ANALYSIS OF ALGORITHM EFFICIENCY

Gustavo Carvalho
(ghpc@cin.ufpe.br)

Universidade Federal de Pernambuco
Centro de Informática, 50740-560, Brazil

Centro de Informática
UFPE

Universidade Federal de Pernambuco

Agenda

Introduction

Efficiency with respect to:

- Running time: time efficiency / complexity
- Memory space: space efficiency / complexity
  (in addition to the space needed for its input and output)

Efficiency: a function in terms of input size

Introduction

Efficiency with respect to:

- Running time: time efficiency / complexity
- Memory space: space efficiency / complexity
  (in addition to the space needed for its input and output)

Efficiency: a function in terms of input size

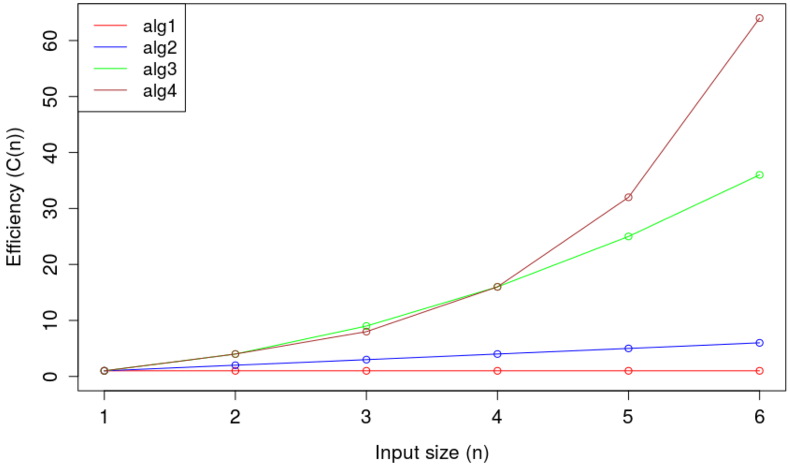
Units for measuring running time

- Absolute: $T(n)$
- Relative: $C(n)$ (in terms of the basic operation *op*)
  - Let c_{op} be the execution time of *op* on a particular computer
    $T(n) \approx c_{op} * C(n)$

Centro de Informática UFPE

Universidade Federal de Pernambuco

Introduction



Introduction[1]

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Performing a trillion (10^{12}) operations per second

- 2^{100} operations would take $4 * 10^{10}$ years
- Estimated age of Earth: $4.5 * 10^9$ years

[1] Source: A. Levitin. Introduction to the Design and Analysis of Algorithms. 2011.

Introduction

In practice, there might be different situations

- Best case
- Average case
- Worst case

Introduction

In practice, there might be different situations

- Best case
- Average case
- Worst case

An alternative: empirical analyses

- Specially for the average case

Agenda

Non-recursive functions: unique elements

In many situations, $C(n)$ can be defined as summation formulae

Algorithm: UniqueElements(A[0..n-1])

for $i \leftarrow 0$ **to** $n-2$ **do**
  for $j \leftarrow i+1$ **to** $n-1$ **do**
    if $A[i] = A[j]$ **then return** *false*;
return *true*;

Parameter for input's size: n

Basic operation: comparison $A[i] = A[j]$

$C(n)$ is not strictly dependent on n

- Best, average, and worst cases

Centro de Informática UFPE

Universidade Federal de Pernambuco

Non-recursive functions: unique elements (*true*)

$$
\begin{aligned}
C_{worst}(n) &= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 \\
&= \sum_{i=0}^{n-2} ((n-1) - (i+1) + 1) \\
&= \sum_{i=0}^{n-2} (n-1-i) \\
&= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i \\
&= (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2} \\
&= (n-1)^2 - \frac{(n-2)(n-1)}{2} \\
&= \frac{2(n^2-2n+1)-(n^2-3n+2)}{2} \\
&= \frac{2n^2-4n+2-n^2+3n-2}{2} \\
&= \frac{n^2-n}{2} \\
&= \frac{(n-1)n}{2}
\end{aligned}
$$

Remember that:

- $\sum_{i=l}^{u} c * a_i = c * \sum_{i=l}^{u} a_i$
- $\sum_{i=l}^{u} (a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i$
- $S_n = \frac{(a_1+a_n)*n}{2}$ (arithm. prog.)

