COMPUTER PROGRAMMING
Unit 2
Analysis of Algorithms. Efficiency. Sorting

Jon Ander Gómez Adrián

jon@dsic.upv.es

Departament de Sistemes Informàtics i Computaciò
Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

25 de enero de 2018

UNIVERSITAT POLITÈCNICA DE VALÈNCIA
DSIC
Escola Tècnica Superior d'Enginyeria Informàtica
etsinf

Contents

Bibliography

- "Empezar a programar usando Java"
  Profesores de las asignaturas IIP y PRG
  Apuntes facilitados vía *PoliformaT*
  Capítulos 12 y 13

- "Introducció a l'anàlisi i disseny d'algorismes"
  Ferri, F,J., Albert, F.V., Martín, G. – Universitat de València, 1998
  Chapter 2
  Chapter 3, except subsections from 3.3.2 to 3.3.5
  Chapter 5, sections from 5.1 to 5.3, and subsection 5.4.1
  **** This book is specially recommended for people able to read and understand the Valencian language.*

- "Estructuras de datos en Java: compatible con Java 2"
  Mark Allen Weiss – Ed. Addison Wesley, 2000 - 2006
  Chapter 5
  Chapter 8, sections from 8.1 to 8.3

- "Fundamentos de Algoritmia"
  G. Brassard y P. Bratley – Pearson – Prentice Hall, 2001
  Chapter 2, sections from 2.1 to 2.6
  Chapter 4, sections from 4.1 to 4.4

- "Una introducción a la programación. Un enfoque algorítmico"
  García, J., Montoya, F, et al. – Ed. Thomson, 2005
  Chapter 6, sections 6.3, 6.4 and 6.5

Introduction to the analysis of algorithms – Concepts

Algorithm

An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.

An algorithm is thus a sequence of computational steps that transform the input into the output.

The specification of an algorithm must provide a precise description of the computational procedure to be followed.

Introduction to the analysis of algorithms – Concepts

Efficiency

If we need to sort a collection of items, which algorithm is the best for a given application depends on –among other factors– the number of items to be sorted, the extent in that the items are already sorted, possible restrictions on the item values, and the kind of storage device to be used. *RAM memory, disk file, database table*

- We need an independent criterion to decide which algorithm is the best one: *Efficiency.*
- The most efficient program is the one which uses the minimal amount of resources when it is executed.
- CPU time and RAM memory are the basic and most important resources.

Introduction to the analysis of algorithms – Concepts

Input size

- The notion of input size depends on the problem being studied.
- For many problems, such as sorting, the most natural measure is the total number of items to be sorted.
- Sometimes, it is more appropriate to describe the size of the input with two numbers rather than one.

  For instance, if the input to an algorithm is a graph, the input size can be described by the numbers of vertices and edges in the graph. To compute the $n!$ the input size is the value of n.
- Before analyzing an algorithm which solves a problem, or kind of problems, we have to determine the input size measure.

Introduction to the analysis of algorithms – Concepts

Instance of a problem

- An instance of a problem consists of the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem.
- It will be usual to deal with algorithms whose running time, for the same input size, depends on the way input data are arranged.
- Using the same example of sorting problem, several sorting algorithms have a better running time if the input items are sorted. Obviously, they will spend more time if the input data are in the reverse order.
- We will use *sample* as synonym of instance of input data.

Introduction to the analysis of algorithms – Concepts

How long an algorithm takes to produce its result?

- **Temporal cost.** A measure of the CPU time that a program consumes when it is executed for the input data.

How much space an algorithm takes to produce its result?

- **Spatial cost.** A measure of the amount of RAM memory that a program uses when it is executed for the input data.

Introduction to the analysis of algorithms – Concepts

The costs of a particular program or algorithm depend on two types of factors.

- **Intrinsic program factors**, such as the strategy and the data structures it uses.
- **Factors of the programming environment**, such as the type of computer, programming language, compiler, system load, etc.

$$f(n) = \alpha \cdot f_{intrinsic}(n) + \beta$$

Introduction to the analysis of algorithms – Concepts

The costs of a program can be calculated in two ways.

- **Theoretical analysis**, also known as *a priori* analysis.

