Techniques for analysis of algorithms

Luciano Bononi and Moreno Marzolla
bononi@cs.unibo.it

Department of Computer Science Engineering, University of Bologna

23 aprile 2012

Computation Model

Let's consider a computation model composed of a register-based machine as follows:

- It has a input and a output device;
- The machine has N memory locations, addressed from 1 to N; every memory location can contain a value (integer, real, etc.);
- Read or write access to each memory location requires constant time;
- The machine has a set of registers to store parameters needed for basic operations, and the pointer to current operation;
- The machine has a program composed by a finite set of instructions.

Computational Cost

Definition

Let $f(n)$ be the amount of resources (execution time or memory requested) needed by an algorithm on a input of size n, executed on a register-based machine.

We want to study the *order of magnitude* of $f(n)$ by ignoring the multiplicative constants and the terms of lower magnitude.

Computational cost metric

Evaluating the real execution time of a program to estimate the computational cost has a number of disadvantages:

- To implement a given algorithm could be time consuming activity;
- Execution time is dependent on the given architecture used (programming language used, machine and CPU characteristics, etc.);
- We could be interested to know the computational cost metric for input size too wide for the machine available;
- To estimate the order of magnitude of the cost metric from empirical measures is nto always possible;

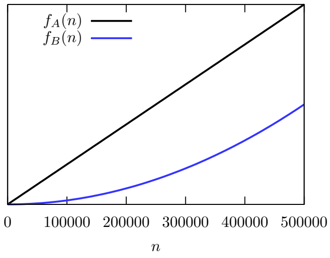
Computational Cost

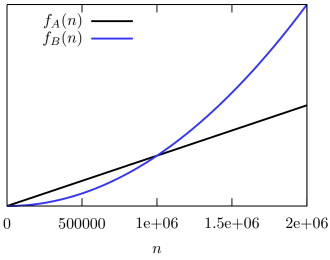
Example

Let's conider two algorithms *A* e *B* resolving the same problem.

- Let $f_A(n) = 10^3 n$ be the computational cost of *A*;
- Let $f_B(n) = 10^{-3} n^2$ be the computational cost of *B*.

Which one is preferable?





Asymptotic notation $O(f(n))$

Definition

Given a cost function $f(n)$, we define the set $O(f(n))$ as the set of functions $g(n)$ such that constants $c > 0$ e $n_0 \geq 0$ exist, such that the following conditions are satisfied:

$$\forall n \geq n_0 \;:\; g(n) \leq cf(n)$$

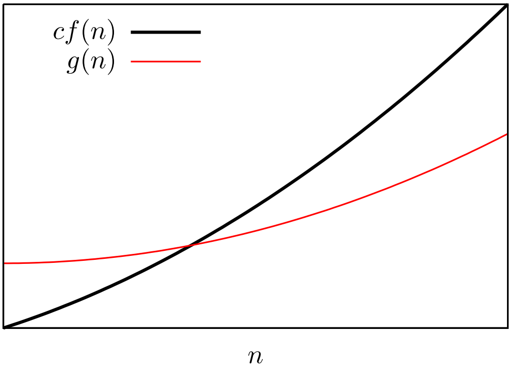
Or synthetically:

$$O(f(n)) = \{g(n) \;:\; \exists c > 0, n_0 \geq 0 \textit{ such that } \forall n \geq n_0 : g(n) \leq cf(n)\}$$

Note: we use the notation (though not formally correct) $g(n) = O(f(n))$ to indicate $g(n) \in O(f(n))$.

Graphical representation

$$g(n) = O(f(n))$$



Example

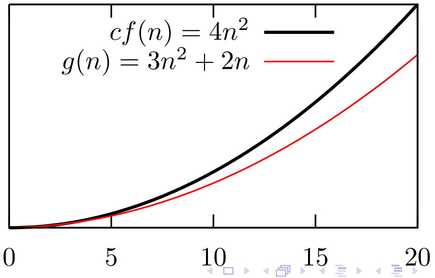
Let $g(n) = 3n^2 + 2n$ and $f(n) = n^2$. We want to prove that $g(n) = O(f(n))$.

