Data structure and Algorithm

By
Elsayed Atlam
2022

Course Contents

- Data Types
  - Overview, Introductory concepts
  - Data Types, meaning and implementation
  - Abstract data types (ADT)
  - Arrays (revisited)
  - Structures
- Stacks (recursion)
- Queues
- Linked Lists
- Trees (traversals, implementation)

Course Contents

- Binary Trees
- Indexing Methods
  - Hashing
- Binary Search Trees
- Balanced Search Trees
  - (AVL Tree) Adelson-Velskii-Landis
- Heaps

Course objectives

- Be familiar with different data structures available to represents data
- Be able to trace algorithms and verify correctness.
- Be able to develop and implement algorithms using different data structures
- Be able to select appropriate data structures and algorithms for given problems
- Be able to use JAVA language to implement different algorithms pseudo codes.

Objectives of the course

- Present in a systematic fashion the most commonly used data structures, emphasizing their abstract properties.
- Discuss typical algorithms that operate each kind of data structure, and analyze their performance.
- Compare different Data Structures for solving the same problem, and choose the best

Readings/references

- Text Book:
  - **Data Structures & Algorithms in JAVA** (5^{th} Edition), by M. Goodrich & R. Tamassia, John Wiley & Sons, inc., 2010.
- Additional Readings:
  - **Data Structures and Problem Solving with JAVA** (3^{rd} Edition), by Mark Allen Weiss, Addison Wesley, 2006.
  - **Lecture slides and handouts**

What is data?

- Data
  - A collection of facts from which conclusion may be drawn
  - e.g. **Data**: Temperature 35°C; **Conclusion**: It is hot.
- Types of data
  - Textual: For example, your name (Muhammad)
  - Numeric: For example, your ID (090254)
  - Audio: For example, your voice
  - Video: For example, your voice and picture
  - (...)

What is the difference between **Data** and **Information**?

- **Data** are a set of collected numbers, words, anything. They do not mean anything until they are organized, arranged or developed.
- **Examples**: *numbers, dates, prices, names (Olive)*
- Once that happens (after they have been **processed**), information is obtained.
- **Information** actually makes sense and is expressed through some sort of comprehensible logic.
- **Examples:** *reports, Tables, Figures (olive Oil)*

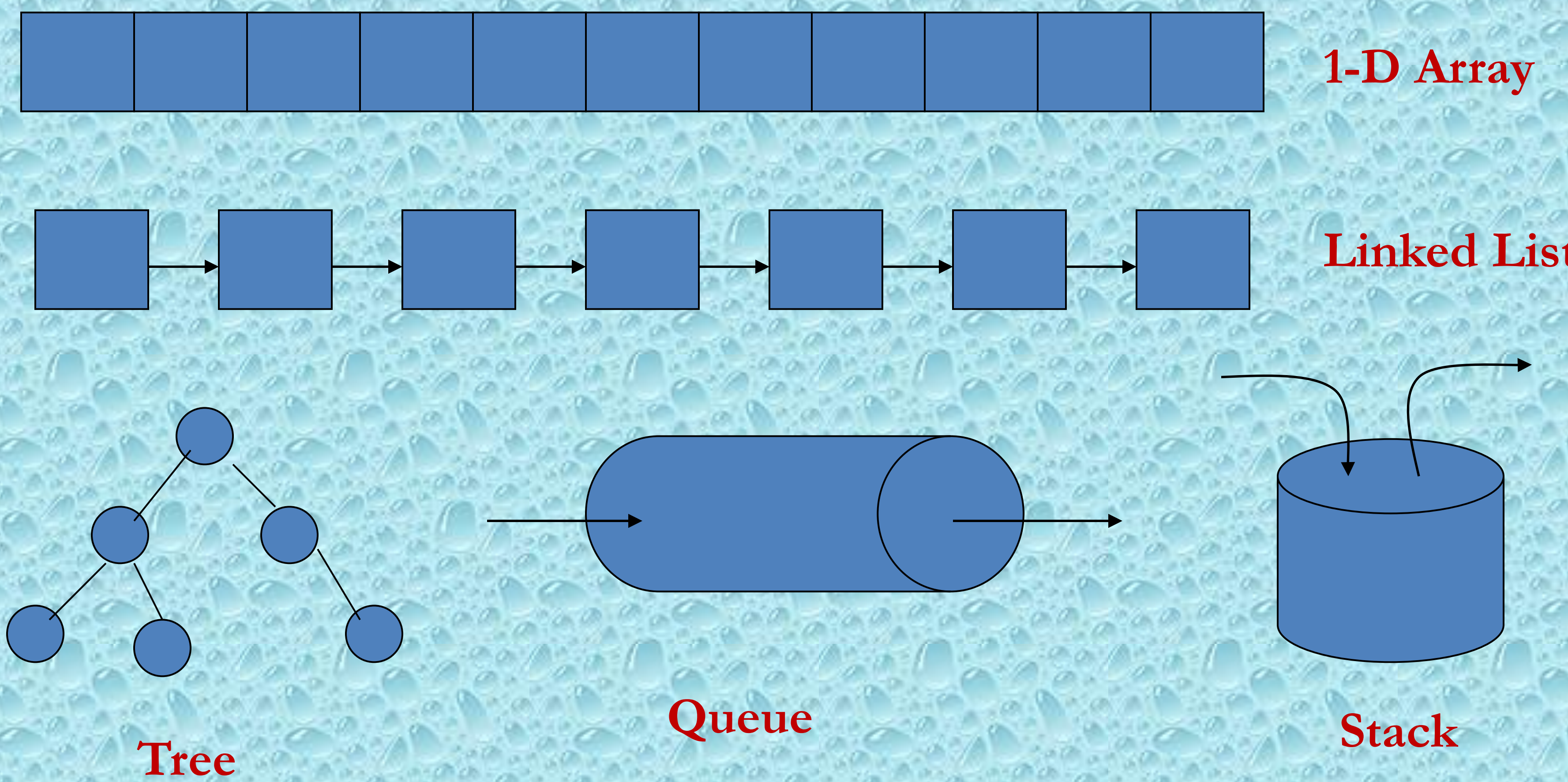
What is the **Processing** that change **Data** to **Information?**

