LIÈGE université
GIGA

INTRODUCTION TO ALGORITHMS

GIGA Doctoral School

Introduction to Scientific Computing

Outline

- Introduction
- Types of algorithms
- Classification of algorithms
- Expressing algorithms
- Constructs of an algorithm
- The concept of sub-algorithm
- Examples
- Algorithm complexity

Introduction

Definition:

- An algorithm is step-by-step procedure with the aim of solving a problem.
- Algorithms are often used in many real life problems
- In computer science, an algorithm has a special meaning. It is defined to have these features:
  - An algorithm must have some data to operate on it
  - It must produce at least one result
  - It must terminate after a finite numbers of steps

Introduction

History:

- History of algorithms can be traced back to the ancient Greeks
- An efficient method for finding the Greatest Common Divisor was proposed by Euclid
- Study of algorithm was done by Mohammed ibn mussa al-Khowarizmi

Types of Algorithms

The types of algorithms depends on the type of **task** to be solved.

- ***Searching***
  - Designed to search for a given item in large data set
- ***Sorting***
  - Used to arrange data items in ascending or descending order
- ***Compression***
  - Meant to reduce the size of data and program files
  - Commonly used for compression of images, audio and video data

Types of Algorithms

- ***Fast Fourier Transforms***
  - Used in Digital Signal Processing (DSP)
- ***Encoding***
  - Used for encryption of data
- ***Geometric***
  - Used for identification of geometric shapes
- ***Pattern Matching***
  - Comparing images and shapes

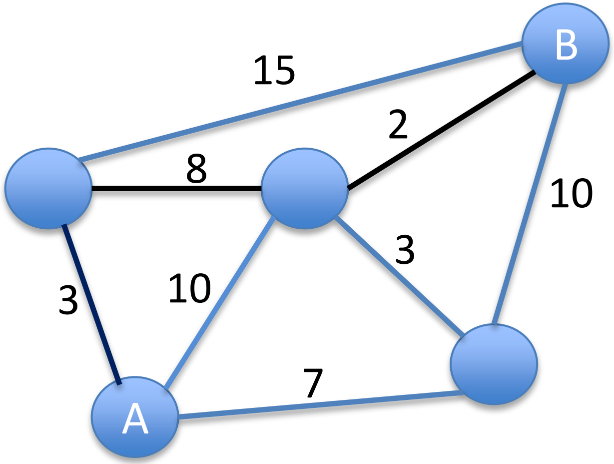
Classification of Algorithms

Depending on the **strategy** used for **solving** a particular problem, algorithms are classified as follows:

- Divide-and-Conquer Algorithms
  - A given problem is fragmented into sub-problems which are solved partially
  - The algorithm is stopped when further sub-division cannot be performed
  - These algorithms are frequently used in searching and sorting problems

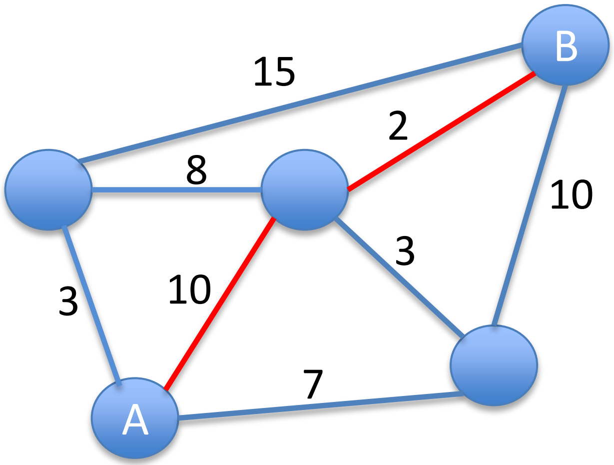
Classification of Algorithms

- Iterative Algorithms
  - Certain steps are repeated in loops, until the goal is achieved
  - An example of an iterative algorithm is sorting an array
- Greedy Algorithms
  - In a Greedy algorithm an immediately available best solution at each step is chosen
  - Useful for solving graph theory



Classification of Algorithms

- Back-Tracking Algorithms
  - In back tracking algorithms, all possible solutions are explored until the end is reached, afterwards the steps are traced back
  - These are useful in graph theory.
  - Back tracking algorithms are used frequently for traversing trees