We must find two constants $c > 0, n_0 \geq 0$ such that $g(n) \leq cf(n)$ for each $n \geq n_0$, in other words:

$$3n^2 + 2n \leq cn^2 \quad (1)$$

$$c \geq \frac{3n^2 + 2n}{n^2} = 3 + \frac{2}{n}$$

as an example, let's select $n_0 = 10$ and $c = 4$, and we see that relation (1) is satisfied.



Asymptotic notation $\Omega(f(n))$

Definition

Given a cost function $f(n)$, we define the set $\Omega(f(n))$ as the set of functions $g(n)$ such that constants $c > 0$ e $n_0 \geq 0$ exist, such that the following conditions are satisfied:

$$\forall n \geq n_0 \ : \ g(n) \geq cf(n)$$

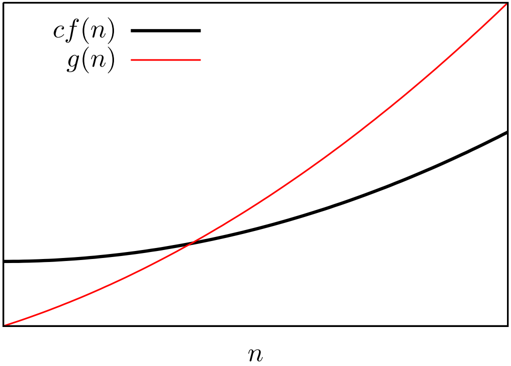
More shortly:

$$\Omega(f(n)) = \{g(n) \ : \ \exists c > 0, n_0 \geq 0 \textit{ such that } \forall n \geq n_0 : g(n) \geq cf(n)\}$$

Note: we use the notation $g(n) = \Omega(f(n))$ to indicate $g(n) \in \Omega(f(n))$.

Graphical representation

$$g(n) = \Omega(f(n))$$



Example

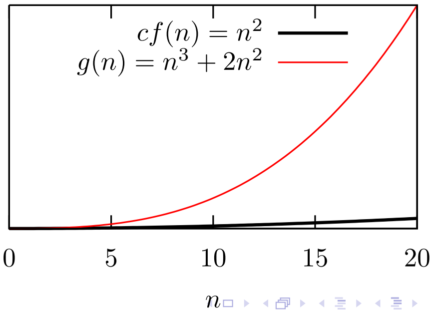
Let $g(n) = n^3 + 2n^2$ and $f(n) = n^2$, and let's prove that $g(n) = \Omega(f(n))$.

We must find two constrants $c > 0, n_0 \geq 0$ such that, for any $n \geq n_0$ then $g(n) \geq cf(n)$, in other words:

$$n^3 + 2n^2 \geq cn^2 \tag{2}$$

$$c \leq \frac{n^3 + 2n^2}{n^2} = n + 2$$

as an example, by selecting $n_0 = 0$ and $c = 1$, the relation (2) holds.



Asymptotical notation $\Theta(f(n))$

Definition

Given a cost function $f(n)$, we define the set $\Theta(f(n))$ as the set of functions $g(n)$ such that constants $c_1 > 0$, $c_2 > 0$ and $n_0 \geq 0$ exist, such that the following conditions are satisfied:

$$\forall n \geq n_0 \ : \ c_1 f(n) \leq g(n) \leq c_2 f(n)$$

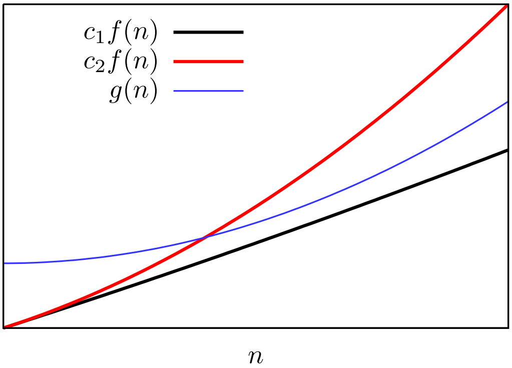
Synthetically:

$$\Theta(f(n)) = \{g(n) \ : \exists c_1 > 0, c_2 > 0, n_0 \geq 0 \text{ such that } \forall n \geq n_0 : c_1 f(n) \leq g(n) \leq c_2 f(n)\}$$

Note: we use the notation $g(n) = \Theta(f(n))$ to indicate $g(n) \in \Theta(f(n))$.