Centro de Informática UFPE

Universidade Federal de Pernambuco

Non-recursive functions: unique elements

For $n \leq 1$

- $C_{best}(0) = 0$
  (unique elements: true)
- $C_{best}(1) = 0$
  (unique elements: true)

For an arbitrary $n > 1$

- $C_{best}(n) = 1$
  (unique elements: false)

Non-recursive functions: sequential search

Algorithm: SequentialSearch(A[0..n-1], K)

$i \leftarrow 0$;
while $i < n \wedge A[i] \neq K$ **do**
  $i \leftarrow i + 1$;
if $i < n$ **then return** i;
else return -1;

Parameter for input's size: n

Basic operation: comparison $A[i] \neq k$

$C(n)$ is not strictly dependent on n

- Best, average, and worst cases

Centro de Informática UFPE
Universidade Federal de Pernambuco

Non-recursive functions: sequential search (-*1*)

Assuming short-circuit evaluation

For $n = 0$

- $C_{worst}(0) = 0$

For an arbitrary $n \geq 1$

$$
\begin{aligned}
C_{worst}(n) &= \sum_{i=0}^{n-1} 1 \\
&= (n - 1) - 0 + 1 \\
&= n
\end{aligned}
$$

Centro de Informática UFPE

Universidade Federal de Pernambuco

Non-recursive functions: sequential search

Assuming short-circuit evaluation

For $n = 0$

- $C_{best}(0) = 0$
  (sequential search: -1)

For an arbitrary $n \geq 1$

- $C_{best}(n) = 1$
  (sequential search: 0)

Centro de Informática UFPE

Universidade Federal de Pernambuco

Non-recursive functions: sequential search

Assuming:

- Probability of successful search: p, where $0 \leq p \leq 1$
- Same probability for every i

$$
\begin{aligned}
C_{avg}(n) &= (1 * \tfrac{p}{n} + 2 * \tfrac{p}{n} + \cdots + i * \tfrac{p}{n} + \cdots + n * \tfrac{p}{n}) + n * (1 - p) \\
&= \tfrac{p}{n}(1 + 2 + \cdots + i + \cdots + n) + n(1 - p) \\
&= \tfrac{p}{n}\tfrac{(1+n)n}{2} + n(1 - p) \\
&= \tfrac{p(n+1)}{2} + n(1 - p)
\end{aligned}
$$

Non-recursive functions: sequential search

Assuming:

- Probability of successful search: p, where $0 \leq p \leq 1$
- Same probability for every i

$$
\begin{aligned}
C_{avg}(n) &= (1 * \tfrac{p}{n} + 2 * \tfrac{p}{n} + \cdots + i * \tfrac{p}{n} + \cdots + n * \tfrac{p}{n}) + n * (1 - p) \\
&= \tfrac{p}{n}(1 + 2 + \cdots + i + \cdots + n) + n(1 - p) \\
&= \tfrac{p}{n}\tfrac{(1+n)n}{2} + n(1 - p) \\
&= \tfrac{p(n+1)}{2} + n(1 - p)
\end{aligned}
$$

Typically, $C_{avg}(n)$ is more difficult to be defined

- For some algorithms, $C_{avg} \ll C_{worst}$

Centro de Informática UFPE

Universidade Federal de Pernambuco

Agenda

Recursive functions: Factorial

Typically, $C(n)$ can be defined as recurrence relations

Algorithm: F(n)

if $n = 0$ **then return** *1*;
else return $F(n-1) * n$;

Parameter for input's size: n

Basic operation: multiplication $F(n-1) * n$

- Assuming that the cost of $*$ is the same for any two numbers

$C(n)$ is strictly dependent on n

Recurrence relation:

