PEOPLE'S DEMOCRATIC REPUBLIC OF ALGERIA
MINISTRY OF HIGHER EDUCATION AND SCIENTIFIC RESEARCH
UNIVERSITY OF RELIZANE
Faculty of Science and Technology
Computer Science department

Course materials

Algorithmics and Distributed Systems

1st year Master Networks and Distributed Systems students

Established by :

Dr. MEDJAHED Seyyid Ahmed

Academic Year 2023-2024

Table of contents

Contents

Table of contents

List of figures

List of Figures

List of tables

List of paintings

Foreword

Foreword

This course materials is intended for 1st year Master Networks and Distributed Systems students . Its objective is to introduce students to advanced algorithmic concepts. This course is organized into five parts:

The first part is a reminder of algorithms. The second part is a state of the art on algorithmic complexity. The third part is devoted to search and sorting algorithms. The fourth part presents an introduction to distributed systems. The last part is dedicated to the exercises.

ASD

1. Reminder on algorithms

2. Definition

An algorithm is a method of solving a problem of size **n**. An algorithm is a finite number of elementary operations constituting a calculation or problem-solving scheme.

A program is the creation (implementation) of an algorithm using a given language (on a given architecture). It is about the implementation of the principle. For example, when programming we will sometimes explicitly deal with memory management (dynamic allocation in C) which is an implementation problem ignored at the algorithmic level.

3. Structure of an algorithm

```
algorithm name of algorithm     // header
const                           // declaration
      constantes
var
      variables
func                            // function
       functions
begin                           // body
      instruction 1
      ……..
      instruction n
end
```

3.1. Read/write operation

```
Var
      x : integer
Begin
      Write(”Write the number x :”)
      Read(x)
end
```

3.2. Conditional structure

```
Var
    x,y : integer
Begin
      If (x <= y) Then
      Begin
            Write(x)
```

3.3. Loops – FOR

```
Var
    x, i, n, s : integer
Begin
      For i From x to n Step s Do
            Write(i)
      EndFor
end
```

3.4. Loops – While

```
Var
    i : integer
Begin
      While condition Do
            Write(i)
      EndWhile
```

3.5. Functions

```
Var
    i : integer
Function add(x, y)
Begin
      return x + y
End
Begin
      add(2, 3)
```

4. Algorithmic Complexity

4.1. Definition

Complexity is a criterion for measuring the quality of an algorithm which is the quantity of resources in terms of space and time consumed by the implemented program.

The complexity of an algorithm quantifies the time required for an algorithm to execute based on the size of the input.

For a given problem, there are several algorithms that solve that problem. To choose the best one, we need to compare and analyze the performance of each algorithm.

When analyzing an algorithm, we mainly consider the study of complexity.

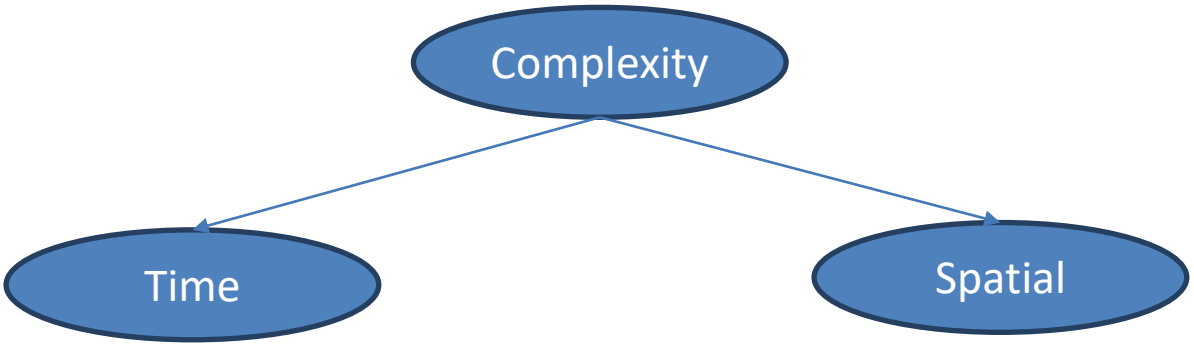

Figure 1. Complexity

4.2. Spatial complexity

Space complexity is related to the size of the memory space occupied by the program during its execution. Today the capacity of MC is very large and therefore we do not need to study spatial complexity, we are only interested in temporal complexity.