- Adding
- Deleting
- Multiplying
- Logical operations:!=,==,etc
- Retrieving, modifying, updating, saving

What is data structure?

- A particular way of **storing and organizing data in a computer** so that it can be used <u>efficiently and effectively</u>.
- Data Structures are **the programmatic way of storing data so that data can be used efficiently**.
- Data structure is the logical or mathematical model of a particular organization of data.
- A group of data elements grouped together under one name.
  - For example, an array of integers

Types of data structures



There are many, but we named a few. We’ll learn these data structures in great detail!

The Need for Data Structures

- Goal: to **organize data**

- Criteria: to facilitate **efficient**
  - **storage** of data
  - **retrieval** of data
  - **manipulation** of data

- Design Issue:
  - **select and design** appropriate data types

  (This is the main motivation to learn and understand data structures)

Why Study?

- A particular way of **storing and organizing data in a computer** so that it can be used efficiently and effectively

- Designed to develop students understanding the impact of ***structuring data to achieve efficiency*** of a solution to a problem

- After completion you will be familiar with important and most often used data structuring techniques.

- It will enable you to understand the manner in which data is organized and presented later.

Data Structure Operations

(Demonstrate using class room example!)

- Traversing
  - Accessing each data element exactly once so that certain items in the data may be processed
- Searching
  - Finding the location of the data element (key) in the structure
- Insertion
  - Adding a new data element to the structure

Data Structure Operations (cont.)

- Deletion
  - Removing a data element from the structure
- Sorting
  - Arrange the data elements in a logical order (ascending/descending)
- Merging
  - Combining data elements from two or more data structures into one

What is algorithm?

- A finite set of instructions which accomplish a particular task
- A method or process to solve a problem
- Transforms input of a problem to output

Algorithm = Input + Process + Output

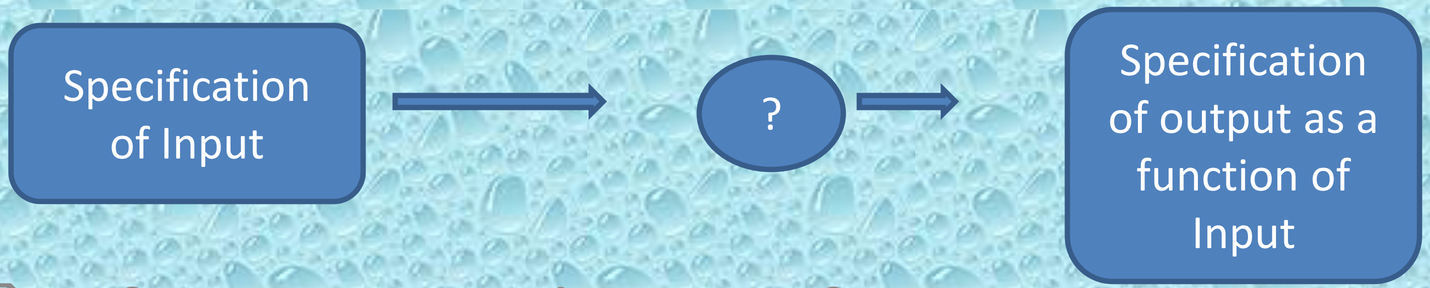
Algorithm development is an art – it needs practice, practice and only practice!

- Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output.
- Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.
- From the data structure point of view, following are some important categories of algorithms −
- **Search** − Algorithm to search an item in a data structure.
- **Sort** − Algorithm to sort items in a certain order.
- **Insert** − Algorithm to insert item in a data structure.
- **Update** − Algorithm to update an existing item in a data structure.
- **Delete** − Algorithm to delete an existing item from a data structure.

Introduction
Data structure and Algorithm

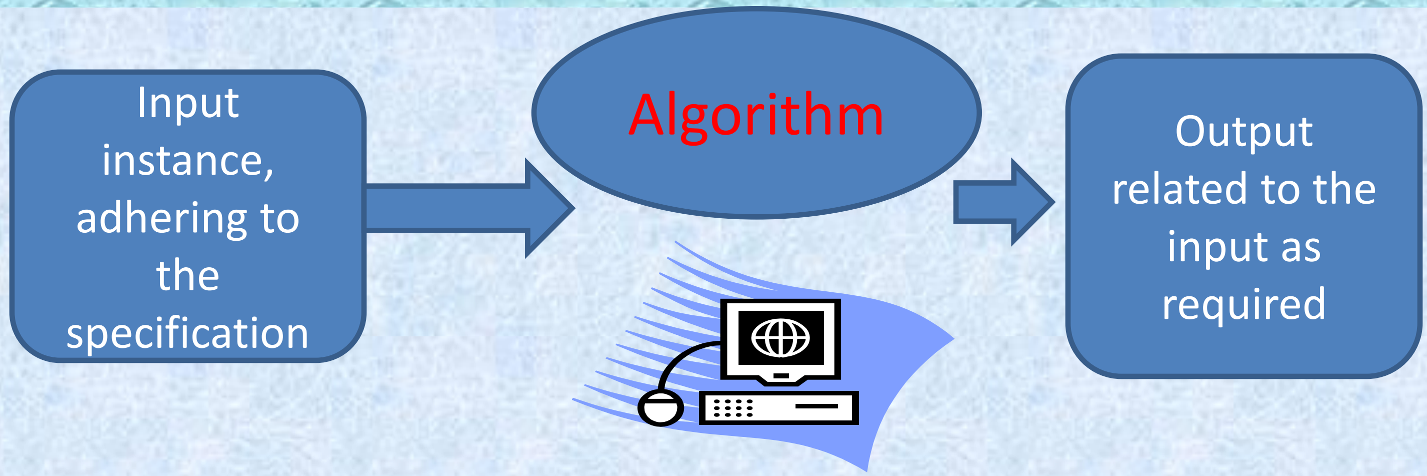
- Algorithm: outline, the essence of a computational procedure, step by step instructions
- Program: an implementation of an Algorithm, written in some specific programming language
- Data Structure: Organization of Data needed to solve the problem

Algorithmic Problem



- Infinite number of input instances satisfying the specification, For example : A sorted, non-decreasing sequence of natural number of non-zero, finite length:
    - 1, 20, 908, 909, 100000, 1000000
    - 3

Algorithmic Solution



- Algorithm describes actions on the input instance to get an output as desires as specified
- Again infinitely many correct algorithms can be used for the same Algorithmic problem

What is a good Algorithm?

- Efficient: Any thing is efficient is good
  - Small Running time
  - Space Used (Less Memory)

What is a good algorithm?

- It must be correct
- It must be finite (in terms of time and size)
- It must terminate
- It must be unambiguous
    - Which step is next?
- It must be space and time efficient

A program is an instance of an algorithm, written in some specific programming language

What is a good Program?

There are a number of facets to good programs: they must

- run correctly
- run efficiently
- be easy to read and understand
- be easy to debug *and*
- be easy to modify.

What does correct mean?

We need to have some formal notion of
the meaning of correct:
thus we define it to mean

"run in accordance with the specifications".

