Discrete Mathematics

CMP-101

Lecture 11
Introduction to Algorithms, Linear Search, Binary Search

Abdul Hameed

http://informationtechnology.pk/

abdul.hameed@pucit.edu.pk

Outline

- Introduction to Algorithms
- Linear Search
- Binary Search

Computational problems

- A computational problem specifies an input-output relationship
  - What does the input look like?
  - What should the output be for each input?
- Example:
  - Input: an integer number N
  - Output: Is the number prime?

Computational problems

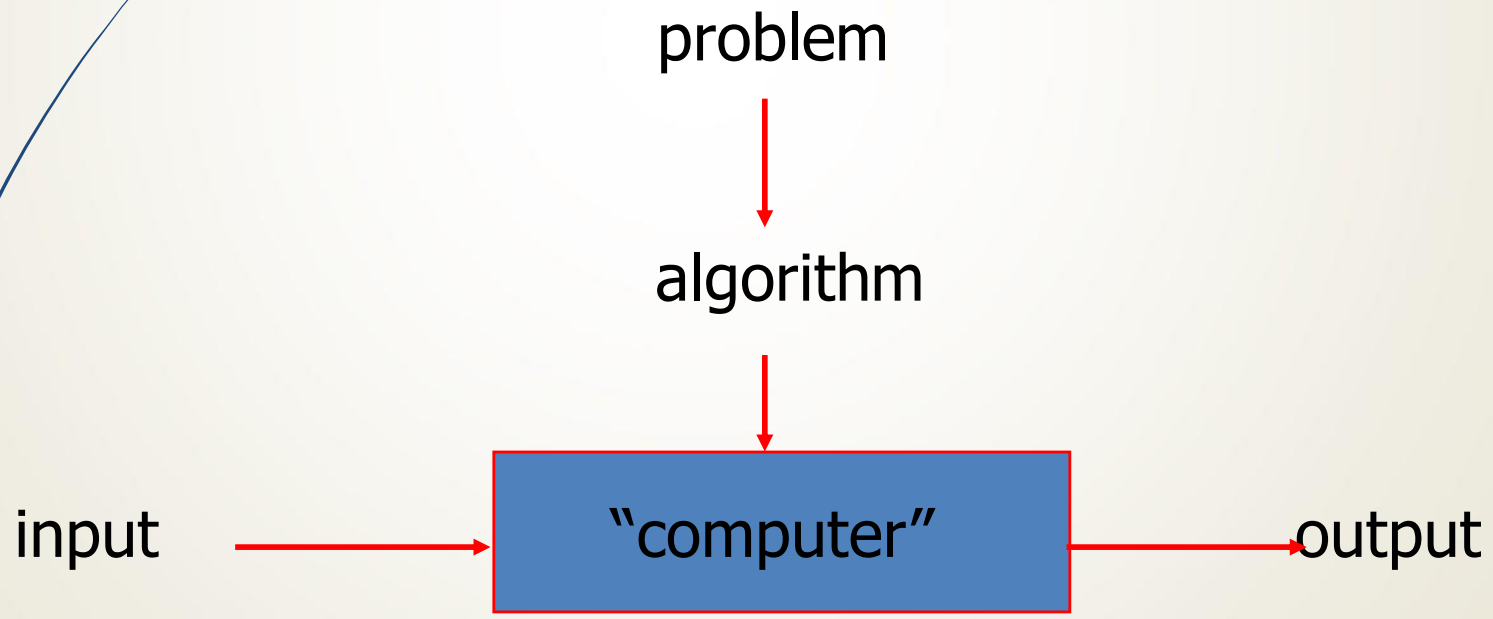
- Example:
  - Input: A list of names of people
  - Output: The same list sorted alphabetically
- Example:
  - Input: A picture in digital format
  - Output: An English description of what the picture shows

Algorithm

- An algorithm is an exact specification of how to solve a computational problem
- An algorithm must specify every step completely, so a computer can implement it without any further “understanding”
- An algorithm must work for all possible inputs of the problem.

Algorithm

An *algorithm* is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.



Algorithm

- Algorithms must be:
  - Correct: For each input produce an appropriate output
  - Efficient: run as quickly as possible, and use as little memory as possible – more about this later
- There can be many different algorithms for each computational problem.

Definition

- In simple terms, an algorithm is a series of instructions to solve a problem (complete a task)

- Problems can be in any form
  - Business
    - Get a part from Vancouver to Ottawa by morning
    - Allocate manpower to maximize profit
  - Life
    - I am hungry. How do I order pizza?
    - Explain how to tie shoelaces to a five year old child

Definition

- We deal with data processing problems
- translated to programs that can be run on a computer
- Since we can only input, store, process & output data on a computer, the instructions in our algorithms will be limited to these functions

Algorithmic Representation of Computer Functions

- Input
  - Get information — Get (input command)
- Storage
  - Store information — Given/Result
    Intermediates/Set
- Process
  - Arithmetic — Let (assignment command)
  - Repeat instructions — Loop
  - Branch conditionals — If
- Output
  - Give information — Give (output command)

Algorithm Description

- Understand the problem before solving it
  - Identify & name each Input/**Givens**
  - Identify & name each Output/**Results**
  - Assign a name to our algorithm (**Name**)
  - Combine the previous 3 pieces of information into a formal statement (**Definition**)
    - Results := Name (Givens)

Method

- Once we have prepared the Algorithm Description, we need to solve the problem

- We develop a series of instructions (limited to those described previously) that, when executed, will compute the desired **Results** from the **Givens** (**Method**)

Assignment command

Syntax

$$X = 5Y + 16$$

On the left side of =, we put the name of a variable and on the right side we put a value or an expression.

Each variable refers to a unique location in computer memory that contains a value.

Interpretation

An assignment is executed in two steps :

1-evaluation of the *expression* found on the right side.

2-setting the returned value in the memory cell corresponding to *variable*.

Example

Let SideSize=15

Let Area=SideSize∗SideSize

Assignment in computer science and equality in mathematics

a) The following instructions are the same in mathematics.

A=B B=A

not in computer science.

Let A=B is different from Let B=A

b) In mathematics we work with relations.

A relation B=A+1 means that it is true all the time

In computer science, we work with assignments. We can have:

A=5
B=A+1
A=2

The relation B=A+1 is true only after the second instruction and before the third one.

Assignment in computer science and equality in mathematics

c) The instruction A=A+3 is false in mathematics.

In computer science Let A=A+3 means: the new value of A is equal to the old one plus three.

d) The instruction A+5=3 is allowed in mathematics (it is an equation).

Let A+5=3 has no meaning in computer science (the left side must be a variable).

Input command

Syntax

Get variable

The variable must be from Givens

Interpretation

Here the user must give a value. The given value is assigned to the variable.

Example

Get Size_Side

Output command

Syntax

Give variable

The variable must be from Results

Interpretation

The value of the variable is displayed.

Example

Give Area

Algorithm 1.1

- Write an algorithm to find the sum of three given numbers
  - NAME: SUM3
  - GIVENS: N1, N2, N3
  - RESULTS: Total
  - DEFINITION: Total := SUM3(N1, N2, N3)
  - ----------
  - METHOD:

    Get N1

    Get N2

    Get N3

    Let Total = N1 + N2 + N3

    Give Total

Algorithm 1.2

- Write an algorithm to find the result of a division operation for the given two numbers X and Y
  - NAME: Division
  - GIVENS: X, Y
  - RESULTS: Quotient
  - DEFINITION: Quotient := Division(X, Y)
  - ----------
  - METHOD:

    Get X

    Get Y

    Let Quotient = X/Y

    Give Quotient

Algorithm 1.3

- Write an algorithm to find the sum and product of the two given numbers
  - NAME: SumTimes
  - GIVENS:Num1, Num2
  - RESULTS: Total, Product
  - DEFINITION: Total & Product := SumTimes(Num1, Num2)
  - ----------
  - METHOD:

    Get Num1

    Get Num2

    Let Total = Num1 + Num2

    Let Product = Num1 * Num2

    Give Total

    Give Product

Algorithm 1.4

- Find the sum and average of three given numbers
  - NAME:AVG3
  - GIVENS:Num1, Num2, Num3
  - RESULTS:Sum , Average
  - DEFINITION:Sum & Average := AVG3(Num1, Num2, Num3)
  - ----------
  - METHOD:

    Get Num1

    Get Num2

    Get Num3

    Let Sum = Num1 + Num2 + Num3

    Let Average = Sum /3

    Give Sum

    Give Average

Variables

- Observe that we have used names for the data items in our Givens and Results
  - Num1, Num2, Num3, Sum, Average in Algorithm 1.4
- Each name refers to a unique location in computer memory (one or more adjacent bytes) that contains a value
- Since that value can change as the instructions in our algorithm are executed, we call each data item a variable

Variables

- In our algorithm, when we use a variable name, we are referring to the value stored in memory for that data item
- Later in this lecture we will learn more about how to define variables

Intermediates

- Occasionally, in an algorithm, we need to have a variable (in addition to those representing Givens or Results) to store a value temporarily
- These are intermediate variables and we identify them in the Algorithm Description as **Intermediates**

