Algorithm Complexity Analysis

Francesco Zanlungo

January 21, 2016

The problem

- When we use an algorithm to solve a problem on a computer, we would like to know how long it will take to execute.
- This depends on many things: the computer, the processor(s), how the program is written, in which language...
- In this course we forget all these problems and focus on the analysis of the algorithm itself, regardless of the details above.

The "solution"

- We look for the algorithm's *basic operation* and compute how many times it is computed.
- Of course we have to make a few basic definitions, first of all we have to define the basic operation itself, and find a function called *time complexity function*, that relates the size of the algorithm's *input* to how many times the operation is performed.
- We obviously need also to be able to properly analyse the result.

Input size

- The size n of an array.
- The size n of a n by n matrix.
- and for a single input?

Basic operation

- There is no unique definition.
- It may be the operation that is performed more times (in the meaning of the theory to be developed).
- Or the most time consuming operation.
- But the better solution is to analyse the complexity of all the operations done by our algorithm.

Complexity function

- It gives the number of operations as a function of the input size n.
- $N = f(n);\ n, N \in \mathcal{N}$
- But is this function uniquely determined by n?
- In general it depends not only on the size n, but on the input itself.

Every case (time) analysis

- For some algorithms the number of operations depends only on the input size.
- In this case we define $N = T(n)$ as the *every-case time complexity function*.
- Examples:
  - simple sort $n(n-1)/2$
  - factorial $n-1$
  - matrix product n^3
  - determinant with minors (no check) $n!$
  - Gaussian elimination $n^3/3 + n^2/2 + n/6$

Worst-case (time) analysis

- In other algorithms (or in different versions of the previous) the number of operations depends on the specific input.
- In this case we may define $N = W(n)$ as the *worst-case time complexity function*.
- Examples:
  - assignment in simple sort
  - sequential search
  - determinants with checks

Best-case (time) analysis

- We may define also $N = B(n)$, the *best-case time complexity function*.
- Examples:
  - assignment in simple sort
  - sequential search
  - determinants with checks

Average-case analysis

- Sometimes we may interested in knowing the worst case.
- The best case is almost always just a curiosity...
- ...unless statistically dominant.
- We may thus introduce an Average-case statistical analysis.
- Examples: search, determinant.

Issues with average

- What is the probability of data?
- Statistical analysis (standard deviation etc.).

Memory complexity

- Tells us how the memory required by the program grows with n.
- It will not be considered in this course.

Analysis using limits

- We may use $\lim_{n\to\infty} f(n)$ to analyse complexity.
- If

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0,$$

defining $h(n) = f(n) + g(n)$ we have

$$\lim_{n\to\infty} \frac{h(n)}{f(n)} = 1.$$

- This means that we may try to define *equivalence classes* between functions for which

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = m, \qquad 0 < m < \infty.$$

Analysis using limits 2

- Using the definition of limit for $n \to \infty$ we may easily see that:
  - if $\lim_{n\to\infty} h(n)/f(n) = m > 1$ (including $m = \infty$) then $\exists N$ such that if $n > N$, $h(n) > f(n)$,
  - and if $\lim_{n\to\infty} g(n)/f(n) = 0$ adding $g(n)$ to $h(n)$ or $f(n)$ does not change this situation.

Examples

- $n^n / n!$
- $n! / e^n$
- e^n / n^m
- $n^m / \ln n$
- $\log_a n / \log_b n$
- etc.

Algorithm complexity notation

- $g(n)$ is O of $f(n)$ or $O(f(n))$ if $\exists\ c > 0$ and N such that if $n > N$ then $g(n) > cf(n)$
- $g(n)$ is Ω of $f(n)$ or $\Omega(f(n))$ if $\exists\ c > 0$ and N such that if $n > N$ then $g(n) < cf(n)$
- $g(n)$ is Θ of $f(n)$ or $\Theta(f(n))$ if $\exists\ b > 0$, $c > 0$ and N such that if $n > N$ then $bf(n) > g(n) > cf(n)$
- $g(n)$ is o of $f(n)$ or $o(f(n))$ if $\exists\ N$ such that if $n > N$ then $\forall c > 0$ $g(n) < cf(n)$

Real world applications

- Taking in account finite n, length of operations, initialisation, etc.