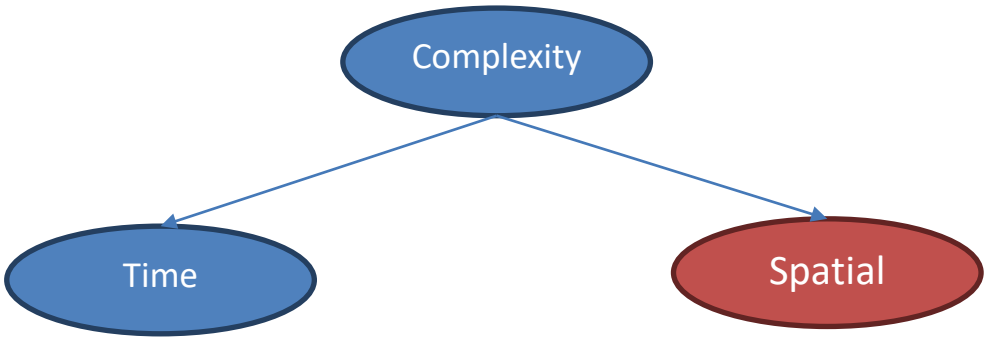

Figure 2. Spatial complexity

4.3. Time complexity

Time complexity describes the time required to execute a program.

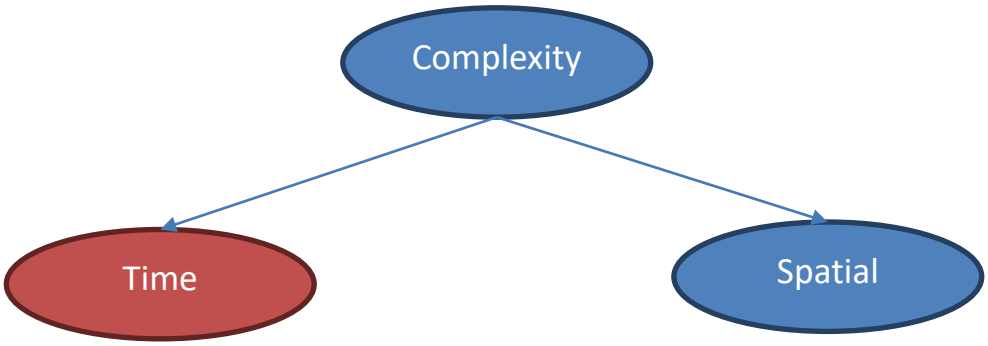

Figure 3. Time complexity

4.4. Practical time

The practical time depends on several factors such as:

- The machine
- The operating system
- The programming language
- The algorithm used

Given these factors, the study of complexity was limited to the theoretical execution time

4.5. Theoretical execution time

To solve a problem, the algorithm must execute a number of instructions (commands). Each instruction is composed of one or more elementary operations; an elementary operation is a fundamental operation with zero, one or two operands (generally) such as assignment, arithmetic, logic, input, output operations.

Table 1. Execution time

Instruction	**Number of operations**	**Explanation**
x ← y	1 elementary operation	Assignment operation
x ← y + z	2 elementary operation	Addition and assignment
x ← y +z +w	3 elementary operation	2 additions and 1 assignment

If the average execution time of an elementary operation is **t** and the number of elementary operations of an algorithm is **n** then:

Theoretical execution time = n*t

Knowing that the average execution time of an elementary operation is a constant linked to several factors (machine), the study of the theoretical execution time was limited to the number of elementary operations carried out.

4.6. Case of complexity

In complexity, when solving a problem, we can find three cases:

- The best-case scenario
- The average case
- The worst of cases

The study of complexity is limited to the worst case.

Studying complexity means calculating the number of elementary Operations in the worst case necessary to solve a problem according to its size.