Algorithm 1.5

- Given 3 assignment marks (out of 50, 20, 70), find the average (calculated as a mark out of 100)
- General Concept
  - How does one figure out the percentage of several marks?
    - Add them all up
    - Divide by the maximum possible mark (50+20+70)
    - Multiply by 100

Algorithm 1.5

- **Given 3 assignment marks (out of 50, 20, 70), find the average, calculated as a mark out of 100**

- NAME: CalcMark
  - GIVENS: A1, A2, A3
  - RESULTS: Mark
  - INTERMEDIATES: Total, MaxMark (Constant)
  - DEFINITION: Mark := CalcMark(A1, A2, A3)
  - ----------
  - METHOD:

    Set MaxMark = 140 (Constant)

    Get A1

    Get A2

    Get A3

    Let Total = A1 + A2 + A3

    Let Mark = Total/MaxMark * 100

    Give Mark

Algorithm 1.6

- Given a two digit number, find the sum of its digits
- General Concept
  - How can we break apart a number?
    - 41 = 4 Tens and 1 Ones
    - so for the number 41, we want 4 + 1 = 5
  - Use integer division
    - DIV returns the integer part of a division
    - MOD returns the remainder of a division

41 / 10 = 4

$$\begin{array}{r} 4 \\ 10\overline{)41} \\ \underline{40} \\ 1 \end{array}$$

41 MOD 10 = 1

Algorithm 1.6

- **Given a two digit number, find the sum of its digits**
  - NAME: SumDig
  - GIVENS: N
  - RESULTS: Sum
  - INTERMEDIATES: Tens, Ones
  - DEFINITION: Sum := SumDig(N)
  - ----------
  - METHOD:

    Get N

    Let Tens = N div10

    Let Ones = N mod 10

    Let Sum = Tens + Ones

    Give Sum

Algorithm 1.7

- **Write an algorithm which swaps the values of two numbers**

- **Example 1**
  - Two car family. The wrong car is at the end of the driveway
    - Move first car out on to the street
    - Move second car out on to the street
    - Move first car back in
    - Move second car back in

- **Example 2**
  - You are looking after a 3 year old. He wants milk and juice. You put the milk in the blue glass and the juice in the red glass. The child is screaming that you did it wrong.
    - Get a third glass (white)
    - Pour the milk from the blue glass to the white glass
    - Pour the juice from the red glass to the blue glass
    - Pour the milk from the white glass to the red glass

Algorithm 1.7

- **Write an algorithm which swaps the values of two numbers**
  - NAME: Swap
  - GIVENS: X, Y
  - RESULTS: X, Y
  - INTERMEDIATES: Temp
  - DEFINITION: Swap (X, Y)
  - ----------
  - METHOD:

    Get X

    Get Y

    Let Temp = X

    Let X = Y

    Let Y = Temp

    Give X

    Give Y

Algorithm 1.8

- **Write an algorithm which adds the given two numbers (X and Y) and returns the sum in the given variable X**
  - NAME: AddXY
  - GIVENS:X, Y
  - RESULTS: X
  - INTERMEDIATES: None
  - DEFINITION: AddXY (X, Y)
  - ----------
  - METHOD:

    Get X

    Get Y

    Let X = X+ Y

    Give X

Recap

Name	The formal name of the algorithm
Givens	Names of the given values for a problem
Results	Names of the resulting values of the problem
Intermediates	Names of the intermediate variables used in the algorithm
Definition	The formal definition of the algorithm, using Name, Givens, and Results
Method	The step by step sequence of statements to solve the problem

Tracing an Algorithm

- The purpose of tracing an algorithm is to ensure that it works
- This is a paper test. It should be completed before writing the computer code
- Tracing involves
  - Executing the sequence of instructions with a sample set of values
  - Computing the value of each variable after each instruction is executed
  - Checking for the correct result

Tracing an Algorithm

- Step 1 - Number every instruction in the algorithm

  Step 2 – Make a table

  - The first column of the table indicates which instruction has been executed
  - Subsequent columns list all the variables of the algorithm (Givens, Results, Intermediates)

Tracing an Algorithm

- Step 3 – Complete the table
  - Each column of the table represents a variable
  - Start at the first line of your algorithm. Identify what will happen to each variable as that instruction is executed
    - Change any values which the instruction changes and leave all other columns blank

Trace 1.1

- **Trace Algorithm 1.4 using the numbers 24, 31, and 35**

METHOD:

1. Get Num1
2. Get Num2
3. Get Num3
4. Let Sum = Num1 + Num2 + Num3
5. Let Average = Sum /3
6. Give Sum
7. Give Average

Line	Num1	Num2	Num3	Sum	Avg
1	24				
2		31			
3			35		
4				90	
5					30
6 output 90					
7 output 30					

Trace 1.2

- **Trace Algorithm 1.5 with the numbers 40, 18, 26**

METHOD:	Ln	A1	A2	A3	MM	Ttl	Mark
(1) Set MaxMark =140	1				140		
(2) Get A1	2	40					
(3) Get A2	3		18				
(4) Get A3	4			26			
(5) Let Total = A1 + A2 + A3	5					84	
(6) Let Mark = Total/MaxMark * 100	6						60
(7) Give Mark	7	output	60				

Trace 1.3

- **Trace Algorithm 1.7 when X and Y have the values 25 and 88, respectively**

METHOD:	LN	X	Y	Temp
(1) Get X	1	25		
(2) Get Y	2		88	
(3) Let Temp = X	3			25
(4) Let X = Y	4	88		
(5) Let Y = Temp	5		25	
(6) Give X	6 output 88			
(7) Give Y	7 output 25			

Data Types

- We know that use of a variable name in an algorithm refers to the value stored at a unique location in computer memory
- Eventually, we will translate the instructions in our algorithm to computer instructions
- Computers need to know they type of each variable

Data Types

- The data type indicates the
  - Kind of data that can be stored in the variable
    - numbers, dates, text, etc.
  - Range of possible values that can be stored in the variable
  - Size of memory required to store the variable
  - Operations that can be performed on the variable

Flow Charts

Flow Charts

- Can be created in MS Visio
- Begin and End with an Oval
- Get/Give use a parallelogram
- Lets use a rectangle
- Flow is shown with arrows

Algorithm 1.1

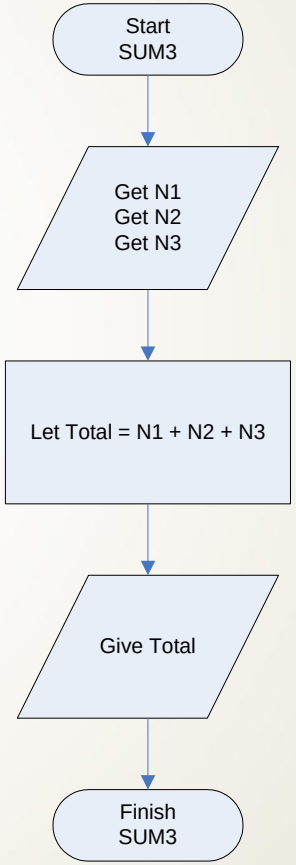
NAME: SUM3

GIVENS: N1, N2, N3

RESULTS: Total

DEFINITION:

Total := SUM3(N1, N2, N3)



Algorithm 1.2

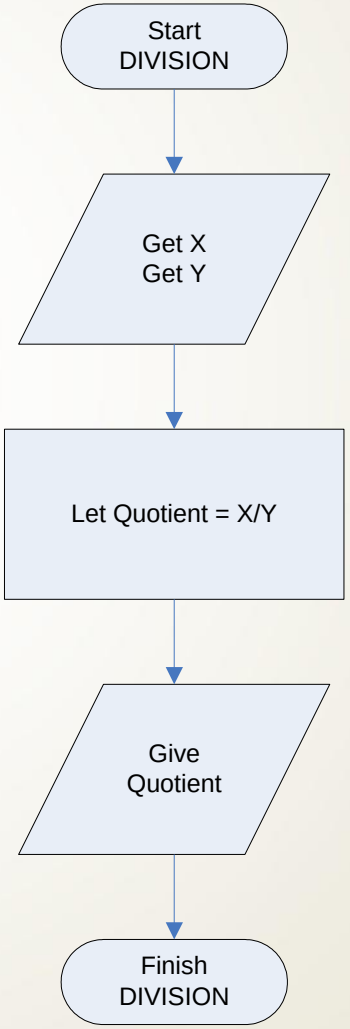
NAME: Division

GIVENS: X, Y

RESULTS: Quotient

DEFINITION:

Quotient := Division(X, Y)



Algorithm 1.3

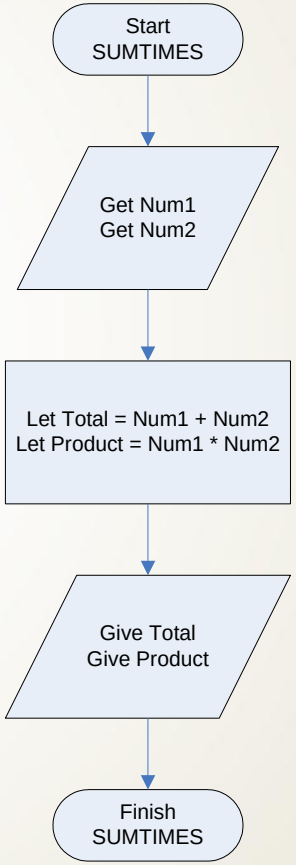
NAME: SumTimes

GIVENS:Num1, Num2

RESULTS: Total, Product

DEFINITION:

Total & Product :=
SumTimes(Num1, Num2)



Algorithm 1.4

NAME:AVG3

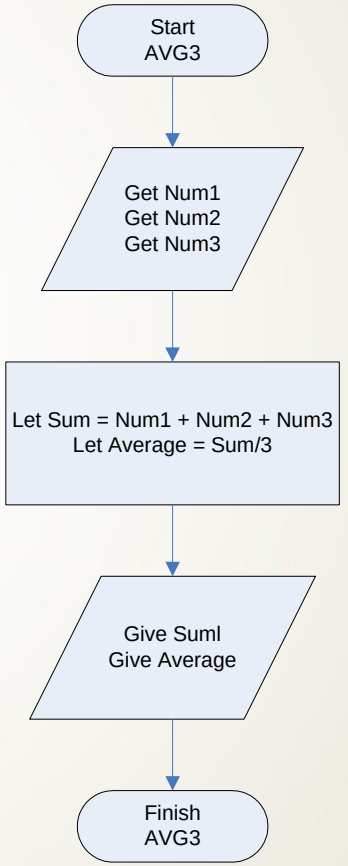
GIVENS:Num1, Num2, Num3

RESULTS:Sum , Average

DEFINITION:

Sum & Average :=

AVG3(Num1, Num2, Num3)



Algorithm 1.5

NAME: CalcMark

GIVENS: A1, A2, A3

RESULTS: Mark

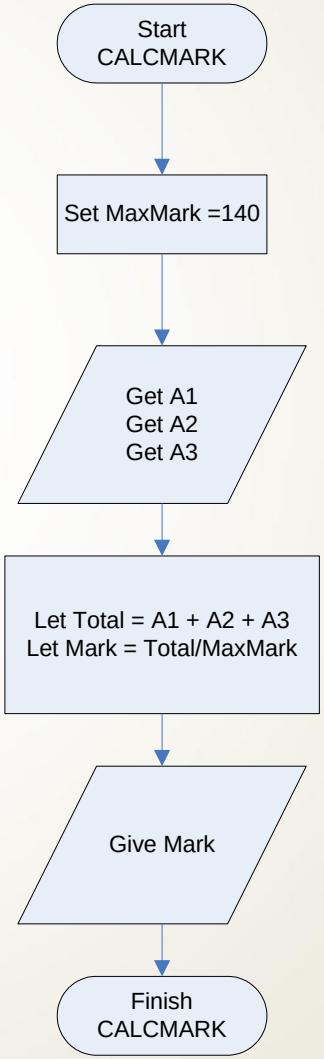
INTERMEDIATES:

Total,

MaxMark (Constant)

DEFINITION:

Mark := CalcMark(A1, A2, A3)



Algorithm 1.6

NAME: SumDig

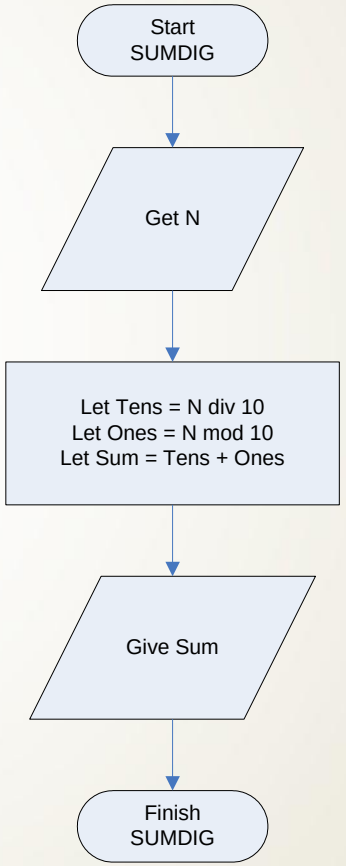
GIVENS: N

RESULTS: Sum

INTERMEDIATES: Tens, Ones

DEFINITION:

Sum := SumDig(N)



Algorithm 1.7

NAME: Swap

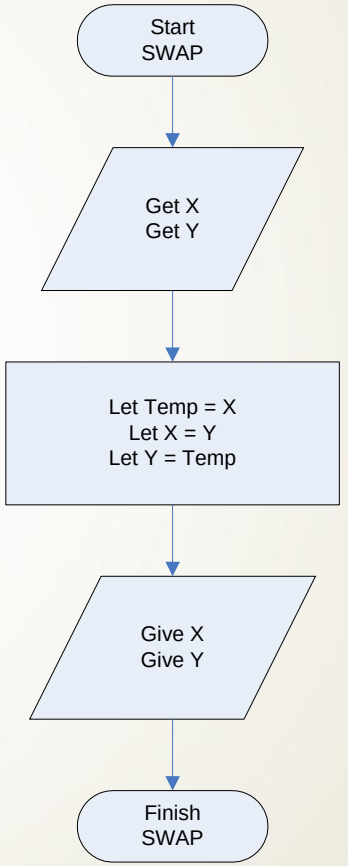
GIVENS: X, Y

Change: X, Y

RESULTS: None

INTERMEDIATES: Temp

DEFINITION: Swap (X, Y)



Algorithm 1.8

NAME: AddXY

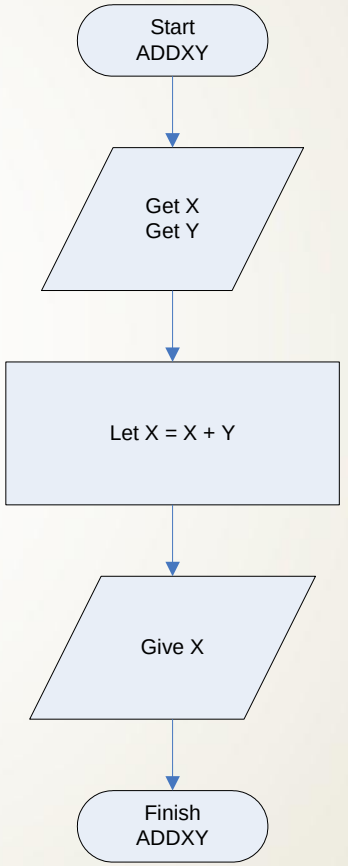
GIVENS:X, Y

Change: X

RESULTS:None

INTERMEDIATES: None

DEFINITION: AddXY (X, Y)



Exercise:

1. Write an algorithm to sum 5 numbers.
   - Trace the algorithm using the numbers 1, 3, 5, 7 and 9.
2. Write an algorithm to compute the average of 5 numbers.
   - Trace the algorithm using the numbers 1, 3, 5, 7 and 9.
3. Write an algorithm to multiply 3 numbers.
   - Trace the algorithm using the numbers 2, 9 and 6.
4. Write an algorithm to multiply 2 numbers and divide the product by a third number.
   - Trace the algorithm using the numbers 2, 9 and 6.

Exercise:

1. Write an algorithm to calculate the average mark out of 100, given three assignment marks, each of which is marked out of 20.
   - Trace the algorithm using 15, 20 and 10 as the three assignment marks.
2. Write an algorithm to calculate the sum of the digits of a three digit number.
   - Trace the algorithm using the number 659.
3. Write an algorithm to swap two numbers.
   - Trace the algorithm using x as 8 and y as 9.
4. Write an algorithm to sum four numbers and return the sum in the variable x.
   - Trace the algorithm using w as 2, x as 3, y as 4 and z as 5.

Summary

- Binary search **reduces the work by half** at each comparison
- If array is not sorted ➔ Linear Search
  - Best Case O(1)
  - Worst Case O(N)
- If array is sorted ➔ Binary search
  - Best Case O(1)
  - Worst Case O(Log_2N)

Searching

- Given a collection and an element (key) to find…
- Output
  - Print a message (“Found”, “Not Found)
  - Return a value (position of key)
- Don’t modify the collection in the search!

Searching Example
(Linear Search)

Linear Search: A Simple Search

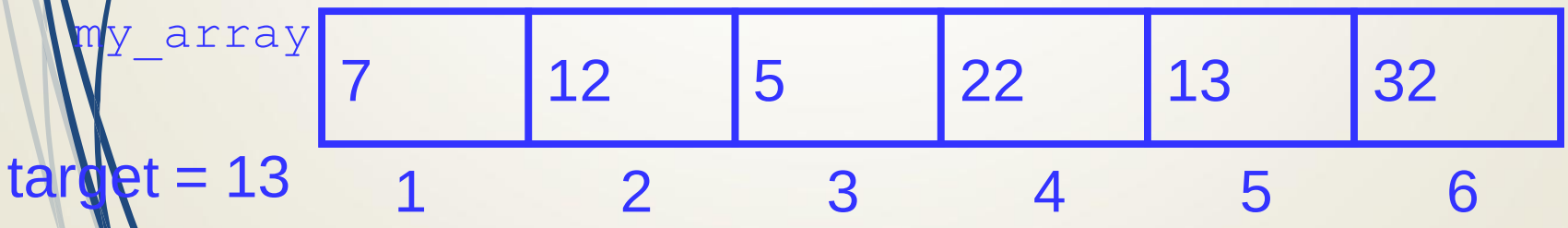
- A search traverses the collection until
  - The desired element is found
  - Or the collection is exhausted
- If the collection is ordered, I might not have to look at all elements
  - I can stop looking when I know the element cannot be in the collection.