  The cost is estimated in function of the program intrinsic factors.

  It is an analysis which is independent of the programming environment.

- **Empirical analysis**, also known as *a posteriori* analysis.

  The cost is computed by measuring, in seconds, the time that the program spends when it is executed, and, in bytes, the amount of memory used while it was in execution.

  This is an analysis performed in a particular programming environment and using a particular set of input data (a sample or instance).

Introduction to the analysis of algorithms – Concepts

Some important details.

- Both kinds of analysis are important, in fact they are complementary. Usually we have to do both analyses, the theoretical analysis to predict the growth rate of the temporal cost function of algorithms, and the empirical one to test if the codification corresponds to algorithms designed.

- Before programming, good programmers do a theoretical analysis of algorithms designed in order to avoid wasting time.

- Efficiency is a criterion that one must keep in mind when designing algorithms.

- A program, or algorithm, must be efficient independently of the computer it runs.

Time and space cost of computer programs

The temporal cost of an algorithm is measured based on the time of execution of elementary operations.

A1:

```
m = n*n;
```

A2:

```
m = 0;
for( int i=0; i < n; i++ ) m+=n;
```

A3:

```
m = 0;
for( int i=0; i < n; i++ )
    for( int j=0; j < n; j++ ) m++;
```

Time and space cost of computer programs

The temporal cost of an algorithm is defined as the sum of the execution times of the elementary operations involved.

$$
\begin{aligned}
T_{A1}(n) &= t_a + t_{op} \\
T_{A2}(n) &= t_a + t_a + (n+1) \cdot t_c + n \cdot t_a + 2 \cdot n \cdot t_{op} \\
T_{A3}(n) &= t_a \\
&+ t_a + (n+1) \cdot t_c + n \cdot t_{op} \\
&+ n \cdot (t_a + (n+1) \cdot t_c + n \cdot t_{op}) \\
&+ n^2 \cdot t_{op}
\end{aligned}
$$

Where t_a is the cost of assignment, t_c is the cost of comparison, and t_{op} is the cost of arithmetic operations.

Time and space cost of computer programs

The comparison of temporal costs using expressions like the previous ones is a hard task.

A first step to simplify the analysis is do not use the detail of the cost of elemntary operations, so ...

$$
\begin{aligned}
T_{A1}(n) &= t_a + t_{op} \equiv k_1 \\
T_{A2}(n) &= t_a + t_a + (n+1)\cdot t_c + n\cdot t_a + 2\cdot n\cdot t_{op} \equiv k_2\cdot n + k_3 \\
T_{A3}(n) &= t_a \\
&+ t_a + (n+1)\cdot t_c + n\cdot t_{op} \\
&+ n\cdot(t_a + (n+1)\cdot t_c + n\cdot t_{op}) \\
&+ n^2\cdot t_{op} \equiv k_4\cdot n^2 + k_5\cdot n + k_6
\end{aligned}
$$

Time and space cost of computer programs

The temporal cost as a function which depends on the input size.

- The **temporal cost** (or spatial cost) of an algorithm is defined as *"a non decreasing function of the amount of CPU time (or RAM memory) it needs to be executed depending on the input size."*
- The CPU time needed depends on the amount of data to be processed.
- The **input size** of a problem is defined as *"the value (or set of values) related to input data which represents a measure of difficulty for its resolution."*

Time and space cost of computer programs

Determining the input size of a problem.

Examples:

Problem	**Input size**
Search for an item within a set	Number of elements in the set
Product of matrices	Size of the matrices
Calculating $n!$	Value of n
Solving a system of linear equations	Number of equations or unknowns
Sort a collection of items	Number of elements to be sorted

Time and space cost of computer programs

From now on,

- the temporal cost function of an algorihtm A is expressed as $T_A(n)$, where n represents the input size, and
- the temporal cost function represents the number of program steps.

Then, from the previous examples we can rewrite the temporal cost functions as

- $T_{A_1}(n) = 1$ program step
- $T_{A_2}(n) = n + 2$ program steps
- $T_{A_3}(n) = n^2 + n + 2$ program steps

Time and space cost of computer programs

Let's define a new concept.