Let **n be** the number of elementary operations of instruction **i** :

$$C = \sum_{i=1}^{n} n_i$$

4.6.1. Conditional

$$C = n_{expression} + max(n_{Bloc\,1}, n_{Bloc\,2})$$

Example

```
x ← a
y ← b + c
if (x < y) Then
      y ← y + 1
      x ← y + x +1
else
      y ← x
      x ← b
Endif
```

C = (1+2) + (1 + max (5, 2)) = 3 + (1 + 5) = 9

4.6.2. FOR loop

$$\boldsymbol{C = (k - j + 1) * (n_{Entete} + n_{Bloc}) + n_{Entete}}$$

Example

```
FOR i FROM 1 to n Do
      a ← a + b
EndFor
```

C = (n-1+ 1)* (1 + 2) + 1 = n*(1 + 2) + 1 = 3n+1

4.6.3. WHILE loop

$$\boldsymbol{C = (n_{Iterations} * (n_{Condition} + n_{Bloc})) + n_{Condition}}$$

- **The number of iterations = n/step (If i tends towards n by addition or subtraction)**

- **The number of iterations = Ln(n)/Ln(steps) (If i tends towards n by multiplication or by division)**

Example

```
i ← 0
WHILE (i < N) Do
      x ← x + y
      i ← i + 3
EndWhile
```

The number of iterations = n/3

C = 1 + [(n/3 * (1 + 4)) + 1] = (5n/3) + 2

4.6.4. Call Function

The cost of calling a function is equal to:

- **The cost of the function body for its parameters**
- **The cost of evaluating its parameters**

Example

```
FUNCTION checkEven(a:Int):bool
Begin
If (a mod 2 = 0) Then       (2)
      return true           (1)
Else
      return false          (1)
End
Begin
If (checkEven(n)==true) Then  1+1+3
      n ← n + 1             (2)
Else
      n ← n - 1             (2)
End
End
```

The cost of the function = 3

The cost of the algorithm = 1 + 1 + 3 +2 = 7

The cost of IF = 1

The cost of loading n = 1

The cost of the function = 3

The cost of the IF test = 2

C = 7

4.6.5. Recursive Function

Mathematical reminder - Arithmetic sequence

$$u_{n+1} = u_n + r$$

$$u_n = u_0 + n * r$$

Generally

$$u_n = u_p + (n - p) * r$$

Geometrical sequence

$$u_{n+1} = q \times u_n$$

$$u_n = u_0 \times q^n$$

Generally _

$$u_n = u_p \times q^{n-p}$$

Example

```
FUNCTION power(a, n:Int):bool
Begin
If (n = 0) Then        (1)
      return 1          (1)
Else
      return (a * power(a, n-1))   (1+1+1+2+ Cn-1)
End
Begin
Read(a , n) (2)
Write(power(a,n)) (1) + 6n + 2 + 2
End
```

1 = return

1 = multiplication

1 = subtraction

2 = loading parameters

Cn-1 = power function call

Cn = 6 + Cn-1

Cn = 6n + C0

We calculate C0 = 2

Cost of function = Cn = 6n + 2

C = 6n + 7

4.7. Asymptotic complexity

Asymptotic complexity consists of studying the approximate behavior of the number of elementary operations of the algorithm when the size of the problem treated is sufficiently large (tends towards infinity).

4.7.1. Concept of the Big O

In algorithmics, the notion of the big O is a metric used to describe the execution time of an algorithm.

4.7.2. Simplifying complexity

- **We only keep the dominant term**
- **We remove the multiplicative constants**

Example

$$\mathbf{VS(n)} = \mathbf{3}\boldsymbol{n}^{\mathbf{5}} - \mathbf{2}\boldsymbol{n}^{\mathbf{3}} + \mathbf{5}\boldsymbol{n}^{\mathbf{2}} + \mathbf{7}$$

We keep the dominant term:$\mathbf{3}\boldsymbol{n}^{\mathbf{5}}$

We remove the multiplicative constants: $\boldsymbol{n}^{\mathbf{5}}$

$$\boldsymbol{C}(\boldsymbol{n}) = \boldsymbol{O}(\boldsymbol{n}^{\mathbf{5}})$$

4.7.3. Type of complexity

The following table gives the types of complexity usually encountered.

