Algorithms: Analysis of Algorithms

Alberto Valderruten (orig.), Carlos Gómez Rodríguez (transl.)

Dept. of Computer Science, University of A Coruña

alberto.valderruten@udc.es, carlos.gomez@udc.es

UNIVERSIDADE DA CORUÑA

Contents

Index

Analysis of the efficiency of algorithms (1)

- **Goal:** to *predict* the algorithm's *behaviour*
  $\Rightarrow$ quantitative aspects:
  execution time (runtime), memory usage
- To have a *metric* of their efficiency:
  - "theoretical"
  - not exact: *approximation*, sufficient to *compare*, *classify*
  $\Rightarrow$ bound $T(n)$: execution time,
  $n =$ *problem size* (sometimes, input size)
  $n \to \infty$: *asymptotic behaviour*
  $\Rightarrow T(n) = O(f(n))$
  $f(n)$: an **upper bound** of $T(n)$, sufficiently adjusted
  $f(n)$ grows faster than $T(n)$

Analysis of the efficiency of algorithms (2)

- **Approximation?**
  1. *Ignore constant factors*:
  20 multiplications per iteration $\rightarrow$ 1 **operation** per iteration
  How many iterations? $\rightarrow$ iterations as a function of n
  2. *Ignore lower order terms (the maximum rule)*: $n + const \rightarrow n$

- **Example 1**:
  2 algorithms (A1 and A2) for the same problem A
  - algorithm A1: $100n$ steps $\rightarrow$ a traversal of the input
    $T(n) = O(n)$: *linear* algorithm
  - algorithm A2: $2n^2 + 50$ steps $\rightarrow$ n traversals of the input
    $T(n) = O(n^2)$: *quadratic* algorithm

Analysis of the efficiency of algorithms (3)

- **Example 1 (Cont'd)**:
  $\Rightarrow$ A1 linear and A2 quadratic:
  - *Compare*: A2 "slower" than A1,
    even though with $n \leq 49$ it's faster
    $\Rightarrow$ **A1 is better**
  - *Classify*: linear, quadratic...
    Typical growth rates:
    $O(1), O(logn), O(n), O(nlogn), O(n^2), O(n^3), \ldots O(2^n), \ldots$
- **Example 2**: (approximation $\Rightarrow$ limitations)
  2 algorithms (B1 and B2) for the same problem B:
  - algorithm B1: $2n^2 + 50$ steps $\rightarrow O(n^2)$
  - algorithm B2: $100n^{1.8}$ steps $\rightarrow O(n^{1.8})$
    $\Rightarrow$ B2 is "better"...

  but only from some value of n between 310 **and** $320 * 10^6$

Index

Asymptotic notations

- **Goal:** To establish a relative order between functions, comparing their growth rates
- O **(big O) notation:**
  $T(n), f(n) : Z^+ \to R^+$
  Definition: $T(n) = O(f(n))$
  if $\exists$ constants $c > 0$ and $n_0 > 0$: $T(n) \leq c * f(n)\ \forall n \geq n_0$
  $\square$
  n_0: threshold
  $T(n)$ is $O(f(n))$, $T(n) \in O(f(n))$
  "the growth rate of $T(n) \leq$ than that of $f(n)$"
  $\to f(n)$ is *an* **upper bound** of $T(n)$
- **Example:** $5n^2 + 15 = O(n^2)$?
  $< c, n_0 > = < 6, 4 >$ in the definition: $5n^2 + 15 \leq 6n^2\ \forall n \geq 4$;
  $\exists$ infinite $< c, n_0 >$ satisfying the unequality

O notation (1)

- **Observation**:
  According to the definition, $T(n)$ could be far below:
  $5n^2 + 15 = O(n^3)$?
  $< c, n_0 > = < 1, 6 >$ in the definition: $5n^2 + 15 \leq 1n^3 \; \forall n \geq 6$
  but it is more precise to say $= O(n^2) \equiv$ to **adjust bounds**
  $\Rightarrow$ **For algorithm analysis, we use the approximations:**
  $5n^2 + 4n \rightarrow O(n^2)$
  $log_2 n \rightarrow O(logn)$
  $13 \rightarrow O(1)$
- **Observation**:
  O notation is also used in expressions like $3n^2 + O(n)$
- **Example 3**:
  How do we obtain a more drastic improvement,
  - improving the efficiency of the algorithm, or
  - improving the computer?