A simple algorithm

- Problem: Find maximum of `a, b, c`
- **Algorithm**
  - Input = `a, b, c`
  - Output = `max`
  - Process

```
o Let max = a
o If b > max then
    max = b
o If c > max then
    max = c
o Display max
```

Order is very important!!!

Algorithm development: Basics

- Clearly identify:
  - what output is required?
  - what is the input?
  - What steps are required to transform input into output
    - The most crucial bit
    - Needs problem solving skills
    - A problem can be solved in many different ways
    - Which solution, amongst the different possible solutions is optimal?

How to express an algorithm?

- A sequence of steps to solve a problem
- We need a way to express this sequence of steps
  - Natural language (NL) is an obvious choice, but not a good choice. Why?
    - NLs are notoriously ambiguous (unclear)
  - Programming language (PL) is another choice, but again not a good choice. Why?
    - Algorithm should be PL independent
  - We need some balance
    - We need PL independence
    - We need clarity
    - Pseudo-code provides the right balance

What is pseudo-code?

- **Pseudo-code** is a short hand way of describing a computer program
- Rather than using the specific syntax of a computer language, more general wording is used
- It is a mixture of NL and PL expressions, in a systematic way
- Using pseudo-code, it is easier for a non-programmer to understand the general workings of the program

Pseudo-code: general guidelines

- Use PLs construct that are consistent with modern high level languages, e.g. C++, Java, ...

- Use appropriate comments for clarity

- Be simple and precise

Pseudo-Code

- A mixture of natural language and high –level programming concepts that describes the main idea behind a generic implementation of a data structure or Algorithm.

- Eg: **Algorithm arrayMax(A,n):**

Input: An array A storing n integers,

Output: the maximum element in A.

```
currentMax ← A[0]
for i← 1 to n-1 do
  if currentMax <A[i] then currentMax← A[i]
return currentMax
```

Pseudo-Code

It is more structured than usual prose but less formal than a programing language

What pseudo-code looks like:

- Expressions:
  - Use standard mathematical symbols to describe numeric and boolen expresions
  - Use ← for assignment (`=` in C)
  - Use = for the equality relationship (`==` in C)

- Method Declaration
  - Algorithm name (param1, Param2)

Pseudo-Code

- Programming Constructions:
  - Decision structure: **if… then… [else….]**
  - While-loops: **while….do**
  - Repeat-loops: **repeat…. Until….**
  - For-loop: **for….do**
  - Array indexing: **A[i], A[I,j]**

- Methods
  - Calls: object method(args)
  - Returns: **return** Value

Components of Pseudo-code With Examples

- **Expressions**
  - Standard mathematical symbols are used
    - Left arrow sign (←) as the assignment operator in assignment statements (equivalent to the = operator in Java)
    - Equal sign **(=)** as the equality relation in Boolean expressions (equivalent to the "= =" relation in Java)
    - For example

      Sum ← 0

      Sum ← Sum + 5

    What is the final value of sum?

Components of Pseudo-code (cont.)

- **Decision structures** (if-then-else logic)
  - **`if`** condition **`then`** true-actions [**`else`** false-actions]
  - We use indentation to indicate what actions should be included in the true-actions and false-actions
  - For example

```
if marks > 50 then
   print "Congratulation, you are passed!"
else
   print "Sorry, you are failed!"
end if
```

What will be the output if marks are equal to 75?

Components of Pseudo-code (cont.)

- Loops (Repetition)
  - **Pre-condition loops**
    - While loops
      - **`while`** <u>condition</u> **`do`** <u>actions</u>
      - We use indentation to indicate what actions should be included in the loop actions
      - For example

```
while counter < 5 do
        print "Welcome to CS204!"
        counter ← counter + 1
end while
```

What will be the output if counter is initialised to 0, 7?

Components of Pseudo-code (cont.)

- Loops (Repetition)
  - **Pre-condition loops**
    - For loops
      - **for** variable-increment-definition **do** actions
      - For example

for counter ← 0; counter < 5; counter ← counter + 2 **do**

print "Welcome to CS204!"

end for

What will be the output?

Components of Pseudo-code (cont.)

- Loops (Repetition)
  - **Post-condition loops**
    - Do loops
      - **`do`** actions **`while`** condition
      - For example

```
do
        print "Welcome to CS204!"
        counter ← counter + 1
while   counter < 5
```

What will be the output, if counter was initialised to 10?

