



Departamento de Informática  
Universidad de Valladolid  
Campus de Segovia

---

# TOPIC 5: ALGORITHM COMPLEXITY

# ALGORITHM COMPLEXITY

- Introduction.
- Asymptotic Behaviour.
- Some practical rules to work out the temporal cost.
- Temporal complexity of searching and sorting algorithms

# DEFINITION OF ALGORITHM

- An algorithm is an accurate description of the steps to follow in order to get a solution of a given problem.
- Each step represents a set of actions or operations performed over a set of objects.



# SOME FEATURES OF AN ALGORITHM

- An algorithm has to satisfy the following features:
  - Accuracy: An algorithm has to be defined without ambiguity.
  - Deterministic: it involves a predictable behaviour. Given a particular input, the algorithm has to produce the same correct output.
  - Finite: The algorithm always reaches a solution

# QUALITIES OF AN ALGORITHM

- Furthermore, an algorithm has to be:
  - As **general** as possible since in this way will be able to work out a wide range of problems.
  - As **Efficient** as possible. The efficiency of an algorithm is related to speed, (the time it takes for an operation to complete), and space, (the memory or non-volatile storage used up by the algorithm).
- In general it is difficult to find an algorithm, given a problem, that satisfies both of them. Therefore, it is necessary to reach a compromise between both features.

# ALGORITHM COMPLEXITY.

- Algorithm complexity is a measure of the temporal and memory space resources spent by an algorithm in order to work out a problem.
- However the temporal aspect (temporal cost) is the most important. For this reason algorithm complexity usually made reference to this last one.
- The temporal cost has to be expressed in terms of the size of the problem, it involves using relative measures instead of an absolute ones.

# THE TEMPORAL COST OF AN ALGORITHM IS MEASURED BY MEANS OF STEPS.

- The measure of temporal cost has to be independent of:
  - the underlying machine
  - the programming language
  - The compiler
  - Any other hardware or software element that may influence over the measurement.
- Since an algorithm is described by means of a set of steps, it is possible to use this concept as an abstract temporal unit.
- Therefore the temporal cost of any algorithm will be given by the number of steps it takes to complete the problem.

# SOME DEPENDENCIES OF THE TEMPORAL COST

- Dependencies of temporal cost:
  - If most significant data is a simple data type then temporal cost depends only on the **data size** ( $T(n)$ ).
  - If most significant data is a composed data type then, besides the data size, it is necessary to assess **three possible situations**:
    - The best case
    - The worst case
    - The average case



# DATA SIZE DEPENDENCE OF TEMPORAL COST

- According to the problem, the data size may affect to temporal cost in different ways
  - Taking into account the magnitude of the number
  - Depending on the number of digits or elements that make up the data.

# DATA SIZE DEPENDENCE OF TEMPORAL COST. EXAMPLES

- Let us suppose the next algorithm that works out the parity of an integer subtracting 2 successively while the result of this operation is greater than one.
  - This algorithm will be achieved after  $(n \text{ DIV } 2)$  subtractions.
- Let us suppose the next slow addition algorithm:

```
while b>0 do begin
  a:=a+1;
  b:=b-1;
end;
```

  - In that case  $T=T(b)$ .

# THE BEST, WORST AND AVERAGE CASES

- In computer science, **best**, **worst** and **average cases** of a given algorithm express what the resource usage is *at least*, *at most* and *on average*, respectively.
- Average performance and worst-case performance are the most used in algorithm analysis.
- In real-time computing, the worst-case execution time is a helpful measure of the temporal cost since it is important to know how much time might be needed *in the worst case* to guarantee that the algorithm would always finish on time.

# THE ORDERED SEQUENTIAL SEARCH ALGORITHM. THE BEST, WORST AND AVERAGE CASES

```
type
tinterval=0..N;
tvector=array[1..N] of integer
FUNCTION Ordered_sequential_search(v:tvector;elem:telem):tinterval;
var
    i:tinterval;
begin
    i:=0;
    repeat
        i:=i+1;
    until (v[i]>=elem) or (i=N);
    if v[i]=elem then
        Ordered_sequential_search:=i
    else
        Ordered_sequential_search:=0
End;
```

**-The best case ( $T_{\min}(n)$ ):** : The element is in the first search position.

**-The worst case ( $T_{\max}(n)$ ):** The element is in the last search position or even is not present.

**-The average case ( $T_{\text{ave}}(n)$ ):** Each position has the same probability of containing the element.

# THE ORDERED SEQUENTIAL SEARCH ALGORITHM. THE BEST, WORST AND AVERAGE CASES

- The next operations are regarded as constant with respect to the temporal cost:
  - addition: 's'
  - comparison: 'c'
  - allocation: 'a'

# THE ORDERED SEQUENTIAL SEARCH ALGORITHM. THE BEST, WORST AND AVERAGE CASES

$T_{\min}$ : when  $v[1] \geq \text{elem}$ .

$$T_{\min} = 3a + 3c + s = \text{constant}$$

$T_{\max}$ : when  $v[n] \leq \text{elem}$

$$T_{\max} = a + n(s + 2c + a) + c + a = n(s + 2c + a) + 2a + c$$

$$T_{\max} = K_1 n + K_2$$

$T_{\text{ave}}$ : Since each position has the same probability of containing the element, then  $T(j) = jK_1 + K_2$

$$T_{\text{ave}}(n) = \sum_{j=1}^n T(j) P \quad \text{donde} \quad P = 1/n$$

$$T_{\text{ave}}(n) = \sum_{j=1}^n (jk_1 + k_2) \frac{1}{n} =$$

$$= \sum_{j=1}^n \frac{k_1}{n} j + \sum_{j=1}^n \frac{k_2}{n} = \frac{k_1}{n} \frac{(n+1)n}{2} + k_2$$

$$T_{\text{ave}}(n) = \frac{k_1(n+1)}{2} + k_2 = \frac{k_1 n}{2} + \frac{k_1}{2} + k_2 = c_1 n + c_2$$

# ASYMPTOTIC BEHAVIOUR OF TEMPORAL COST

- If data size is very large then the asymptotic behaviour of temporal cost has to be considered.