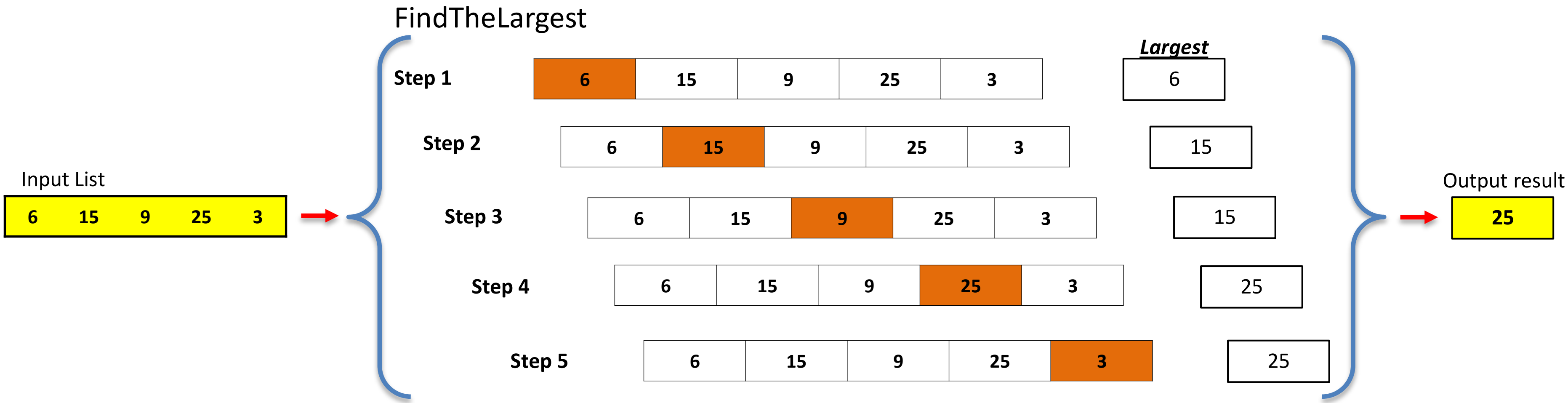
Expressing Algorithms

- Describing algorithms requires a **notation** for expressing a **sequence** of steps to be performed.
- Algorithms can be expressed in many kinds of notation, including **natural languages**, **pseudocode**, **flowcharts**

Natural Language

- English words and sentences can be used to express statements and processing steps
  - For example, words like read, write, compute and set can be used for Input-Output operations, computations and assigning values to variables.
  - Comparison operations are expressed as equal to, less than, greater than
  - Arithmetical operations are expressed using words like add, subtract, divide and multiply
  - Control structures are expressed using sentences like repeat for, while, if, halt, exit
- Example: Find the largest element in a list/array of five integers.

What you would do?



What does it mean in natural language?

Input List

6	15	9	25	3

FindTheLargest

Step 1: Set Largest to the first number.

Step 2: If the second number is greater than Largest, set Largest to the second number.

Step 3: If the third number is greater than Largest, set Largest to the third number

Step 4: If the fourth number is greater than Largest, set Largest to the fourth number

Step 5: If the fifth number is greater than Largest, set Largest to the fifth number

Output result

25

Could you express it in a more simple way?

FindTheLargest

Step 0: Set Largest to 0

Step 1: If the current number is greater than Largest, set Largest to the current number.

…

…

Step 5: If the current number is greater than Largest, set Largest to the current number

Input List

6	**15**	**9**	**25**	**3**

Output result

25

FindTheLargest

Input List

Set Largest to 0.

Repeat the following N times:

If the current number is greater than Largest, set Largest to the current number.

Output result

Input/read: list of N integers

Set Largest to 0

Repeat the following N times

If the current number is greater than Largest, Set Largest to the current number

Output Largest

End

Expressing Algorithms

Use of Pseudocode

- Algorithms in natural language tend to be wordy and verbose
- Pseudocode provides an alternative way of expressing algorithms
- It is a mixture of natural language and programming notation
- In practice several conventions are used to write pseudocode

```
Input/read: list of N integers
Set Largest to 0
Repeat the following N times
If the current number is greater than Largest,
        Set Largest to the current number
Output Largest
End
```

Expressing Algorithms

Use of Pseudocode

- Algorithm is identified by a name
- Comments are enclosed in square brackets
- Assignment statement is coded using left arrow
- Operators : (+, -, *, /, <, >, =, !=)
- Input and Output : read and write
- Control Structures : if-then, if-then-else
- Repetitive operations : Repeat, for, while, until

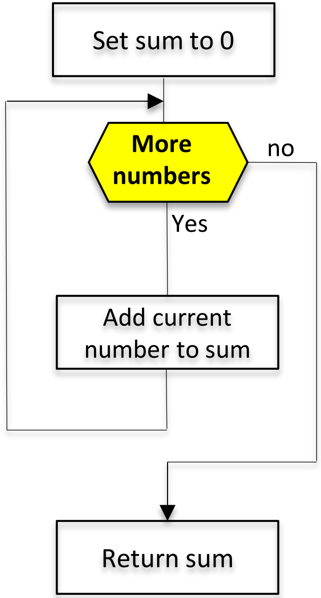
FindTheLargest

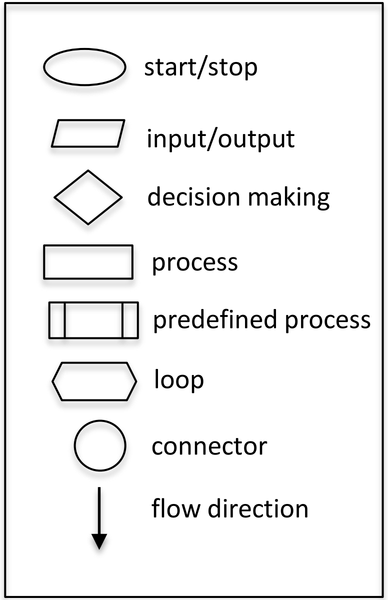
Input: A list of positive integers

1. Set Largest to 0
2. while (more integers)
3. if (the current integer is greater than Largest)
4. then
5. Set Largest to the value of the current integer
6. end if
7. End while
8. Return Largest
9. End

Expressing Algorithms

Flowchart





Flowchart Rules:

1. Flowchart is generally drawn from **top** to **bottom**
2. All boxes of flowchart must be **connected**
3. All flowchart start with **terminal** or **process symbol**
4. Decision symbol have **2 exit points**, one for YES (TRUE) and another for NO (FALSE)

