Algorithm efficiency analysis

Mădălina Răschip, Cristian Gațu

Faculty of Computer Science
"Alexandru Ioan Cuza" University of Iași, Romania

DS 2018/2019

Content

Efficiency classes

Class	Notation	Example
logarithmic	$O(\log n)$	binary search
linear	$O(n)$	sequential search
quadratic	$O(n^2)$	insertion sort
cubic	$O(n^3)$	multiplication of two matrices $n \times n$
exponential	$O(2^n)$	processing of the subset of an n element set
factorial	$O(n!)$	processing of the permutation of n order

Algorithm efficiency empirical analysis

- Employed when the theoretical analysis is difficult.
- Aim:
  - formulation of an hypothesis regarding the algorithm efficiency;
  - verification of this hypothesis;
  - comparison of algorithms;
  - efficiency analysis of some implementation.

Algorithm efficiency empirical analysis

- The analysis aim is set.
- An efficiency measure is chosen
  Example: elementary operation counting, running time, etc.
- Input data characteristics are set.
- The algorithm is implemented.
- Input data are generated.
- The program is run for for all input data; the result are stored.
- The results are analyzed.

Content

Algorithm efficiency analysis

Recursive function analysis

- Recursive functions
- Substitution method
- Iteration method
- Recurrence trees
- Master theorem

Content

Recursive functions

- A function f() **calls directly** a function g() if the definition of f() contains at least one call to g().
- A function f() **calls indirectly** a function g() if f() calls directly a function h(), and h() calls directly or indirectly the function g().
- A function f() is **recursively defined** if it calls itself directly or indirectly.

Recursive functions

A recursive function definition:

- **Base case testing** – recursive calls stopping condition.
- **Recursive (general case)**: some variable (integer) is passed as parameter to the function itself such that after a finite number of calls to reach the base case.

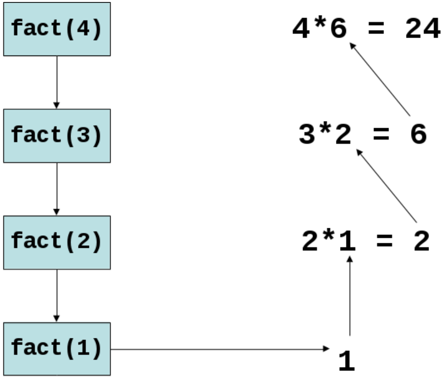
Remark: There exists recursive function with no parameters.

Example 1. Factorial function definition:

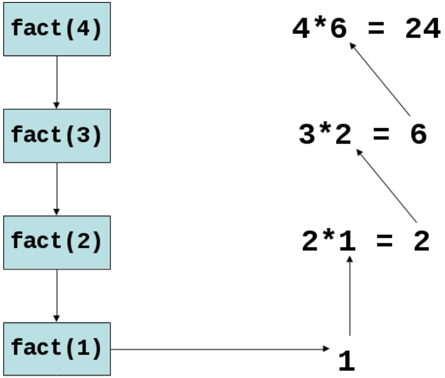
- Base case: $0! = 1$;
- General case: $n! = n \times ((n-1)!), n > 0$.

Function *factorial(n)*
begin
    if $n <= 1$ **then**
        return 1
    else
        return $(n * factorial(n-1))$
end

factorial(4) - recursive calls



factorial(4) - recursive calls



- recursive algorithms: easy to implement;
- additional cost: each recursive call places information in a specific memory zone (program stack).

factorial(n) - iterative version

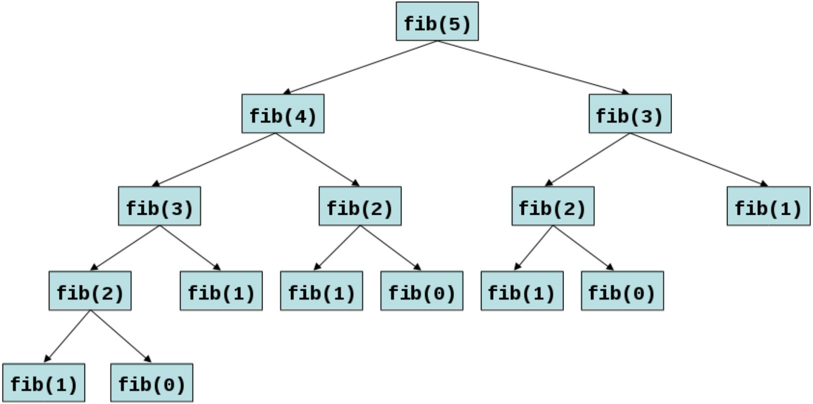
Function *factorial(n)*
begin
    $product \leftarrow 1$
    while $n > 1$ **do**
        $product \leftarrow product * n$
        $n \leftarrow n - 1$
    return *product*
end

