what is the difference of $2F(n)$ and $F(n)$

$2F(n) - F(n) = 2^{n+1} - 1 \Longrightarrow F(n) = 2^{n+1} - 1$.

(★) what about $G(n) = \sum_{i=1}^{n} i\,2^i = 1\cdot 2^1 + 2\cdot 2^2 + \cdots n\cdot 2^n$.

Again the same technique:

$2G(n) = 1\cdot 2^2 + 2\cdot 2^3 + 3\cdot 2^4 + \cdots + n\cdot 2^{n+1}$

Thus $2G(n) - G(n) = n\,2^{n+1} - (1\cdot 2^1 + 1\cdot 2^2 + \cdots + 1\cdot 2^n) = n\,2^{n+1} - (F(n)-1) =$

$n\cdot 2^{n+1} - (2^{n+1} - 2) = (n-1)\,2^{n+1} + 2$.