Algorithmics

Bruno MARTIN,
University of Nice - Sophia Antipolis
mailto:Bruno.Martin@unice.fr
http://deptinfo.unice.fr/~bmartin/mathmods.html

- Analysis of algorithms
- Introduction to recursive functions
- Some classical data structures
- Sorting
- Searching
- Hashing
- Graph algorithms
- Untractable problems

Some information

10 lectures: this building
Every monday except maybee the week starting apr.23 'till apr.29

- 10h-11h00 lecture
- 11h15-12h15 exercises

Office hours: monday afternoon (please drop a mail)
Mail: mailto:Bruno.Martin@unice.fr
Web: http://deptinfo.unice.fr/~bmartin/mathmods.html
Assignments: one? two? tests and one final exam.

Programming details: Ruby desuka?

Ruby desu: http://www.ruby-lang.org/en/

Install (1.9.3): http://www.ruby-lang.org/en/downloads/

Learn:
http://www.ruby-lang.org/en/documentation/quickstart/
More interactive: http://tryruby.org/

One-page doc: http://ruby.on-page.net/

Your first homework: install it and learn it by yourself

Analysis of Algorithms

Given a problem :

- how do we find an **efficient algorithm** for its solution ?

Once we have found an algorithm :

- how can we **compare** this algorithm **with other algorithms** that solve the same problem ?
- how should we judge **the efficiency** of an algorithm ?

These questions interest both :

- programmers
- computer scientists

Outline
Introduction to the analysis of algorithms
Introduction to recursive functions

Time and Space Complexity

Associate with a problem a size (an integer n) :

- which is a measure of the quantity of input data (via an adequate coding)
  - size of a matrix,
  - size of a file,
  - degree of a polynomial,
  - number of nodes in a graph, ...

The time needed by the execution of an algorithm is expressed as a function of this size and is called the **time complexity**

The space needed by the execution of an algorithm is expressed as a function of this size and is called the **space complexity**

Bruno MARTIN, University of Nice - Sophia Antipolis mailto:Br Algorithmics

Outline
Introduction to the analysis of algorithms
Introduction to recursive functions

Evaluating Algorithms

Algorithms can be evaluated by a **variety of criteria** :

- input/output, disk access, energy consumption...

The most often we are interested in their growth :

- **in time** and **in space**
- when solving **larger and larger** instances of the problem

The input's size (usually n) is one of the main parameters

Bruno MARTIN, University of Nice - Sophia Antipolis mailto:Br Algorithmics

Outline
Introduction to the analysis of algorithms
Introduction to recursive functions

Asymptotic Complexity

Our interest: behavior of the complexity as the size increases ($n \to \infty$) : the **asymptotic complexity**

Determines the **size of problems** algorithmically solvable

When an algorithm processes data of size n in time $c_1 \times n^2 + c_2$ (c_1, c_2 constants) then its time complexity is :
$O(n^2)$ i.e. is in order of n^2 i.e. is proportional to n^2

A function $g(n)$ is said to be $O(f(n))$ if :
there exists constants $c_0 > 0$ and n_0 such that $g(n) \leq c_0 \times f(n)$ for all $n > n_0$

Bruno MARTIN, University of Nice - Sophia Antipolis mailto:Br Algorithmics

Other Orders of Magnitude

A function $g(n)$ is said to be $\Omega(f(n))$ if :
there exists constants $c > 0$ and n_0 such that for all $n > n_0$

$$0 \leq c \times f(n) \leq g(n)$$

A function $g(n)$ is said to be $\Theta(f(n))$ if :
there exists constants $c_1, c_2 > 0$ and n_0 such that for all $n > n_0$

$$0 \leq c_1 \times f(n) \leq g(n) \leq c_2 \times f(n)$$

Performance Analysis of an Algorithm

The O notation gives an "upper bound" to the complexity.
It guides designers in the search of the "best" algorithm

The **goal** of the **study of complexity** is : *if you provide an algorithm with an* **"upper bound" complexity***, and if you can* **demonstrate** *that your* **problem** *has a* **"lower bound" complexity** *and that* **they match***, then you can* **stop searching a better algorithm** *and focus on the implementation*

You often provide an **"upper bound"** by counting and analyzing the frequencies of the statements of the algorithm

Providing a **"lower bound" complexity** is very difficult. One needs to consider an abstract model - Turing machines - and determine which fundamental operations must be performed **by any algorithm** to **solve the problem**

Importance of the Constants

This notation O is meaningful for large values of n
The O notation says nothing about the time complexity when :