Example 2. Fibonacci numbers:

- $f(0) = 0, f(1) = 1,$
- $f(n) = f(n-1) + f(n-2), n > 1.$

Function *fib(n)*
begin
    if $n <= 1$ **then**
        return n
    else
        return $fib(n-1) + fib(n-2)$
end

Recursive Fibonacci: call tree



$O(\phi^n)$

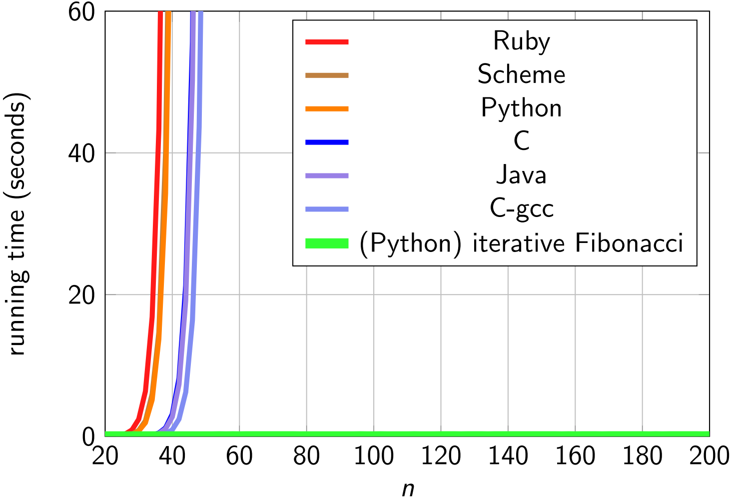
Call number

n	fib(n)	calls
2	1	3
24	46'368	150'049
42	267'914'296	866'988'873
43	433'494'437	1'402'817'465

Recursion vs. iteration: iterative Fibonacci

Function *ifib(n)*
begin
    $f0 \leftarrow 0$
    $f1 \leftarrow 1$
    if $n <= 1$ **then**
        return n
    else
        for $k \leftarrow 2$ **to** n **do**
            $temp \leftarrow f1$
            $f1 \leftarrow f1 + f0$
            $f0 \leftarrow temp$
        return $f1$
end

Recursive / iterative Fibonacci comparison



Recursive algorithm efficiency

- To estimate the running time:
  - a recurrence is set in order to express the relation between the execution time of the initial problem and the execution time of the reduced problem;
  - the recurrence is solved.
- Example: for the factorial, the recurrence that estimates the running time is:

$$T(n) = \begin{cases} 0, & n = 0 \\ T(n-1) + 1, & n > 1 \end{cases}$$

How to solve recurrences

1. **Substitution method.** A limit is guessed and is proved by induction.

2. **Iteration method.** The recurrence is iterated and expressed as a sum of terms that depends only on the problem size and the initial condition (base cases).

3. **Recurrence tree.** Expresses the recurrence as a tree (nodes corresponds to costs).

4. **Master theorem.** Provides limits for recurrences of type

$$T(n) = aT(n/b) + f(n)$$

.

Content

Algorithm efficiency analysis

Recursive function analysis

- Recursive functions
- Substitution method
- Iteration method
- Recurrence trees
- Master theorem

1. Substitution method

- The solution is guessed.
- Mathematical induction is employed to determine the constants and to proof that the solution is correct.

Substitution method – example

Find an upper bound for the recurrence $T(n) = 2T(\lfloor n/2 \rfloor) + n$

Substitution method – example

Find an upper bound for the recurrence $T(n) = 2T(\lfloor n/2 \rfloor) + n$

- Guessed solution: $T(n) = O(n \log n)$.
- To be proved by induction: $T(n) \leq cn \log n$, for $c > 0$.