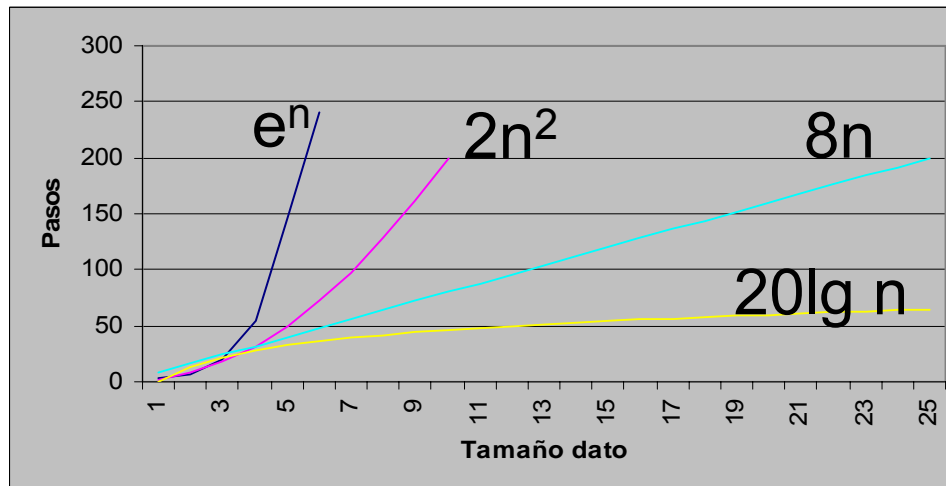
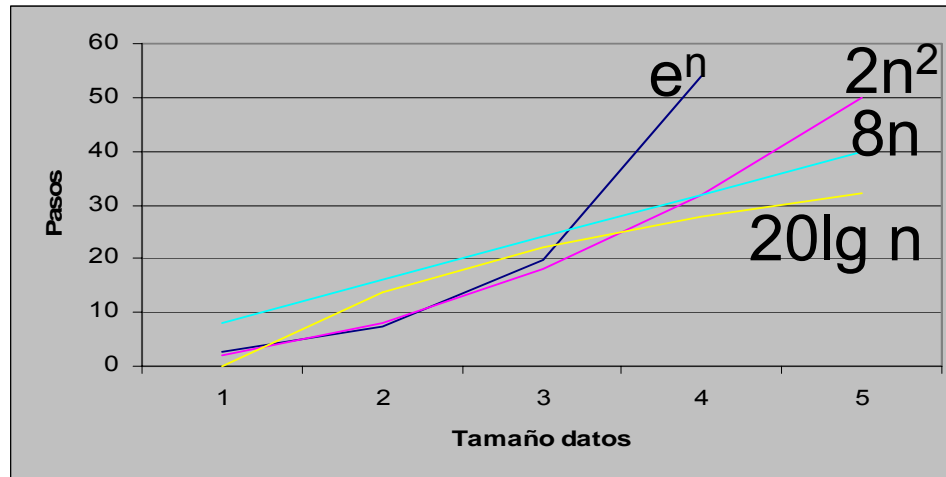
# ASYMPTOTIC BEHAVIOUR OF THE TEMPORAL COST

- In the picture below the time spent in working out a problem using algorithms with different temporal cost is shown .
- These temporal costs are expressed by means of a functional representation of their asymptotic behaviours.
- In this calculation a computer executing 1 million of operations per second is regarded.

n \ T(n)	log n	n	n log n	n <sup>2</sup>	n <sup>3</sup>	2 <sup>n</sup>	n!
10	3.3 10 <sup>-6</sup>	10 <sup>-5</sup>	3.3 10 <sup>-5</sup>	10 <sup>-4</sup>	0.001	0.001	3.63
50	5.6 10 <sup>-6</sup>	5 10 <sup>-5</sup>	2.8 10 <sup>-4</sup>	0.0025	0.125	unapproach	unapproach
100	6.6 10 <sup>-6</sup>	10 <sup>-4</sup>	6.6 10 <sup>-4</sup>	0.01	1	unapproach	unapproach
10 <sup>3</sup>	10 <sup>-5</sup>	0.001	0.01	1	1000	unapproach	unapproach
10 <sup>4</sup>	1.3 10 <sup>-5</sup>	0.01	0.13	100	10 <sup>6</sup>	unapproach	unapproach
10 <sup>5</sup>	1.6 10 <sup>-5</sup>	0.1	1.6	10 <sup>4</sup>	unapproach	unapproach	unapproach
10 <sup>6</sup>	2 10 <sup>-5</sup>	1	19.9	10 <sup>6</sup>	unapproach	unapproach	unapproach