Graphical representation

$$g(n) = \Theta(f(n))$$



Intuitive explanation

- If $g(n) = O(f(n))$ this means that the order of magnitude of $g(n)$ is "less or equal" than $f(n)$;
- If $g(n) = \Theta(f(n))$ this means that $g(n)$ and $f(n)$ have the same order of magnitude;
- Se $g(n) = \Omega(f(n))$ this means that the order of magnitude of $g(n)$ is "greater or equal" than $f(n)$

Some properties of the asymptotical notation

Simmetry

$$g(n) = \Theta(f(n)) \text{ if and only if } f(n) = \Theta(g(n))$$

Transposed Simmetry

$$g(n) = O(f(n)) \text{ iff } f(n) = \Omega(g(n))$$

Transitivity

If $g(n) = O(f(n))$ and $f(n) = O(h(n))$, then $g(n) = O(h(n))$.
The same holds for Ω and Θ.

Orders of magnitude

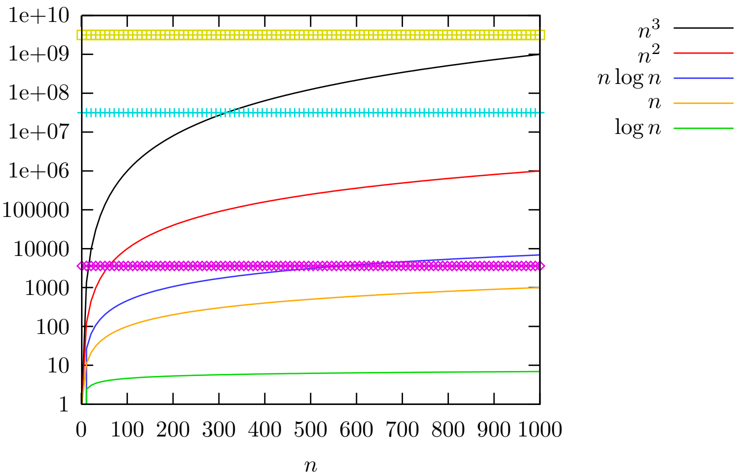
In ascending order of cost:

	Order	Example
$O(1)$	constant	determine if a number is even
$O(\log n)$	logaritmic	search of an element in an ordered array
$O(n)$	linear	search of an element in an unordered array
$O(n \log n)$	pseudolinear	Merge Sort ordering of an array
$O(n^2)$	quadratic	Bubble Sort ordering of an array
$O(n^3)$	cubic	matrix product $n \times n$ with “intuitive” algorithm
$O(c^n)$	exponential, base $c > 1$	
$O(n!)$	factorial	Computation of matrix determinant by expansion of minor
$O(n^n)$	exponential, base n	

In general:

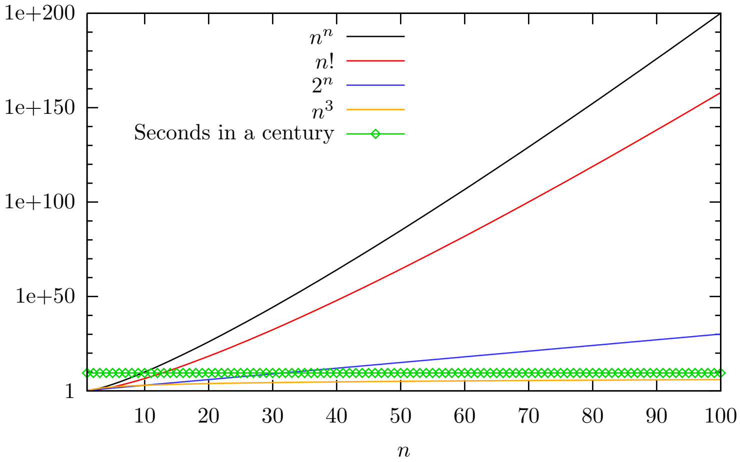
- $O(n^k)$ when $k > 0$ is polinomial order
- $O(c^n)$ when $c > 1$ is exponential order