- **Program step.** Sequence of one or more elementary operations with a running time independent of the problem input size.
- A program step is considered a valid unit of time to express the cost of an algorithm.
- So, the analysis of the cost of algorithms gets independent of the running time of elementary operations.

Time and space cost of computer programs

Determining an elementary operation as reference of program step.

Problem	**Elementary operation**
Search for an item within a set	Comparison between the item and each element of the set
Product of matrices	Product of components of matrices
Calculating $n!$	The product
Solving a system of linear equations	The sum
Sort a collection of items	Comparison between each pair of items

Growth of functions – Asymptotic notation

- The temporal cost of an algorithm is expressed as a non decreasing function depending on input size, $T(n)$.
- Determine which algorithm is better than others for solving the same problem is as easy as comparing the rates of growth of non-decreasing functions.
- In general, $T(n)$ is a polynomial function, so we are interested in higher-order terms. They are who determine the growth rate when n takes large values.

Growth of functions – Asymptotic notation

Let's see a table with typical functions commonly used

Function	Name
c	constant
$\log n$	logarithmic
$\log^2 n$	squared logarithmic
n	linear
$n \log n$	n-log-n
n^2	squared
n^3	cubic
2^n	exponential

Growth of functions – Asymptotic notation

- Downloading a file from Internet is a good example of linear cost.
- The transfer rate is always the same and the temporal cost depends on the size of the file.
- Linear cost means that the temporal cost is proportional to the input size.
- The linear cost is a reasonable cost, the logarithmic one is better, and for many problems a cost as $n \log n$ is an acceptable growth rate.
- The remaining typical functions set hard limitations to the size of the problems we can solve.

Growth of functions – Asymptotic notation

Using asymptotic notation.

- When analyzing the cost of an algorithm we are interested in its kind of growth.
- By identifying the types of growth as typical functions, we can compare algorithms, and so choose.
- Therefore, the *a priori* analysis of algorithms consists in measuring the growth rate of temporal cost functions, and expressing it by using asymptotic notation.

Growth of functions – Asymptotic notation

Lower and upper bounds of temporal cost functions.

- The order of growth of the running time of an algorithm gives a simple characterization of the algorithm's efficiency,
- and allow us to compare the relative performance of alternative algorithms.
- For large enough inputs, the multiplicative constants and lower-order terms of an exact running time are dominated by the effects of the input size itself.

Growth of functions – Asymptotic notation

Lower and upper bounds of temporal cost functions.

- When we only put our attention in large-enough input sizes, in order to study the growth rate of the running time, we are studying the asymptotic behaviour of algorithms.
- That is, we are concerned how the running time of an algorithm increases as the input size grows *in the limit*.
- Usually, an algorithm that is asymptotically more efficient will be the best choice for all but very small inputs.

Growth of functions – Asymptotic notation

Lower and upper bounds of temporal cost functions.

For a given function $g(n)$, we denote by $O(g(n))$ the *set of functions*

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

For a given function $g(n)$, we denote by $\Omega(g(n))$ the *set of functions*

$$\Omega(g(n)) = \{f(n) : \exists\ c_1 > 0 \text{ and } n_0 > 0 \text{ such that } f(n) \geq c_1 g(n)\ \forall\ n \geq n_0\}$$

$f(n) \in O(g(n)) \implies g(n)$ is an ***asymptotic upper bound*** for $f(n)$

$f(n) \in \Omega(g(n)) \implies g(n)$ is an ***asymptotic lower bound*** for $f(n)$

Growth of functions – Asymptotic notation

Lower and upper bounds of temporal cost functions.

For a given function $g(n)$, we denote by $\Theta(g(n))$ the *set of functions*

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

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\ \forall\ n \geq n_0\}$$

If $f(n) \in \Theta(g(n))$ we say that $g(n)$ is an ***asymptotically tight bound*** for $f(n)$

Θ-notation asymptotically bounds a function from above and below.

Growth of functions – Asymptotic notation

Remember!

- For large enough values of input size,
- the temporal cost function value is determined by the highest order term,
- the remaining terms become no significant.
- The asymptotic notation sets a relative order of temporal cost functions by comparing the highest-order terms.