Constructs of an algorithm

FindTheLargest

Input: A list of positive integers

1. Set Largest to 0
2. while (more integers)
3. if (the current integer is greater than Largest)
4. then
5. Set Largest to the value of the current integer
6. end if
7. End while
8. Return Largest
9. End

Sequence

do action 1
do action 2
…
…
…
do action n

Decision

if a condition is true.
Then
do a series of actions
Else
do a series of actions

Repetition

While a condition is true.

do action 2
…
…
…
do action n

Constructs of an algorithm

Constructs & pseudocode

```
action 1
action 2
…
…
…
action n
```

Sequence

```
If (condition)
        then
              action
              action
              …
        else
              action
              action
              …
End if
```

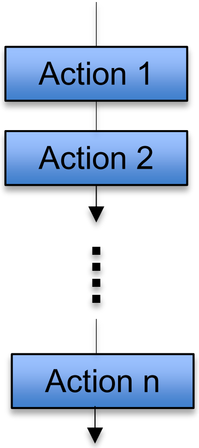
Decision

```
While (condition)
        action
        action
        …
End while
```

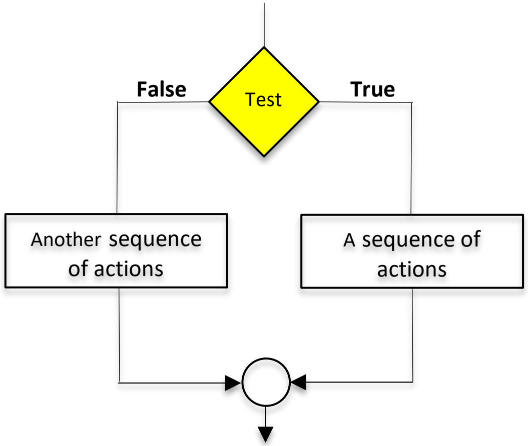
Repetition

Constructs of an algorithm

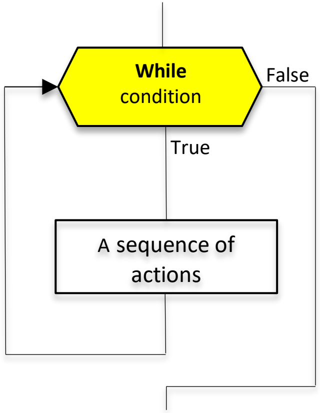
Constructs & Flowcharts



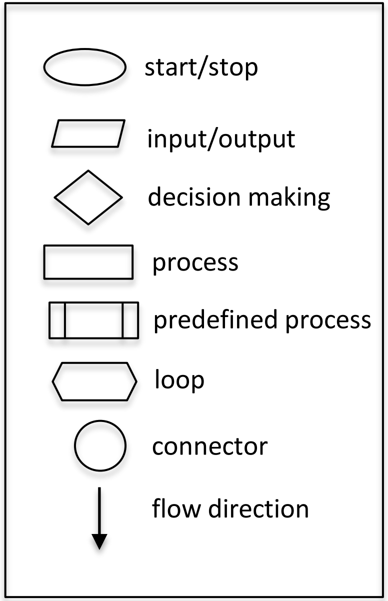
Sequence



Decision



Repetition



The concept of sub-algorithm

FindTheLargest

Input: A list of positive integers

1. Set Largest to 0
2. while (more integers)

    2.1 FindLarger

    End while

3. Return Largest

End

FindLarger

Input: Largest and integer

if (integer greater than Largest)

then

1.1 Set Largest to the value of the integer

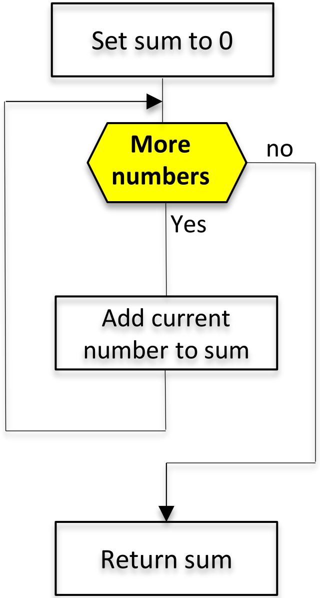
End if

End

Examples of algorithms

Summation/Multiplication

Flowchart



Pseudocode

Summation

Input: A list of integers

1. Set Sum to 0
2. While(more integers)

    2.1. Add current number to sum

    End of while
3. Return Sum

End

Pseudocode

Multiplication

Input: A list of integers

1. Set product to 1
2. While(more integers)

    2.1. Multiply current number by product

    End of while
3. Return product

End

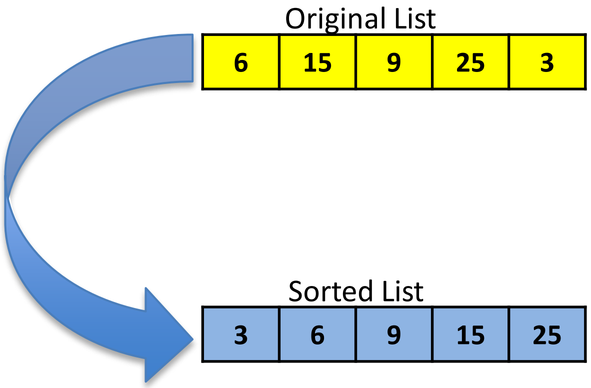
Examples of algorithms

Sorting algorithms

- Given a list, put it into some order

  Input: sequence $(a_1, a_2, \ldots, a_n)$ of numbers.