Substitution method – example

Find an upper bound for the recurrence $T(n) = 2T(\lfloor n/2 \rfloor) + n$

- Guessed solution: $T(n) = O(n \log n)$.
- To be proved by induction: $T(n) \leq cn \log n$, for $c > 0$.
  Suppose that the relation is true for all positive values $m < n$, in particular for $m = \lfloor n/2 \rfloor$: $T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)$.

$$\begin{aligned} T(n) &\leq 2(c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)) + n \\ &\leq cn \log(\lfloor n/2 \rfloor) + n \\ &= cn \log n - cn \log 2 + n \\ &= cn \log n - cn + n \\ &\leq cn \log n, \text{ for } c \geq 1 \end{aligned}$$

Substitution method – example

Find an upper bound for the recurrence $T(n) = 2T(\lfloor n/2 \rfloor) + n$

- Guessed solution: $T(n) = O(n \log n)$.
- To be proved by induction: $T(n) \leq cn \log n$, for $c > 0$.
  Suppose that the relation is true for all positive values $m < n$, in particular for $m = \lfloor n/2 \rfloor$: $T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)$.

$$\begin{aligned} T(n) &\leq 2(c \lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)) + n \\ &\leq cn \log(\lfloor n/2 \rfloor) + n \\ &= cn \log n - cn \log 2 + n \\ &= cn \log n - cn + n \\ &\leq cn \log n, \text{ for } c \geq 1 \end{aligned}$$

  It has to be shown that the relation holds for the base case (limit conditions). $T(1) = 1 \leq c1 \log 1 = 0$
  Base cases: T(2) and T(3) ($n_0 = 2$)
  $T(2) = 4$ and $T(3) = 5$, $T(2) \leq c2 \log 2$ and $T(3) \leq c3 \log 3 \Rightarrow c \geq 2$.

Substitution method – tips

- Subtraction of an inferior term order (to consolidate the inductive hypothesis).

Example: $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$

  - Guessed solution: $T(n) = O(n)$.
  - It is shown by induction that $T(n) \leq cn$, for $c > 0$.

$$\begin{aligned} T(n) &\leq c\lfloor n/2 \rfloor + c\lceil n/2 \rceil + 1 \\ &= cn + 1 \end{aligned}$$

  - It is shown by induction that $T(n) \leq cn - d$, $d >= 0$ const.

$$\begin{aligned} T(n) &\leq (c\lfloor n/2 \rfloor - d) + (c\lceil n/2 \rceil - d) + 1 \\ &= cn - 2d + 1 \\ &\leq cn - d, \text{ for } d \geq 1 \end{aligned}$$

The c constant has to be chosen such that the limit conditions are satisfied.

Substitution method – tips

- Traps

Example: $T(n) = 2T(\lfloor n/2 \rfloor) + n$

It is shown ("false") that $T(n) = O(n)$ guessing that $T(n) \leq cn$ and:

$$
\begin{aligned}
T(n) &\leq 2(c\lfloor n/2 \rfloor) + n \\
&\leq cn + n \\
&= O(n), \quad \Leftarrow \text{false!!}
\end{aligned}
$$

Error: it was not proven the *exact form* of the induction hypothesis.

Substitution method – tips

- Variable changes.

  Example: $T(n) = 2T(\lfloor sqrt(n) \rfloor) + \log n$

  The recurrence is simplified by variable change: $m = \log n$.
  $T(2^m) = 2T(2^{m/2}) + m$

  Rename $S(m) = T(2^m)$, and it follows $S(m) = 2S(m/2) + m$.
  $S(m) = O(m \log m)$,
  $T(n) = T(2^m) = S(m) = O(m \log m) = O(\log n \log \log n)$.

Content

2. Recurrence iteration

Substitution method: implies to guess the solution (!)

Recurrence iteration:

- **direct**
  - starts from the base case and builds successive terms using the recurrence;
  - the expression of the general term $T(n)$ is identified;
  - it is proved by direct computations or mathematical induction.
- **reverse**
  - it starts from the $T(n)$ and is replaced by $T(h(n))$ with corresponding value, then $T(h(h(n)))$ is replaced and so on until the base case is reached;
  - $T(n)$ is obtained.