O notation (2)

- **Example 3** (cont'd):

$T(n)$	$time_1$ 1000 steps/s	$time_2$ 2000 steps/s	$time_3$ 4000 steps/s	$time_4$ 8000 steps/s
$log_2 n$	0.010	0.005	0.003	0.001
n	1	0.5	0.25	0.125
$nlog_2 n$	10	5	2.5	1.25
$n^{1.5}$	32	16	8	4
n^2	1,000	500	250	125
n^3	1,000,000	500,000	250,000	125,000
1.1^n	10^{39}	10^{39}	10^{38}	10^{38}

Table: Execution times (in s) for 7 algorithms with different complexities (n=1000).

- **Example 4**: Sort 100.000 random integers:
  * 17 s on a 386 + Quicksort
  * 17 min on a 100 times faster processor + Bubble sort

O notation (3)

Practical rules to work with O:

Definition: $f(n)$ is **monotonically increasing** if $n_1 \geq n_2 \Rightarrow f(n_1) \geq f(n_2)$

$\square$

- **Theorem**: $\forall c > 0, a > 1, f(n)$ monotonically increasing:

$$(f(n))^c = O(a^{f(n)})$$

$\equiv$ *"An exponential function (e.g.: 2^n) grows faster than a polynomial function (e.g.: n^2)"*

$$\rightarrow \begin{cases} n^c = O(a^n) \\ (log_a n)^c = O(a^{log_a n}) = O(n) \end{cases}$$

$$\rightarrow (logn)^k = O(n)\ \forall k \text{ const.}$$

$\equiv$ *"n grows faster than any power of a logarithm"*

$\equiv$ *"logarithms grow very slowly"*

O notation (4)

Practical rules to work with O (Cont'd):

- **Sum and product**:

$$T_1(n) = O(f(n)) \wedge T_2(n) = O(g(n)) \Rightarrow$$
$$\begin{cases} (1) & T_1(n) + T_2(n) = O(f(n) + g(n)) = max(O(f(n)), O(g(n))) \\ (2) & T_1(n) * T_2(n) = O(f(n) * g(n)) \end{cases}$$

$$\text{Application: } \begin{cases} \text{(1) Sequence:} & 2n^2 = O(n^2) \wedge 10n = O(n) \\ & \Rightarrow 2n^2 + 10n = O(n^2) \\ \text{(2) Loops} & \end{cases}$$

Observation: **Do not** extend the rule to substraction or division

$\leftarrow$ relation $\leq$ in the definition of O

... sufficient to order the majority of functions

Other asymptotic notations (1)

1. $T(n), f(n) : Z^+ \to R^+$, **Definition**: O
2. **Definition**: $T(n) = \Omega(f(n))$
   iff $\exists$ constants c and n_0: $T(n) \geq cf(n)\ \forall n \geq n_0$
   $\square$
   $f(n)$: **lower bound** of $T(n) \equiv$ minimum work of the algorithm
   Ejemplo: $3n^2 = \Omega(n^2)$: more adjusted lower bound...
   but $3n^2 = O(n^2)$ also! $(O \wedge \Omega)$
3. **Definition**: $T(n) = \Theta(f(n))$
   iff $\exists$ constants c_1, c_2 and n_0: $c_1 f(n) \leq T(n) \leq c_2 f(n)\ \forall n \geq n_0$
   $\square$
   $f(n)$: **exact bound** of $T(n)$, of the exact order
   Example: $5nlog_2 n - 10 = \Theta(nlogn)$:
   $$\begin{cases} (1) \text{ prove } O \to < c, n_0 > \\ (2) \text{ prove } \Omega \to < c', n'_0 > \end{cases}$$

Other asymptotic notations (2)