- $C(0) = 0$, for $n = 0$
- $C(n) = C(n-1) + 1$, for $n > 0$

Centro de Informática UFPE

Universidade Federal de Pernambuco

Recursive functions: Factorial

Solving the recurrence using the method of backward substitutions

$$
\begin{aligned}
C(n) &= C(n-1)+1 \\
&= (C(n-1-1)+1)+1 = C(n-2)+2 \\
&= (C(n-2-1)+1)+2 = C(n-3)+3 \\
&= \cdots \\
&= C(n-i)+i \\
&= \cdots \\
&= C(n-n)+n \\
&= C(0)+n \\
&= 0+n \\
&= n
\end{aligned}
$$

Recursive functions: binary search[2]

Binary search: a decrease-and-conquer approach

- Sequential search: a brute force approach

Assuming that the input is a sorted array in ascending order

- Let A be the input array
- Let K be the key that is being searched

$$\begin{array}{c} K \\ \updownarrow \\ \underbrace{A[0] \ldots A[m-1]}_{\substack{\text{search here if} \\ K < A[m]}} \quad A[m] \quad \underbrace{A[m+1] \ldots A[n-1]}_{\substack{\text{search here if} \\ K > A[m]}}. \end{array}$$

[2] Source: A. Levitin. Introduction to the Design and Analysis of Algorithms. 2011.

Recursive functions: binary search[3]

Assuming that $K = 70$

3	14	27	31	39	42	55	70	74	81	85	93	98

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	3	14	27	31	39	42	55	70	74	81	85	93	98
iteration 1	l						m						r
iteration 2								l		m			r
iteration 3								l,m	r				

[3] Source: A. Levitin. Introduction to the Design and Analysis of Algorithms. 2011.

Recursive functions: binary search

Algorithm: BS(A[0..n-1],l,r,K))

```
if r ≥ l then
    m ← ⌊(l + r)/2⌋;
    if K = A[m] then
        return m;
    else if K < A[m] then
        return BS(A, l, m − 1, K);
    else
        return BS(A, m + 1, r, K);
else
    return −1;
```

Centro de Informática UFPE

Universidade Federal de Pernambuco

Recursive functions: binary search

Algorithm: BS(A[0..n-1],l,r,K))

```
if r ≥ l then
    m ← ⌊(l + r)/2⌋;
    if K = A[m] then
        return m;
    else if K < A[m] then
        return BS(A, l, m − 1, K);
    else
        return BS(A, m + 1, r, K);
else
    return −1;
```

Algorithm: BS(A[0..n-1],K)

```
l ← 0;
r ← n − 1;
while l ≤ r do
    m ← ⌊(l + r)/2⌋;
    if K = A[m] then
        return m;
    else if K < A[m] then
        r ← m − 1;
    else
        l ← m + 1;
return −1;
```

Centro de Informática UFPE

Universidade Federal de Pernambuco

Recursive functions: binary search

Parameter for input's size: n

Basic operation: comparison between A and K

- Assuming that a single comparison is enough, which is reasonable

$C(n)$ is not strictly dependent on n

- Best, average, and worst cases

Recurrence relation:

- $C_{worst}(n) = 0$, for $n = 0$
- $C_{worst}(n) = 1 + C_{worst}(\lfloor n/2 \rfloor)$, for $n > 0$

Centro de Informática UFPE

Universidade Federal de Pernambuco

Recursive functions: binary search

Assuming (for simplicity) that both sides of A have the same size

$$C_{worst}(n) = C_{worst}(\frac{n-1}{2}) + 1$$

$$C_{worst}(\frac{n-1}{2}) = 1 + C_{worst}(\frac{\frac{n-1}{2}-1}{2}) = 1 + C_{worst}(\frac{n-2^0-2^1}{2^2})$$

$$C_{worst}(\frac{n-2^0-2^1}{2^2}) = 1 + C_{worst}(\frac{n-2^0-2^1-2^2}{2^3})$$

$$\vdots$$

$$C_{worst}(\frac{n-2^0-2^1-\cdots-2^j}{2^{j+1}}) = 1 + C_{worst}(\frac{n-2^0-2^1-\cdots-2^{j+1}}{2^{j+2}})$$