Recurrence iteration – example $n!$

$$T(n) = \begin{cases} 0, & n = 1 \\ T(n-1) + 1, & n > 1 \end{cases}$$

Recurrence iteration – example $n!$

$$T(n) = \begin{cases} 0, & n = 1 \\ T(n-1) + 1, & n > 1 \end{cases}$$

Direct iteration

$T(1) = 0$
$T(2) = 1$
$T(3) = 2$
$\ldots$
$T(n) = n - 1$

Recurrence iteration – example $n!$

$$T(n) = \begin{cases} 0, & n = 1 \\ T(n-1) + 1, & n > 1 \end{cases}$$

Direct iteration

$T(1) = 0$
$T(2) = 1$
$T(3) = 2$
...
$T(n) = n - 1$

Reverse iteration

$T(n) = T(n-1) + 1$
$T(n-1) = T(n-2) + 1$
...
$T(2) = T(1) + 1$
$T(1) = 0$

$T(n) = n - 1$

Recurrence iteration – example

$T(n) = 3T(\lfloor \frac{n}{4} \rfloor) + n$

$$
\begin{aligned}
T(n) &= n + 3(\lfloor \frac{n}{4} \rfloor) + 3T(\lfloor \frac{n}{16} \rfloor) \\
&= n + 3\lfloor \frac{n}{4} \rfloor + 9(\lfloor \frac{n}{16} \rfloor + 3T(\lfloor \frac{n}{64} \rfloor)) \\
&= n + 3\lfloor \frac{n}{4} \rfloor + 9\lfloor \frac{n}{16} \rfloor + 27T(\lfloor \frac{n}{64} \rfloor) \\
\ldots \\
&\leq n \sum_{i=0}^{\infty} (\frac{3}{4})^i + \Theta(n^{log_4 3} x T(1)) \\
&= 4n + \Theta(n^{log_4 3} x T(1)) \\
&= O(n)
\end{aligned}
$$

Remark: using geometric series:
$1 + x + x^2 + ... + x^n = \frac{1-x^{n+1}}{1-x}$, for $x \neq 1$
$1 + x + x^2 + ... = \frac{1}{1-x}$, for $|x| < 1$

Content

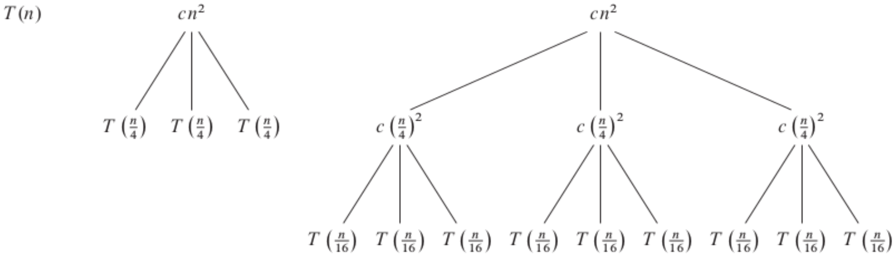
- **Recurrence trees**:
  - allows to visualize a recurrence;
  - each node corresponds to the cost of a sub-problem;
  - the costs are sum on levels and then they are summed to get the total cost.
- The recurrence tree can be used to generate a value for the substitution method.

Recurrence trees – example

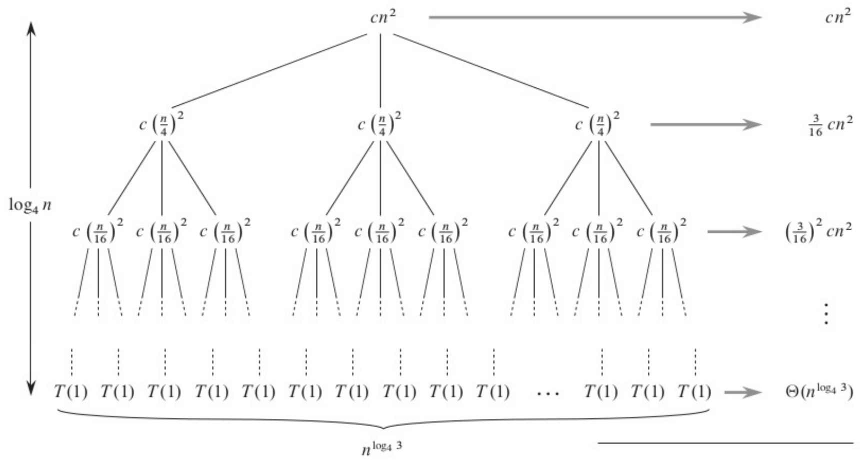
$$T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$$

