Where were we….

- Comparing ***worst case*** performance of algorithms.
- Do it in ***machine-independent*** way.
- ***Time usage*** of an algorithm: how many basic steps does an algorithm perform, as a function of the input size.
- For example: given an array of length N (=input size), how many steps does linear search perform?

Rate of Growth

We don't know how long the steps actually take; we only know it is some constant time. We can just lump all constants together and forget about them.

What we are left with is the fact that the time in linear search grows linearly with the input, while in binary search it grows logarithmically - much slower.

Linear vs logarithmic growth

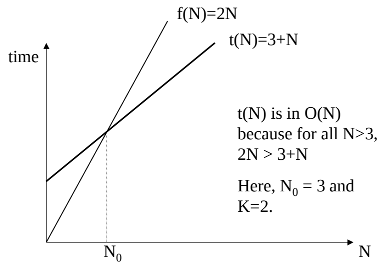
Input size	Linear growth: T(N) = N* c	Logarithmic growth: T(N) = c $\log_2$ N
10	10c	c $\log_2$ 10 = 4c
100	100c	c $\log_2$ 100 = 7c
1000	1000c	c $\log_2$ 1000 = 10c
10000	10000c	c $\log_2$ 10000 = 16c

O() complexity measure

Big O notation gives an asymptotic upper bound on the actual function which describes time/memory usage of the algorithm: logarithmic, linear, quadratic, etc.

The complexity of an algorithm is O(f(N)) if there exists a constant factor K and an input size N_0 such that the actual usage of time/memory by the algorithm on inputs greater than N_0 is always less than K f(N).

Upper bound example



t(N) is in O(N) because for all N>3, 2N > 3+N

Here, N_0 = 3 and K=2.

In other words

If an algorithm actually makes g(N) steps,
(for example $g(N) = C_1 + C_2\log_2 N$)
there is an input size N' and
there is a constant K, such that
for all N > N' , g(N) ≤ K f(N)
then the algorithm is in O(f(N).
Binary search is O(log N):
$C_1 + C_2\log_2 N \le (C_1 + C_2) \log_2 N$ for N > 2

Example

What is the big(O) upper bound on this function:

$C_1 N^2 + C_2 N + C_3$? (assume that C_1 , C_2 , C_3 are positive).

Proof that it is in $O(N^2)$

We need to show that there exist N_0 and K such that

$C_1 N^2 + C_2 N + C_3 \leq K N^2$ for all $N > N_0$.

There are many suitable N_0 and K : for example, $N_0 = 1$, $K = C_1 + C_2 + C_3$. It is easy to check that

$C_1 N^2 + C_2 N + C_3 \leq (C_1 + C_2 + C_3) N^2$

for $N > 1$.

Proof that it is not in $O(N)$

We need to show that there are no N_0 and K such that

$C_1 N^2 + C_2 N + C_3 \leq K N$ for all $N > N_0$.

Suppose there are such N_0 and K (reasoning by contradiction). Then for all $N > N_0$,

$C_1 N^2 + C_2 N + C_3 \leq K N$. Since C_1 , C_2 and C_3 are positive, $C_1 N^2 \leq K N$. Dividing both sides by N, $C_1 N \leq K$. This is a contradiction: it says that no matter how large N becomes, $C_1 N$ is less than some fixed number (thanks to one of the students for suggested simplification).

Comments

Obviously lots of functions form an upper bound, we try to find the closest.

We also want it to be a simple function, such as

constant O(1)

logarithmic O(log N)

linear O(N)

quadratic, cubic, exponential...

Typical complexity classes

Algorithms which have the same O( ) complexity belong to the same *complexity class*.

Common complexity classes:

- O(1) constant time: independent of input length
- O(log N) logarithmic: usually results from splitting the task into smaller tasks, where the size of the task is reduced by a constant fraction
- O(N) linear: usually results when a given constant amount of processing is carried out on each element in the input.

Contd.

- O(N log N) : splitting into subtasks and combining the results later
- $O(N^2)$: quadratic. Usually arises when all pairs of input elements need to be processed
- $O(2^N)$: exponential. Usually emerges from a brute-force solution to a problem.

Practical hints

- Find the actual function which shows how the time/memory usage grows depending on the input N.
- Omit all constant factors.
- If the function contains different powers of N, (e.g. $N^4 + N^3 + N^2$), leave only the highest power (N^4).
- Similarly, an exponential (2^N) eventually outgrows any polynomial in N.

Warning about O-notation

- O-notation only gives sensible comparisons of algorithms when N is large
  Consider two algorithms for same task:
  Linear: $g(N) = 1000\,N$ is in $O(N)$
  Quadratic: $g'(N) = N^2/1000$ is in $O(N^2)$
- The quadratic one is faster for N < 1 000 000.
- Some constant factors are machine dependent, but others are a property of the algorithm itself.

Summary

- Big O notation is a rough measure of how the time/memory usage grows as the input size increases.
- Big O notation gives a machine-independent measure of efficiency which allows comparison of algorithms.
- It makes more sense for large input sizes. It disregards all constant factors, even those intrinsic to the algorithm.

Recommended reading

- Shaffer, Chapter 3 (note that we are not going to use Ω and Θ notation in this course, only the upper bound O()).

Informal coursework

Which statements below are true?

- If an algorithm has time complexity $O(N^2)$, it always makes precisely N^2 steps, where N is the size of the input.
- An algorithm with time complexity $O(N)$ is always runs slower than an algorithm with time complexity $O(\log_2(N))$, for any input.
- An algorithm which makes $C_1 \log_2(N)$ steps and an algorithm which makes $C_2 \log_4(N)$ steps belong to the same complexity class (C_1 and C_2 are constants).