Growth of functions – Asymptotic notation

Function	Name	Asymptotic notation
c	constant	$\Theta(1)$
$\log n$	logarithmic	$\Theta(\log n)$
$\log^2 n$	squared logarithmic	$\Theta(\log^2 n)$
n	linear	$\Theta(n)$
$n \log n$	n-log-n	$\Theta(n \log n)$
n^2	squared	$\Theta(n^2)$
n^3	cubic	$\Theta(n^3)$
2^n	exponential	$\Theta(2^n)$

Growth of functions – Asymptotic notation

- As computers become faster and faster may seem that it is not worth investing our time in designing more efficient algorithms. And if we wait for the next generation of computers? Why look for efficiency?

- Suppose we have available an algorithm with exponential cost to solve a concrete problem, executed on today computers needs $10^{-4} \times 2^n$ seconds of CPU time to solve the problem given an input of size n.

n	running time	
10	$10^{-4} \times 2^{10}$	$\sim$ 0,1 seconds.
20	$10^{-4} \times 2^{20}$	$\sim$ 2 minutes.
30	$10^{-4} \times 2^{30}$	more than 1 day.
38	$10^{-4} \times 2^{38}$	$\sim$ 1 year.

Growth of functions – Asymptotic notation

- Suppose we buy a new computer one hundred times faster than the last one. Now we can solve, using the same algorithm and also one year of CPU time, an instance of the problem with input size $n = 45$.

- Generally, using the same algorithm, if the old computer takes T seconds for solving an instance of the problem with input size n, the new computer can solve, on the same time, instances with input size $n + \log n$.

Growth of functions – Asymptotic notation

- Suppose that we invest in the design of algorithms and we find an algorithm with a temporal cost function $\in \Theta(n^3)$. This algorithm needs $10^{-2} \times n^3$ seconds for solving instances of the same problem of size n.

n	running time	
10	$10^{-2} \times 10^3$	$\sim$ 10 seconds.
20	$10^{-2} \times 20^3$	more than 1 minute.
30	$10^{-2} \times 30^3$	$\sim$ 4,5 minutes.
200	$10^{-2} \times 200^3$	$\sim$ 1 day.
1500	$10^{-2} \times 1500^3$	$\sim$ 1 year.

Growth of functions – Asymptotic notation

- The new algorithm not only offers an improvement much better than buying new computers, if one can invest in new hardware the new acquisition will become more profitable.

- Generally speaking, improvements on applications due to change of the programming language, or due to buying new computers, are less significant than improvements on the strategy implying a lower growth rate.

Worst, best and avarage cases

- The running time of an algorithm is a non decreasing function depending on the input size.

- Given a particular input size, n, the running time of an algorithm also depends on how the input data are distributed. In other words, it depends on the instance of the problem.

- An instance of a problem represents all possible configurations of input data, for a given input size, for which the temporal cost function of the algorithm is similar.

Worst, best and avarage cases

If when analyzing the behavior of an algorithm we detect significant instances, i.e., the running time is different depending on how input data are distributed, then different cases must be considered:

- worst-case, $T^w(n)$, the algorithm consumes the maximum CPU time for solving the problem.
- best-case, $T^b(n)$, the algorithm consumes the minimum CPU time for solving the problem.
- average-case, $T^\mu(n)$, the algorithm consumes a quantity of CPU time for solving the problem which corresponds to the running time needed for the most probably distribution of input data.

When analyzing algorithms we are interested in worst-case and average-case. Particularly in the worst-case because it represents an upper bound for the running time.

Worst, best and avarage cases

Algorithms for vector traversal and search within vectors

- The vector traversal problem has a unique instance, we can't distinguish between worst-case and best-case.

- The search on vectors has multiple instances: n instances if the element we are looking for exists in the vector, and one if it not exists. When the element exists, it can be found at any position within the vector.

- If the search algorithm performs the linear search from the begin of the vector, we have: **best-case**, the element exists and it is located at the first position, **worst-case**, the element does not exist or it is located at the last position.