Graphical comparison of orders of magnitude



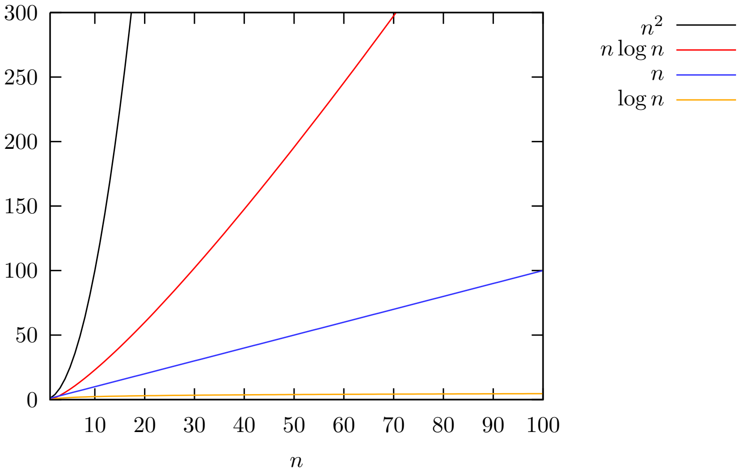
Note: *y* scale is logaritmic; horizontal lines indicate the number of seconds in an hour, a year, a century (respectively, bottom to up)

Graphical comparison of orders of magnitude



Note: *y* scale is logaritmic

Graphical comparison of orders of magnitude



Quiz: true or false?

$6n^2 = \Omega(n^3)$?

By applying the definition, we must prove that

$$\exists c > 0, n_0 \geq 0 \ : \ \forall n \geq n_0 \quad 6n^2 \geq cn^3$$

That is, $c \leq 6/n$.

Given c we can always select a value of n sufficiently large such that $6/n < c$, hence the assertion is false. □

Quiz: true or false?

$10n^3 + 2n^2 + 7 = O(n^3)$?

By applying the definition, we must prove that

$$\exists c > 0, n_0 \geq 0 \ : \ \forall n \geq n_0 \quad 10n^3 + 2n^2 + 7 \leq cn^3$$

In other words:

$$\begin{aligned} 10n^3 + 2n^2 + 7 &\leq 10n^3 + 2n^3 + 7n^3 \qquad (\text{se } n \geq 1) \\ &= 19n^3 \end{aligned}$$

Hence the inequality is true e.g. when $n_0 = 1$ and $c = 19$. □

Questions

- Demonstrate $\log_2 n = O(n)$;
- What is the difference if the base of logarithm is 2?
- Demonstrate $n \log n = O(n^2)$;
- Demonstrate, for all $\alpha > 0$, $\log n = O(n^\alpha)$ (hint: see above, we can say $\log n^\alpha = O(n^\alpha)$, hence...)
- Find the good location for $O(\sqrt{n})$ in the table of the orders of magnitude. Why?

Cost of execution

Definition

An algorithm $\mathcal{A}$ has execution cost $O(f(n))$ on an instance of the input of size n with respect to a given computation resource, if given the amount $r(n)$ of the resource sufficient for execution of $\mathcal{A}$, for every instance of size n, the following relation holds: $r(n) = O(f(n))$.

Note Computation resources in our case means execution time or memory occupation.

Problem complexity

Definition

A problem $\mathcal{P}$ has complexity $O(f(n))$, with respect to a given computation resource, if an algorithm exists which resolves $\mathcal{P}$, whose execution cost with respect to the resource is $O(f(n))$.