# ASYMPTOTIC BEHAVIOUR OF SOME FUNCTIONS



# NOTATION TO DESCRIBE THE ASYMPTOTIC BEHAVIOUR OF FUNCTIONS

- In order to describe the asymptotic behavior of  $T(n)$  for very large inputs is necessarily to dispose of a suitable notation.
- Its purpose is to characterize a function's in a simple but rigorous way that enables comparison to other functions.
- These notations are:
  - Big- O
  - Big Omega
  - Big Theta

# BIG-O NOTATION

- **Big O notation** or **Big Oh notation**, and also **Landau notation** or also called **asymptotic notation**, is a mathematical notation used to describe the asymptotic behavior of functions.
- The symbol  $O$  is used to describe an asymptotic upper bound for the magnitude of a function in terms of another, usually simpler, function.
- Informally, the  $O$  notation is commonly employed to describe an asymptotic tight bound, but tight bounds are more formally and precisely denoted by the  $\Theta$  (capital theta).

# BIG-O NOTATION. DEFINITION



- Definition: if  $f, g: \mathbb{Z}^+ \rightarrow \mathbb{R}^+$ , then  $f \in O(g)$  or  $g$  is a tight bound of  $f$  if there are constants such as  $n_0 \in \mathbb{Z}^+$  and  $\lambda \in \mathbb{R}^+$  such that:

$$f(n) \leq \lambda g(n) \text{ for } n \geq n_0$$

- It means that  $f$  does not grow as quick as  $g$ . In that way the asymptotic behaviour of the function is upperly bounded.
- For Ordered sequential search algorithm:

$$T_{\max}(n) = k_1 n + k_2 \in O(n) \quad \text{since} \\ k_1 n + k_2 \leq \lambda n \text{ for } n \geq k_2 / (\lambda - k_1)$$

- The Big O notation for any constant function with respect to the time is  $O(1)$ .

# SOME PROPERTIES OF BIG-O NOTATION. SCALABILITY.

- $O(\log_a n) = O(\log_b n)$
- For this reason it is not necessary to specify the base of the logarithm  $O(\log n)$ .

# SOME PROPERTIES OF BIG-O NOTATION. THE ADDITION RULE.

- **The addition rule:** if  $f_1 \in O(g_1)$  and  $f_2 \in O(g_2)$  then  $f_1 + f_2 \in O(\max(g_1, g_2))$ .
- A generalization of this rule that involves the scalability property may be expressed as follows:
  - if  $f_i \in O(f)$  for  $i=1 \dots k$  then:  
$$c_1 f_1 + \dots + c_k f_k \in O(f).$$
  - So, any polynomial may be expressed as  $p_k(n) \in O(n^k)$

# SOME PROPERTIES OF BIG-O NOTATION. THE SUMMATORY RULE.

- **The summatory rule:** if  $f \in O(g)$  and  $g$  is a growing function then:

$$\sum_{i=1}^n f(i) \in O\left(\int_1^{n+1} g(x) dx\right)$$

- if  $f(i)=i$ .

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

$$\int_1^{n+1} x dx = \frac{x^2}{2} \Big|_1^{n+1} = \frac{(n+1)^2}{2} - \frac{1}{2} = \frac{n^2}{2} + n$$

- Since any polynomial may be expressed as  $p_k(n) \in O(n^k)$
- Then this last expression is an upper bound of the previous function

# A USEFULL CONSEQUENCE OF SUMMATORY RULE

$$\sum_{i=1}^n i^k \in O(n^{k+1})$$

$$\int_1^{n+1} x^k dx = \frac{x^{k+1}}{k+1} \Big|_1^{n+1} = \frac{(n+1)^{k+1}}{k+1} - \frac{1}{k+1} \approx$$

$$k_1 (n+1)^{k+1} + k_2 \in O(n^{k+1})$$



# HIERARCHY OF MOST FREQUENT FUNCTIONS

- The most frequent asymptotic behaviours may be sorted in growing order as follows:

$$1 \ll \log n \ll n \ll n \log n \ll n^2 \ll n^3 \ll \dots \ll 2^n \ll n!$$

# SOME PRACTICAL RULES IN ORDER TO WORK OUT THE TEMPORAL COST OF AN ALGORITHM

- Next, a general rules about how to work out the temporal complexity in the worst case are shown.
- These rules have to take into account the temporal cost of:
  - Simple instruction
  - Composition of instructions
  - Selective instructions
  - Loops
  - Subprograms

# SIMPLE INSTRUCTION

- The following simple instructions are supposed to be constant:
  - The arithmetics operations and comparison between simple data, as long as this last one has a constant size during the calculations.
  - The allocation, writing, and reading of simple data operations
  - Any operation that involves access to an array component, a record field or the next position of a file.
- All these operation are regarded as  $\Theta(1)$ .

