Introduction to Computer Science

Algorithms and data structures

Piotr Fulmański

Faculty of Mathematics and Computer Science,
University of Łódź, Poland

November 19, 2008

Table of Contents

Name

Term *algorithm* comes from the name of Persian astronomer and mathematician lived between VIII and IX AD. In 825 AD Muhammad ibn Musa al-Chorezmi (al-Khawarizmy) wrote treatise *On Calculation with Hindu Numerals*, where he precisely described many mathematical rules (e.g. addition or multiplication of decimal numbers). It was translated into Latin in the 12th century as *Algoritmi de numero Indorum*, which title was likely intended to mean *Algoritmi on the numbers of the Indians*, where *Algoritmi* was the translator's rendition of the author's name; but people misunderstanding the title treated Algoritmi as a Latin plural and this led to the word *algorithm* (Latin algorismus) coming to mean *calculation method*.

An informal definition

No generally accepted formal definition of *algorithm* exists yet. As the term is popularly understood, *algorithm* mean the way of doing sth, recipe for sth or formula for sth.

More formal definition

In mathematic and computer science, *algorithm* mean finite, ordered sequence of clearly defined actions, needed to perform some task. Algorithm should meet the following conditions:

- explicitness or uniqueness (jednoznaczność)
- standed out the beginning and the end
- discreteness
- versatility
- effectiveness

An informal definition

No generally accepted formal definition of *algorithm* exists yet. As the term is popularly understood, *algorithm* mean the way of doing sth, recipe for sth or formula for sth.

More formal definition

In mathematic and computer science, *algorithm* mean finite, ordered sequence of clearly defined actions, needed to perform some task. Algorithm should meet the following conditions:

- explicitness or uniqueness (jednoznaczność)
- standed out the beginning and the end
- discreteness
- versatility
- effectiveness

An informal definition

No generally accepted formal definition of *algorithm* exists yet. As the term is popularly understood, *algorithm* mean the way of doing sth, recipe for sth or formula for sth.

More formal definition

In mathematic and computer science, *algorithm* mean finite, ordered sequence of clearly defined actions, needed to perform some task. Algorithm should meet the following conditions:

- explicitness or uniqueness (jednoznaczność)
- standed out the beginning and the end
- discreteness
- versatility
- effectiveness

An informal definition

No generally accepted formal definition of *algorithm* exists yet. As the term is popularly understood, *algorithm* mean the way of doing sth, recipe for sth or formula for sth.

More formal definition

In mathematic and computer science, *algorithm* mean finite, ordered sequence of clearly defined actions, needed to perform some task. Algorithm should meet the following conditions:

- explicitness or uniqueness (jednoznaczność)
- standed out the beginning and the end
- discreteness
- versatility
- effectiveness

Algorithm

An informal definition

No generally accepted formal definition of *algorithm* exists yet. As the term is popularly understood, *algorithm* mean the way of doing sth, recipe for sth or formula for sth.

More formal definition

In mathematic and computer science, *algorithm* mean finite, ordered sequence of clearly defined actions, needed to perform some task. Algorithm should meet the following conditions:

- explicitness or uniqueness (jednoznaczność)
- standed out the beginning and the end
- discreteness
- versatility
- effectiveness

An informal definition

No generally accepted formal definition of *algorithm* exists yet. As the term is popularly understood, *algorithm* mean the way of doing sth, recipe for sth or formula for sth.

More formal definition

In mathematic and computer science, *algorithm* mean finite, ordered sequence of clearly defined actions, needed to perform some task. Algorithm should meet the following conditions:

- explicitness or uniqueness (jednoznaczność)
- standed out the beginning and the end
- discreteness
- versatility
- effectiveness

An informal definition

No generally accepted formal definition of *algorithm* exists yet. As the term is popularly understood, *algorithm* mean the way of doing sth, recipe for sth or formula for sth.

More formal definition

In mathematic and computer science, *algorithm* mean finite, ordered sequence of clearly defined actions, needed to perform some task. Algorithm should meet the following conditions:

- explicitness or uniqueness (jednoznaczność)
- standed out the beginning and the end
- discreteness
- versatility
- effectiveness

The place

The place of algorithm in a process of creating program solving stated problem.

- problem
- computer (time, internal data representation, software)
- programming language (available construction and data types)
- **algorithm**
- program

The place

The place of algorithm in a process of creating program solving stated problem.

- problem
- computer (time, internal data representation, software)
- programming language (available construction and data types)
- **algorithm**
- program

The place

The place of algorithm in a process of creating program solving stated problem.