  Output: permutation $(a'_1, a'_2, \ldots, a'_n)$ such that $a'_1 \leq a'_2, \leq \ldots \leq a'_n$.



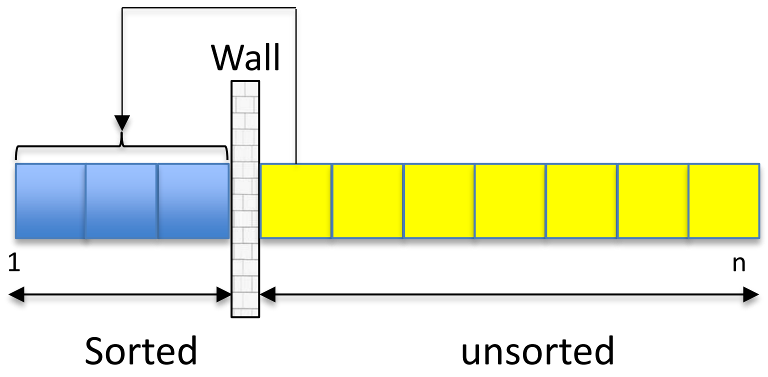
- We will see three types
  - Insertion sort
  - Selection sort
  - Bubble sort

Examples of algorithms

Sorting algorithms

Insertion-Sort

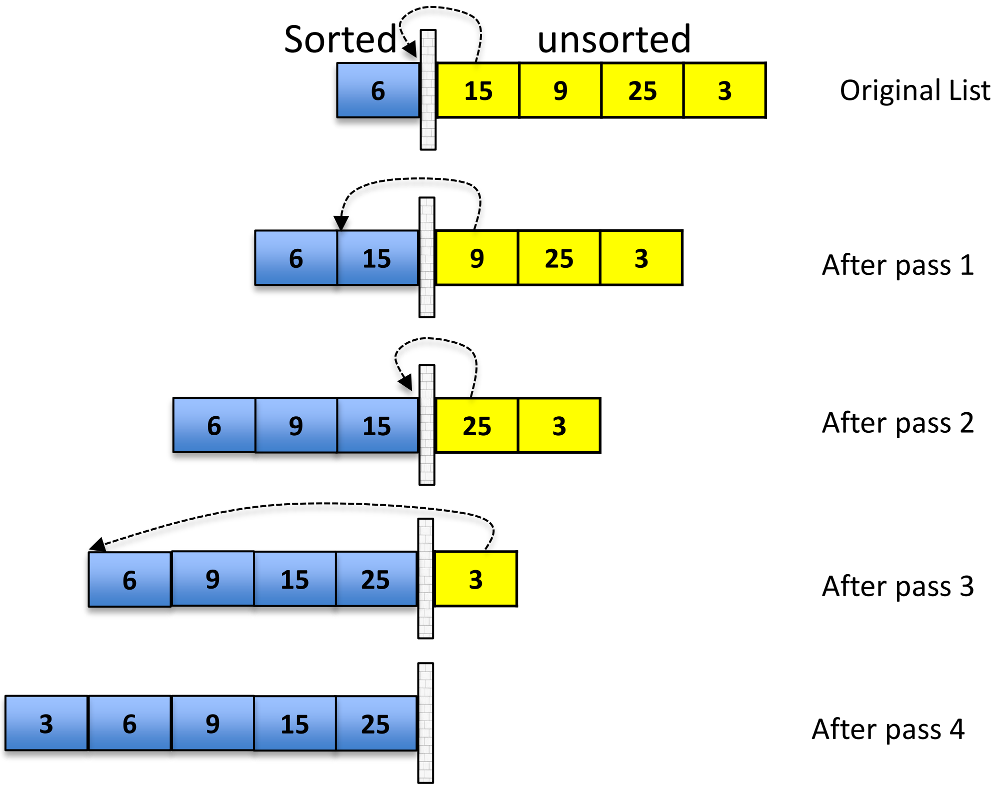
- It starts with a list with one element, and inserts new elements into their proper place in the sorted part of the list.



Examples of algorithms

Sorting algorithms

Insertion-Sort



Examples of algorithms

Sorting algorithms

Pseudocode

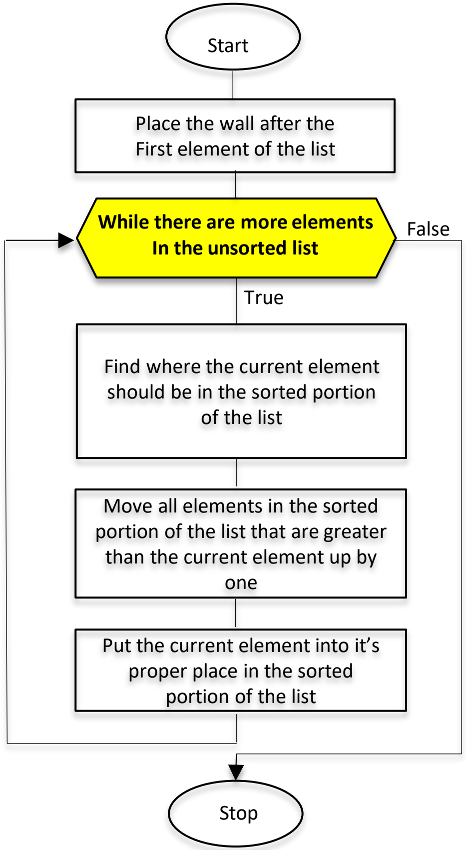
Insertion-Sort

Input: A list of integers (a_1, a_2,…, a_n)

```
for j = 2 to A.length
    value = A[j]
    Insert A[j] into the sorted sequence A[1 . . j-1]
    i = j-1
    While( i > 0 and A[i] > value)
        A[i+1] = A[i]
        i = i -1
    End of while
    A[i+1] = value
End of for
```