Table 2. Type of complexity

Big O notation	**Name**
O(1)	Constant complexity
O(log n)	Logarithmic complexity
O(n)	Linear complexity
O(n log n)	Quasi-linear complexity
$O(n^2)$	Quadratic complexity
$O(2^n)$	Exponential complexity
O(n!)	Factorial complexity

Big O notation	**Name**
O(1)	Polynomial Complexity
O(log n)	
O(n)	
O(n log n)	
$O(n^2)$	
$O(2^n)$	Non-Polynomial Complexity
O(n!)	

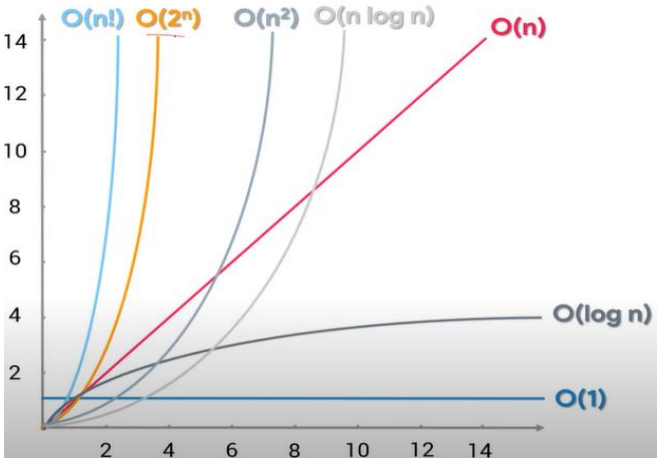

Figure 4. Type of complexity

5. Search algorithms

A search algorithm is an algorithm that searches for an element or the position of an element in a list or array.

A search algorithm is an algorithm that locates specific data among a collection of data (table, linked list, tree, graph, etc.).

There are two main search categories that only apply to tables:

5.1. Sequential search

Sequential search (linear search) is the simplest technique for searching for an element in an array.

The algorithm starts from the beginning of the array and checks each element until the desired element is found.

Sequential search can be applied on a sorted or unsorted array.

```
Function sequential_search(table, size, x)
    For i from 0 to size - 1
        if table[i] = x
                return i
        End If
    End for
return -1
End function
```

5.2. Dichotomous search

Dichotomous search (binary search) is a search algorithm for finding the position of an element in a **drawn array** . It is based on the divide and conquer methodology.

The algorithm splits the entire array into two subarrays. If the element is in the box in the middle of the array it returns the location, otherwise it jumps to the left or right subarray and repeats the same process until it finds the element.

```
Fonction dichotomous_search(table, x, inf, sup)
mil = (inf + sup)/2
while (inf <= sup) do
if table[mil] = x
      return mil
else if  table[mil] < x
          inf = mil +1
else
          sup = mil - 1
Fin Si
End while
return -1
end function
```

6. Sorting algorithms

6.1. Definition

A sorting algorithm is an algorithm that puts the elements of a list or array in order.

The most frequently used orders are numerical order and alphabetical order and either ascending or descending.

Sorting is important to optimize the efficiency of other algorithms such as search algorithms.

It is easier and faster to locate elements in a sorted array than an unsorted one.

Sorting is used to represent data in more human-readable formats.

6.2. Sort category

Sort by comparison	Divide and rule	Sort by counting
Selection Insertion Bubble Heap Shell	Fast Merger	Counting Base

6.2.1. Sort by comparison

In this category, the items to be sorted are compared two by two, using comparison operations to determine their relative order.

The most common algorithms in this category are:

- Sort by selection (selection sort),
- Insertion sort (insertion sort),
- Bubble sort ,
- Sort by heap (heap sort),
- Shell sort (shell sort).

6.2.2. Divide and rule

It is a problem-solving strategy that involves dividing a problem into smaller sub-problems, solving them separately, and then combining the results to solve the original problem. The most common algorithms in this category are:

- Quick sort (quick sort).
- Merge sort.

6.2.3. Sort by counting

Count sort is a sorting method that counts the number of occurrences of each item in a list and arranges them in ascending or descending order. The most common algorithms in this category are:

- Sort by counting (counting sort),
- Sort by radix (radix sort).