Some useful laws

Sum

If $g_1(n) = O(f_1(n))$ and $g_2(n) = O(f_2(n))$, then
$g_1(n) + g_2(n) = O(f_1(n) + f_2(n))$

Product

If $g_1(n) = O(f_1(n))$ and $g_2(n) = O(f_2(n))$, then
$g_1(n) \cdot g_2(n) = O(f_1(n) \cdot f_2(n))$

constants elimination

If $g(n) = O(f(n))$, then $a \cdot g(n) = O(f(n))$ for every constant $a > 0$

Observation

Dealing with the orders of magnitude, every basic operation (instruction) has cost $O(1)$; a different contribute comes from conditional and iterative instructions.

```
if ( F_test ) {
  F_true
} else {
  F_false
}
```

Assuming:

- $\text{F\_test} = O(f(n))$
- $\text{F\_true} = O(g(n))$
- $\text{F\_false} = O(h(n))$

The execution cost of the if-then-else block is

$$O(\max\{f(n), g(n), h(n)\})$$

Analysis of the best, worst and average case

Let $\mathcal{I}_n$ be the set of all possible *istances of the input* of size n. Let $T(I)$ be the execution time of the algorithm on the intance $I \in \mathcal{I}_n$.

- The (*worst case*) cost is defined as

$$T_{\text{worst}}(n) = \max_{I \in \mathcal{I}_n} T(I)$$

- The (*best case*) cost is defined as

$$T_{\text{best}}(n) = \min_{I \in \mathcal{I}_n} T(I)$$

- The (*average case*) cost is defined as

$$T_{\text{avg}}(n) = \sum_{I \in \mathcal{I}_n} T(I) P(I)$$

where $P(I)$ is the probability of occurrence of the instance I.

Analyisis of non recursive algorithms

Search of min value in non-empty array

```
// Return position of minimum element in A
algorithm Minimum( A[1..n] of float ) -> int
  int m:=1; // Position of min element
  for i:=2 to n do
    if ( A[i]<A[m] ) then
      m = i;
    endif
  endfor
  return m;
}
```

Analysis

- Let n be the length of array v.
- the cycle body is executed $n - 1$ times;
- Every iteration has cost $O(1)$
- Il time cost of the execution of `Minimum` is $O(n)$ (or, more precisely, $\Theta(n)$: why?).

Sequential search
Best and Worst cases

```
// Returns the position of first occurrence of ``val''
// in the array A[1..n].
// Returns -1 if the value is not included.
algorithm Find( array A[1..n] of int, int val ) -> int
  for i:=1 to n do
    if ( A[i]==val ) then
      return i;
    endif
  endfor
  return -1;
```

- In the best case the searched element is the first of the list. Hence $T_{\text{best}}(n) = O(1)$
- In the worst case the searched element is the last one (or it is not present). Hence $T_{\text{worst}}(n) = \Theta(n)$
- and in the average case?

Sequential search
Analysis of the average case

We do not know the probability of occurrence of the values in the list, so we make some assumptions.

Given an array of n elements, we assume the probability P_i that the element is in position i ($i = 1, 2, \ldots n$) to be $P_i = 1/n$, for every i (we assume the element is always present in the array).

The time $T(i)$ needed to find element in the i-th position is $T(i) = i$.

Hence we conclude that:

$$T_{\text{avg}}(n) = \sum_{i=1}^{n} P_i T(i) = \frac{1}{n} \sum_{i=1}^{n} i = \frac{1}{n} \frac{n(n-1)}{2} = \Theta(n)$$

Example
An iterative ordering algorithm

```
public class SortingAlgo {
  // compute index of min element in the set
  // v[i], v[i+1]... v[j]
  static int Min( int v[], int i, int j )
  { /* ... /* }
  // v[] must be non-empty
  public static void Sort( int v[] )
  {
    for ( int i=0; i<v.length-1; ++i ) {
      int m = SortingAlgo.Min( v, i, v.length-1 );
      // Swap v[i] e v[m]
      int tmp = v[i];
      v[i] = v[m];
      v[m] = tmp;
    }
  }
}
```

Analysis of the sorting algorithm

- The call of Min(v,i,v.length−1) finds the min element in the array $v[i], v[i+1], \ldots v[n-1]$. The time needed is proportional to $n-i$, $i = 0, 1, \ldots n-1$ (why?);
- the swap operation has execution time cost $O(1)$;
- The body of the **for** cycle is executed n times.