End (a'_1, a'_2, …, a'_n) are sorted

Flowchart

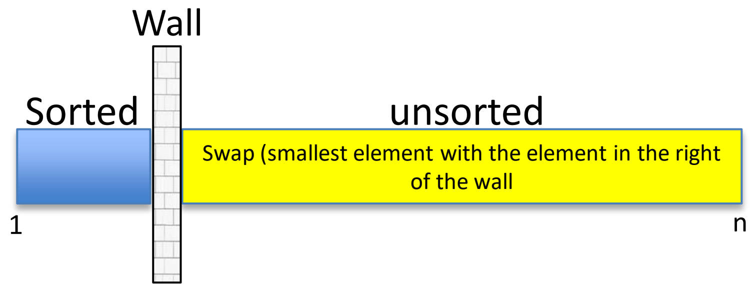


Examples of algorithms

Sorting algorithms

Selection-Sort

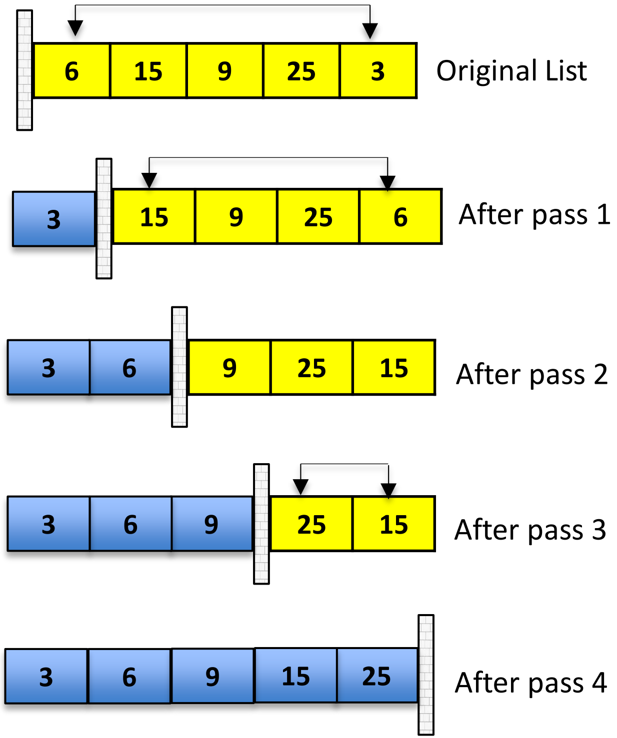
- Find the smallest element in the unsorted list and swap it with the first element of the unsorted list.



Examples of algorithms

Sorting algorithms

Insertion-Sort



Sorting algorithms

Selection-Sort

Pseudocode

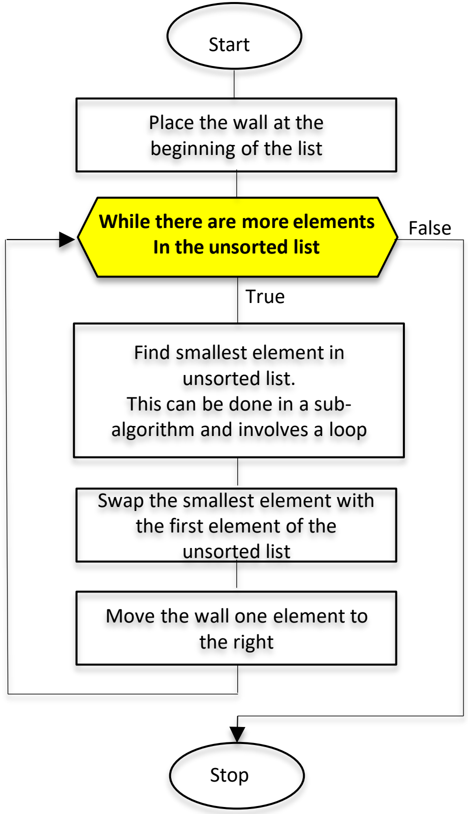
Selection-Sort

Input: A list of integers (a_1, a_2,…, a_n)

```
for i = 1 to A.length -1
    min = i
    /*  check the element to  be minimum */
    for j = i+1 to A.length
        if A[j] > A[min] then
            Min = j
        end if
    end for
    /*  swap the minimum element with the current element */
    If indexMin  != i  then
        swap A[min]   and   A[i]
    end if
End for
```