The body of a post-condition loop must execute at least once

Homework

1. **Write an algorithm to find the largest of a set of 10 numbers.**

2. **Write an algorithm in pseudocode that finds the average of (10) numbers.**

Components of Pseudo-code (cont.)

- **Method declarations**
  - Return_type method_name (parameter_list) method_body
  - For example

    integer *sum* (integer num1, integer num2)

    start

    result ← num1 + num2

    end
- **Method calls**
  - object.method (args)
  - For example

    mycalculator.sum(num1, num2)

Components of Pseudo-code (cont.)

- **Method returns**
  - **`return`** value
  - For example

integer *sum* (integer num1, integer num2)

start

result ← num1 + num2

`return` result

end

Components of Pseudo-code (cont.)

- Comments
  - /* Multiple line comments go here. */
  - // Single line comments go here
  - Some people prefer braces {}, for comments
- Arrays
  - *A[i]* represents the *i*th cell in the array *A*.
  - The cells of an *n*-celled array *A* are indexed from *A*[0] to *A*[$n - 1$] (consistent with Java).

Algorithm Design: Practice

- Example : Determining even/odd number
  - A number divisible by 2 is considered an even number, while a number which is not divisible by 2 is considered an odd number. Write pseudo-code to display first N odd/even numbers.

Even/ Odd Numbers

```
Input range
for      num←0;      num<=range;
 num←num+1 do
 if num % 2 = 0 then
   print num is even
 else
   print num is odd
 endif
endfor
```

1. Write an algorithm to find the largest of a set of 10 numbers.

Input: 10 positive integers

Output: Max integer

Process:

```
Range=10;
Max← 0;
Counter← 1;
for              counter←0;counter<=range;
  counter←counter+1 do
  if  integer>=  max then
     max=integer;
     endif
Endfor
Return max;
```

FindLargest

Input: 1000 positive integers

1. Set Largest to 0
2. Set Counter to 0
3. while (Counter less than 1000)
   - 3.1 if (the integer is greater than Largest)
     - then
       - 3.1.1 Set Largest to the value of the integer
     - End if
   - 3.2 Increment Counter

   End while
4. Return Largest

End

1. **Write an algorithm in pseudocode that finds the average of (10) numbers.**

Input: 10 positive integers

Output: average of 10 integers

Process:

```
sum← 0;
for i←0; i<=10; i←i+1 do
    input x;
        sum=sum+x;
Endfor
Avg=sum/10;
Return Avg;
```

Write an algorithm which requires a number between 10 and 20, until the response is appropriate. If the number is more than 20, it will display a message: “Bigger!" If the number is less than 10, it will display “smaller!"

Begin
Input: num
Output: numbers between 10 and 20
Process:
Start
    if (num<10) Then
        print “Smaller !”
    elseif (num >20)
        print “Bigger !”
    End if
End

What are the values of the variables A, B and C after execution of the following instructions?

Begin

$A \leftarrow 3$

$B \leftarrow 10$

$C \leftarrow A + B$

$B \leftarrow A + B$

$A \leftarrow C$

End

Write an algorithm to swap the value the 2 variables A and B.

Input: A and B and C

Output: Swapping

Process:

```
Start
    C← A;
    A← B;
    B← C;
    Return A and B;
End
```

Write pseudocode that will take a number as input and tells whether a number is positive, negative or zero.

```
Begin
WRITE “Enter a number”
READ num
IF num> 0  THEN
    WRITE “The number is positive”
ELSE IF num = 0 THEN
   WRITE “The number is zero”
ELSE
    WRITE “The number is negative”
    ENDIF
ENDIF                                        End
```

Question?

- Write a pseudo-code to count (calculate) the submission of the first 100 normal number?

- what is a good Algorithm?

- What is a good program?

Measuring the Running time

How should we measure the running time of an **Algorithm?**

Experimental Study

- Write a certain **program** that implements the algorithm
- Run the program with data sets of varying size (large or small) and composition
- Clock the time by: Use a method like **System.currentTimeMillis()** to get an accurate measure of the actual running time

Limitations of Experimental Studies

- It is necessary to **implement** and test the algorithm in order to determine its running time.
- Experiments can be done only on **a limited set of inputs,** and may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same **Hardware and software environments** should be used.

Beyond Experimental Studies

We will develop a **general methodology** for analyzing running time of algorithms. This approach (we want to)

- Uses a **high-level description** of the algorithm instead of testing one of its implementations.
- Takes into account **all possible inputs**
- Allows one to evaluate the efficiency of any algorithm in a way that is **independent of the hardware and software environment**

Analysis of Algorithms

- **Primitive Operation:** low-level operation independent of programming language.