4. **Definition**: $T(n) = o(f(n))$
   iff $\forall$ constant $C > 0$, $\exists n_0 > 0$: $T(n) < Cf(n)\ \forall n \geq n_0$
   $\square$
   $\equiv O \wedge \neg\Theta \equiv O \wedge \neg\Omega$
   $f(n)$: **strict upper bound** of $T(n)$: $\lim_{n\to\infty} \frac{T(n)}{f(n)} = 0$
   Examples: $\frac{n}{log_2 n} = o(n)$ $\quad \frac{n}{10} \neq o(n)$
5. **Definition**: $T(n) = \omega(f(n))$
   iff $\forall$ constant $C > 0$, $\exists n_0 > 0$: $T(n) > Cf(n)\ \forall n \geq n_0$
   $\square$
   $\leftrightarrow f(n) = o(T(n))$
   $\rightarrow f(n)$: **strictly lower bound** of $T(n)$
6. **OO notation** [Manber]: $T(n) = OO(f(n))$ if it is $O(f(n))$
   but with too large constants for practical cases
   Ref: Example 2 (p. 4): $B1 = O(n^2)$, $B2 = OO(n^{1.8})$

Other asymptotic notations (3)

Practical rules (Cont'd):

- $T(n) = a_0 + a_1 n + a_2 n^2 + \ldots + a_k n^k \Rightarrow T(n) = \Theta(n^k)$
  (polynomial of degree k)
- **Theorem**: $\forall c > 0, a > 1, f(n)$ monotonically increasing:

$$(f(n))^c = o(a^{f(n)})$$

$\equiv$ *"An exponential function* ***grows faster*** *than a polynomial function"*
$\rightarrow$ they never get equal

Index

Model of computation (1)

- Calculate O for $T(n) \equiv$ number of "steps" $\rightarrow f(n)$? *step*?
- **Model of computation**:
  - sequential computer
  - instruction $\leftrightarrow$ step (there are no complex instructions)
  - inputs: single type ("integer") $\rightarrow$ seq(n)
  - infinite memory + *"everything is in memory"*
- Alternatives: A *step* is...
  1. **Elementary operation**:
     Operation whose runtime is bounded above by a constant that only depends on the implementation $\rightarrow = O(1)$
  2. **Main operation** [Manber]:
     Operation that is *representative* of the algorithm's work:
     The number of main operations being executed must be *proportional* to the total number of operations (verify!).
     Example: comparison in a sorting algorithm

Model of computation (2)

- The hypothesis of main operations is more abstract/a wider approximation!
- In general, **we will use the hypothesis of elementary operations**.
- In any case, we ignore: programming language, processor, OS, load...

  $\Rightarrow$ We only consider algorithm, problem size ...
- **Weaknesses**:
  - different-cost operations
    ("everything in memory" $\Rightarrow$ disk read = assignment)
    $\rightarrow$ count separately according to operation type and then *weight* $\equiv$ factors $\equiv$ implementation-dependent
    $\Rightarrow$ costly and generally useless
  - page faults ignored
  - etc.

  $\rightarrow$ *Approximation*

Case analysis

- **Case analysis**:
  We consider different functions for $T(n)$:

$$\begin{cases} T_{best}(n) & \\ T_{average}(n) & \leftarrow \text{representative, more complicated to obtain} \\ T_{worst}(n) & \leftarrow \text{generally, the most used} \end{cases}$$

$$T_{best}(n) \leq T_{average}(n) \leq T_{worst}(n)$$

- Response time is critical?
  $\rightarrow$ *Real-Time Systems*

Insertion Sort (1)

```
procedure Insertion Sort (var T[1..n])
    for i:=2 to n do
        x:=T[i];
        j:=i-1;
        while j>0 and T[j]>x do
            T[j+1]:=T[j];
            j:=j-1
        end while;
        T[j+1]:=x
    end for
end procedure
```

Insertion Sort (2)

3	$\boxed{1}$	4	1	2	9	5	6	5	3
1	3	$\boxed{4}$	1	2	9	5	6	5	3
1	3	**4**	$\boxed{1}$	2	9	5	6	5	3
1	**1**	3	4	$\boxed{2}$	9	5	6	5	3
1	1	**2**	3	4	$\boxed{9}$	5	6	5	3
1	1	2	3	4	**9**	$\boxed{5}$	6	5	3
1	1	2	3	4	**5**	9	$\boxed{6}$	5	3
1	1	2	3	4	5	**6**	9	$\boxed{5}$	3
1	1	2	3	4	5	**5**	6	9	$\boxed{3}$
1	1	2	3	**3**	4	5	5	6	9