# COMPOSITION OF INSTRUCTIONS

- Let us suppose that the instructions  $I_1$  and  $I_2$  have, in the worst case, the temporal complexities  $T_1(n)$  and  $T_2(n)$  respectively. Then, the temporal cost of the composition that involves both of them is:

$$T_{I_1, I_2}(n) = T_1(n) + T_2(n)$$

- Then, taking into account the addition rule, the temporal cost of this composition will be:

$$T_{I_1, I_2}(n) = \max(T_1(n), T_2(n))$$

# SELECTIVE INSTRUCTIONS

- Let us consider the next selective instructions:
  - if <condition> then  $I_1$  else  $I_2$

$$T_{\text{selection}}(n) = T_{\text{condition}}(n) + \max(T_1(n), T_2(n))$$

- Case <expression> of
  - case1:  $I_1$ ;
  - case2:  $I_2$ ;
  - .....
  - .....
  - casen:  $I_n$ ;
- end; {case}

$$T_{\text{selection}}(n) = T_{\text{expression}}(n) + \max(T_1(n), \dots, T_n(n))$$

# ITERATIVE INSTRUCTIONS: LOOPS

- The most simple case is the FOR-loop type:

- for  $i:=1$  to  $m$  do  $I$

$$T_{bucle}(n) = m + \sum_{j=1}^m T_{I_j}(n)$$

- Where “ $m$ ” represents how many times “ $i$ ” is increased and the test operation to check if “ $i$ ” is inside of the loop interval.
- If the body of the loop is made up of instructions with a constant temporal cost then the previous equation may be written as follows:

$$T_{bucle}(n) = m (1 + T_I(n))$$

- The “while” and “repeat” loops have not got a general rule. In that case is necessary to consider each situation individually.

# RECURSIVE ALGORITHMS

- For working out the temporal cost of a recursive algorithm it is necessary to distinguish between the case base and the recurrent one.

```
FUNCTION Fac(n:integer):integer;
```

```
Begin
```

```
  if n=0 then
```

```
    Fac:=1
```

```
  else
```

```
    Fac:=n*Fac(n-1)
```

```
End;{Fac}
```

- The temporal cost of the case base is:  $T_{\text{fac}}(0)=1$
- The temporal cost of the recurrent case is:

$$T_{\text{fac}}(n)=1+T_{\text{fac}}(n-1)=$$

$$T_{\text{fac}}(n)=1+1+T_{\text{fac}}(n-2)=$$

.....

$$T_{\text{fac}}(n)=1+1+\dots+1+T_{\text{fac}}(0) = n+1 \in O(n)$$

# SEARCHING AND SORTING ALGORITHMS

- Array-Searching algorithms.
  - Sequential search
  - Ordered Sequential search
  - Binary search
- Array-sorting algorithms.
  - Selection sort
  - Insertion sort
  - Bubble sort
  - Quick Sort
  - Merge Sort



# ARRAY SEARCHING ALGORITHMS

- These algorithms represent techniques of finding a particular value in a linear array.
- Searching(vector,element):
  - $i \in \{1, \dots, n\}$  if the element exists
  - 0 otherwise
- Data Structure in Pascal:

```
const
    N=100;
type
    tInterval=0..N;
    tvector=array[1..N] of tElem {ordinal type}
```

# ARRAY SEARCHING ALGORITHMS

- Sequential search
- Ordered sequential search
- Binary search

# SEQUENTIAL SEARCH

- The sequential search consists in looking for sequentially an specific element going through every component of the linear array.
- This process is over when this element is localized or if the end of the linear array is reached
- First design step:
  - ind:=0
  - Searching inside the array
  - if vector[ind]=element then
    - sequentialsearch:=ind
  - else
    - sequentialsearch:=0

# SEQUENTIAL SEARCH IMPLEMENTATION IN PASCAL

FUNCTION Sequentialsearch(v:tvector ; elem:telem):tInterval;  
{Returns "i" if v[i]=elem otherwise "0"}

VAR

    i:tInterval;

BEGIN

    i:=0;

    repeat

        i:=i+1;

    until (v[i]=elem) or (i=N);

    if v[i]=elem then

        Sequentialsearch:=i

    else

        Sequentialsearch:=0

END; {Sequentialsearch}

The number of times this algorithm is executed depends on the loop. The worst case is  $O(n)$ .

# ORDERED SEQUENTIAL SEARCH

- The previous search algorithm may be improved if the array is ordered. (i.e. a growing array).
- In this way, if an array component greater than the searched element is found, it means that this one does not exist in this collection.

# ORDERED SEQUENTIAL SEARCH, IMPLEMENTATION IN PASCAL

```
FUNCTION Orderedsequentialsearch(v:tvector ; elem:telem):tInterval;  
{Returns "i" if v[i]=elem otherwise "0"}
```

```
VAR
```

```
    i:tInterval;
```

```
BEGIN
```

```
    i:=0;
```

```
    repeat
```

```
        i:=i+1;
```

```
    until (v[i]≥elem) or (i=N);
```

```
    if v[i]=elem then
```

```
        Orderedsequentialsearch:=i
```

```
    else
```

```
        Orderedsequentialsearch:=0
```

```
END; {Orderedsequentialsearch}
```

\*This algorithm, on the worst case, is  $O(n)$ .

# BINARY SEARCH

- This algorithm is called binary since during the search process the array is divided successively into two parts.
- Regarding an ordered array, the binary search consist in:
  - Checking if the middle position array component is the searched element.
  - If it is so, then the process is over. Otherwise the process is repeated up to find the element, considering only the half part of the array where this element may be localized, or up to reach the end of the array.
- A binary search is an example of a divide and conquer algorithm

# BINARY SEARCH, IMPLEMENTATION IN PASCAL

```
FUNCTION Binarysearch(v:tvector ; elem:telem):tInterval;
```

```
{Prec. "v" is growing ordered}
```

```
{Returns "i" if v[i]=elem otherwise "0"}
```

```
VAR
```

```
    lowered,upperend,midpos:tInterval;
```

```
    found:boolean;
```

```
BEGIN
```

```
    lowerend:=1; upperend:=N; found:=false;
```

```
    while not found and (upperend $\geq$ lowerend) do begin
```

```
        midpos:=(upperend+lowerend) DIV 2;
```

```
        if elem=v[midpos] then
```

```
            found:=true
```

```
        else if elem>v[midpos] then
```

```
            lowerend:=midpos+1
```

```
        else
```

```
            upperend:=midpos-1
```

```
    end {while}
```