Un-Ordered Iterative Array Search

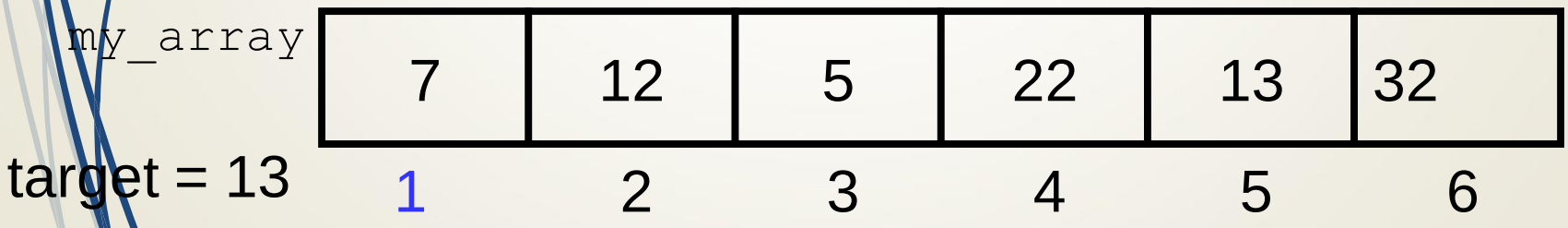
```
procedure Search(my_array Array,
                target Num)
  i Num
  i <- 1
  loop
    exitif((i > MAX) OR (my_array[i] = target))
    i <- i + 1
  endloop

  if(i > MAX) then
    print("Target data not found")
  else
    print("Target data found")
  endif
endprocedure // Search
```

Scan the array

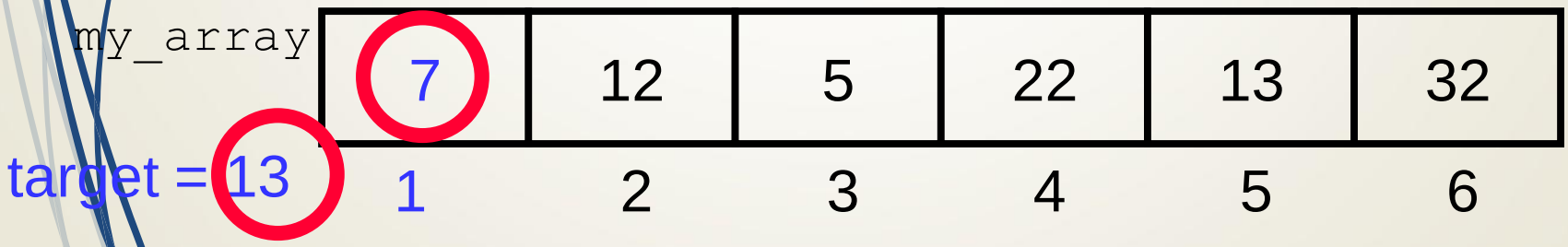
```
procedure Search(my_array Array,
                target Num)
  i isoftype Num
  i <- 1
  loop
    exitif((i > MAX) OR (my_array[i] = target))
    i <- i + 1
  endloop

  if(i > MAX) then
    print("Target data not found")
  else
    print("Target data found")
  endif
endprocedure // Search
```



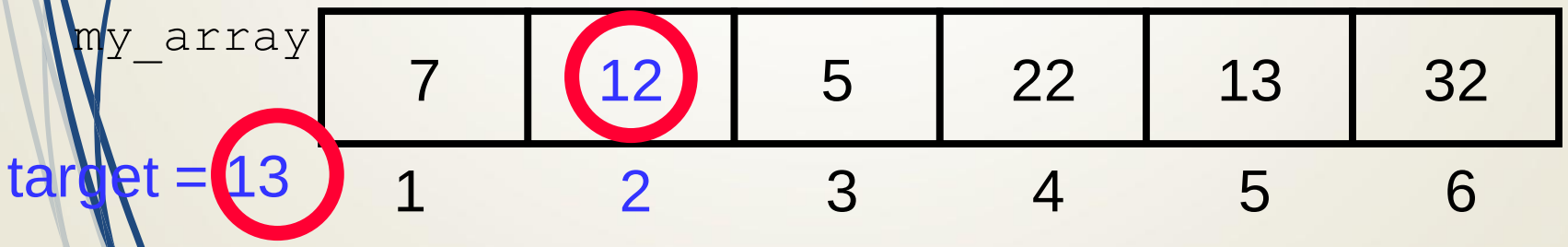
```
procedure Search(my_array isoftype in NumArrayType,
                target isoftype in Num)
  i isoftype Num
  i <- 1
  loop
    exitif((i > MAX) OR (my_array[i] = target))
    i <- i + 1
  endloop

  if(i > MAX) then
    print("Target data not found")
  else
    print("Target data found")
  endif
endprocedure // Search
```



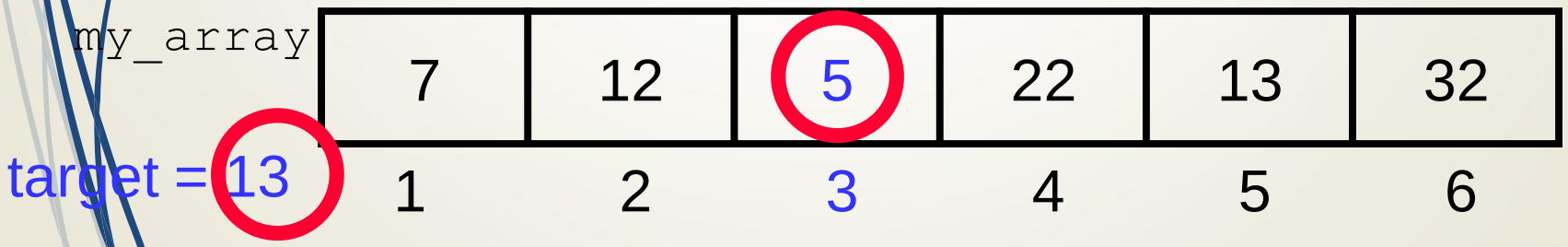
```
procedure Search(my_array isoftype in NumArrayType,
                target isoftype in Num)
   i isoftype Num
   i <- 1
   loop
     exitif((i > MAX) OR (my_array[i] = target))
     i <- i + 1
   endloop

   if(i > MAX) then
     print("Target data not found")
   else
     print("Target data found")
   endif
endprocedure // Search
```



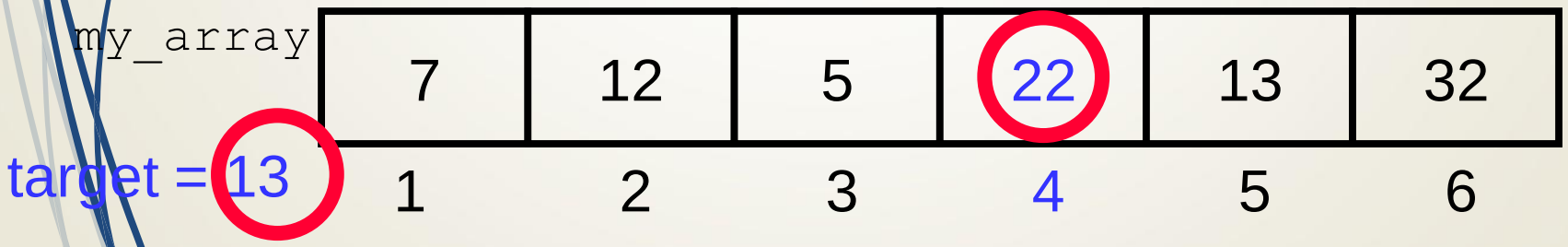
```
procedure Search(my_array isoftype in NumArrayType,
                target isoftype in Num)
   i isoftype Num
   i <- 1
   loop
     exitif((i > MAX) OR (my_array[i] = target))
     i <- i + 1
   endloop

   if(i > MAX) then
     print("Target data not found")
   else
     print("Target data found")
   endif
endprocedure // Search
```