Worst, best and avarage cases

Sequence of steps to analyze an algorithm:

1. Determine the input size of the problem, i.e., study which are the parameters the temporal cost function depends on.

2. Select an elementary operation to be considered as a program step.

3. Detect if there exist significant instances for a given input size, i.e., the algorithm behaves differently depending on how input data are distributed.

4. Obtain the temporal cost function $T(n)$. If there are significant instances then obtain the temporal cost function for the best-case $T^b(n)$ and for the worst-case $T^w(n)$.

5. Express the bounds of the temporal cost function by using the asymptotic notation.

Analysis of iterative algorithms

- Detecting one or more *critical instructions* is the first step to obtain the temporal cost function for an iterative algorithm.

- A **critical instruction (or barometer)** is an elementary operation which fulfills the following two conditions:

  1. It is repeated as many times as any other, usually in the most inner loop.
  2. Its running time is independent of input data, and independent of the nature of the problem.

- By considering a critical instruction as a program step we can obtain the temporal cost function. This means writing the expression which represents the growth rate as a function of the input size.

Analysis of iterative algorithms

Efficiency of a traversal algorithm

- Let v a vector of n items, which can be traversed by using an algorithm like the following:

```
for( int i=0; i <n; i++ ) op(v[i]);
```

- Suppose that `op(v[i])` is an operation with a fixed running time.
- The temporal cost function $T(n)$ depends on the input size, n.
- The algorithm has no significant instances, $T(n)$ can be obtained by counting the number of times `op(v[i])` is executed.
- The possible critical instructions are: `op(v[i]), i++, i<n`
- The cost of the algorithm is: $T(n) = n$, so $T(n) \in \Theta(n)$

Analysis of iterative algorithms

Efficiency of a linear search algorithm

- Let v a vector with n items, the following algorithm searches in v the position of an item which fulfills a predefined property.

```
int i=0;
while( i < n  &&  !(PROPERTY(v[i]) ) i++;
return ( i < n ) ? i : -1;
```

- $PROPERTY(e)$ is the property that the element we are looking for must fulfill.
- The temporal cost function depends on the input size n.
- In this case the temporal cost function also depends on how the input data are distributed.

Analysis of iterative algorithms

Efficiency of a linear search algorithm

- This algorihtm presents $n+1$ significant instances.
- We consider the loop condition as the critical instruction:

```
i<n && !PROPERTY(v[i])
```

- Worst-case: $T^w(n) = n \in \Theta(n)$ linear
- Best-case: $T^b(n) = 1 \in \Theta(1)$ constant
- Upper bound: $T(n) \in O(n)$
- Lower bound: $T(n) \in \Omega(1)$

Analysis of iterative algorithms

Efficiency of a linear search algorithm

- We need to know the probability distribution of the different instances of the problem for estimating the temporal cost in the average case.

- Simplifying, we consider two possibilities:

  1. The search ends with success, and the probability that the element we are looking for is located at any position is the same.
  2. The probability that the element we are looking for is found in the vector is the same that it is not found. If it is found in the vector, then we consider the previous possibility.

Analysis of iterative algorithms

Efficiency of a linear search algorithm. First possibility

- There are n possible instances, each one with a probability of appearance equal to $\frac{1}{n}$

- The cost of each instance is the number of times the loop is repated, and matches with the position where the element i-th is located.

$$T^{\mu}(n) = \sum_{i=1}^{n} \frac{i}{n} = \frac{1}{n} \sum_{i=1}^{n} i = \frac{n(n+1)}{2n} = \frac{(n+1)}{2} \in \Theta(n)$$

Analysis of iterative algorithms

Efficiency of a linear search algorithm. Second possibility

- There are $n+1$ possible instances:
  - The element is not found within the vector with a probability equal to $\frac{1}{2}$
  - The element is found in any position of the vector with probability $\frac{1}{2n}$

- The cost of the first instance is $n+1$, the cost of the remaining instances is i, being i the position where the element is located.

$$\begin{aligned} T^{\mu}(n) &= \tfrac{n+1}{2} + \sum_{i=1}^{n} \tfrac{i}{2n} = \tfrac{n+1}{2} + \tfrac{n(n+1)}{4n} \\ &= \tfrac{2n(n+1)}{4n} + \tfrac{n(n+1)}{4n} = \tfrac{3n(n+1)}{4n} = \tfrac{3}{4}(n+1) \in \Theta(n) \end{aligned}$$

Analysis of recursive algorithms

The temporal cost function of a recursive algorithm is influenced by the following factors:

- The number of recursive calls every time the method is executed.
- The decreasing rate of the input size. Usually it can be a substraction $(n - c)$ or a division (n/c).

  c is a constant, $c > 1$ in the case of division.
- The temporal cost of the reamaining instructions excluding the recursive calls.