- problem
- computer (time, internal data representation, software)
- programming language (available construction and data types)
- **algorithm**
- program

The place

The place of algorithm in a process of creating program solving stated problem.

- problem
- computer (time, internal data representation, software)
- programming language (available construction and data types)
- **algorithm**
- program

The place

The place of algorithm in a process of creating program solving stated problem.

- problem
- computer (time, internal data representation, software)
- programming language (available construction and data types)
- **algorithm**
- program

The place

The place of algorithm in a process of creating program solving stated problem.

- problem
- computer (time, internal data representation, software)
- programming language (available construction and data types)
- **algorithm**
- program

Data processing

Limited information

- Information stored and processed by computer is a small fragment of reality containing essential data to solve stated problem.
- We have to think which informations are essential, which can help us and which are completely useless.
- We have to think how we will represent choosen informations.

The last point lead us to notion of data type (data structure).

Limited information

- Information stored and processed by computer is a small fragment of reality containing essential data to solve stated problem.
- We have to think which informations are essential, which can help us and which are completely useless.
- We have to think how we will represent choosen informations.

The last point lead us to notion of data type (data structure).

Limited information

- Information stored and processed by computer is a small fragment of reality containing essential data to solve stated problem.
- We have to think which informations are essential, which can help us and which are completely useless.
- We have to think how we will represent choosen informations.

The last point lead us to notion of data type (data structure).

Limited information

- Information stored and processed by computer is a small fragment of reality containing essential data to solve stated problem.
- We have to think which informations are essential, which can help us and which are completely useless.
- We have to think how we will represent choosen informations.

The last point lead us to notion of data type (data structure).

Limited information

- Information stored and processed by computer is a small fragment of reality containing essential data to solve stated problem.
- We have to think which informations are essential, which can help us and which are completely useless.
- We have to think how we will represent choosen informations.

The last point lead us to notion of data type (data structure).

A data structure

A data structure is a way of storing data in a computer so that it can be used efficiently. Often a carefully chosen data structure will allow the most efficient algorithm to be used.

Data type

The most popular division distinguish primitive types, also known as built-in types or basic types and composite types — types composed of basic types.
As a primitive types we consider:

- numerical type (e.g. integer, floating-point number, fixed-point number)
- character type (alphanumeric symbols)
- boolean type

As a composite types (also known as data structures) we consider:

- array
- dictionary
- set
- record
- file
- queue
- stack
- tree

Data type

The most popular division distinguish primitive types, also known as built-in types or basic types and composite types — types composed of basic types.

As a primitive types we consider:

- numerical type (e.g. integer, floating-point number, fixed-point number)
- character type (alphanumeric symbols)
- boolean type

As a composite types (also known as data structures) we consider:

- array
- dictionary
- set
- record
- file
- queue
- stack
- tree

Data type

The most popular division distinguish primitive types, also known as built-in types or basic types and composite types — types composed of basic types.

As a primitive types we consider:

- numerical type (e.g. integer, floating-point number, fixed-point number)
- character type (alphanumeric symbols)
- boolean type

As a composite types (also known as data structures) we consider:

- array
- dictionary
- set
- record
- file
- queue
- stack
- tree

Data type

The most popular division distinguish primitive types, also known as built-in types or basic types and composite types — types composed of basic types.

As a primitive types we consider:

- numerical type (e.g. integer, floating-point number, fixed-point number)
- character type (alphanumeric symbols)
- boolean type

As a composite types (also known as data structures) we consider:

- array
- dictionary
- set
- record
- file
- queue
- stack
- tree

Data type

The most popular division distinguish primitive types, also known as built-in types or basic types and composite types — types composed of basic types.

As a primitive types we consider:

- numerical type (e.g. integer, floating-point number, fixed-point number)
- character type (alphanumeric symbols)
- boolean type

As a composite types (also known as data structures) we consider:

- array
- dictionary
- set
- record
- file
- queue
- stack
- tree

Data type

The most popular division distinguish primitive types, also known as built-in types or basic types and composite types — types composed of basic types.

As a primitive types we consider:

- numerical type (e.g. integer, floating-point number, fixed-point number)
- character type (alphanumeric symbols)
- boolean type

As a composite types (also known as data structures) we consider:

- array
- dictionary
- set
- record
- file
- queue
- stack
- tree

Data type

The most popular division distinguish primitive types, also known as built-in types or basic types and composite types — types composed of basic types.