End (a'_1, a'_2, …, a'_n) are sorted

Flowchart

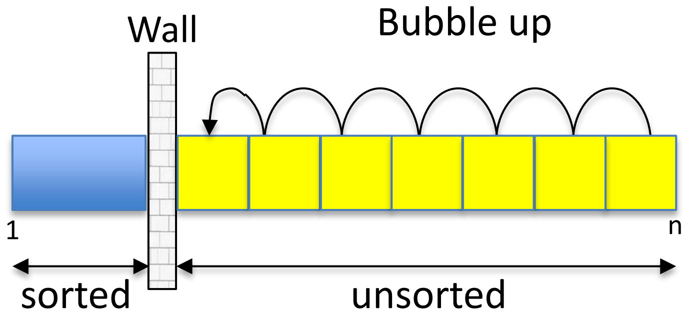


Examples of algorithms

Sorting algorithms

Bubble-Sort

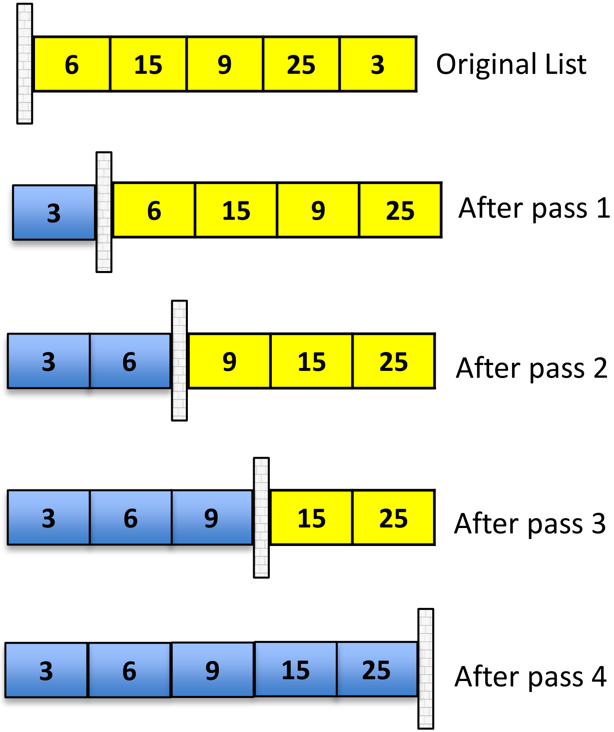
- One of the least efficient algorithms
- It takes successive elements and « bubbles » them up/down in the list.



Examples of algorithms

Sorting algorithms

<u>Bubble-Sort</u>



Examples of algorithms

Sorting algorithms

Pseudocode

Bubble-Sort

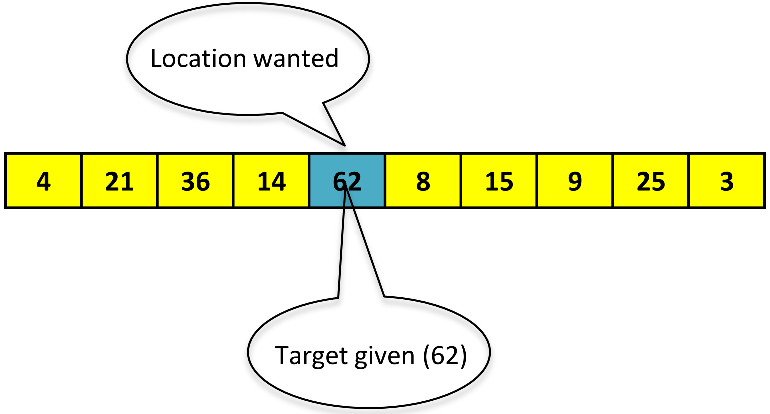
Input: A list of integers (a_1, a_2,…, a_n)

```
for i = 1 to A.length
      swapped = false
      for j = 1 to A.length
          [compare the adjacent elements]
           if A[j] > A[j+1] then
              [ swap them ]
               swap(A[j], A[j+1])
               swapped = true
                end if
          end for
          [if no number was swapped that means
             list is sorted now, break the loop. ]
          if(not swapped) then
             break
            end if
   End for
End  (a'1, a'2, …, a'n ) are sorted
```