Can be identified in pseudo-code. For eg:

  - Data movement (assign)
  - Control (Branch, subroutine call, return)
  - Arithmetic an logical operations (e.g. addition, comparison)

- By inspecting the pseudo-code, we can count the number of primitive operations executed by the algorithm

Example: Sorting

INPUT

Sequence of numbers

a1,a2,a2,……an

2 5 4 10 7

Sort

OUTPUT

A permutation of the sequence of numbers

b1,b2,b3,….,bn

2 4 5 7 10

Correctness (requirements for the output)

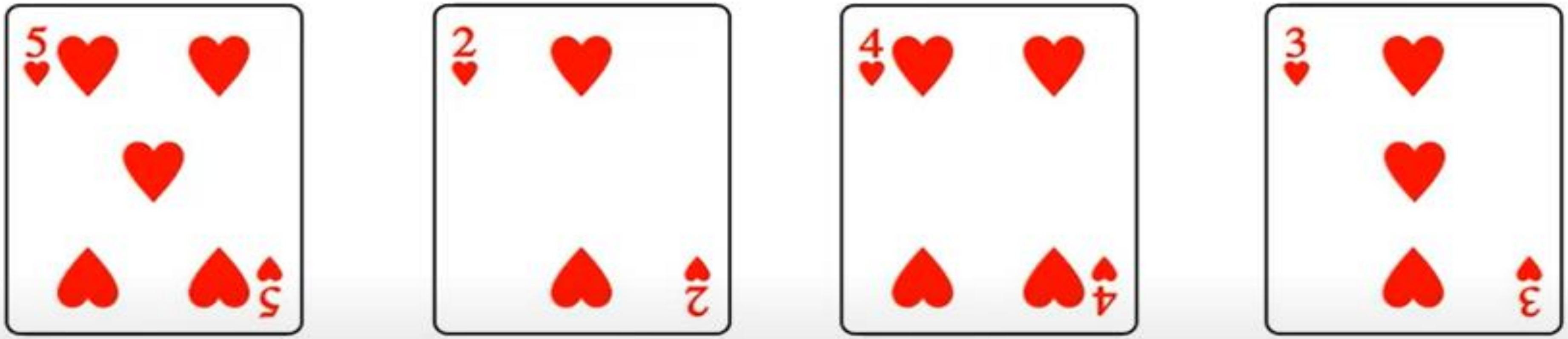
For any given input the algorithm halts with the output

- b1 < b2 < b3……< bn
- b1,b2,…..bn is a permutation of a1,a2,…..an

Running time Depend on

- Number of element (n)
- How (partially) sorted they are
- Algorithm

Insertion Sort



Cards Hand Play

```
void insertionSort(int arr[], int n)
{
    int key, j;                     //0   1  2  3  4  5  6
    for (int i = 1; i < n; i++)//80 |90| 60 30 50 70 40
    {
        key = arr[i];//90
        j = i - 1;//0

        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;//90
    }
```

```
void insertionSort(int arr[], int n)
{
    int key, j;                       //0   1  2  3  4  5  6
    for (int i = 1; i < n; i++)//60  80 90 |30 50 70 40
    {
        key = arr[i];//60                 i=2
        j = i - 1;//0

        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;//90
    }
```

Example: Sorting

A	3	4	6	8	9	7	2	5	1
	1					j→			n

←i

Strategy

- Start empty handed
- Insert a card in the right position of the already sorted hand
- Continue until all cards are inserted sorted

INPUT:A[1…n]- an array of integers
OUTPUT: a permutation of A such that A[1]<A[2]….<A[n]

for j=2 to n do
Key← A[j]
Insert A[j] into the sorted sequence A[1,j-1]
i←j-1
While i>0 **and** A[i]>Key
  do A[i+1]← A[i]
    i- -
  A[i+1]← key

Analysis of Algorithms

Algorithm	Cost	Times
for j=2 to n do	C1	n-1
Key← A[j]	C2	n-1
Insert A[j] into the sorted sequence A[1,j-1]	0	n-1
i←j-1	C3	n-1
While i>0 **and** A[i]>Key	C4	$\sum_{j=2}^{n} t_j$
do A[i+1]← A[i]	C5	$\sum_{j=2}^{n} t_j - 1$
i- -	C6	$\sum_{j=2}^{n} t_j - 1$
A[i+1]← key	C7	n-1