- n happens to be less than n_0
- and c_0 is hiding a large amount of "overhead"

For small values of n :

- you'd prefer an algorithm in $O(n^2)$ time complexity
- rather than one in $O(n)$ but with a big constant c_0

$10n^2$ is faster than $500n$ for $n < 50$ and slower if $n > 50$
Constants often hide implementation details - initialisations -

Complexity in the Best, Average and Worst Case

We are interested in the average case : the amount of time a program takes on a **typical input data**

And in the worst case : the amout of time a program takes on the **worst possible input configuration**

Many programs are extremely sensitive to their input data and performance might fluctuate wildly depending on the input

When studying an algorithm it is interesting to evaluate the average and the worst case

But the average case might be a mathematical fiction that is not representative of the actual data

The worst case might be a bizarre construction that would never occur in practice (consider LP for instance)

Moore's Law

Moore's original statement that transistor counts had doubled every year can be found in "Cramming more components onto integrated circuits", Electronics Magazine 19 April 1965:
The complexity for minimum component costs has increased at a rate of roughly a factor of two per year ... Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000. I believe that such a large circuit can be built on a single wafer.

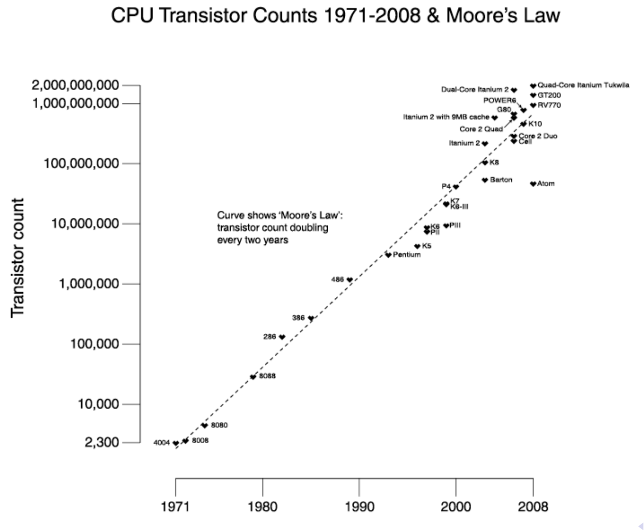
Classifying Algorithm by Performances

The O notation is extremely useful for classifying algorithms by performances. Suppose you have seven algorithms with the following time complexity:

$\log n$	$\sqrt{(n)}$	n	$n \log n$	n^2	n^3	2^n
3	3	10	30	100	1,000	1024
6	10	100	600	10,000	1,000,000	10^{30}
9	31	1,000	9,000	1,000,000	10^9	∞
13	100	10,000	130,000	10^8	10^{12}	∞
16	316	100,000	1,600,000	10^{10}	10^{15}	∞
19	1,000	1,000,000	19,000,000	10^{12}	10^{18}	∞

Even with the Moore law, some algorithms are still intractable

Moore's Law



Other Formulations

in terms of:

- number of transistors per integrated circuit
- cost per transistors
- computing performance per unit cost
- power consumption
- HD storage cost per bit
- ...

Have a look at `http://en.wikipedia.org/wiki/Moore's_law`

Conclusion by Examples of Time Complexities

Suppose that the time complexities are really $1000n$, $100n\log n$, $10n^2$, n^3 and 2^n then:

2^n would be the best for problem of size $2 \leq n \leq 9$
$10n^2$ would be the best for problem of size $10 \leq n \leq 58$
$100n\log n$ the best for problem of size $59 \leq n \leq 1024$
$1000n$ the best for problem of size $1024 < n$

Classification of Algorithms Complexity

- 1 **constant**: all the instructions of a program are executed once or at most only a few times
- $\log n$ **logarithmic**: solve a problem by splitting it into smaller pieces
- n **linear**: a small amount of processing is done on each element
- $n\log n$ **quasilinear**: solve a problem by splitting it in smaller subproblems, solving them independently and then combining the solution
- n^2 **quadratic**: process all pairs of data items (perhaps in a double-nested loop)
- n^3 **cubic**: process all triples of data items (perhaps in a triple-nested loop)
- 2^n **exponential**: a brute-force solution to a problem

Recursive Functions

Recursive functions are quite common in mathematics
In CS, a recursive function is one that **calls itself**
If a recursive function calls itself in any branch : the **definition is circular** and the program **won't stop**
The function must have a **termination condition** to stop calling itself

Should we Use Recursive Functions ?