Examples of algorithms

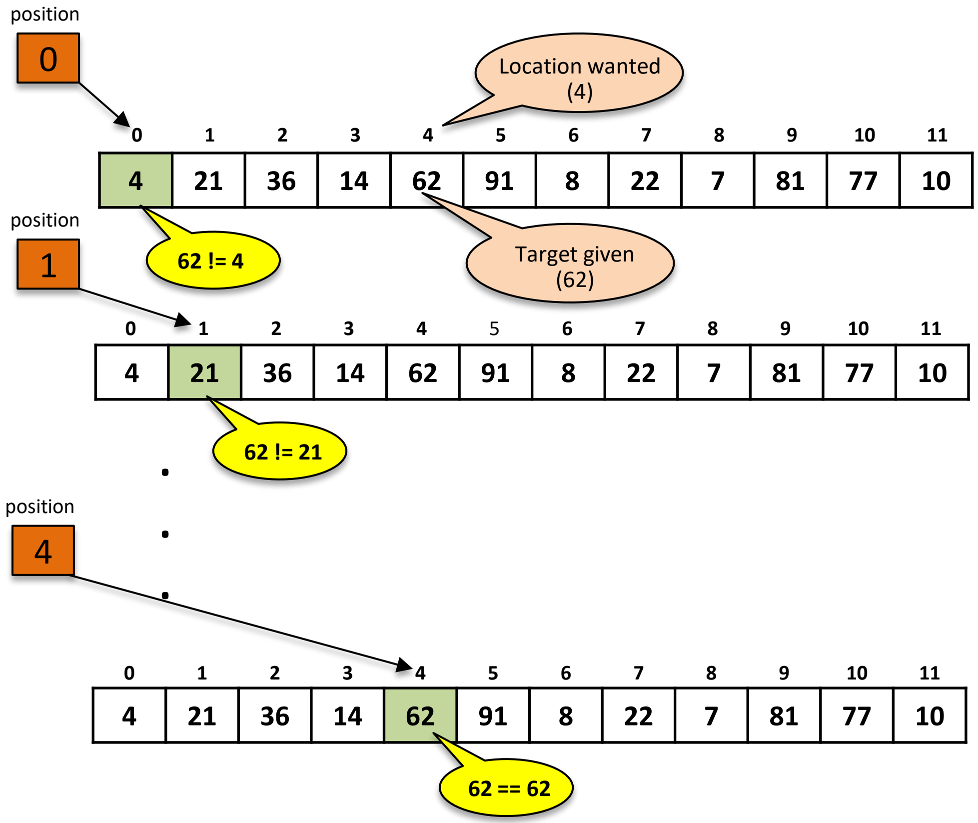
Searching algorithms

- Given a list, find a specific element in the list
- We will see two types
  - Linear search (sequential search)
  - Binary search



Examples of algorithms

Searching algorithms



<u>Linear search</u>

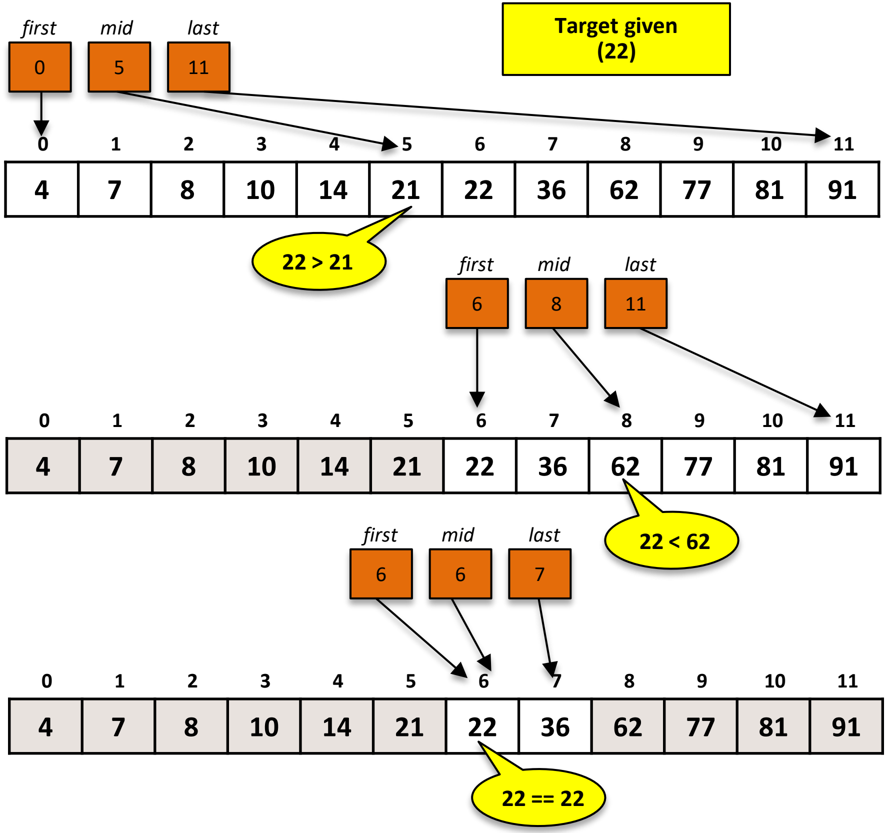
Linear search running time

- How long does this take?
- If the list has n elements , worst case scenario is that it takes n « steps »
- Here, a step is considered a single step through the list

Examples of algorithms

Searching algorithms

Binary search = List MUST be sorted!



Binary search running time

- How long does this take (worst case)?
- If the list has 8 elements
  - It takes 3 steps
- If the list has 16 elements
  - It takes 4 steps
- If the list has 64 elements
  - It takes 6 steps
- If the list has n elements
  - It takes $\log_2(n)$ steps

Algorithm complexity

Space complexity

- How much space is required?

Time complexity

- How much time does it take to run algorithm?

Often, we deal with estimates!

Algorithm complexity

Space complexity

- Space complexity S(p) of an algorithm is the **total space** in memory taken by the algorithm to complete its execution with respect to the input size

S(p) = CONSTANT SPACE + AUXILARY SPACE

Constant space : is the space fixed for that algorithm, generally equals to space used by input and local variables

Auxiliary space : is the extra/temporary space used by an algorithm

ONLY THE AUXILIARY PART SHOULD BE CONSIDERED

S(p) = C + S(auxiliary) = S(auxiliary)

Algorithm complexity

Space complexity

Summation

```
Input: a, b, c
    return a + b + c
End
```

S(p) = 1 + 1 +1 = 3 ➔ No Auxilary

Summation

```
Input (a, n)
Sum = 0
for i in range (n)
        sum = sum + a[i]
end for
return Sum
End
```