Case analysis: Insertion Sort

- **Worst case** $\rightarrow$ "insert always at the first position"
  $\equiv$ input in reverse order
  $\Rightarrow$ the inner loop executes 1 time in the first iteration,
  2 times in the second, $\ldots$, $n-1$ times in the last:
  $\Rightarrow \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$ iterations of the inner loop

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

  $\Rightarrow T(n) = \frac{n(n-1)}{2}c_1 + (n-1)c_2 + c_3$: polynomial of degree 2
  $\Rightarrow \boxed{T(n) = \Theta(n^2)}$
- **Best case** $\rightarrow$ "never insert" $\equiv$ ordered input
  $\Rightarrow$ the inner loop is never ran
  $\Rightarrow T(n) = (n-1)c_1 + c_2$: polynomial of degree 1
  $\Rightarrow \boxed{T(n) = \Theta(n)}$

$\Rightarrow T(n)$ depends *also* of the initial state of the input

Selection Sort (1)

```
procedure Selection Sort (var T[1..n])
    for i:=1 to n-1 do
        minj:=i;
        minx:=T[i];
        for j:=i+1 to n do
            if T[j]<minx then
                minj:=j;
                minx:=T[j]
            end if
        end for;
        T[minj]:=T[i];
        T[i]:=minx
    end for
end procedure
```

Selection Sort (2)

3	$\boxed{1}$	4	1	2	9	5	6	5	3
1	3	4	$\boxed{1}$	2	9	5	6	5	3
1	**1**	4	3	$\boxed{2}$	9	5	6	5	3
1	1	**2**	$\boxed{3}$	4	9	5	6	5	3
1	1	2	**3**	4	9	5	6	5	$\boxed{3}$
1	1	2	3	**3**	9	5	6	5	$\boxed{4}$
1	1	2	3	3	**4**	$\boxed{5}$	6	5	9
1	1	2	3	3	4	**5**	6	$\boxed{5}$	9
1	1	2	3	3	4	5	**5**	$\boxed{6}$	9
1	1	2	3	3	4	5	5	**6**	$\boxed{9}$

Case analysis: Selection Sort

- $\boxed{T(n) = \Theta(n^2)}$ regardless of initial order (exercise)
  $\leftrightarrow$ the inner comparison is ran the same number of times
  Empirically: $T(n)$ does not vary more than 15%

algorithm	minimum	maximum
Insertion	0,004	5,461
Selection	4,717	5,174

Tabla: Times (in seconds) obtained for $n = 4000$

- **Comparison**:

algorithm	worst case	average case	best case
Insertion	$\Theta(n^2)$	$\Theta(n^2)$	$\boxed{\Theta(n)}$
Selection	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Quicksort	$O(n^2)$	$\boxed{O(nlogn)}$	$O(nlogn)$

Rules to calculate O (1)

1. elementary operation = 1 $\leftrightarrow$ Model of Computation

Rules to calculate O (2)

2. **sequence**: $S_1 = O(f_1(n)) \wedge S_2 = O(f_2(n))$
$\Rightarrow \boxed{S_1; S_2} = O(f_1(n) + f_2(n)) = O(max(f_1(n), f_2(n)))$

- *Also with* Θ

Rules to calculate O (3)

3. **condition**: $B = O(f_B(n)) \wedge S_1 = O(f_1(n)) \wedge S_2 = O(f_2(n))$
   $\Rightarrow \boxed{\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2} = O(max(f_B(n), f_1(n), f_2(n)))$

- If $f_1(n) \neq f_2(n)$ and $max(f_1(n), f_2(n)) > f_B(n) \leftrightarrow$ **Worst case**
- Average case?
  $\rightarrow f(n)$: average of f_1 and f_2 weighted with the frequencies of each branch
  $\rightarrow O(max(f_B(n), f(n)))$

Rules to calculate O (4)

4. **iteration**: $B; S = O(f_{B,S}(n)) \wedge$ nº iter$= O(f_{iter}(n))$

   $\Rightarrow \boxed{\textbf{while } B \textbf{ do } S} = O(f_{B,S}(n) * f_{iter}(n))$
   iff the cost of iterations does not vary, else: $\sum$ indiv. costs.

   $\Rightarrow \boxed{\textbf{for } i \leftarrow x \textbf{ to } y \textbf{ do } S} = O(f_S(n) *$nº iter$)$
   iff the cost of iterations does not vary, else: $\sum$ indiv. costs.

