Asymptotic Analysis

Angel Noé Martínez González

June 25, 2015

MOTIVATION I

WHY ANALYSING ALGORITHMS?

- Predict resources an algorithm requires: memory, bandwidth, etc.
- Identify the most efficient algorithm between several others.
- Improve efficiency of an existing algorithm.
- For fun!

MOTIVATION II

HOW DO WE ANALYSE ALGORITHMS?

- Model of the technology: RAM, PC.
- Mathematical tools or analysis (not so damn complicated!).
- Asymptotic analysis, i.e. when n goes to infinity.

WE ONLY CONSIDER TWO IMPORTANT FACTORS:

1. Input size $n \in \mathbb{N}$. Assume n large.
2. Running time $T(n)$: number of primitive operations or "steps" executed for the given input.

GETTING STARTED: THE SORTING PROBLEM I

Given a sequence of numbers $A = [a_1, a_2, ..., a_n]$ as input, generate a permutation of the input such that $a'_1 \leq a'_n \leq ... \leq a'_n$

```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1 .. j − 1].
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

GETTING STARTED: THE SORTING PROBLEM II

ANALYSIS

- Assume each instruction has a constant given time.
- Look how many times the instruction is executed.
- Take the product of the two above.
- The total running time is the sum of all instructions running times.

Getting started: the sorting problem III

Insertion-Sort(A)	*cost*	*times*
for $j = 2$ **to** $A.length$	c_1	n
$key = A[j]$	c_2	$n-1$
// Insert $A[j]$ into the sorted sequence $A[1 \,..\, j-1]$.	0	$n-1$
$i = j - 1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] = key$	c_8	$n-1$

GETTING STARTED: THE SORTING PROBLEM IV

The running time $T(n)$ for insertion sort is

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

Solving recurrences

$$T(n) = (c_5 + c_6 + c_7)\frac{1}{2}n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n - (c_2 + c_5 + c_5 + c_8)$$

Which can be represented as

$$T(n) = a \cdot n^2 + b \cdot n + c$$

WORST AND AVERAGE CASE I

AVERAGE CASE

1. Analyse the average running time of an algorithm.
2. Requires domain knowledge.
3. Analyse for specific inputs.
4. Bit more complex to analyse input distributions, etc.

Worst and Average Case II

Worst case

1. Upper bounds on the running time of an algorithm.
2. Worst case occurs fairly often!
3. Analysis hold for every input.
4. Useful for general purpose algorithms.
5. Mathematically more tractable.

We will focus mainly in the worst case analysis.

Big O Notation I

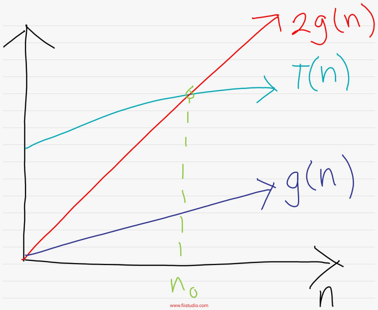
Denotes asymptotic upper bound. For a given function $g(n)$, we denote by $O(g(n))$ the set of functions

$$
\begin{aligned}
O(g(n)) \quad = \quad & \{f(n) : \text{There exists positive constans } c \text{ and } n_0 \\
& \text{such that } 0 \leq f(n) \leq cg(n) \text{ for all } n > n_0\}
\end{aligned}
$$