# BINARY SEARCH, IMPLEMENTATION IN PASCAL

```
if found then
    binarysearch:=midpos
else
    binarysearch:=0
END; {Binarysearch}
```

# BINARY SEARCH

- The algorithm complexity of Binary search may be work out as follows:

Taking into account that:

$$2^k \leq N \leq 2^{k+1}$$

and

$$2^k \leq N$$

$$\log 2^k \leq \log N$$

$$k \leq \log N$$

Then, on the worst case the algorithm complexity is :

$$T(n) \approx O[\log N]$$

# ARRAY SORTING ALGORITHM

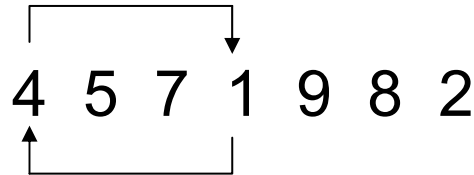
- Next, the most frequent array sorting algorithm are presented:
  - Quadratic algorithms:
    - Selection sort
    - Insertion sort
    - Bubble sort
  - Advanced sorting algorithm:
    - Quick Sort
    - Merge Sort

# SELECTION SORT

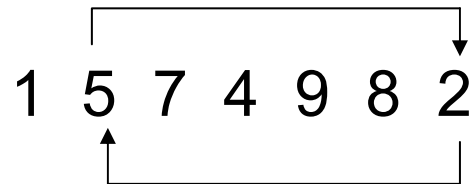
- First design step: Selection sort goes through the array searching the lesser element in order to put it in the correct position. This operation is repeated until the array is ordered.

# SELECTION SORT

- 1. The lesser element among  $v[1], \dots, v[n]$  is put in  $v[1]$ . In order to perform this operation,  $v[1]$  and  $v[i]$  {where  $v[i]=\min(v)$ } are interchanged.



- 2. The lesser element among  $v[2], \dots, v[n]$  is put in  $v[2]$ . In order to perform this operation,  $v[2]$  and  $v[i]$  {where  $v[i]=\min(v)$ } are interchanged.



- (n-1). The lesser element among  $v[n-1]$  and  $v[n]$  is put in  $v[n-1]$ . In order to perform this operation,  $v[n-1]$  and  $v[i]$  {where  $v[i]=\min(v)$ } are interchanged.

1 2 4 5 7 8 9

# SELECTION SORT, IMPLEMENTED IN PASCAL

```
PROCEDURE Selectionsort( var v:tvector);
```

```
{Returns an growing ordered array}
```

```
VAR
```

```
    i, j, lessposition:tInterval;
```

```
    lessvalue, aux:telem;
```

```
BEGIN
```

```
    for i:=1 to N-1 do begin
```

```
        lessvalue:=v[i]
```

```
        lessposition:=i
```

```
        for j:=i+1 to N do
```

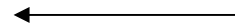
```
            if v[j]<lessvalue then begin
```

```
                lessvalue:=v[j];
```

```
                lessposition:=j
```

```
            end; {if }
```

Searchs the lesser element  
among  
i+1,.....,N



# SELECTION SORT, IMPLEMENTED IN PASCAL

```
if lessposition<>i then begin
    aux:=v[i];
    v[i]:=v[lessposition]; ← Performs an interchange if
    v[lessposition]:=aux    lessposition is different to "i"
end {if}
end{for i}
END; {Selectionsort}
```

# INSERTION SORT

- First design step: Insertion goes through the array inserting  $v[i]$  in its correct position among the already ordered elements.



# INSERTION SORT

- 1.  $v[1]$  is regarded as the first element.
- 2.  $v[2]$  is inserted in its correct position with respect to  $v[1]$ .
- 3.  $v[3]$  is inserted in its correct position with respect to  $v[1]$  and  $v[2]$ .
- i.  $v[i]$  is inserted in its correct position with respect to  $v[1], \dots, v[i-1]$ .
- n. This operation is repeated with the last element of the array.

# INSERTION SORT, IMPLEMENTED IN PASCAL

```
PROCEDURE Insertionsort( var v:tvector);  
{Returns an growing ordered array}
```

```
VAR
```

```
  i, j:tInterval;
```

```
  aux:telem;
```

```
BEGIN
```

```
  for i:=2 to N do begin
```

```
    aux:=v[i]
```

```
    j:=i-1
```

```
    while (j≥1) and (v[j]>aux) do begin
```

```
      v[j+1]:=v[j];
```

```
      j:=j-1
```

```
    end; {while}
```

```
    v[j+1]:=aux
```


```
  end {for}
```

```
END; {insertionsort}
```

Shifting the lesser value



Inserting the value in its  
correct position



# BUBBLE SORT

- First design step: Bubble sort goes through the array searching the lesser element from the last position up to the current one where it is inserted.

# BUBBLE SORT

- 1. Inserts the lesser element in the first position.
  - The last element is compared with respect to the previous last one interchanging their values if they are in decreasing ordered. This operation is repeated until the first one is reached. (then the lesser element is located at the first position).
  
- 2. Inserts the second lesser element in the second array position.
  - It is performed as we mentioned before but finishing when the second position is reached.
  
- 3. repeats until the array is ordered interchanging in each case if it is necessary.