The recurrence ends when the argument of C_{worst} is equal to 0:

$$C_{worst}(\frac{n-2^0-2^1-\cdots-2^{j+1}}{2^{j+2}}) = C_{worst}(0) = 0$$

Centro de Informática UFPE

Universidade Federal de Pernambuco

Recursive functions: binary search

It is also true that:

$$\begin{aligned} C_{worst}(n) &= 1 + 1 + 1 + \cdots + 1 + C_{worst}(0) \\ &= 1 + 1 + 1 + \cdots + 1 + 0 \end{aligned}$$

We have $j + 2$ sums of 1 before reaching $C_{worst}(0)$:

$C_{worst}(n) = (j + 2) * 1 + 0 = j + 2$

Centro de Informática UFPE

Universidade Federal de Pernambuco

Recursive functions: binary search

Remember that

- $\frac{n-2^0-2^1-\cdots-2^{j+1}}{2^{j+2}} = 0$
- $S_n = \frac{a_1*(q^n-1)}{q-1}$ (geometric progression)

We can find the value of j as follows:

$$n = 2^0 + 2^1 + \cdots + 2^{j+1}$$

$$n = \sum_{i=0}^{j+1} 2^i$$

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

$$n = 2^{j+2} - 1$$

$$n + 1 = 2^{j+2}$$

Recursive functions: binary search

Applying *log_2* on both sides:

$$\begin{aligned} log_2(n+1) &= log_2 2^{j+2} \\ &= (j+2) \cdot log_2 2 \\ &= (j+2) \end{aligned}$$

Therefore, we have that $j = log_2(n+1) - 2$.

And $C_{worst}(n)$ can be defined as follows:

$$\begin{aligned} C_{worst}(n) &= j+2 \\ &= log_2(n+1) - 2 + 2 \\ &= log_2(n+1) \end{aligned}$$

Agenda

Asymptotic analysis

How do we compare the efficiency of different algorithms?

- $log_2(n+1)$ vs. $log_2 n$
- $n^3 + 100n^2 + 10000$ vs. n^3

We compare the order of growth

- O: big oh (no worse than)
- Ω: big omega (no better than)
- Θ: big theta (similar to)

Centro de Informática UFPE

Universidade Federal de Pernambuco

Asymptotic analysis

$O, \Omega, \Theta : \mathcal{F} \nrightarrow \mathbb{P}\,\mathcal{F}$

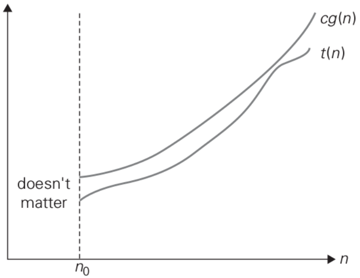
- $\mathcal{F}$ the set of all functions
- $dom(O) = dom(\Omega) = dom(\Theta)$ = non-negative functions over $\mathbb{N}$

Examples

- $O(n^2)$ = all non-negative functions with an order of growth no worse than n^2
- $\Omega(n^2)$ = all non-negative functions with an order of growth no better than n^2
- $\Theta(n^2)$ = all non-negative functions with an order of growth similar to n^2

Asymptotic analysis[4]

$$\forall\, t, g \bullet t \in dom(O) \land g \in dom(O) \Rightarrow$$
$$t \in O(g) \Leftrightarrow$$
$$\exists\, c, n_0 \bullet c \in \mathbb{R} \land c > 0 \land$$
$$n_0 \in \mathbb{N} \land n_0 \geq 0 \land$$
$$\forall\, n \bullet n \in \mathbb{N} \land n \geq n_0 \Rightarrow$$
$$t(n) \leq c * g(n)$$