As a primitive types we consider:

- numerical type (e.g. integer, floating-point number, fixed-point number)
- character type (alphanumeric symbols)
- boolean type

As a composite types (also known as data structures) we consider:

- array
- dictionary
- set
- record
- file
- queue
- stack
- tree

Data type

The most popular division distinguish primitive types, also known as built-in types or basic types and composite types — types composed of basic types.

As a primitive types we consider:

- numerical type (e.g. integer, floating-point number, fixed-point number)
- character type (alphanumeric symbols)
- boolean type

As a composite types (also known as data structures) we consider:

- array
- dictionary
- set
- record
- file
- queue
- stack
- tree

Data type

The most popular division distinguish primitive types, also known as built-in types or basic types and composite types — types composed of basic types.

As a primitive types we consider:

- numerical type (e.g. integer, floating-point number, fixed-point number)
- character type (alphanumeric symbols)
- boolean type

As a composite types (also known as data structures) we consider:

- array
- dictionary
- set
- record
- file
- queue
- stack
- tree

Data type

The most popular division distinguish primitive types, also known as built-in types or basic types and composite types — types composed of basic types.

As a primitive types we consider:

- numerical type (e.g. integer, floating-point number, fixed-point number)
- character type (alphanumeric symbols)
- boolean type

As a composite types (also known as data structures) we consider:

- array
- dictionary
- set
- record
- file
- queue
- stack
- tree

Data type

The most popular division distinguish primitive types, also known as built-in types or basic types and composite types — types composed of basic types.
As a primitive types we consider:

- numerical type (e.g. integer, floating-point number, fixed-point number)
- character type (alphanumeric symbols)
- boolean type

As a composite types (also known as data structures) we consider:

- array
- dictionary
- set
- record
- file
- queue
- stack
- tree

Data type

The most popular division distinguish primitive types, also known as built-in types or basic types and composite types — types composed of basic types.

As a primitive types we consider:

- numerical type (e.g. integer, floating-point number, fixed-point number)
- character type (alphanumeric symbols)
- boolean type

As a composite types (also known as data structures) we consider:

- array
- dictionary
- set
- record
- file
- queue
- stack
- tree

Data type

The most popular division distinguish primitive types, also known as built-in types or basic types and composite types — types composed of basic types.

As a primitive types we consider:

- numerical type (e.g. integer, floating-point number, fixed-point number)
- character type (alphanumeric symbols)
- boolean type

As a composite types (also known as data structures) we consider:

- array
- dictionary
- set
- record
- file
- queue
- stack
- tree

Examples of array usage

```
Ada:
-- definition of array type
type TableType is array(1 .. 100) of Integer;
-- variable definition of specyfic array type
MyTable : TableType;

Visual Basic:
Dim a(1 to 5,1 to 5) As Double
Dim MyIntArray(10) As Integer
Dim MySingleArray(3 to 5) As Single
```

Examples of array usage

```
C:
char my_string[40];
int my_array[] = {1,23,17,4,-5,100};

Java:
int [] counts;
counts = new int[5];

PHP:
$first_quarter = array(1 =>'January','February','March');

Python:
mylist = ["List item 1", 2, 3.14]
```

Example of dictionary usage

Python:

```
d = {"key1":"val1", "key2":"val2"}
x = d["key2"]
d["key3"] = 122
d[42] = "val4"
```

Methods of algorithm description

- Natural language
  - (theoretically) easy to write (enumerate actions)
  - problems with implementation
- block diagram or flowchart (also spelled flow-chart and flow chart)
  - high clarity
  - reflect structure of algorithm pointing out all branches (decisions points)
  - problems with implementation
- pseudocode
  - facilitate implementation
  - not so clear as natural language or flowchart

Methods of algorithm description

- Natural language
  - (theoretically) easy to write (enumerate actions)
  - problems with implementation
- block diagram or flowchart (also spelled flow-chart and flow chart)
  - high clarity
  - reflect structure of algorithm pointing out all branches (decisions points)
  - problems with implementation
- pseudocode
  - facilitate implementation
  - not so clear as natural language or flowchart

Methods of algorithm description

- Natural language
  - (theoretically) easy to write (enumerate actions)
  - problems with implementation
- block diagram or flowchart (also spelled flow-chart and flow chart)
  - high clarity
  - reflect structure of algorithm pointing out all branches (decisions points)
  - problems with implementation
- pseudocode
  - facilitate implementation
  - not so clear as natural language or flowchart

Methods of algorithm description

- Natural language
  - (theoretically) easy to write (enumerate actions)
  - problems with implementation
- block diagram or flowchart (also spelled flow-chart and flow chart)
  - high clarity
  - reflect structure of algorithm pointing out all branches (decisions points)
  - problems with implementation
- pseudocode
  - facilitate implementation
  - not so clear as natural language or flowchart

Euclidean algorithm

Consider the Euclidean algorithm (also called Euclid's algorithm) which is an algorithm to determine the greatest common divisor (GCD) of two positive integers.