The time execution cost of the whole function `Sort` is:

$$\sum_{i=0}^{n-1}(n-i) = n^2 - \sum_{i=0}^{n-1} i = n^2 - \frac{n(n-1)}{2} = \frac{n^2+n}{2}$$

which is $\Theta(n^2)$.

```
public class BinarySearch {

  static int FindRec( int val, int v[], int i, int j ) {
    if ( i>j ) { return -1; }
    else {
      int m = (i+j)/2;
      if ( v[m] == val ) { return m; } // found
      else {
        if ( v[m] > val ) {
          return FindRec( val, v, i, m-1 );
        } else {
          return FindRec( val, v, m+1, j );
        }
      }
    }
  }

  // Finds the position of an element with value val in the
  // array v[], ordered in ascending order.
  public static int Find( int val, int v[] ) {
    BinarySearch.FindRec( val, v, 0, v.length-1 );
  }
}
```

Analysis of the binary search algorithm

Let $T(n)$ be the execution time of function `FindRec` on an array of $n = j - i + 1$ elements.

In general, $T(n)$ depends both on the number of elements in the array, and on the position of the searched element (or the fact that the element is missing).

- In the most favorable case (best case) the searched element is in the central position; in this case $T(n) = O(1)$.
- In the less favorable case (worst case) the searched element does not exist. Which function is $T(n)$ in this case?

Analysis of the binary search algorithm
Iteration method

We can define $T(n)$ with a recurrence, as follows.

$$T(n) = \begin{cases} c_1 & \text{if } n = 0 \\ T(\lfloor n/2 \rfloor) + c_2 & \text{if } n > 0 \end{cases}$$

The iteration method consists in developing the recurrence equation and intuitively define the equation:

$$T(n) = T(n/2) + c_2 = T(n/4) + 2c_2 = T(n/8) + 3c_2 = \ldots = T(n/2^i) + i \times c_2$$

Assuming that n is a power of 2, we stop when $n/2^i = 1$, that is $i = \log n$. At the end we get

$$T(n) = c_1 + c_2 \log n = O(\log n)$$

Verifying recurrence equations
Substitution method

We apply the principle of induction to verify the solution of a recurrence equation.

Example We prove that $T(n) = O(n)$ is a solution for

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(\lfloor n/2 \rfloor) + n & \text{if } n > 1 \end{cases}$$

Proof By induction, we verify that $T(n) \leq cn$ for n sufficiently large.

- Base step: $T(1) = 1 \leq c \times 1$. It is sufficient to choose $c \geq 1$.
- Inductive step:

$$\begin{aligned} T(n) &= T(\lfloor n/2 \rfloor) + n \\ &\leq c\lfloor n/2 \rfloor + n \qquad \text{(inductive assumption)} \\ &\leq cn/2 + n = f(c)n \end{aligned}$$

with $f(c) = (c/2 + 1)$. The proof of the inductive step works when $f(c) \leq c$, that is $c \geq 2$.

□

Theorem

The recurrence relation:

$$T(n) = \begin{cases} aT(n/b) + f(n) & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases} \quad (3)$$

has solution:

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

Example

Application of the master theorem

$$T(n) = \begin{cases} aT(n/b) + f(n) & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$$

1. In the binary search, we have $T(n) = T(n/2) + O(1)$. Hence $a = 1$, $b = 2$, $f(n) = O(1)$; this is the second case of the theorem, hence $T(n) = \Theta(\log n)$.
2. Considering $T(n) = 9T(n/3) + n$; in this case $a = 9$, $b = 3$ and $f(n) = O(n)$. This is the first case, $f(n) = O(n^{\log_b a - \epsilon})$ with $\epsilon = 1$, that is $T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$.