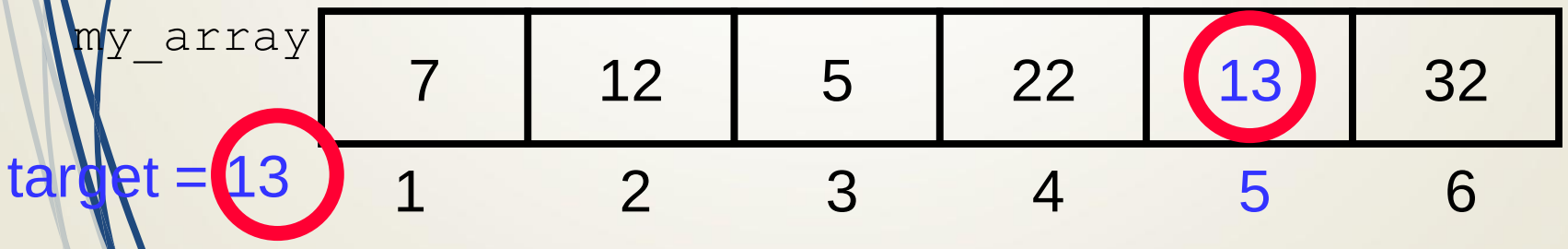
```
procedure Search(my_array isoftype in NumArrayType,
                target isoftype in Num)
   i isoftype Num
   i <- 1
   loop
     exitif((i > MAX) OR (my_array[i] = target))
     i <- i + 1
   endloop

   if(i > MAX) then
     print("Target data not found")
   else
     print("Target data found")
   endif
endprocedure // Search
```



```
procedure Search(my_array isoftype in NumArrayType,
                target isoftype in Num)
   i isoftype Num
   i <- 1
   loop
     exitif((i > MAX) OR (my_array[i] = target))
     i <- i + 1
   endloop

   if(i > MAX) then
     print("Target data not found")
   else
     print("Target data found")
   endif
endprocedure // Search
```



```
procedure Search(my_array isoftype in NumArrayType,
                target isoftype in Num)
   i isoftype Num
   i <- 1
   loop
     exitif((i > MAX) OR (my_array[i] = target))
     i <- i + 1
   endloop

   if(i > MAX) then
     print("Target data not found")
   else
     print("Target data found")
   endif
endprocedure // Search
```



```
procedure Search(my_array isoftype in NumArrayType,
                target isoftype in Num)
   i isoftype
   i <- 1
   loop
     exitif((
     i <- i +
   endloop

   if(i > MAX) then
     print("Target data not found")
   else
     print("Target data found")
   endif
endprocedure // Search
```

Target data found

rget))

my_array	7	12	5	22	13	32
	1	2	3	4	5	6

target = 13

Linear Search Analysis: Best Case

```
procedure Search(my_array Array,
                target Num)
  i Num
  i <- 1
  loop
    exitif((i > MAX) OR (my_array[i] = target))
    i <- i + 1
  endloop

  if(i > MAX) then
    print("Target data not found")
  else
    print("Target data found")
  endif
endprocedure // Search
```

Scan the array

Best Case:
1 comparison

Best Case: match with the first item

Linear Search Analysis: Worst Case

```
procedure Search(my_array Array,
                target Num)
  i Num
  i <- 1
  loop
    exitif((i > MAX) OR (my_array[i] = target))
    i <- i + 1
  endloop

  if(i > MAX) then
    print("Target data not found")
  else
    print("Target data found")
  endif
endprocedure // Search
```

Scan the array

Worst Case:
N comparisons

Worst Case: **match with the last item (or no match)**

Searching Example
(Binary Search on Sorted List)

The Scenario

- We have a **sorted array**
- We want to determine if a **particular element** is in the array
  - Once **found**, print or return (index, boolean, etc.)
  - If **not found**, indicate the element is not in the collection

7	12	42	59	71	86	104	212

A Better Search Algorithm

- Of course we could use our simpler search and traverse the array
- But we can use the fact that the array is sorted to our advantage
- This will allow us to reduce the number of comparisons

Binary Search

- Requires a **sorted array** or a ***binary search tree.***
- Cuts the "search space" **in half** each time.
- Keeps cutting the search space in half until the **target is found** or has **exhausted the all possible locations.**

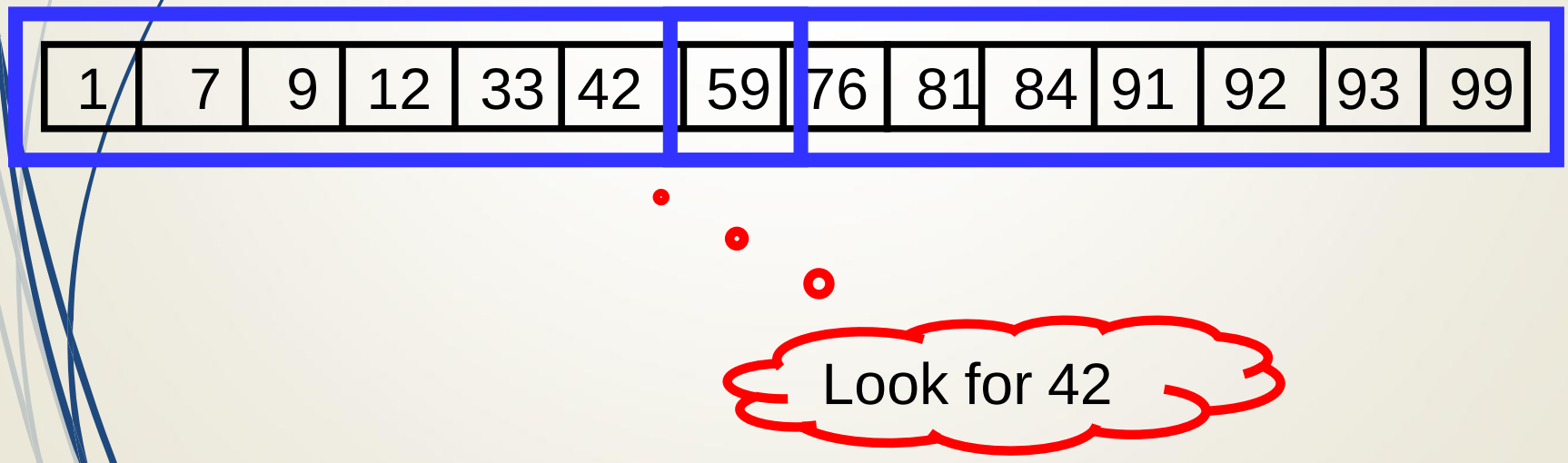
Binary Search Algorithm

look at "middle" element

if no match then

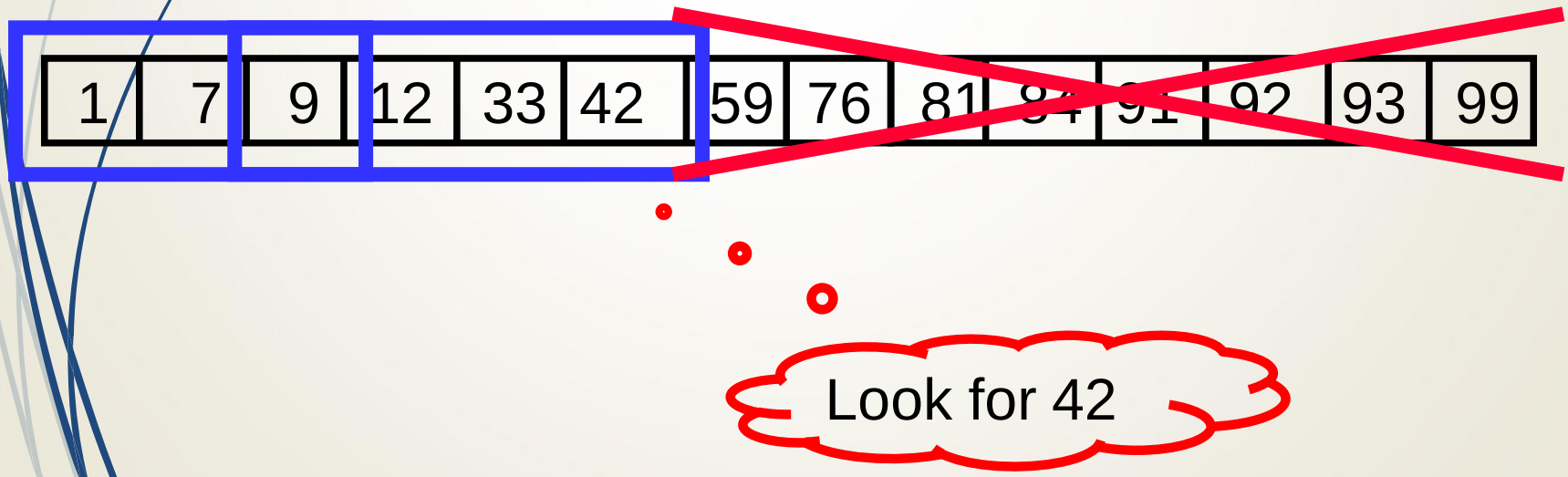
look *left* (if need smaller) or

right (if need larger)