- **1.** Consider two positive integer numbers: a i b.
- **2.** If $b = 0$ then go to step **3.**, else:
  - **2.1.** If $a > b$ then $a := a - b$.
  - **2.2.** else $b := b - a$.
  - **2.3.** Go to step **2.**
- **3.** Return a as result.
- **4.** The end

Euclidean algorithm

Consider the Euclidean algorithm (also called Euclid's algorithm) which is an algorithm to determine the greatest common divisor (GCD) of two positive integers.

- **1.** Consider two positive integer numbers: a i b.
- **2.** If $b = 0$ then go to step **3.**, else:
  - **2.1.** If $a > b$ then $a := a - b$.
  - **2.2.** else $b := b - a$.
  - **2.3.** Go to step **2.**
- **3.** Return a as result.
- **4.** The end

Euclidean algorithm

Consider the Euclidean algorithm (also called Euclid's algorithm) which is an algorithm to determine the greatest common divisor (GCD) of two positive integers.

- **1.** Consider two positive integer numbers: a i b.
- **2.** If $b = 0$ then go to step **3.**, else:
  - **2.1.** If $a > b$ then $a := a - b$.
  - **2.2.** else $b := b - a$.
  - **2.3.** Go to step **2.**
- **3.** Return a as result.
- **4.** The end

Euclidean algorithm

Consider the Euclidean algorithm (also called Euclid's algorithm) which is an algorithm to determine the greatest common divisor (GCD) of two positive integers.

- **1.** Consider two positive integer numbers: a i b.
- **2.** If $b = 0$ then go to step **3.**, else:
  - **2.1.** If $a > b$ then $a := a - b$.
  - **2.2.** else $b := b - a$.
  - **2.3.** Go to step **2.**
- **3.** Return a as result.
- **4.** The end

Euclidean algorithm

Consider the Euclidean algorithm (also called Euclid's algorithm) which is an algorithm to determine the greatest common divisor (GCD) of two positive integers.

- **1.** Consider two positive integer numbers: a i b.
- **2.** If $b = 0$ then go to step **3.**, else:
  - **2.1.** If $a > b$ then $a := a - b$.
  - **2.2.** else $b := b - a$.
  - **2.3.** Go to step **2.**
- **3.** Return a as result.
- **4.** The end

Euclidean algorithm

Consider the Euclidean algorithm (also called Euclid's algorithm) which is an algorithm to determine the greatest common divisor (GCD) of two positive integers.

- **1.** Consider two positive integer numbers: a i b.
- **2.** If $b = 0$ then go to step **3.**, else:
  - **2.1.** If $a > b$ then $a := a - b$.
  - **2.2.** else $b := b - a$.
  - **2.3.** Go to step **2.**
- **3.** Return a as result.
- **4.** The end

Euclidean algorithm

Consider the Euclidean algorithm (also called Euclid's algorithm) which is an algorithm to determine the greatest common divisor (GCD) of two positive integers.

- **1.** Consider two positive integer numbers: a i b.
- **2.** If $b = 0$ then go to step **3.**, else:
  - **2.1.** If $a > b$ then $a := a - b$.
  - **2.2.** else $b := b - a$.
  - **2.3.** Go to step **2.**
- **3.** Return a as result.
- **4.** The end

Symbols

- beginning and the end
- block of instructions
- decision/condition
- link
- read/write

Symbols

- beginning and the end
- block of instructions
- decision/condition
- link
- read/write

Symbols

- beginning and the end
- block of instructions
- decision/condition
- link
- read/write

Symbols

- beginning and the end
- block of instructions
- decision/condition
- link
- read/write

Symbols

- beginning and the end
- block of instructions
- decision/condition
- link
- read/write

Symbols

- beginning and the end
- block of instructions
- decision/condition
- link
- read/write

Rules

1. blocks are linked by oriented lines (ended by arrow)
2. always we performe either all instructions in block or none
3. subsequent operations not depend on previous unless the dependence are transmited with the use of data
4. the order of performing operation is strictly determine by oriented lines
5. into each blocks leads only one line
6. lines can be joined into one in point called meeting point