Analysis of recursive algorithms

- Commonly, relations of recurrence are used for analyzing recursive algorithms.
- Relations of recurrence allow us to obtain $T(n)$ for the different cases of recursive algorithms, namely, trivial case and general case.
- The *substitution method* is the most commonly used technique for resolving recurrences, therefore, it is very useful for obtaining the running time $T(n)$ of a recursive algorithm.
- It is based on mathematical induction. All we need is the inductive hypotesis.

Analysis of recursive algorithms

Example: analysis for $n!$

```
int factorial( int n )
{
    if ( n == 0 ) return 1;
    else return n*factorial(n-1);
}
```

- **Input size:** n, the value of the argument.
- **Base case:** $n = 0$, $T(n) = k_1$ constant.
- **General case:** $n > 0$, $T(n) = T(n-1) + k_2$
- **Relation of recurrence** or inductive hipotesis:

$$T(n) = \begin{cases} k_1 & n = 0 \\ T(n-1) + k_2 & n > 0 \end{cases}$$

Analysis of recursive algorithms

Example: analysis for $n!$

- Let's see how to apply the substitution method for $5!$

$$T(n) = \begin{cases} k_1 & n = 0 \\ T(n-1) + k_2 & n > 0 \end{cases}$$

$$\begin{aligned} T(5) &= T(4) + k_2 \\ &= T(3) + k_2 + k_2 = T(3) + 2 \cdot k_2 \\ &= T(2) + 3 \cdot k_2 \\ &= T(1) + 4 \cdot k_2 \\ &= T(0) + 5 \cdot k_2 = k_1 + 5 \cdot k_2 \end{aligned}$$

$$T(n) = k_1 + n \cdot k_2 \in \Theta(n)$$

Analysis of sorting algorithms

http://www.sorting-algorithms.com

- Algorithms based on comparisons.
  - Slow algorithms.
    - Selection-Sort
    - Insertion-Sort
    - Bubble-Sort
  - Fast algorithms.
    - Merge-Sort
    - Quick-Sort
    - Heap-Sort
- Algorithms not based on comparisons.
  - Counting-Sort
  - Radix-Sort
  - Bucket-Sort

Analysis of sorting algorithms

The Selection-Sort algorithm

```
void selectionSort( int V[] )
{
    for( int i=0; i < V.length-1; i++ ) {
        int posMin = i;
        for( int j=i+1; j < V.length; j++ ) {
            if ( V[j] < V[posMin] ) posMin=j;
        }
        int aux = V[posMin];
        V[posMin] = V[i];
        V[i] = aux;
    }
}
```

- At any iteration of the external loop, the subvector $V[0..i-1]$ is sorted, and all the values contained in it are smaller than any of the values contained in the subvector $V[i..n]$
- The statement of finding the minimum is a vector traversal operation.

Analysis of sorting algorithms

The Selection-Sort algorithm

```
void selectionSort( int V[] ) {
    for( int i=0; i < V.length-1; i++ ) {
        int k = posMin( V, i, V.length-1 );
        swap( V, i, k );
    }
}
int posMin( int V[], int start, int end ) {
    int pos=start;
    for( int i=start+1; i <= end; i++ ) {
        if ( V[i] < V[pos] ) pos=i;
    }
    return pos;
}
void swap( int V[], int i, int j ) {
    int aux = V[i]; V[i] = V[j]; V[j] = aux;
}
```

Analysis of sorting algorithms

Analysis of the Selection-Sort algorithm

- The input size is the number of elements to be ordered: `V.length`
- The condition of the inner loop is the best critical instruction: `j <V.length`
- The behavior of this algorithm does not change depending on how the numbers are arranged within the vector, so it doesn't present significant instances.