Recursive expression of programs is often more **simple** and **natural** to write than its iterative counterpart
For example, **simple mathematic recurrence relations** can be expressed easily in simple recursive programs
The recurrence relation of the factorial function is

$$N! = N.(N-1)! \textit{ for } N \geq 1 \textit{ with } 0! = 1$$

The recurrence relation of the fibonacci numbers is

$$U_N = U_{N-1} + U_{N-2} \textit{ for } N > 1 \textit{ with } U_0 = U_1 = 1$$

Recursive Implementation of Factorial –2nd–

```
class Integer
def factoorial
     if self == 0
       1
     else
       self * (self-1).factoorial
     end
   end
end
```

Call with

```
number.factorial
```


Recursive Implementation of Factorial –1st–

```
def factorial(n)
  if n == 0
    1
  else
    n * factorial(n-1)
  end
end
```

The recursive expression of the factorial function is **efficient**
To compute the factorial of n you need $n+1$ recursive calls to the *factorial* function
It has a linear number of recursive calls.

Default

```
[neon:~/Desktop] bmartin% irb
>> load "Factorial.rb"
=> true
>> factorial(7)
=> 5040
>> load "Factoorial.rb"
=> true
>> 7.factorial
NoMethodError: private method `factorial' called for 7:Fixnum
        from (irb):4
>> 7.factoorial
=> 5040
>> 
```

Recursive Implementation of Fibonacci

```
def fib(n)
  if n <= 2
    1
    else
    fib(n-1)+fib(n-2)
  end
end
```

The recursive expression of the fibonacci numbers is simple and natural
Do you think it is **efficient** ?

Executing the Fibonacci Function fib(5)

- Stop of the rec. calls when the corresp. value for `fib` is 1 (leaves in the execution tree). the number of 1's $=$ `fib`(N).
- Thus, the rec. algo. decomposes `fib`$(N)= 1 + \ldots + 1$ and does `fib`$(N) - 1$ sums.

Proposition

There are $\mathtt{fib}(N) - 1$ recursive calls for computing $\mathtt{fib}(N)$.

Thus an **exponential-time** algorithm for the fibonacci numbers since $\mathtt{fib}(N) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^N$ when $N \to \infty$

Executing `fib(5)`

```
                 fib(5)
                /       \
         fib(4)          fib(3)
        /      \          /  \
   fib(3)     fib(2)  fib(2)    fib(1)
   /    \       |       |         |
fib(2)   fib(1) 1       1         1
  |        |
  1        1
```

The recursive calls indicate that `fib(3)` and `fib(2)` should be computed **repeatedly** and you need `fib`(5)-1 recursive calls. (Don't count the leaves).
But in fact you **certainly would use** the computation of `fib(3)` to compute `fib(4)` and decrease the number of **function calls**

Iterative Implementation of Fibonacci

```
def fibonacciter(n)
  t1,t2,t = 1,1,1
  for i in 3..n
    t =t1 + t2
    t2 =t1
    t1 =t
  end
  puts(t)
end
```

This is a linear-time program to compute the fibonacci numbers

Divide-and-Conquer Methods

You can sometimes split your input into two halves and apply the algorithm recursively on each half

It is the divide-and-conquer method

Divide-And-Conquer methods normally lead to more efficient algorithms when the input is divided without overlap

Recursive fibonacci algorithm lead to excessive recomputation because of overlap

Conclusion about Recursive Programs

Recursion should not be used blindly or it might become **not practical** like for the recursive fibonacci algorithm

Don't forget that the recursion depth is stored in the **execution stack**

You must understand clearly the behaviour of your recursive function. But recursion stay a natural and simple way to express algorithms

Complexity

Theorem

Let $a \geq 1$ and $b > 1$ two constant integers, $f(n)$ a function and $T(n)$ inductively defined:

$$T(n) = a.T(n/b) + f(n).$$

An asymptotic bound on $T(n)$ is:

1. $T(n) = \Theta(n^{\log_b(a)})$ *if* $f(n) = O(n^{\log_b(a-\varepsilon)})$ *for* $\epsilon > 0$ *constant*
2. $T(n) = \Theta(n^{\log_b(a)}\log(n))$ *for* $f(n) = \Theta(n^{\log_b(a)})$
3. $T(n) = \Theta(f(n))$ *for* $f(n) = \Omega(n^{\log_b(a+\varepsilon)})$ *and if* $a.f(n/b) \leq c.f(n)$ *for a constant* $c < 1$ *and* n *sufficiently large.*

Or use Mathematica `RSolve` function.