Rules

1. blocks are linked by oriented lines (ended by arrow)
2. always we performe either all instructions in block or none
3. subsequent operations not depend on previous unless the dependence are transmited with the use of data
4. the order of performing operation is strictly determine by oriented lines
5. into each blocks leads only one line
6. lines can be joined into one in point called meeting point

Rules

1. blocks are linked by oriented lines (ended by arrow)
2. always we performe either all instructions in block or none
3. subsequent operations not depend on previous unless the dependence are transmited with the use of data
4. the order of performing operation is strictly determine by oriented lines
5. into each blocks leads only one line
6. lines can be joined into one in point called meeting point

Rules

1. blocks are linked by oriented lines (ended by arrow)
2. always we performe either all instructions in block or none
3. subsequent operations not depend on previous unless the dependence are transmited with the use of data
4. the order of performing operation is strictly determine by oriented lines
5. into each blocks leads only one line
6. lines can be joined into one in point called meeting point

Rules

1. blocks are linked by oriented lines (ended by arrow)
2. always we performe either all instructions in block or none
3. subsequent operations not depend on previous unless the dependence are transmited with the use of data
4. the order of performing operation is strictly determine by oriented lines
5. into each blocks leads only one line
6. lines can be joined into one in point called meeting point

Rules

1. blocks are linked by oriented lines (ended by arrow)
2. always we performe either all instructions in block or none
3. subsequent operations not depend on previous unless the dependence are transmited with the use of data
4. the order of performing operation is strictly determine by oriented lines
5. into each blocks leads only one line
6. lines can be joined into one in point called meeting point

Rules

1. blocks are linked by oriented lines (ended by arrow)
2. always we performe either all instructions in block or none
3. subsequent operations not depend on previous unless the dependence are transmited with the use of data
4. the order of performing operation is strictly determine by oriented lines
5. into each blocks leads only one line
6. lines can be joined into one in point called meeting point

Flowchart of the Euclidean algorithm

Statements

Pseudocode does not actually obey the syntax rules of any particular language. There is no systematic standard form, although any particular writer will generally borrow the appearance of a particular language. Popular sources include C, Java, PHP, Python etc. Details not relevant to the algorithm (such as memory management code) are usually omitted. Blocks of code, for example code contained within a loop, may be described in a one-line natural language sentence.

We will use the following notation

- assignment statement

```
x:=y;
age:=12.6;
name:="Piotr";
```

Statements

Pseudocode does not actually obey the syntax rules of any particular language. There is no systematic standard form, although any particular writer will generally borrow the appearance of a particular language. Popular sources include C, Java, PHP, Python etc. Details not relevant to the algorithm (such as memory management code) are usually omitted. Blocks of code, for example code contained within a loop, may be described in a one-line natural language sentence.
We will use the following notation

- assignment statement

  ```
  x:=y;
  age:=12.6;
  name:="Piotr";
  ```

Statements

- block (block of statements)

```
begin
  block consist of
  instructions/statements
end
```

Statements

- if statement (condition)

```
if (CONDITION) then
begin
  TRUE
end
```

```
if (CONDITION) then
begin
  TRUE
end
else
begin
  FALSE
end
```

- `CONDITION` — expression which is true or false, e.g.

```
x=7
x>12
x>12 and y<3
x=5 and (y=1 or z=2)
```

- `TRUE` (`FALSE`) — block performed when condition is true (false)

Statements

- do-while and while statement (loop)

```
do
begin
  instructions
end
while (CONDITION);
```

```
while (CONDITION)
begin
  instructions
end
```

Statements

- for statement (loop)

```
for i:=1 to 10 step 1 do
begin
  instructions
end

for i in NAME do
begin
  instructions
end
```

- `NAME` — variable represented list, dictionary, queue, set etc.

Function

Function as a black box for doing defined task.

- function call:

```
FunctionName(arguments);
x:=Function(arg1,arg2,arg3);
```

- definition of a function (body of a function):

```
function FunctionName(arguments)
begin
  statements/instructions
  return returnedValue;
end
```

Function

Function as a black box for doing defined task.

- function call:

```
FunctionName(arguments);
x:=Function(arg1,arg2,arg3);
```

- definition of a function (body of a function):

```
function FunctionName(arguments)
begin
  statements/instructions
  return returnedValue;
end
```

Function

Function as a black box for doing defined task.