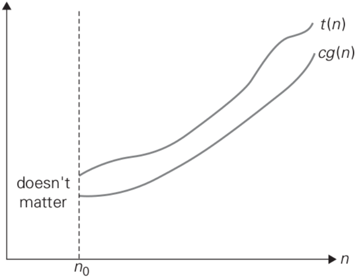
Example:

- Let $t(n) = 100n + 5$ and $g(n) = n$, $t \in O(g)$ is true
- Also said that $100n + 5 \in O(n)$
- Proof (informal)
  - $n_0 = 1$
  - $c = 105$

[4] Source: A. Levitin. Introduction to the Design and Analysis of Algorithms. 2011.

Asymptotic analysis[5]

$$\begin{array}{l}\forall t, g \bullet t \in dom(\Omega) \wedge g \in dom(\Omega) \Rightarrow \\ \quad t \in \Omega(g) \Leftrightarrow \\ \quad \exists c, n_0 \bullet c \in \mathbb{R} \wedge c > 0 \wedge \\ \qquad n_0 \in \mathbb{N} \wedge n_0 \geq 0 \wedge \\ \qquad \forall n \bullet n \in \mathbb{N} \wedge n \geq n_0 \Rightarrow \\ \qquad\quad t(n) \geq c * g(n)\end{array}$$



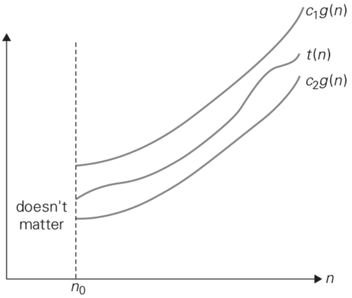
Example:

- $n^3 \in \Omega(n^2)$
- Proof (informal)
  - $n_0 = 0$
  - $c = 1$

[5] Source: A. Levitin. Introduction to the Design and Analysis of Algorithms. 2011.

Asymptotic analysis[6]

$$\forall t, g \bullet t \in dom(\Theta) \wedge g \in dom(\Theta) \Rightarrow$$
$$t \in \Theta(g) \Leftrightarrow$$
$$\exists c_1, c_2, n_0 \bullet c_1 \in \mathbb{R} \wedge c_1 > 0 \wedge$$
$$c_2 \in \mathbb{R} \wedge c_2 > 0 \wedge$$
$$n_0 \in \mathbb{N} \wedge n_0 \geq 0 \wedge$$
$$\forall n \bullet n \in \mathbb{N} \wedge n \geq n_0 \Rightarrow$$
$$c_2 * g(n) \leq t(n) \leq c_1 * g(n)$$



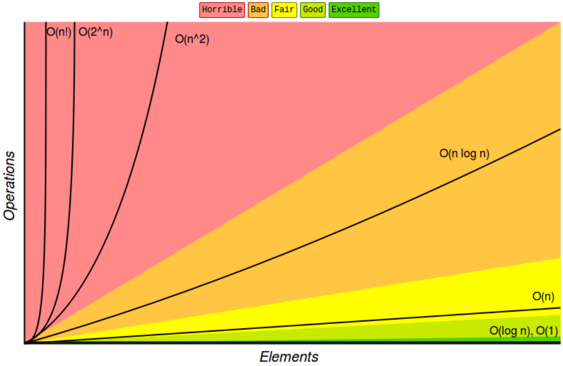
Exemple:

- $\frac{(n-1)n}{2} \in \Theta(n^2)$
- Proof (informal)
  - $n_0 = 2$
  - $c_1 = 1/2$
  - $c_2 = 1/4$

[6] Source: A. Levitin. Introduction to the Design and Analysis of Algorithms. 2011.