Total time= n(C1+C2+C3+C7)+ $\sum_{j=2}^{n} t_j (C4+C5+C6)$ - (C1+C2+C3+C5+C6+C7)

Best/Worst/average Case (1)

Total time= n(C1+C2+C3+C7)+ $\sum_{j=2}^{n} t_j (C4 + C5 + C6)$ - (C1+C2+C3+C5+C6+C7)

- **Best Case**: elements already sorted; tj=1,

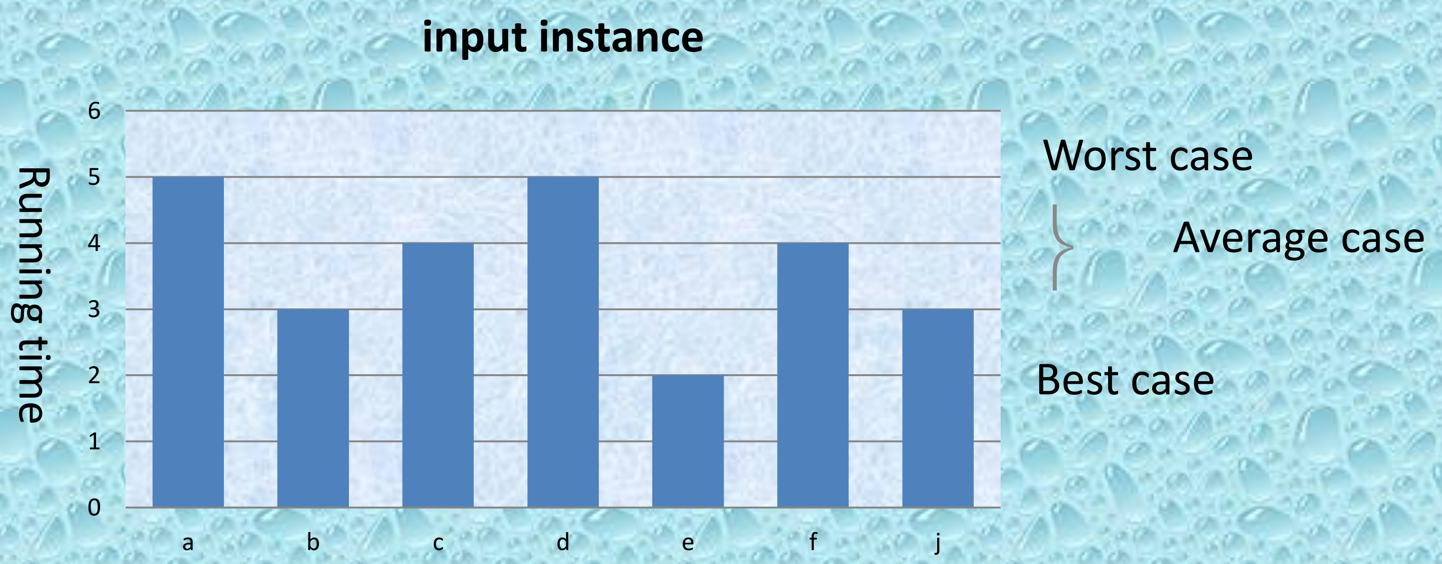
Running time= f(n), i.e. Linear time

- **Worst Case**: elements are sorted in inverse order, tj=j, running time =f(n^2), i.e quadratic time
- **Average case**: tj=j/2, running time = f(n^2)

i.e quadratic time

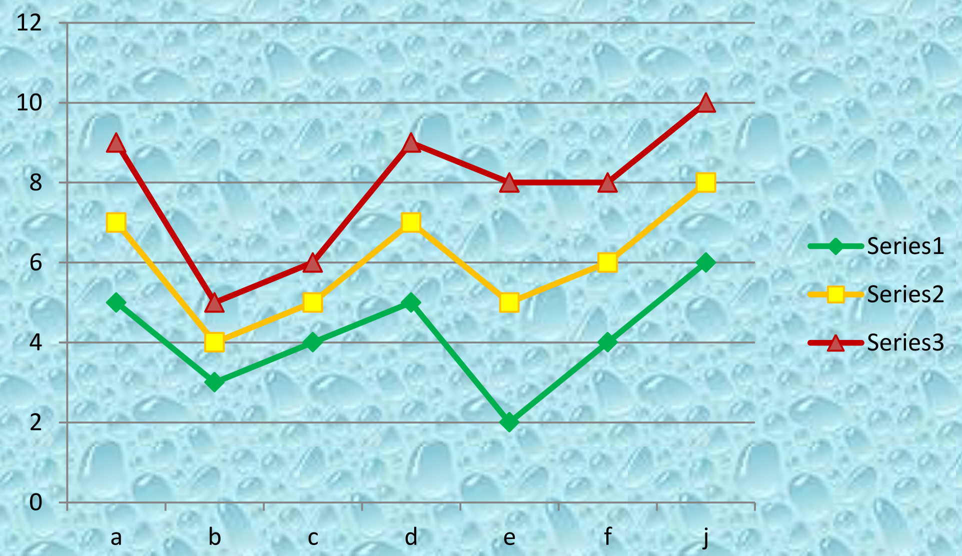
Best/Worst/average Case (2)