6.3. Sort by selection

Selection sorting consists of each step finding the smallest element not yet sorted and placing it after the already sorted elements. Its principle is as follows:

1. Find the smallest element in the array and swap it with the first element in the array.
2. Find the ^{2nd} smallest element in the array and swap it with the second element in the array.
3. Repeat the process until the array is sorted.

```
Function selection_sort(table)
    for i from 0 to size(table) - 1
        index_min <- i
        for j from i+1 to size(table)
            If table[j] < table[index_min]
                index_min <- j
            end if
        end for
        if index_min ≠ i
            swap table[i] et table[index_min]
        End if
    End for
End function
```

```
Function swap(a, b)
      tmp <- a
      a <- b
      b <- tmp
```

6.4. Insertion sort

Insertion sort consists of inserting each element of the array in the correct place among the elements already sorted. Its principle is as follows:

1. Sort the first two elements of the array in the correct order
2. The 3rd element is inserted in its place among the 2 others
3. The 4th is inserted in its place among the other 3
4. Etc.

```
Function insertion_sort(table)
    for i from 1 to size(table) - 1
        element_to_insert <- table[i]
        j <- i - 1
        while j >= 0 and table[j] > element_to_insert
            table[j + 1] <- table[j]
            j <- j - 1
        end while
        table[j + 1] <- element_to_insert
    end for
end function
```

6.5. Bubble sort

Bubble sort is the simplest sorting algorithm that works by comparing adjacent elements and swapping them if they are out of order.

```
Function bubble_sort(table)
    n <- size(table)
    for i from 0 to n-1
        for j from 0 to n-i-1
            if table[j] > table[j+1]
                swap table[j] et table[j+1]
            End if
        End for
    End for
End function
```

```
Function swap(a, b)
      tmp <- a
      a <- b
      b <- tmp
end Function
```

6.6. Quick sort

Quicksort is a recursive sorting algorithm. It is based on the divide and conquer paradigm. Its principle is as follows:

1. Select a pivot element (the last one in the table).

2. Divide the array into two subarrays with the elements lower than the pivot on the left and the upper ones on the right.

3. Repeat the process recursively on the two subarrays until you have arrays of length 1.

4. Combine the arrays to construct the sorted array.

```
Function quick_sort(table, b, h)
Begin
if h > b
pivot = partition(T, b, h)
quick_sort (T, b, pivot-1)
quick_sort (T, pivot+1, h)
end if
end function
Function partition(table, b, h)
begin
pivot = table[h]
i=(b-1)
for j from b to h
if table[j]<=pivot
i++
Echanger(table[i], table[j])
End if
End for
swap(table[i+1], table[h])
Return (i+1)
```

```
Function swap(a, b)
      tmp <- a
      a <- b
      b <- tmp
end Function
```

6.7. Merge sort

Merge sort is an algorithm invented by John von Neumann in 1945 that is based on the divide and conquer paradigm . Its principle is:

1. Divide the array into two arrays of identical sizes and repeat the operation until you have an array of size 1.

2. Recursively sort the two subarrays.

3. Combine the two sorted subarrays into a single sorted array.

6.8. Counting

Count sorting is based on how often elements appear in the array to be sorted.

It counts the number of occurrences of elements then placing them in the array sorted by number of occurrences.

6.9. Base

Sort by radix is based on sort by counting.

The algorithm sorts the elements according to the ones digits then the tens digits, until reaching the maximum number of digits contained in an element.

7. Distributed/Distributed Systems

7.1. Historical

- 1945 – 1980 uses of mainframes (Large machines)
- 1985 – LAN, WAN, computing power
- Remote access