Basic asymptotic efficiency classes[7]

constant	$\triangleq 1$
logarithmic	$\triangleq log\ n$
linear	$\triangleq n$
lineararithmic	$\triangleq n\ log\ n$
quadratic	$\triangleq n^2$
cubic	$\triangleq n^3$
exponential	$\triangleq 2^n$
factorial	$\triangleq n!$



[7] Source: http://bigocheatsheet.com/

Asymptotic efficiency related to some data structures[8]

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	O(1)	O(1)	O(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	O(log(n))	O(log(n))	O(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	O(log(n))	O(log(n))	O(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Centro de Informática UFPE

Universidade Federal de Pernambuco

[8] Source: http://bigocheatsheet.com/

Using limits for comparing orders of growth[9]

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & t(n) \text{ has a smaller order of growth than } g(n) \\ c & t(n) \text{ has the same order of growth as } g(n) \\ \infty & t(n) \text{ has a larger order of growth than } g(n) \end{cases}$$

Example

$$\lim_{n\to\infty} \frac{\frac{(n-1)n}{2}}{n^2} = \frac{1}{2} \lim_{n\to\infty} \frac{n^2 - n}{n^2} = \frac{1}{2} \lim_{n\to\infty} (1 - \frac{1}{n}) = \frac{1}{2}$$

therefore, $\frac{(n-1)n}{2} \in \Theta(n^2)$

[9] The limit may not exist, but this rarely happens. Still, this approach is less general than the one based on the definitions of O, Ω, and Θ

Useful property involving asymptotic notations

Theorem: $\forall t_1, t_2, g_1, g_2 \bullet \{t_1, t_2, g_1, g_2\} \subseteq dom(O) \wedge$
$t_1 \in O(g_1) \wedge t_2 \in O(g_2) \Rightarrow t_2 \circ t_1 \in O(max(g_1, g_2))$

The same applies for Ω and Θ

Practical implication: the algorithm's overall efficiency is determined by the part with a higher order of growth

Useful property involving asymptotic notations

Theorem: $\forall\, t_1, t_2, g_1, g_2 \bullet \{t_1, t_2, g_1, g_2\} \subseteq dom(O) \wedge$
$t_1 \in O(g_1) \wedge t_2 \in O(g_2) \Rightarrow t_2 \circ t_1 \in O(max(g_1, g_2))$

The same applies for Ω and Θ

Practical implication: the algorithm's overall efficiency is determined by the part with a higher order of growth

Example: *algorithm* = *sorting* + *binary search*

- Let $C_{worst}(n) \in O(n\ log\ n)$ for *sorting*
- Therefore, $C_{worst}(n) \in O(n\ log\ n)$ for *algorithm*
- The order of growth of *n log n* is larger than *n* (sequential search)
- Is *algorithm* worse than *sequential search*?

Agenda

Amortised analysis

Main idea: to amortise the efficiency over time (i.e., not a single run of the algorithm, but a sequence of runs)

- Some runs will have a different efficiency
- Compute $C_{amort}(n) = \frac{C(n)}{n}$, where $C(n)$ is the cost of n-runs

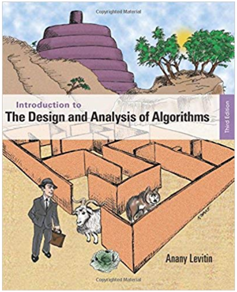
Example: *algorithm* = *sorting* + *binary search*

- First run: *sorting* + *binary search*
- Second run: *binary search*
- n-th run: *binary search*
- Amortised analysis
  - $C_{amort}(n) = \frac{1*(n\ log\ n)+n*log\ n}{n} = log\ n + log\ n = 2 * log\ n$
  - $2 * log\ n \in \Theta(log\ n)$

Centro de Informática UFPE

Universidade Federal de Pernambuco

Bibliography



Chapter 2 (pp. 41–95), Chapter 3 (pp. 97–98,104), Chapter 4 (pp. 150–152. Anany Levitin. *Introduction to the Design and Analysis of Algorithms*. 3rd edition. Pearson. 2011.

Centro de Informática UFPE

Universidade Federal de Pernambuco

ANALYSIS OF ALGORITHM EFFICIENCY

Gustavo Carvalho
(ghpc@cin.ufpe.br)

Universidade Federal de Pernambuco
Centro de Informática, 50740-560, Brazil