   - B is to compare 2 integers $= O(1)$; nº iter $= y - x + 1$

Rules to calculate O (5)

1. elementary operation = 1 $\leftrightarrow$ Model of Computation
2. **sequence**: $S_1 = O(f_1(n)) \wedge S_2 = O(f_2(n))$
   $\Rightarrow \boxed{S_1 ; S_2} = O(f_1(n) + f_2(n)) = O(max(f_1(n), f_2(n)))$
   - Also with Θ
3. **condition**: $B = O(f_B(n)) \wedge S_1 = O(f_1(n)) \wedge S_2 = O(f_2(n))$
   $\Rightarrow \boxed{\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2} = O(max(f_B(n), f_1(n), f_2(n)))$
   - If $f_1(n) \neq f_2(n)$ and $max(f_1(n), f_2(n)) > f_B(n) \leftrightarrow$ **Worst case**
   - Average case? $\rightarrow f(n)$: average of f_1 and f_2 weighted with the frequencies of each branch $\rightarrow O(max(f_B(n), f(n)))$
4. **iteration**: $B; S = O(f_{B,S}(n)) \wedge$ n^{o} iter$= O(f_{iter}(n))$
   $\Rightarrow \boxed{\textbf{while } B \textbf{ do } S} = O(f_{B,S}(n) * f_{iter}(n))$
   iff the cost of iterations does not vary, else: $\sum$ indiv. costs.
   $\Rightarrow \boxed{\textbf{for } i \leftarrow x \textbf{ to } y \textbf{ do } S} = O(f_S(n) *$n^{o} iter$)$
   iff the cost of iterations does not vary, else: $\sum$ indiv. costs.
   - B is to compare 2 integers $= O(1)$; n^{o} iter $= y - x + 1$

Rules to calculate O (6)

- Usage of the rules:
  - analysis "from the inside out"
  - analyse subprograms first
  - recursivity: try to treat it as a loop, without solving recurrence relation
- **Example**: $\sum_{i=1}^{n} i^3$

```
    function sum (n:integer) : integer
{1}   s:=0;
{2}   for i:=1 to n do
{3}        s:=s+i*i*i;
{4}   return s
    end function
```

$\Theta(1)$ in $\{3\}$ and there are no variations
$\Rightarrow \Theta(n)$ in $\{2\}$ (rule 4)
$\Rightarrow T(n) = \Theta(n)$ (rule 2)

- *The reasoning already includes approximations*

Selection Sort (3)

```
    procedure Selection Sort (var T[1..n])
{1}     for i:=1 to n-1 do
{2}         minj:=i; minx:=T[i];
{3}         for j:=i+1 to n do
{4}             if T[j]<minx then
{5}                 minj:=j; minx:=T[j]
                end if
            end for;
{6}         T[minj]:=T[i]; T[i]:=minx
        end for
    end procedure
```

Selection Sort (4)

- $\Theta(1)$ in $\{5\}$ (rule 2)
  $\Rightarrow O(max(\Theta(1), \Theta(1), 0)) = \Theta(1)$ in $\{4\}$
  (rule 3: **we aren't in the worst case**)
- $S = \Theta(1)$; nº iter=$n-i \Rightarrow \Theta(n-i)$ in $\{3\}$ (rule 4)
- $\Theta(1)$ in $\{2\}$ and in $\{6\}$ (rule 2)
  $\Rightarrow \Theta(n-i)$ in $\{2\text{-}6\}$ (rule 2)
- $S = \Theta(n-i)$ **varies**: $\begin{cases} i = 1 & \rightarrow \Theta(n) \\ i = n-1 & \rightarrow \Theta(1) \end{cases}$

  $\Rightarrow \sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i$ in $\{1\}$ (rule 4)
  $= (n-1)n - \frac{n(n-1)}{2}$:polynomial of degree 2

  $\Rightarrow T(n) = \Theta(n^2)$ **in any case**

Case analysis: exponentiation (1)

- Power1: $x^n = x * x * \ldots * x$ (loop, n times x)
  Main operation: multiplication
  Number of multiplications? $f_1(n) = n - 1 \Rightarrow T(n) = \Theta(n)$
- Power2 (recursive):
  $$x^n = \begin{cases} x^{\lfloor n/2 \rfloor} * x^{\lfloor n/2 \rfloor} & \text{if n even} \\ x^{\lfloor n/2 \rfloor} * x^{\lfloor n/2 \rfloor} * x & \text{if n odd} \end{cases}$$
  Number of multiplications? $f_2(n)$?