- function call:

```
FunctionName(arguments);
x:=Function(arg1,arg2,arg3);
```

- definition of a function (body of a function):

```
function FunctionName(arguments)
begin
  statements/instructions
  return returnedValue;
end
```

Iteration and recursion

Iteration

Iteration (lat. *iteratio*) is an action of repeting (often many times) the same instruction or block of instructions.

Recursion

Recursion (lat. *recurrere*, going back) means a function or definition calling itself.

Iteration and recursion

Iteration

Iteration (lat. *iteratio*) is an action of repeting (often many times) the same instruction or block of instructions.

Recursion

Recursion (lat. *recurrere*, going back) means a function or definition calling itself.

The iteration method of factorial

n! = 1 * 2 * 3 * ... * n

The recursive method of factorial

n! = n * (n-1)!

The factorial

```
function SilniaI(n)
begin
  i:=0;
  s:=1;
  while (i<n) do
  begin
    i:=i+1;
    s:=s*i;
  end
  return s;
end
```

```
function SilniaR(n)
begin
  if (n=0) then
  begin
    return 1;
  end
  else
  begin
    return n*SilniaR(n-1);
  end
end
```

The iteration method of factorial

```
n! = 1 * 2 * 3 * ... * n
```

The recursive method of factorial

```
n! = n * (n-1)!
```

The factorial

```
function SilniaI(n)
begin
  i:=0;
  s:=1;
  while (i<n) do
  begin
    i:=i+1;
    s:=s*i;
  end
  return s;
end
```

```
function SilniaR(n)
begin
  if (n=0) then
  begin
    return 1;
  end
  else
  begin
    return n*SilniaR(n-1);
  end
end
```

The iteration method of factorial

```
n! = 1 * 2 * 3 * ... * n
```

The recursive method of factorial

```
n! = n * (n-1)!
```

The factorial

```
function SilniaI(n)
begin
  i:=0;
  s:=1;
  while (i<n) do
  begin
    i:=i+1;
    s:=s*i;
  end
  return s;
end
```

```
function SilniaR(n)
begin
  if (n=0) then
  begin
    return 1;
  end
  else
  begin
    return n*SilniaR(n-1);
  end
end
```

Tree of recursion calls for 4!

```
5*SilniaR(4)
.         |
.         4*SilniaR(3)
.         .         |
.         .         3*SilniaR(2)
.         .         .         |
.         .         .         2*SilniaR(1)
.         .         .         .         |
.         .         .         .         1*SilniaR(0)
.         .         .         .         .         |
.         .         .         .         . <-------1
.         .         .         .         . |
.         .         .         . <-------1*1
.         .         .         . |
.         .         . <-------2*1
.         .         . |
.         . <-------3*2
.         . |
<---------4*6
|
24
```

Definition of Fibonacci numbers

For any $n > 1$ we define

$$fib_n = fib_{n-1} + fib_{n-2}.$$

Term 1. and 0. takes 1 as its value.

The recursive method of computing Fibonacci sequence

```
function FibR(n)
begin
  if ( n=0 or n=1) then
  begin
    return 1;
  end

  return FibR(n-1)+FibR(n-2);
end
```

Definition of Fibonacci numbers

For any $n > 1$ we define

$$fib_n = fib_{n-1} + fib_{n-2}.$$

Term 1. and 0. takes 1 as its value.

The recursive method of computing Fibonacci sequence

```
function FibR(n)
begin
  if ( n=0 or n=1) then
  begin
    return 1;
  end

  return FibR(n-1)+FibR(n-2);
end
```

Time

Tree of recursion calls for 5th term of Fibonacci sequence

```
FibR(5)
     |
     +--FibR(4)
     |       |
     |       +--FibR(3)
     |       |       |
     |       |       +--FibR(2)
     |       |       |       |
     |       |       |       +--FibR(1)
     |       |       |       +--FibR(0)
     |       |       |
     |       |       +--Fib(1)
     |       |
     |       +--FibR(2)
     |               |
     |               +--FibR(1)
     |               +--FibR(0)
     +--FibR(3)
             |
             +--FibR(2)
                ...
```

Number of calls

0	1	12	
1	1	14	
2	3	16	
3	5	18	
4	9	20	21891
5	15	22	
6	25	24	
7	41	26	
8	67	28	
9	109	30	2692537
10	177	32	
		34	
		36	
		38	
		40	

The iteration method of computing Fibonacci sequence

```
function FibI(n)
begin
  i:=1;
  x:=1;
  y:=1;

  while (i<n)
  begin
    z:=x;
    i:=i+1;
    x:=x+y;
    y:=z;
  end

  return x;
end
```