The Fibonacci sequence is defined as:

$$F_n = \begin{cases} 1 & \text{if } n = 1,\ 2 \\ F_{n-1} + F_{n-2} & \text{if } n > 2 \end{cases}$$

Let's consider the execution time of the trivial recursive algorithm to compute F_n, whose execution time $T(n)$ satisfies the occurrence relation:

$$T(n) = \begin{cases} c_1 & \text{if } n = 1,\ 2 \\ T(n-1) + T(n-2) + c_2 & \text{if } n > 2 \end{cases}$$

We want to evaluate a lower and upper bound for $T(n)$

Upper bound. We exploit the fact that $T(n)$ is non-decreasing function:

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + c_2 \\ &\leq 2T(n-1) + c_2 \\ &\leq 4T(n-2) + 2c_2 + c_2 \\ &\leq 8T(n-3) + 2^2 c_2 + 2c_2 + c_2 \\ &\leq \dots \\ &\leq 2^k T(n-k) + c_2 \sum_{i=0}^{k-1} 2^i \\ &\leq \dots \\ &\leq 2^{n-1} c_3 \end{aligned}$$

for a given constant c_3. Hence $T(n) = O(2^n)$.

Analysis of recursive algorithms

Fibonacci–lower bounds

Lower bounds. Again, we exploit the fact that $T(n)$ is a non-decreasing function:

$$
\begin{aligned}
T(n) &= T(n-1) + T(n-2) + c_2 \\
&\geq 2T(n-2) + c_2 \\
&\geq 4T(n-4) + 2c_2 + c_2 \\
&\geq 8T(n-6) + 2^2 c_2 + 2c_2 + c_2 \\
&\geq \dots \\
&\geq 2^k T(n-2k) + c_2 \sum_{i=0}^{k-1} 2^i \\
&\geq \dots \\
&\geq 2^{\lfloor n/2 \rfloor} c_4
\end{aligned}
$$

for a given constant c_4. Hence $T(n) = \Omega(2^{\lfloor n/2 \rfloor})$.

Note

Attention $2^{\lfloor n/2 \rfloor} = O(2^n)$, but $2^{\lfloor n/2 \rfloor} \neq \Theta(2^n)$. In other words, the two functions, both exponential, belong to different classes of complexity. (Why?).

Amortized cost

The amortized analysis studies the average cost of a sequence of operations.

Definition

Let $T(n, k)$ be the total time needed by an algorithm, in the worst case, to execute k operation on input instances of size n. We define amortized cost a sequence of k operations

$$T_\alpha(n) = \frac{T(n, k)}{k}$$

Example
Amortized analysis

Problem: given a sequence of binary digits, initialized to zero, we define a function which increments by one the decimal value represented by the binary digits.

```
// v[0] is the most significant bit
public static void increment( int[] v )
{
  for ( int i=v.length-1; i>0; --i ) {
    v[i] = 1-v[i]; // invert the bit
    if ( v[i] == 1 ) {
      break;
    }
  }
}
```

Example

value	$v[0]$	$v[1]$	$v[2]$	$v[3]$	$v[4]$	$v[5]$	**Cost**
0	0	0	0	0	0	0	
1	0	0	0	0	0	**1**	1
2	0	0	0	0	**1**	**0**	2
3	0	0	0	0	1	**1**	1
4	0	0	0	**1**	**0**	**0**	3
5	0	0	0	1	0	**1**	1
6	0	0	0	1	**1**	**0**	2
7	0	0	0	1	1	**1**	1
8	0	0	**1**	**0**	**0**	**0**	4
9	0	0	1	0	0	**1**	1
10	0	0	1	0	**1**	**0**	2

Analysis

The cost of operation `invert` is the number of inverted bits.

- The first bit (`v[n-1]`) is inverted in each call;
- The second bit (`v[n-2]`) is inverted every 2 calls;
- The third bit (`v[n-3]`) is inverted every 4 calls;
- ...
- The *i*-th bit (`v[n-i]`) is inverted every 2^{i-1} calls;

The total execution time for k operatinos is given as:

$$T(n,k) = k + \lfloor k/2 \rfloor + \lfloor k/4 \rfloor + \ldots + 2 + 1 = \sum_{i=0}^{\log_2 k} \lfloor k/2^i \rfloor \leq k \sum_{i=0}^{\infty} 1/2^i = 2k$$

Hence

$$T_\alpha(n) = \frac{T(n,k)}{k} = O(1)$$