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

Analysis of sorting algorithms

The Insertion-Sort algorithm

```
void insertionSort( int V[] )
{
    for( int i=1; i < V.length; i++ ) {
        int j = i, x = V[i];
        while( j > 0 && x < V[j-1] ) {
            V[j] = V[j-1];
            j--;
        }
        V[j] = x;
    }
}
```

- This algorithm maintains the subvector `V[1..i-1]` sorted. The inner loop puts the new element `V[i]` in the correct position.
- At each iteration of the outer loop a new element is correctly inserted. The sorted subvector grows up in one element.

Analysis of sorting algorithms

Analysis of the Insertion-Sort algorithm

- The input size is the number of elements to be ordered: `V.length`
- The condition of the inner loop is the best critical instruction: `j >0 && V[j] <V[j-1]`
- The inner loop can stop before `j>0` evaluates false. So, we can say this algorithm has significant instances.
- It behaves differently depending on whether the vector is sorted or reversely sorted.
- When the vector is unsorted the inner loop will iterate an undefined number of times, depending on the correct position for the new value `V[i]`.

Analysis of sorting algorithms

Analysis of the Insertion-Sort algorithm

- Best-case analysis. When the vector is already sorted the inner loop never iterates, the condition is always evaluated to false. In this case:

$$T^b(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n) \implies T(n) \in \Omega(n)$$

- Worst-case analysis. When the vector is inversely sorted the inner loop iterates until `j>0` becomes false, i.e., it iterates `i` times. In this case:

$$T^w(n) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \in \Theta(n^2) \implies T(n) \in O(n^2)$$

Analysis of sorting algorithms

The Bubble-Sort algorithm

```
void bubbleSort( int V[] )
{
    for( int i=1; i < V.length; i++ ) {
        for( int j=1; j < V.length; j++ ) {
            if ( V[j] < V[j-1] ) swap( V, j, j-1 );
        }
    }
}
```

- Simple and inefficent sorting algorithm.
- It traverses the vector and interchanges those pairs of consecutive values that they are not in the correct order.
- As can be expected, the computational complexity of this algorithm is $\Theta(n^2)$.
- How do you think could be improved?

Analysis of sorting algorithms

The Natural-Merge algorithm

- Statement: given two sorted vectors, merge them to obtain a new sorted vector. The size of input vectors can differ.

- The Natural-Merge algorithm solves the problem in two stages

  1. The first stage compares items from both vectors and copies them into the new vector. This stage ends when the end of one of the input vectors is reached.
  2. The second stage copies the remaining elements without comparing.

Analysis of sorting algorithms

The Natural-Merge algorithm

```
int [] naturalMerge( int a[], int b[] )
{
    int c[] = new int [ a.length + b.length ];
    int k=0,i=0,j=0;

    while( i < a.length && j < b.length ) {
        if ( a[i] < b[j] ) {
            c[k++] = a[i++];
        } else {
            c[k++] = b[j++];
        }
    }
    while( i < a.length ) c[k++] = a[i++];
    while( j < b.length ) c[k++] = b[j++];

    return c;
}
```

Analysis of sorting algorithms

Analysis of the Natural-Merge algorithm

- **Input size:** `n = a.length + b.length`
- **Significative instances:** No, it always copies `n` items into the output vector.
- **Critical Instruction:** There are two phases, so we must compute the temporal cost of each phase, then accumulate both. In the case of this algorithm, we can use the same critical instruction for both phases: `k++`

$$T(n) = T_{phase_1}(n) + T_{phase_2}(n) = X + (n - X) = n \in \Theta(n)$$

where X is the number of items copied to the output vector during the first phase, regardless of the input vector which were extracted.

Analysis of sorting algorithms

The Merge-Sort algorithm

- The strategy of this algorithm is known as "divide and conquer".
  - Split the vector into two subvectors of the same size.
  - Sort each subvector separately.
  - Merge the two subvectors preserving the order in the merged vector.

- It is a recursive algorithm whose trivial case is when the size of the subvectors becomes 1. One element always is sorted.