For a specific size of input n, investigate running times for different input instance



Best/Worst/average Case (3)

For inputs of all sizes:



Best/Worst/average Case (4)

- **Worst Case**: is usually used: it is an upper bound and in certain application domains (e.g. air traffic control, surgery) knowing the **worst case** time complexity is of crucial important.
- For some algorithms **worst case** occurs fairly often
- **Average case**: is often as bad as the **worst case**
- Finding **average case** can be very difficult

Asymptotic Analysis

- Goal: to simplify analysis of running time by getting rid of details which may be affected by specific implementation and hardware
  - Like *rounding*:1,000,001=1,000,000
  - $3n^2=n^2$
- **Capturing the essence**: how the running time of an algorithm increases with the size of the input in the limit.
  - **Asymptotic** more efficient algorithms are best for all but small inputs

Asymptotic Analysis of Running time

- Using O-notation to express number of primitive operations executed as function of input size.
- Comparing asymptotic running times:
  - An Algorithm that runs in O(n) is better than one runs in $O(n^2)$ time
  - Similarly , O(log n) is better than O(n)
  - Hierarchy of functions: $\log n < n < n^2 < n^3 < 2^n$
- **Caution!** Beware of very large constant factors. An algorithm running in time 1,000,000 n is still O(n) but might be less efficient than one running in time $2n^2$ which is $O(n^2)$

Example of Asymptotic Analysis

Algorithm of prefix Averages1(X):

Input: An n-element array X of Numbers

Output: An n-element array A of numbers such that A[i] is the average of elements X[0],….,X[i]

for i←0 to n-1 do

a← 0

for j← 0 to i do

a← a +X[j]

A[j]←a/(i+1)

return array A

i iterations with
i=0,1, n-1
Executed I times

n iterations

Analysis: running time is O(n^2)

Example of Asymptotic Analysis (A Better Algorithm)

Algorithm of prefix Averages2(X):

Input: An n-element array X of Numbers

Output: An n-element array A of numbers such that A[i] is the average of elements X[0],….,X[i]

S← o

for i← 0 to n do

S← S +X[i]

A[i]←S/(i+1)

return array A

Analysis: running time is O(n)

Example of Asymptotic Analysis (A Better Algorithm)

Algorithm of prefix Averages1(X):

Input: An n-element array X of Numbers

Output: An n-element array A of numbers such that A[i] is the average of elements X[0],….,X[i]

for i←0 to n-1 do

a← 0

for j← 0 to i do

a← a +X[j]

A[j]←a/(i+1)

return array A

Analysis: running time is O(n^2)

Algorithm of prefix Averages2(X):

Input: An n-element array X of Numbers

Output: An n-element array A of numbers such that A[i] is the average of elements X[0],….,X[i]

S← o

for i← 0 to n do

S← S +X[i]

A[i]←S/(i+1)

return array A

Analysis: running time is O(n)

Comparison of running Times

Running Time	Maximum Problem Size (n)		
	1 Second	1 minute	1 hour
400n	2500	150000	9000000
20 n log n	4096	166666	7826087
$2n^2$	707	5477	42426
n^4	31	88	244
2^n	19	25	31

you can see what is the largest size of the problem you can solve in one second , 1 minutes and 1 hour,

Notice

- نلاحظ ان سرعة معالجة البيانات تعتمد علي عدة عوامل إضافة إلي العامل الزمني اللازم للمعالجة مثل:
- عوامل تحديد الذاكرة الرئيسية
- عوامل تحديد وحدات الإدخال والإخراج
- عوامل تحديد تفاعل وحدات الإدخال والإخراج مع الذاكرة الرئيسية
- What is Big-O notation?
- Explanation about Array?(delete, insert,….)