Flynn's taxonomy - 1972

SISD: **Single instruction on single data-pc uniprocessor**

SIMD: **Single instruction multiple vector data-machine**

MISD: **Multiple instructions single data-pipeline**

MIMD: **Multiple instructions on multiple data**

With MIMD we have: Multi-processor and Distributed system

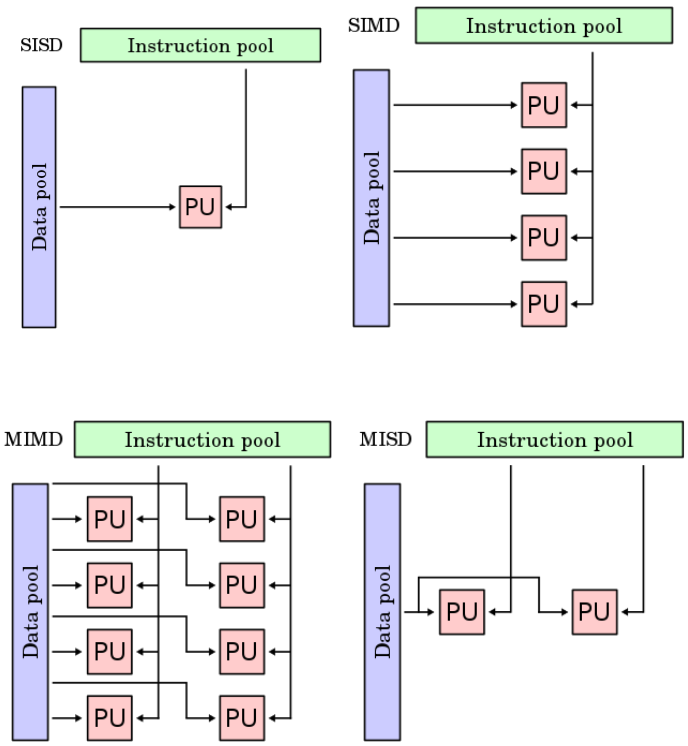

Figure 5. Flynn's taxonomy - 1972

7.2. Definition

7.3. Distributed System

A distributed system emphasizes the distribution of tasks or processes among multiple computers or network nodes.

In a distributed system, tasks or processes are executed concurrently on different nodes in the system, often with the goal of improving performance, increasing availability, or distributing the workload.

Nodes in a distributed system can operate independently and communicate with each other over a network to coordinate their activities.

A common example of a distributed system is a cluster of servers where each server runs part of an overall application or service.

7.4. A distributed system

A distributed system emphasizes the distribution of data or resources, rather than the distribution of tasks or processes.

In a distributed system, data, files, objects or other resources are distributed across different nodes in the network to facilitate their access and use in a transparent manner.

Distributed resources can be shared and used by multiple users or applications simultaneously, as in the case of a distributed database or distributed file system.

The primary goal of a distributed system is to provide seamless access to distributed resources, regardless of the physical location of those resources.

However, the two terms are often used interchangeably, as many modern computing systems can have both task distribution and data distribution aspects.

Systems which consist of multiple computing resources and which are autonomous resources (working alone or collaboratively) and independent (no need for information from other

systems) and can be of different types (pc, phone, tablet, etc.) connected through a communications network.

These resources have no common memory or common clock and we cannot have an instantaneous global state of the entire system.

The goal is to collaborate to achieve a common task.

From a user point of view, the distributed system is transparent (centralized system).

7.5. Development

To develop a distributed system you must use middleware.

Middleware is a type of software or software component that acts as an intermediate layer (hence the name) between different applications, systems or computer components to facilitate communication, data management and coordination of operations.

It is an intermediate software layer that allows distributed applications to communicate transparently.

7.6. Middleware models

7.6.1. Synchronous

- RMI (Remote Method Invocation)
- CORBA (Common Object Request Broker Architecture)
- Web services
    1. SOAP (http1.1+XML)
    2. REST (http1.1, JSON, XML, etc.)
    3. GraphQL (http1.1, JSON)

4. GRPC (http2, ProtoBuf)

7.6.2. Asynchronous

(MOM: Message Oriented Middleware)

Communication via a Broker

- RabbitMQ
- KAFKA
- ActiveMQ

8. Exercises

8.1. Series No. 1: Reminder on algorithms

Exercise 1

Write an algorithm that calculates the sum of integers from 1 to n.

Solution 1

Beginning

Read n

sum ← 0

For i from 1 to n Do

sum ← sum + i

End For

Show "The sum of integers from 1 to", n, "is", sum

END

Exercise 2

Write an algorithm that calculates the PGCD (Greatest Common Divisor).

Solution 2

Beginning

Read a, b

While b ≠ 0 Do

temp ← b

b ← a % b

a ← temp

End While

Show "The GCD of", a, "and", b, "is", a

END

Exercise 3

Searching for a maximum element in an array.