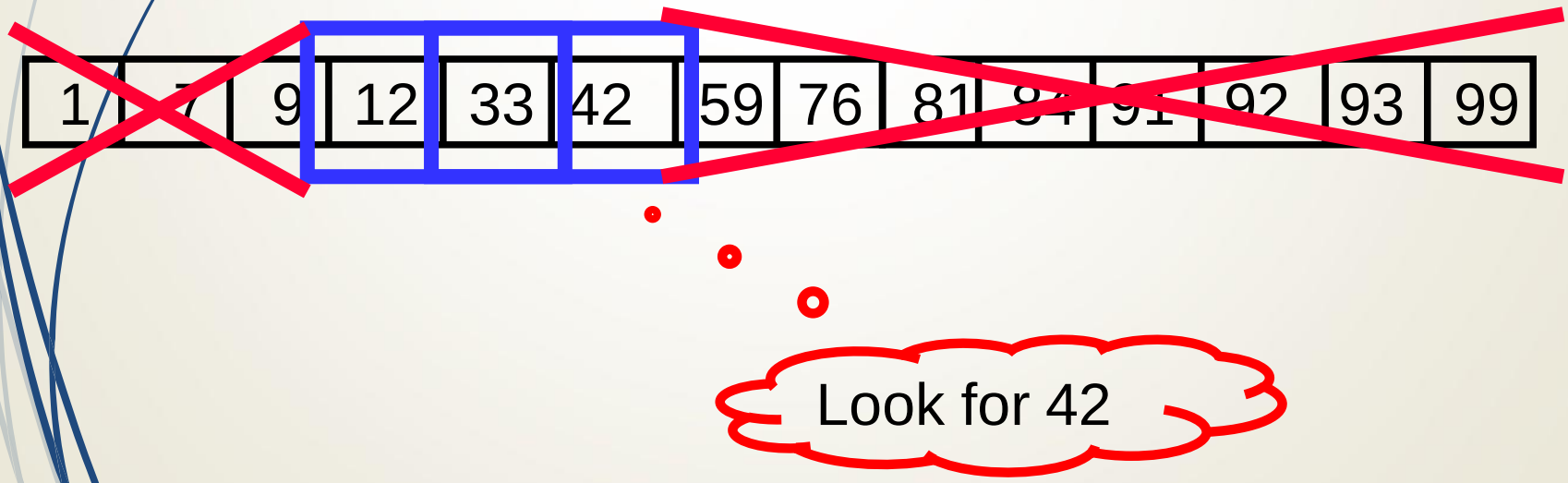
The Algorithm

```
look at "middle" element
if no match then
   look left or right
```



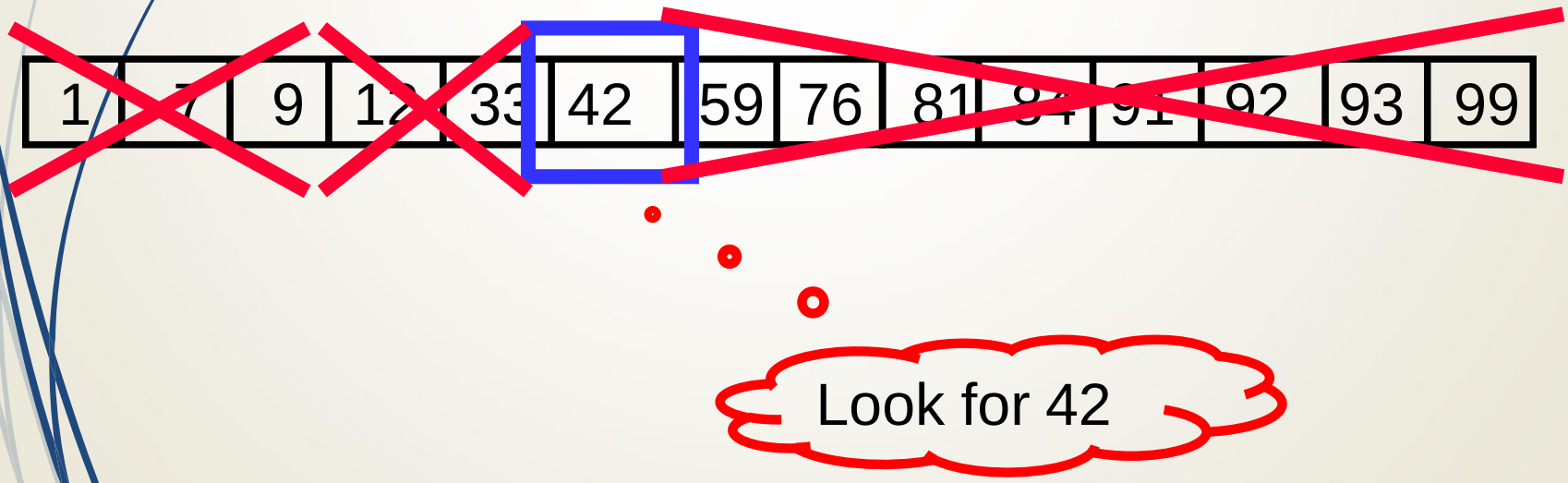
The Algorithm

```
look at "middle" element
if no match then
   look left or right
```



The Algorithm

```
look at "middle" element
if no match then
  look left or right
```



The Binary Search Algorithm

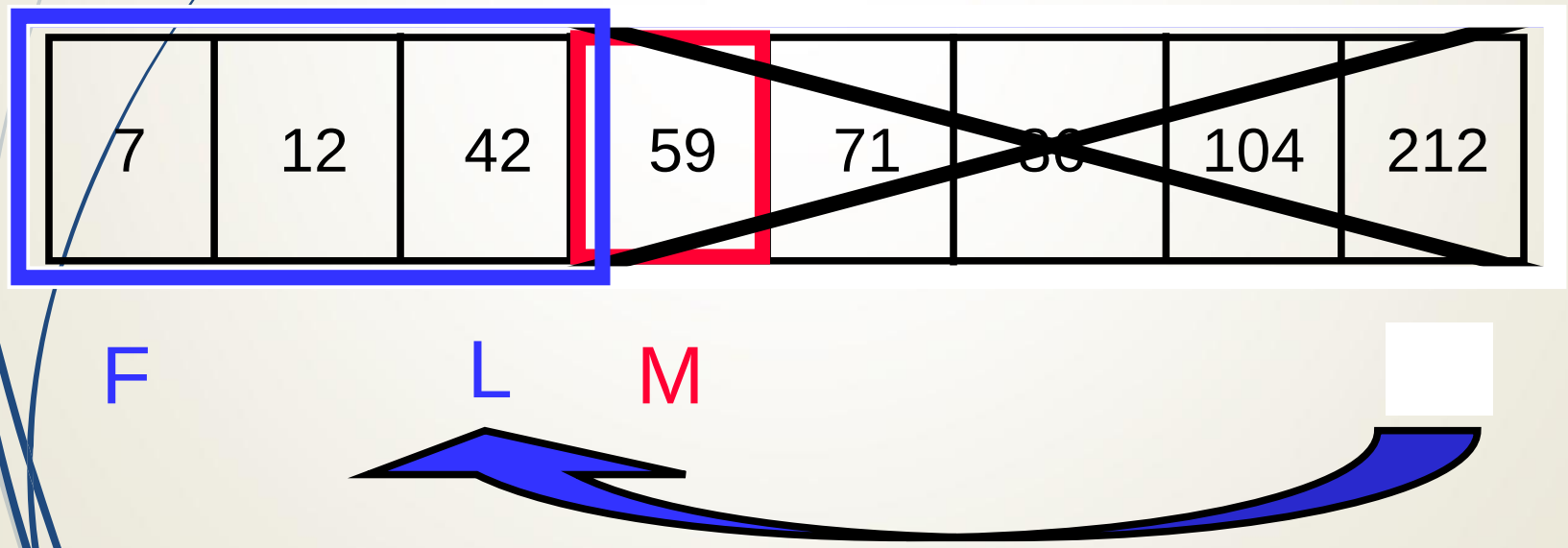
- **Return found or not found (true or false), so it should be a function.**
- **When move *left* or *right*, change the array boundaries**
  - **We'll need a first and last**

The Binary Search Algorithm

```
calculate middle position
if (first and last have "crossed") then
  "Item not found"
elseif (element at middle = to_find) then
  "Item Found"
elseif to_find < element at middle then
  Look to the left
else
  Look to the right
```