# BUBBLE SORT, IMPLEMENTED IN PASCAL

```
PROCEDURE bubblesort( var v:tvector);  
{Returns a growing ordered array}  
VAR  
    i, j:tInterval;  
    aux:telem;  
BEGIN  
    for i:=1 to N-1 do  
        for j:=N downto i+1  
            {The lesser element is searched from the last element going  
            backward and inserting in its corresponding place  $v_i$ }  
            if  $v[j-1]>v[j]$  then begin  
                aux:=v[j];  
                 $v[j]:=v[j-1]$ ; ← Interchange  
                 $v[j-1]:=aux$ ;  
            end; {if}  
        end;  
    end;  
END; {Bubblesort}
```

# ADVANCED SORTING ALGORITHM

- Quick sort
- Merge sort

# QUICK SORT

- This sorting algorithm consist in dividing a vector in two blocks. On the first one are located those elements less than a certain value (reference), while the remaining ones are placed on the second block.
- This procedure is repeated dividing successively each block into two new ones and locating the elements as above.
- The condition for stopping is satisfied when any block of one element (ordered block) is reached.
- This algorithm follows the “Divide and Win” scheme.

# QUICK SORT, PSEUDOCODE

If “v” is a one element block then

    The array is already ordered

else

    divides “v” into two blocks A and B satisfying that any element from A is less than any element from B.

Endif

Sorts A y B using Quick Sort

Returns “v” already ordered.

- Where “Divides “v” into two blocks A and B” may be described as follows:

    Chose an element as reference of “v”

    For each element of “v” do:

        if element is less than reference then it has to be placed in A  
        otherwise in B.



# QUICK SORT, IMPLEMENTED IN PASCAL

```
PROCEDURE Quicksort( var v:tvector);  
{Returns a growing ordered array}  
  PROCEDURE Sort_from_upto(var v:tvector;left,right:tinterval);  
  {returns 'v[left..right]' as a growing ordered array}  
  {next page}
```

```
BEGIN {Quicksort}  
  Sort_from_upto(v,1,n);  
END; {Quicksort}
```

```
PROCEDURE Sort_fro_upto(var v:tvector;left,right:tinterval);  
  {returns 'v[left..right]' as a growing ordered array}  
  VAR  
    i, j:tInterval;  
    p,aux:telem;  
  BEGIN
```

.....

# QUICK SORT, IMPLEMENTED IN PASCAL

BEGIN

i:=left; j:=right; p:=v[(left+right) DIV 2];

while i<j do begin {both blocks are reorganized}

while v[i]<p do

i:=i+1;

while p<v[j] do

j:=j-1;

if i ≤ j then begin {interchanging elements}

aux:=v[i];

v[i]:=v[j]; ← Interchange

v[j]:=aux;

i:=i+1; j:=j-1; {updating positions}

end; {if}

end; {while}

if izq<j then sort\_from\_upto(v, left, j);

if i<der then sort\_from\_upto(v, i, right);

END; {Sort\_from\_upto}

# QUICK SORT, TRACE OF AN EXAMPLE

V=[0,5,15,9,11]

1. Sort\_from\_upto(v,1,5) p=v[3]=15

i=1 [0,5,15,9,11] 0<15

i=2 [0,5,15,9,11] 5<15

i=3 [0,5,15,9,11] 15 not <15

i=3 j=5 [0,5,15,9,11] 11 not >15

i=4 j=4 [0,5,11,9,15] interchange. Loop exit condition

1.1.sort\_from\_upto(v,1,4) p=v[2]=5

i=1 [0,5,11,9] 0<5

i=2 [0,5,11,9] 5 not < 5

i=2 j=4 [0,5,11,9] 9>5

i=2 j=3 [0,5,11,9] 11>5

i=2 j=3 [0,5,11,9] 5 not>5

i=3 j=1 [0,5,11,9] interchange. Loop exit condition

# QUICK SORT, TRACE OF AN EXAMPLE

1.1.1.sort_from_upto(v,3,4)		p=v[3]=11
	i=3 [11,9]	11 not <11
	i=4 [11,9]	9 not > 11
i=4	j=3 [9,11]	interchange. Loop exit condition

V=[0,5,15,9,11]

1.2 sort_from_upto(v,4,5)		p=v[4]=11
	i=4 [11,15]	11 not <11
i=4	j=5 [11,15]	15>11
i=4	j=4 [11,15]	11 not >11
i=5	j=3 [11,15]	interchange. Loop exit condition

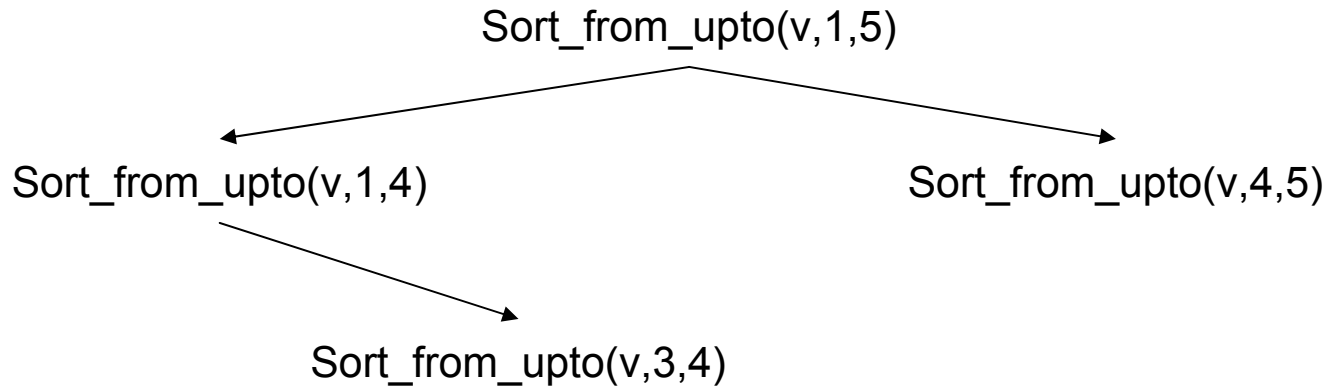
izq=4 not <j=3

i=5 not <der=5

recursion is over.