Case analysis: exponentiation (2)

- Power2 (recursive) (Cont'd)
  Calculation of $f_2(n)$:

$$\begin{cases} \text{min: n even in each call} & \rightarrow n = 2^k, k \in Z^+ \leftrightarrow \text{best case} \\ \text{max: n odd in each call} & \rightarrow n = 2^k - 1, k \in Z^+ \leftrightarrow \text{worst case} \end{cases}$$

- *Best case*: $f_2(2^k) = \begin{cases} 0 & \text{if } k = 0 \\ f_2(2^{k-1}) + 1 & \text{if } k > 0 \ (1) \end{cases}$
- *Worst case*: $f_2(2^k - 1) = \begin{cases} 0 & \text{if } k = 1 \\ f_2(2^{k-1} - 1) + 2 & \text{if } k > 1 \ (2) \end{cases}$

$\rightarrow$ *recurrence relations*

Case analysis: exponentiation (3)

- *Best case*: $f_2(2^k) = \begin{cases} 0 & \text{if } k = 0 \\ f_2(2^{k-1}) + 1 & \text{if } k > 0 \ (1) \end{cases}$

$$\begin{array}{llllll} k = & 0 & \rightarrow f_2(& 1 &) = & 0 \\ & 1 & & 2 & & 1 \\ & 2 & & 4 & & 2 \\ & 3 & & 8 & & 3 \\ & \ldots & & & & \end{array}$$

$\Rightarrow$ Induction hypothesis: $\boxed{f_2(2^\alpha) = \alpha}$: $0 \leq \alpha \leq k - 1$

Induction step:

$$\begin{aligned} (1) \rightarrow f_2(2^k) &= f_2(2^{k-1}) + 1 \\ &= (k-1) + 1 \\ &= k \end{aligned}$$

correct explicit form
of the recurrence relation

Case analysis: exponentiation (4)

- *Worst case*: $f_2(2^k-1) = \begin{cases} 0 & \text{if } k=1 \\ f_2(2^{k-1}-1)+2 & \text{if } k>1 \text{ (2)} \end{cases}$

$$\begin{array}{llll} k = & 1 & \rightarrow f_2(\quad 1 \quad) = & 0 \\ & 2 & \quad\quad 3 & 2 \\ & 3 & \quad\quad 7 & 4 \\ & 4 & \quad\quad 15 & 6 \\ & 5 & \quad\quad 31 & 8 \\ & 6 & \quad\quad 63 & 10 \\ & \ldots & & \end{array}$$

$\Rightarrow$ Induction hypothesis: $\boxed{f_2(2^\alpha-1) = 2(\alpha-1)}$: $1 \le \alpha \le k-1$

Induction step: $\quad$ (2)$\rightarrow f_2(2^k-1)$

$$\begin{aligned} f_2(2^k-1) &= f_2(2^{k-1}-1)+2 \\ &= 2(k-1-1)+2 \\ &= 2(k-1) \end{aligned}$$

Case analysis: exponentiation (5)

- $n = 2^k$ (best case):
  $f_2(2^k) = k$ for $k \geq 0$
  $\rightarrow f_2(n) = log_2 n$ for $n = 2^k$ y $k \geq 0$ (as $log_2 2^k = k$)
  $\Rightarrow \boxed{f_2(n) = \Omega(logn)}$
- $n = 2^k - 1$ (worst case):
  $f_2(2^k - 1) = 2(k-1)$ for $k \geq 1$
  $\rightarrow f_2(n) = 2[log_2(n+1) - 1]$ for $n = 2^k - 1$ y $k \geq 1$
  $\Rightarrow \boxed{f_2(n) = O(logn)}$
- $\Rightarrow f_2(n) = \Theta(logn)$
  Model of computation: main operation = multiplication
  $\Rightarrow \boxed{T(n) = \Theta(logn)}$

best case $\leftrightarrow \Omega$

worst case $\leftrightarrow O$