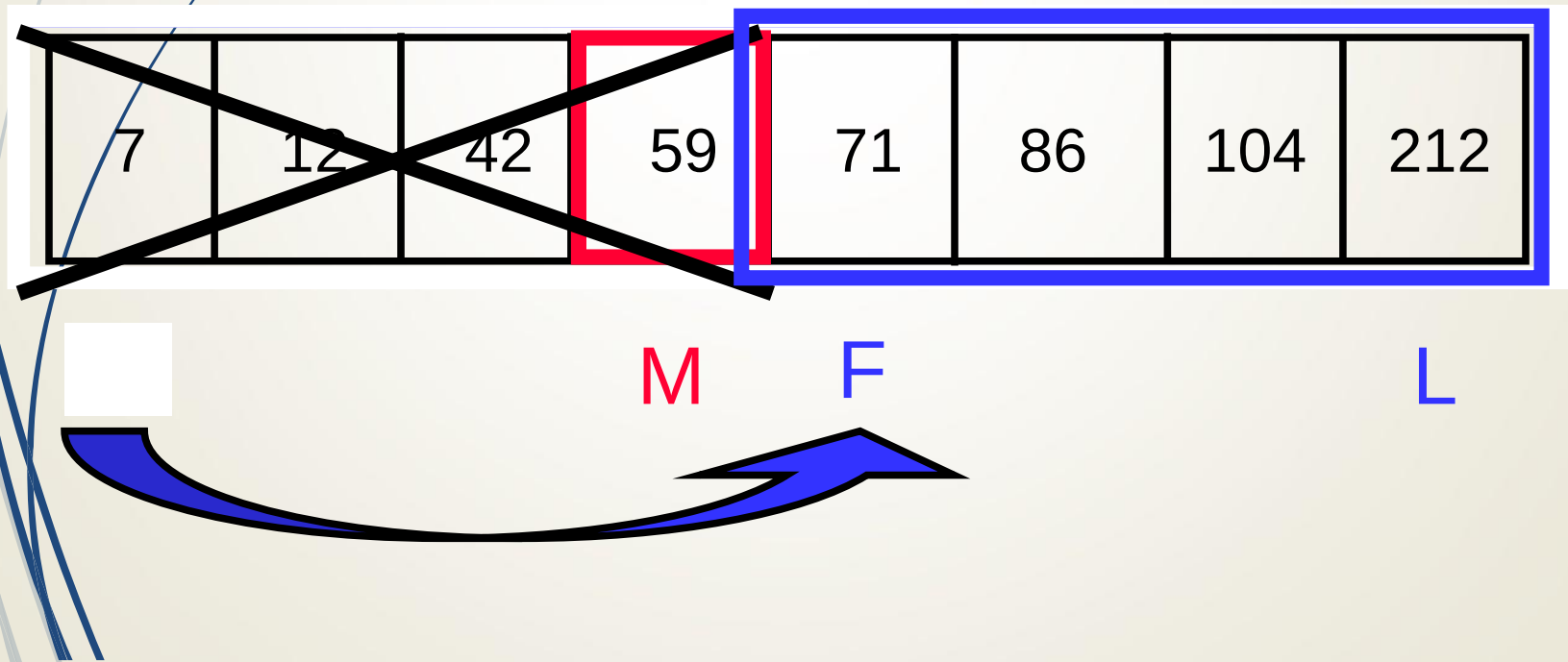
Looking Left

- Use indices "first" and "last" to keep track of where we are looking
- Move left by setting last = middle – 1

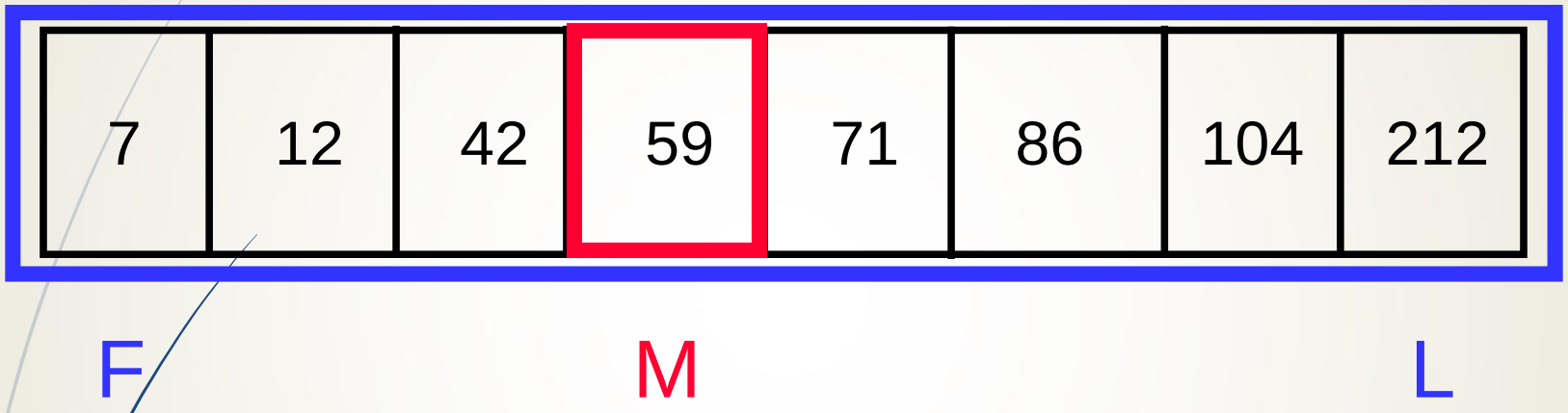


Looking Right

- Use indices "first" and "last" to keep track of where we are looking
- Move right by setting first = middle + 1