growing ordered array: [0,5,9,11,15]

# QUICK SORT, TRACE OF AN EXAMPLE



# QUICK SORT, WORKING OUT ITS TEMPORAL COMPLEXITY

- On the worst case: if the first element is chosen as the reference and the array is decreasing ordered then the loop for each element is executed in:

$$(n-1)+(n-2)+(n-3)+\dots+1 \text{ steps}$$

where each member of this summatory comes from each recursive invocation. The following expression may be obtained from this summatory :

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

that it can be represented by means of a quadratic order  $T(n) \in O(n^2)$

- However if the chosen reference value is the array middle point and this array is not ordered then this algorithm has a temporal complexity of  **$O(n \log n)$** .

Size of the array

Number of divisions

# MERGE SORT

- In that case the scheme is similar to Quick sort, since the underlying idea is “divide and win”. However the main effort is not devoted to divide the array and reorganize each block after division but building the ordered array by means of merging the different generated blocks.
- The main idea consist in dividing “v” into two blocks, A and B, in order to merge both of them, mantaning the growing order, once these ones have been merged in this way (it means, using the same merge sort algorithm).

# MERGE SORT, PSEUDOCODE

```
If v is a one element block then
    the array v is already ordered
else
    divide v into two blocks "A" and "B"
endif
```

Sorts "A" and "B" using Mergesort

Merges the ordered "A" and "B" blocks for building the whole ordered array.

- where: "divide v into two blocks "A" and "B"" may be described in more detail as follows:

Allocate to "A" the block  $[v_1, \dots, v_{n \text{DIV} 2}]$

Allocate to "B" the block  $[v_{n \text{DIV} 2 + 1}, \dots, v_n]$

- and "...Merges the ordered "A" and "B" blocks.." consist in merging the already ordered components of "A" and "B".



# MERGE SORT, IMPLEMENTED IN PASCAL

```
PROCEDURE Mergesort( var v:tvector);
{Returns a growing ordered array}
  PROCEDURE Merge_from_upto(var v:tvector; izq, der: tinterval);
  {returns 'v[left..right]' as a growing ordered array}
  VAR
  centro :tInterval;
      PROCEDURE Merge(var v:tvector; left,mid,right:tinterval; var
        w:tvector);
  BEGIN {Merge_from_upto}
      mid:=(left+right)DIV2;
      if left<mid then
          Merge_from_upto(v,left,mid);
      if mid<right then
          Merge_from_upto(v,mid+1,right);
      Merge(v,left,mid,right,v)
  END; {Merge_from_upto}
```

# MERGE SORT, IMPLEMENTED IN PASCAL

```
BEGIN {Mergesort}  
    Merge_from_upto(v,1,n)  
END; {Mergesort}
```

# MERGE SORT, IMPLEMENTED IN PASCAL

PROCEDURE Merge(var v:tvector; left,mid,right:tinterval; var w:tvector);  
{Returns an ordered array as a result of merging orderly the blocks v[left..ce] and  
v[mid+1..right] }

VAR

i, j,k:tInterval;

BEGIN

i:=left; j:=mid+1; k:=left; {k goes through w}

while( i ≤ mid) and (j ≤right) do begin

if v[i]<v[j] then begin

w[k]:=v[i];

i:=i+1

end; {if}

else begin

w[k]:=v[j];

j:=j+1;

end; {else}

k:=k+1

end; {while}

# MERGE SORT, IMPLEMENTED IN PASCAL

```
for k:=j to de do
```

```
    w[k]:=v[k]
```

```
    for k:=i to mid do
```

```
        w[k+right-mid]:=v[k]
```

```
END; {Merge}
```

# MERGE SORT, TRACE OF AN EXAMPLE

V=[8,5,7,3]

1. Merge\_from\_upto(v,1,4), middle=2

1.1. left(1) < middle(2) Merge\_from\_upto(v,1,2), middle=1

1.1.1. left(1) not < middle(1)

1.1.2. middle(1) < right(2)

1.1.2.1 left(2) not < middle(2)

1.1.2.2. middle(2) not < right(2)

1.1.2.3. Merge(v,2,2,2,v) trivial

Merge(v1,1,2,v)----- w1[5,8]

# MERGE SORT, TRACE OF AN EXAMPLE

1.2.  $\text{middle}(2) < \text{right}(4)$  Merge\_from\_upto( $v,3,4$ ),  $\text{middle}=3$