Solution 3

Beginning

Read table

n ← length(array)

max _element ← array[0]

For i from 1 to n-1 Do

If array[i] > max_element Then

max _element ← array[i]

End if

End For

Show "Maximum element in array is", max_element

END

Exercise 4

Write an algorithm that searches for an element in an array.

Solution 4

Beginning

Read table

Read item to search for

found ← False

For each element in array Do

If element = element to search Then

found ← True

Leave the loop

End if

End For

If found = True Then

Show "The element", element to search for, "was found in the table."

Otherwise

Show "The element", element to search for, "was not found in the table."

End if

END

Exercise 5

Write an algorithm that calculates the factorial of a number.

Solution 5

Beginning

Read n

result ← 1

For i from 1 to n Do

result ← result * i

End For

Show "The factorial of", n, "is", result

END

8.2. Series No. 2: Algorithmic Complexity

Exercise 1

Consider the following algorithm:

```
Algorithm A1
Var A, B, i , N: integer
Beginning
Read( N) A←5 B←3 (3)
If N mod 2=0 then                ( 2)
```

```
   For i=1 a N do          ( 3n+1)

A←A+B

   EndFor

Otherwise

As long as N != 0 do     (Ln(N)/Ln(2))(4+1)+1

      B←B+A

N←N div 2

   FinTq

End if

Write( A,B )  (1)

END
```

C=(3n+8)

```
Procedure Read ( T[ ],N: integer)
Var
i: integer
Beginning
For i=1 a N make 2n+1
Read(T[i])
EndFor
END
```

```
Procedure Sort ( T[ ],N: int )
Var
i,k ,s : integer
Beginning
For i= 1 a N- 1 TO DO          ( n-1)*[(ni)*5+1]+1
For K= i+1 a N do       ( nor)*5+1
If (T[i]>T[k ]) then     4
          s←T [i]
T[i]←T[k]
```

```
T[k]←s
      End if
   EndFor
EndFor
```

The variable i repeats itself N-1 times

arithemetic sequence because i is repeated N-1 times
(N- 1)* i= i+i+i …+i=1+2+…+N-1=n*(n-1)/2

Exercise 2

Expression	Dominant term
$5 + 0.001n^3 + 0.025n$	$0.001n^3$
$500n + 100n^{1.5} + 50nlog_{10}n$	$100n^{1.5}$
$0.3n + 5n^{1.5} + 2.5n^{1.75}$	$2.5n^{1.75}$
$n^2log_2n + n(log_2n)^2$	n^2log_2n
$nlog_3n + nlog_2n$	$nlog_3n, nlog_2n$
$0.01n + 100n^2$	$100n^2$

8.3. TD Series No. 3: Algorithmic Complexity (function)

Exercise 1

Calculate the complexity of the following function:

```
function ( n: integer):

Beginning
if n==0 then
   return 0
Otherwise
   return n + sum(n-1)
End if
END
```

Cn = 1(if)+1(return)+1(sum)+1(subtraction)+1(param)C $_{n-1}$

= 5+Cn-1=5*n+C $_{0}$= 5n+2

Exercise 2

Calculate the complexity of the following algorithm:

```
Function f1 (x: integer):
Beginning
   x ←x+1
END
Algorithm
Var
i,n : integer
Beginning
Read(n)
For i= 1 to do
f1(i)
EndFor
END
```

C=n*(1(param)+2(function))+ 1= 3n+1

Exercise 3

Calculate the complexity of the following algorithm:

```
Function f1 (x: integer):
Beginning
For i= 1 to do
x←i+2    ( 2n+1)
EndFor
END
Function f2 (x: integer):
Beginning
For i =1 to do
f1(i) m*(2n+ 3)+ 1
EndFor
END

Algorithm
Beginning
Read(y)
f 2(y)
END
```

2 +[m *(2n+3)+1]

References

Course Professor Mahseur Mohammed, Houari Boumediene University of Science and Technology

Course Professor Dalila Chiadmi , Mohammadia School of Engineering

Course Professor Mohamed Youssfi , Hassan II University Casablanca

Course Professor Hassan El Bahi , School of Management, Cadi Ayyad University

www.resilio.com

www. atlassian.com

Aaron McCoy, Declan Delaney, Tomas E Ward. Game-state fidelity across distributed interactive games . The ACM Magazine for Students, Vol.12, No.1