$$T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$$

- The recurrence tree is created for $T(n) = 3T(n/4) + cn^2$, $c > 0$



Recurrence trees – example



Recurrence trees – example

- The dimension of a sub-problem corresponding to a node on level i: $n/4^i \Rightarrow$ sub-problem size becomes $n = 1$ when $n/4^i = 1 \Leftrightarrow i = log_4 n \Rightarrow$ the tree has $log_4 n + 1$ levels.
- The number of nodes on level i: 3^i.
- The cost of any node on level i : $c(n/4^i)^2$.
- The total cost of nodes on level i: $3^i c(n/4^i)^2 = (3/16)^i cn^2$ (last level $log_4 n$: $n^{log_4 3} T(1)$.)

Recurrence trees – example

$$
\begin{aligned}
T(n) &= cn^2 + \frac{3}{16}cn^2 + (\frac{3}{16})^2 cn^2 + \dots + (\frac{3}{16})^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3}) \\
&= \sum_{i=0}^{\log_4 n - 1} (\frac{3}{16})^i cn^2 + \Theta(n^{\log_4 3}) \\
&< \sum_{i=0}^{\infty} (\frac{3}{16})^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{1}{1-(3/16)} cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \\
&= O(n^2)
\end{aligned}
$$

Recurrence trees – example

- The substitution method is employed to verify that $T(n) = O(n^2)$ is an upper bound for the relation $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$.

- it is shown that $T(n) \le dn^2$, for $d > 0$

$$\begin{aligned} T(n) &\le 3T(\lfloor n/4 \rfloor) + cn^2 \\ &\le 3d\lfloor n/4 \rfloor^2 + cn^2 \\ &\le 3d(n/4)^2 + c(n^2) \\ &= \frac{3}{16}dn^2 + cn^2 \\ &\le dn^2, \text{ for } d \ge (16/13)c \end{aligned}$$

Content

4. Master theorem

- Provides a method to solve recurrences of type $T(n) = aT(n/b) + f(n)$ where $a \geq 1$ and $b > 1$ are constants, and $f(n)$ is a function asymptotically positive.

- **Master theorem**:
  Let $a \geq 1$ and $b > 1$ constants, $f(n)$ a function and $T(n)$ defined on non-negative integer numbers by the recurrence relation: $T(n) = aT(n/b) + f(n)$. Then:

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

Master theorem – examples

- $T(n) = 9T(n/3) + n$

  $a = 9, b = 3, f(n) = n$ and $n^{log_b a} = n^{log_3 9} = \Theta(n^2)$.
  Since $f(n) = O(n^{log_3 9 - \epsilon})$, with $\epsilon = 1$, the case 1 of Master theorem can be applied $\Rightarrow T(n) = \Theta(n^2)$.

- $T(n) = T(2n/3) + 1$

  $a = 1, b = 3/2, f(n) = 1$ and $n^{log_b a} = n^{log_{3/2} 1} = n^0 = 1$.
  Since $f(n) = \Theta(n^{log_b a}) = \Theta(1)$, the case 2 of Master theorem can be applied $\Rightarrow T(n) = \Theta(\log n)$.

Master theorem – examples

- $T(n) = 3T(n/4) + n \log n$

  $a = 3, b = 4, f(n) = n \log n$ and $n^{log_b a} = n^{log_4 3} = O(n^{0.793})$
  Since $f(n) = \Omega(n^{log_4 3+\epsilon})$, with $\epsilon \approx 0.2$, the case 3 of Master theorem can be applied if: $af(n/b) = 3(n/4)\log(n/4) \leq (3/4)n \log n = cf(n)$ for $c = 3/4$ and n large enough. It follows $T(n) = \Theta(n \log n)$.

- The Master theorem cannot be applied for $T(n) = 2T(n/2) + n \log n$

  $a = 2, b = 2, f(n) = n \log n$ and $n^{log_b a} = n$
  Since $f(n) = n \log n$ is asymptotically larger the $n^{log_b a} = n$, it follows hat the case 3 can be applied (false!!).
  $f(n)$ is not *polynomial* larger.
  $f(n)/n^{log_b a} = (n \log n)/n = \log n$ is asymptotically smaller than n^{ϵ}, for any positive constant ϵ.