We say that $T(n) = O(g(n))$ when $T(n)$ is bounded above for a constant multiple of $g(n)$.

Big O Notation II

In other words



$T(n) = O(g(n))$ if and only if exist $c, n_0 > 0$ such that

$$T(n) \leq cg(n)$$

for all $n \geq n_0$ and c, n_0 independent of n.

HOW TO COMPARE? I

DIVIDE AND CONQUER INTRO: MERGE SORT

MERGE-SORT(A, p, r)

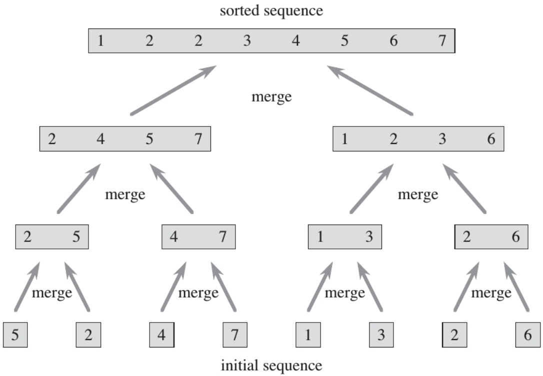
if $p < r$ **then**
    $q = \lfloor (p + r)/2 \rfloor$
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q, r)
    MERGE(A, p, q, r)
end if

MERGE(A, p, q, r)

$B = 1^{st}$ part of array.
$C = 2^{nd}$ part of array.
$i = 1, j = 1$
for $k = 1$ to n **do**
    if $B[i] < C[j]$ **then**
        $A[k] = B[i]$
    else
        $A[k] = C[j]$
    end if
end for

HOW TO COMPARE? II

AN EXAMPLE TREE



At each level $j = 0, 1, ..., log_2(n)$, there are 2^j subproblems of size $n/2^j$.

HOW TO COMPARE? III

Now let's say that subroutine MERGE(A, p, q, r) takes $m \cdot x$, with $x = n/2^j$ and a constant m. Then for each level MERGE(A, p, q, r) costs

$$2^j \cdot m \cdot \frac{n}{2^j} = m \cdot n$$

Then, the running time of MERGE-SORT(A, p, r) is

$$T(n) = m \cdot n \cdot log_2(n) + m \cdot n$$

Finally we get that $O(m \cdot n \cdot log_2(n) + m \cdot n)$ is better than $O(a \cdot n^2 + b \cdot n + c)$.

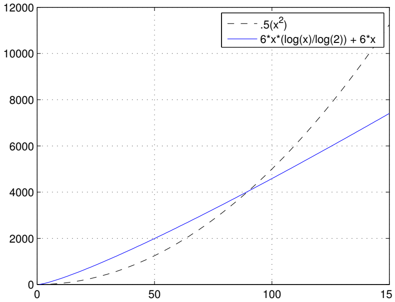
Guiding Principles I

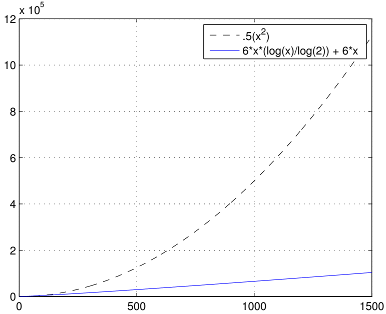
When analysing algorithms

- Don't pay attention in constant factors: dependent on architecture, compiler, programmer, etc.
- Consider each input as equally likely.
- Assume very large input problems, i.e. large n.
- Use the abstraction power of asymptotic analysis: $O(n \cdot log_2(n))$ is better than $O(n^2)$.

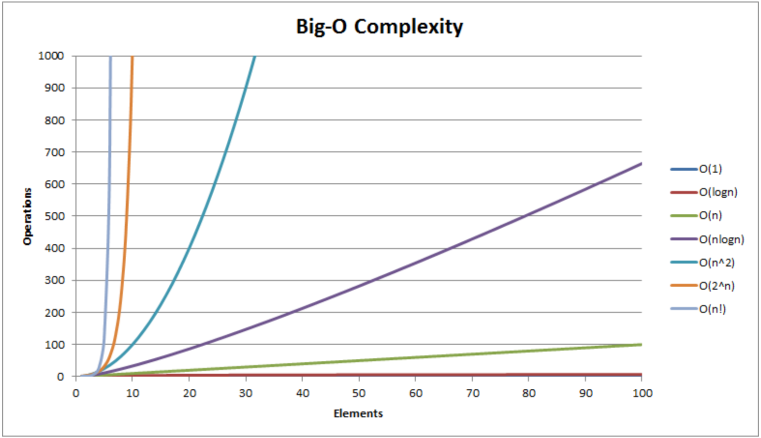
Guiding Principles II

Sometimes look deeper!





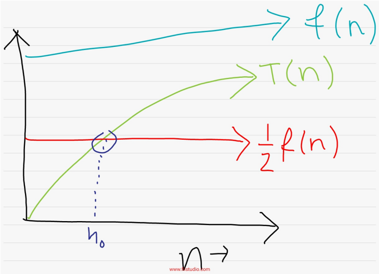
GUIDING PRINCIPLES III



From *http://bigocheatsheet.com/*

BIG Ω AND Θ I

BIG Ω: provides an asymptotic lower bound.



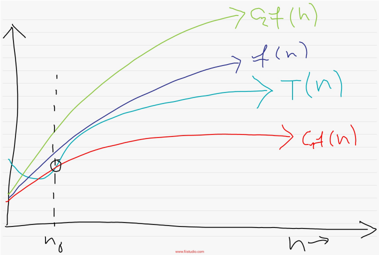
$T(n) = \Omega(f(n))$ if and only if exist constants c and n_0 such that

$$T(n) \geq cf(n)$$

For all $n \geq n_0$.

BIG Ω AND Θ II

BIG Θ: provides asymptotic ranges.



$T(n) = \Theta(f(n))$ if and only if exist constants c_1, c_2 and n_0 such that

$$c_1 f(n) \leq T(n) \leq c_2 f(n)$$

For all $n \geq n_0$.

Note $T(n) = \Theta(f(n))$ implies that $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$.

LITTLE o, ω AND θ I

Little o, ω and θ lose tightness in the bounds.

$$
\begin{aligned}
o(g(n)) &= \{f(n) : \exists\, c, n_0 > 0 \mid 0 \leq f(n) < cg(n) \;\forall\, n \geq n_0 \,\} \\
\omega(g(n)) &= \{f(n) : \exists\, c, n_0 > 0 \mid 0 \leq cg(n) < f(n) \;\forall\, n \geq n_0 \,\} \\
\theta(g(n)) &= \{f(n) : \exists\, c_1, c_2, n_0 > 0 \mid c_1 g(n) < f(n) < c_2 g(n) \;\forall\, n \geq n_0 \,\}
\end{aligned}
$$