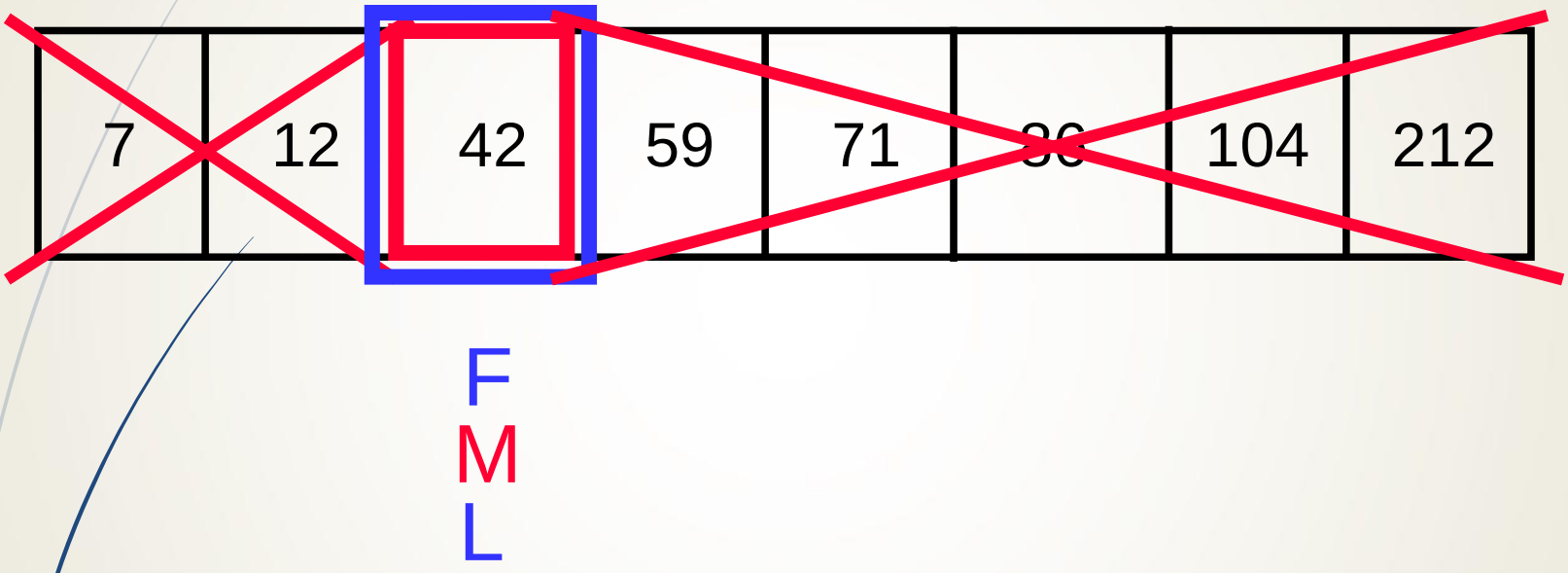


Binary Search Example – Found



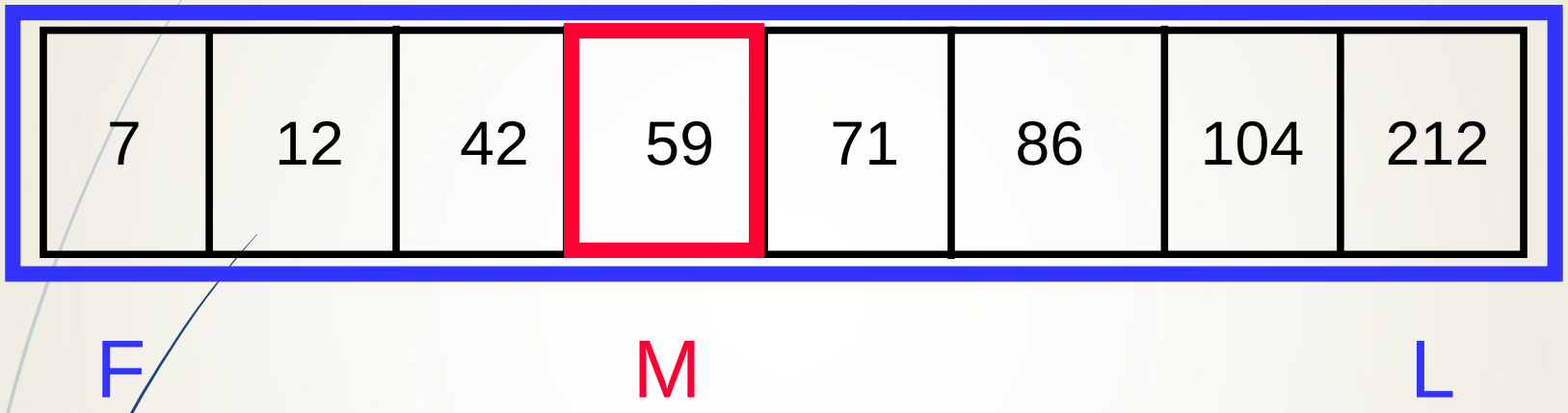
Looking for 42

Binary Search Example – Found



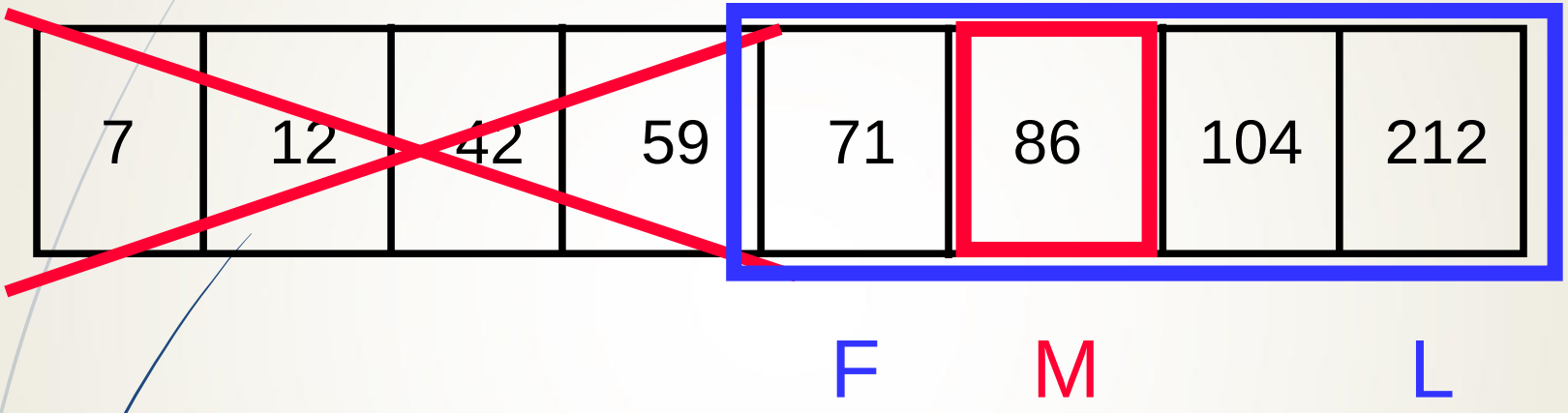
42 found – in 3 comparisons

Binary Search Example – Not Found



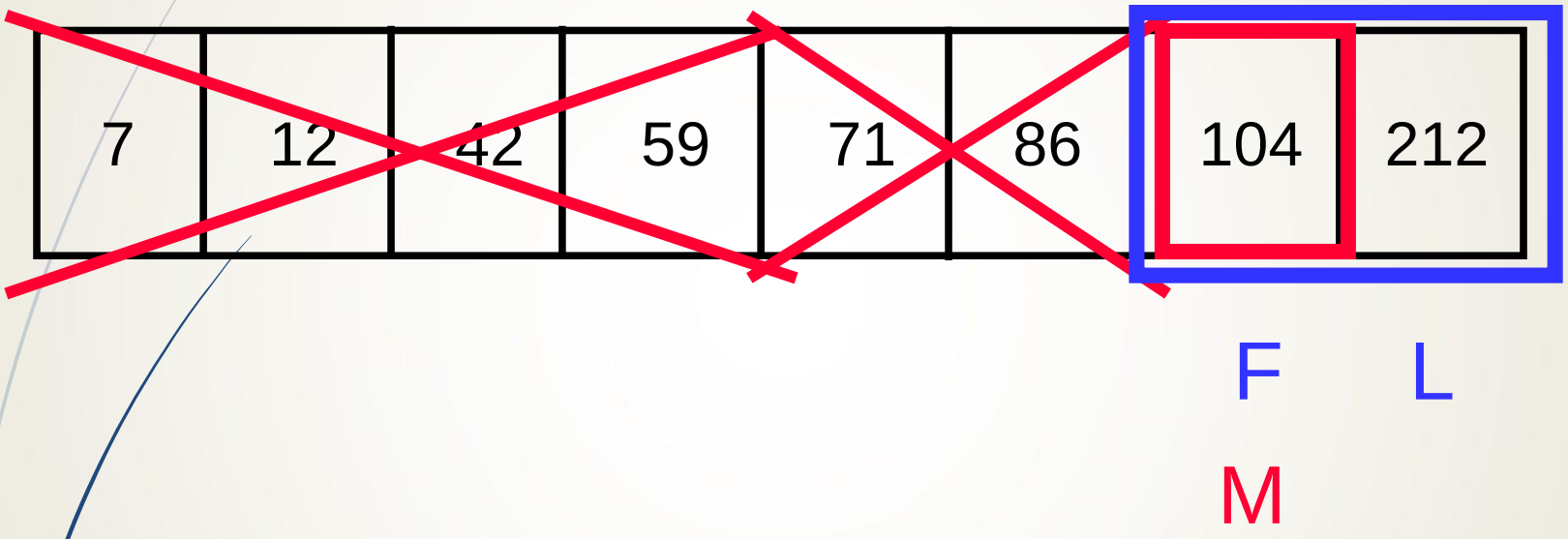
Looking for 89

Binary Search Example – Not Found



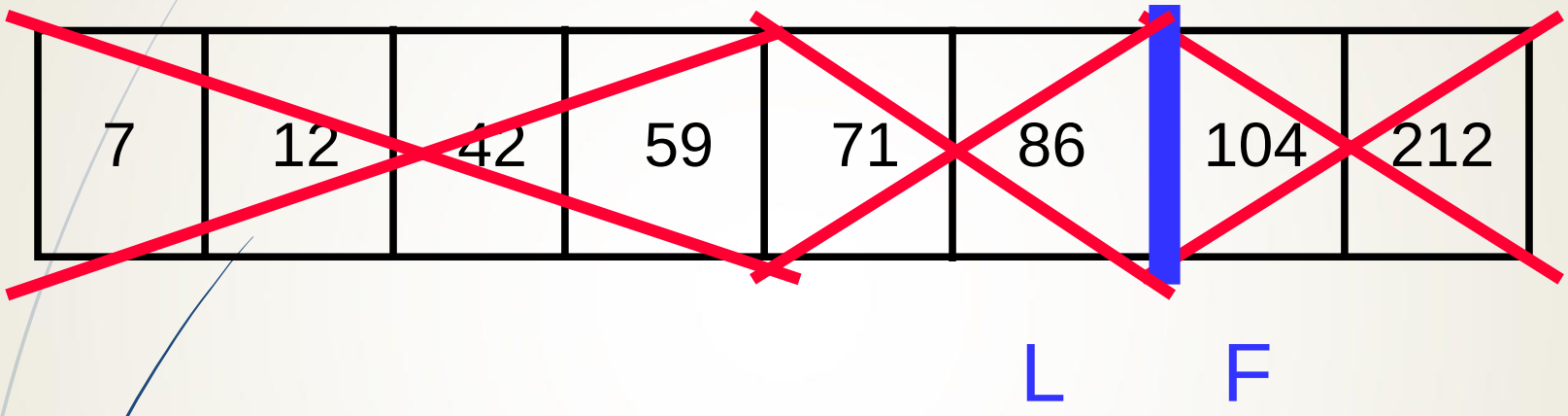
Looking for 89

Binary Search Example – Not Found



Looking for 89

Binary Search Example – Not Found



89 not found – 3 comparisons

Binary Search Function

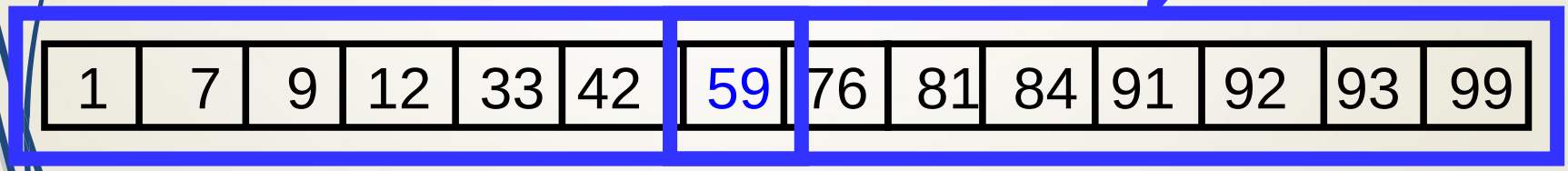
```
Function Find return boolean (A Array, first, last, to_find)

  middle <- (first + last) div 2

  if (first > last) then
    return false
  elseif (A[middle] = to_find) then
    return true
  elseif (to_find < A[middle]) then
     return Find(A, first, middle-1, to_find)
  else
    return Find(A, middle+1, last, to_find)
endfunction
```

Binary Search Analysis: Best Case

```
Function Find return boolean (A Array, first, last, to_find)
  middle <- (first + last) div 2
  if (first > last) then
    return false
  elseif (A[middle] = to_find) then
    return true
  elseif (to_find < A[middle]) then
     return Find(A, first, middle-1, to_find)
  else
    return Find(A, middle+1, last, to_find)
endfunction
```

Best Case:
1 comparison

Best Case: match from the firs comparison



Target: 59

Binary Search Analysis: Worst Case

```
Function Find return boolean (A Array, first, last, to_find)
  middle <- (first + last) div 2
  if (first > last) then
    return false
  elseif (A[middle] = to_find) then
    return true
  elseif (to_find < A[middle]) then
     return Find(A, first, middle-1, to_find)
  else
    return Find(A, middle+1, last, to_find)
endfunction
```

How many comparisons??

Worst Case: divide until reach one item, or no match.