S(p) = (n*1 + 1 +1) + 1 = n + 1 ➔ Auxilay = 1

Algorithm complexity

Time complexity

We analyze time complexity only for :

a) Very large input-size

b) Worst case scenario

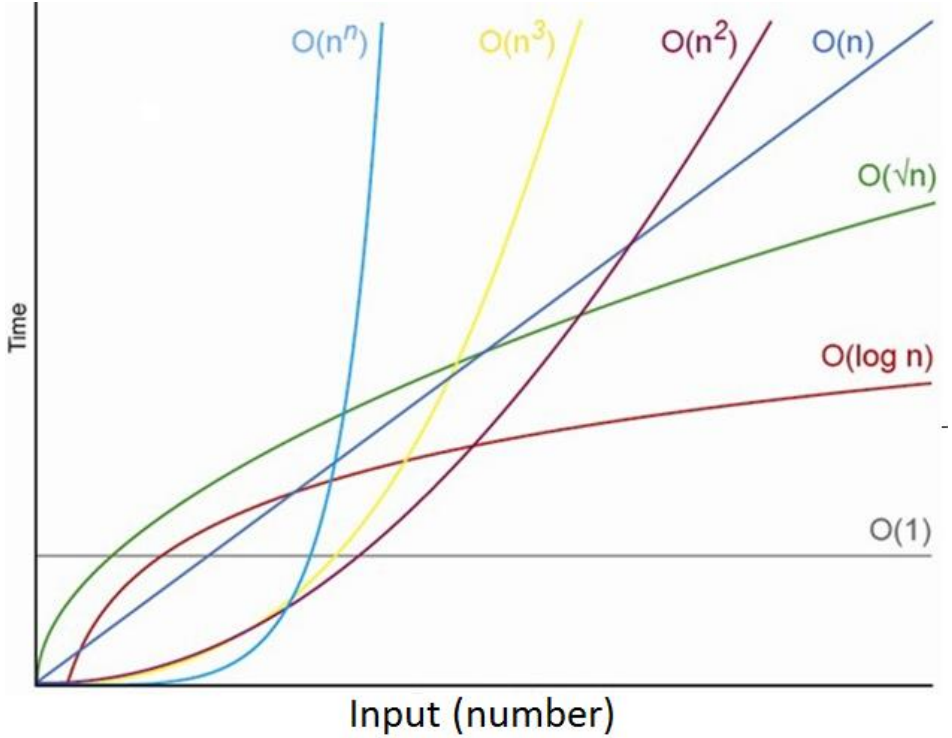
- Time complexity of an algorithm signifies the total **time** required by the program to run till its completion.

The time complexity of algorithms is most commonly expressed using the ***Big O notation***.

Big O notation gives an **uper bound** of the complexity in the **worst** case, helping to quantify performance as the input size becomes **arbitrarily large**.

Algorithm complexity

Time complexity



<u>Big O notation</u>

n: the size of the input

Complexities ordered from smallest to largest

Constant Time: $O(1)$

Linear Time: $O(n)$

Quadratic Time: $O(n^2)$

Cubic Time: $O(n^3)$

Algorithm complexity

Time complexity

Big O properties:

T(n) is a function describing the running time of a particular algorithm for an input of size n:

$T(n) = n^3 + 3n^2 + 4n + 7$

$T(n) \approx n^3 \quad (n \rightarrow \infty)$

$\approx c\, n^3 = O(n^3)$

Rule 1:

a) Lower order terms should not be considered

b) Constant multiplier should not be considered

Example: $T(n) = 17\, n^4 + 3 + 4n + 8 = O(n^4)$

Algorithm complexity

Time complexity

Big O properties:

Rule: Running Time = ∑ Running Time of all fragments

```
int a;
a = 5
a+1;
```

Simple statements

Fragment 1

O(1)

n = length of your list

```
for i = 0 to n
        //simple statements
end of for
```

Simple loop

Fragment 2

O(n)

n = length of your list

```
for i = 0 to n
    for j = 0 to n
        //simple statements
    end of for
end of for
```

nested loop

Fragment 3

$O(n^2)$

Algorithm complexity

Time complexity

```
function
int a;
a = 5;        } O(1)
a+1;
If (some condition)
    for (i = 0 to n
          // simple statements     } O(n)
    end of for
else
    for (i = 0 ; i<n ; i++)
        for (j = 0; j<n; j++)
            //simple statements    } O(n²)
        end of for
    end of for
end of if
end of function
```

$T(n) = O(1) + O(n)$

or

$T(n) = O(1) + O(n^2) \approx O(n^2)$

Rule:

Conditional Statements:

Pick complexity of condition which is worst case

Take-home messages

- Algorithm is a **step-by-step procedure** to **solve problems**
- The types of algorithms depends on the type of **task** to be solved.
- Algorithms are classified based on the **strategy** used for **solving** problems.
- Algorithms can be expressed in : **natural languages, pseudocode, and flowcharts.**
- In one algorithm you could call another algorithm “concept of **sub-algorithm**”.
- Algorithm complexity is seen as **Space complexity** and **time complexity.**

Lecture Videos

COURSE HOME
SYLLABUS
CALENDAR
READINGS
LECTURE NOTES
LECTURE VIDEOS
RECITATION VIDEOS
ASSIGNMENTS
EXAMS
RELATED RESOURCES
DOWNLOAD COURSE MATERIALS

RSS Subscribe to this collection

» Lecture 1: Algorithmic Thinking, Peak Finding

» Lecture 2: Models of Computation, Document Distance

» Lecture 3: Insertion Sort, Merge Sort

» Lecture 4: Heaps and Heap Sort

» Lecture 5: Binary Search Trees, BST Sort

Thank you for your attention

LIÈGE université
GIGA