1.2.1.  $\text{left}(3) \text{ not} < \text{middle}(3)$

1.2.2.  $\text{middle}(3) < \text{right}(4)$

1.2.2.1.  $\text{left}(4) \text{ not} < \text{middle}(4)$

1.2.2.2.  $\text{middle}(4) \text{ not} < \text{right}(4)$

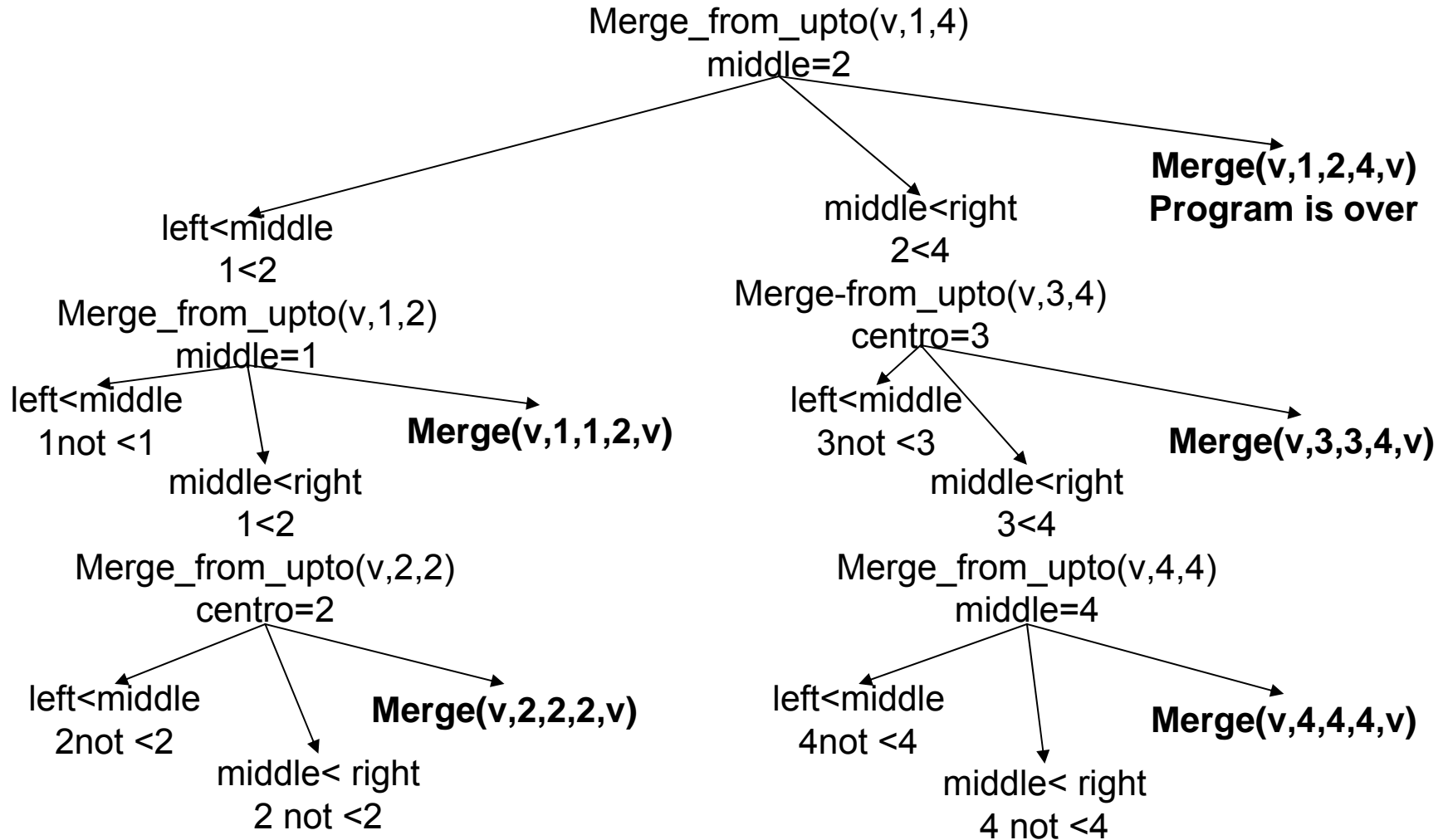
1.1.2.3. Merge( $v,4,4,4,v$ ) trivial

Merge( $v3,3,4,v$ )-----  $w2[3,7]$

1.3. Merge( $v,1,2,4,v$ )----- $w[3,5,7,8]$

That represents the Ordered merge of  $w1$  and  $w2$

# MERGE SORT, TRACE OF AN EXAMPLE



# MERGE SORT, DETERMINING THE ALGORITHM COMPLEXITY

- On the worst case: Since the time to perform the merge is proportional (the length of the array) then the temporal cost might be expressed in terms of the following recurrence:

$$T(n) = \begin{cases} k_1 & \text{si } n = 1 \\ 2T\left(\frac{n}{2}\right) + k_2n + k_3 & \text{si } n > 1 \end{cases}$$

- The general term of this recurrence may be work out as follows (regarding n as a power of 2 (such as n=2<sup>j</sup>)).

$$\begin{aligned} T(n) &= 2(2T(n/4) + k_2(n/2) + k_3) + k_2n + k_3 \\ &= 4T(n/4) + 2k_2n + k_4 = \\ &= 4(2T(n/8) + k_2(n/4) + k_3) + 2k_2n + k_4 = \\ &= 8T(n/8) + 3k_2n + k_5 = \\ &= \dots\dots\dots \dots\dots\dots \dots\dots\dots \dots\dots\dots \dots\dots\dots \dots\dots\dots = \\ &= 2^j T(1) + jk_2n + k_j \end{aligned}$$

- Where if j ≤ log n then the temporal cost order is O(n log n).