Analysis of sorting algorithms

The Merge-Sort algorithm

```
void mergeSort( int v[] )
{
    mergeSort( v, 0, v.length-1 );
}

void mergeSort( int v[], int start, int end )
{
    if ( start < end ) {
        int half = (start+end)/2;
        mergeSort( v, start, half );
        mergeSort( v, half+1, end );
        naturalMerge( v, start, half, end );
    }
}
```

Analysis of sorting algorithms

The Merge-Sort algorithm

```
void naturalMerge( int v[], int start, int half, int end )
{
    int aux[] = new int [end-start+1];
    int a=start, b=half+1, c=0;
    while( a <= half && b <= end ) {
        if ( V[a] < V[b] )
            aux[c++] = V[a++];
        else
            aux[c++] = V[b++];
    }
    while( a <= half ) aux[c++] = V[a++];
    while( b <= end  ) aux[c++] = V[b++];

    for( int i=0; i < aux.length; i++ )
        V[start+i] = aux[i];
}
```

Analysis of sorting algorithms

Analysis of the Merge-Sort algorithm

- **Input size:** `V.length`, the number of elements to be sorted.
- At each instance of the recursive call the input size is `(end-start+1)`.
- **Critical instruction:** It is difficult to determine a critical instruction for analyzing this algorithm. In this case it is better to use a relation of recurrence.
- We know the temporal cost of Natural-Merge is $\Theta(n)$.

Analysis of sorting algorithms

Analysis of the Merge-Sort algorithm

- **Relation of recurrence:**

$$T(n) = \begin{cases} c_1 & 0 \leq n \leq 1 \\ 2 * T(n/2) + \Theta(n) & n > 1 \end{cases}$$

- Applying **the substitution method**:

$$\begin{aligned} T(8) &= 2 * T(4) + 8 = 2^2 * T(2) + 2 * 4 + 8 \\ &= 2^3 * T(1) + 2^2 * 2 + 8 + 8 = 2^3 * c_1 + 8 + 8 + 8 \end{aligned}$$

- Generally,

$$T(n) = 2^k * T(1) + \sum_{i=1}^{k} n = c_1 * n + n \log n \in \Theta(n \log n)$$

where $k = \log_2 n$

Other algorithms. The Binary Search

- There are many situations where it is needed search for items intensively.
- If data are not sorted it must be used the linear search, whose cost is $O(n)$.
- If data is sorted the binary search can be used, whose cost is $O(\log n)$.

Other algorithms. The Binary Search

- The binary search algorihtm looks for an element inside a slice of the vector defined by two indices, the left end and the right end.

- Initially these indices are 0 and $v.length - 1$ respectively.

- The strategy consists in testing if the central element of the slice, $v[k]$, is equal to the searched element x, then we have three possibilites:
  - $v[k] = x \implies$ The algorihtm finishes. x has been found. The position k is returned.
  - $v[k] < x \implies$ Repeat the search in the half left portion. This updates right end to the value $k - 1$.
  - $v[k] > x \implies$ Repeat the search in the right half portion. This updates the left end to the value $k + 1$.

Other algorithms. The Binary Search

```
int binarySearch( int v[], int x )
{
    int left=0, right=v.lengh-1, k=0;
    boolean found=false;

    while( left <= right && !found ) {
        k=(left+right)/2;
        if ( x == v[k] ) {
            found=true;
        } else if ( v[k] < x ) {
            left=k+1;
        } else {
            right=k-1;
        }
    }
    return ( found ) ? k : -1;
}
```

Other algorithms. The Binary Search

- **Input size:** the size of array: `v.length`
- **Critical instruction:** condition of the while loop:

  `(left <= right && found)`

- **Significant instances:** Yes, `x` can be found in the central position of vector `v`, the first position that is checked, or `x` is not in the vector.

$$\text{Best-case:} \quad T^b(n) = c_1 \in \Theta(1) \implies T(n) \in \Omega(1)$$

$$\text{Worst-case:} \quad T^w(n) = c_2 * \log_2 n \in \Theta(\log n) \implies T(n) \in O(\log n)$$