Data Structures (CSE2PM01A / AID2PM01A)

S. Y. B. Tech CSE **Semester – III**

DEPARTMENT OF COMPUTER ENGINEERING & TECHNOLOGY

Data Structures (CSE2PM01A / AID2PM01A)

- Pre-requisites
  - Programming Foundations

- Course Objectives
- **Knowledge**
  - Learn the data structure and its fundamental concept.
- **Skills**
  - Understand the different Linear and nonlinear data structures such as Arrays, Stacks, Queues Trees and Graph.
  - Study the different Sorting Techniques and concepts of Heap and Trees
- **Attitude**
  - Learn how to apply Data Structures to solve Real-life Problems

Data Structures (CSE2PM01A)

- Course Outcomes
  - After completion of this course students will be able to:
  - To demonstrate the use of sequential data structures
  - To analyse different sorting algorithms so as to understand their applications
  - To develop skills for writing and analysing algorithms to solve domain problems
  - To choose appropriate non-linear data structures to solve a given problem

Data Structures (CSE2PM01A)

- **Unit 1 – Introduction to Data Structures**: Data, Data Objects, Abstract Data types (ADT) and Data Structures, Types of data structures (Linear and Non-linear, Static and dynamic) Introduction to algorithms Analysis of Algorithms- Space complexity, Time complexity, Asymptotic notations-Big-O, Theta and Omega, **Sorting**: Types of Sorting-Internal and External Sorting, General Sort Concepts-Sort Order, Stability, Efficiency, and Number of Passes, Comparison Based Sorting Methods-Bubble Sort, Insertion Sort, Selection Sort, Shell Sort, Comparison of all sorting algorithm
- **UNIT 2 – Linear Data Structures:** Arrays, Array as an Abstract Data Type, Sequential Organization, Storage Representation and their Address Calculation: Row major and Column Major, Multidimensional Arrays: Concept of Ordered List, Single Variable Polynomial: Representation using arrays, Polynomial as array of structure, Polynomial addition, Polynomial evaluation and Polynomial multiplication. **Sparse Matrix:** Sparse matrix representation using array, Sparse matrix addition, Transpose of sparse matrix- Simple and Fast Transpose, Time and Space trade-off.

Data Structures (CSE2PM01A)

- **UNIT – 3: Stacks and Queues: Stacks:** Stack as an Abstract Data Type, Representation of Stack Using Sequential Organization, stack operations, Applications of Stack- Expression Conversion and Evaluation, Linked Stack and Operations, Recursion. **Queues:** Queue as Abstract Data Type, Representation of Queue Using Sequential Organization, Queue Operations Circular Queue, Advantages of Circular queues, Linked Queue and Operations Deque-Basic concept, types (Input restricted and Output restricted), Application of Queue: Job scheduling.

Data Structures (CSE2PM01A)

- **UNIT – 4: Linked List:** Linked List as an Abstract Data Type, Representation of Linked List Using Sequential Organization, Representation of Linked List Using Dynamic Organization, Types of Linked List: singly linked, Circular Linked Lists,Doubly Linked List, Primitive Operations on Linked List, Polynomial Manipulations-Polynomial addition. Generalized Linked List (GLL) concept, Representation of Polynomial using GLL.

  Case Study: Garbage Collection

- **Unit – 5**

  Basic Terminology, Binary Tree- Properties, Converting Tree to Binary Tree, Representation using Sequential and Linked organization, Binary tree creation and Traversals, Binary Search Tree (BST) and its operations, threaded binary tree- Creation and Traversal of In-order Threaded Binary tree.

  Heap- Heap as a priority queue, Heap sort.

Data Structures Lab

1. Write a program to create a student database using an array of structures. Apply sorting techniques (Bubble sort , Insertion Sort, Selection Sort).
2. Write a C program for sparse matrix realization and operations on it- Simple Transpose, Fast Transpose.
3. Implement stack as an ADT and apply it for different expression conversions (infix to postfix or infix to prefix (Any one), prefix to postfix or prefix to infix, postfix to infix or postfix to prefix (Any one)).
4. Queues are frequently used in computer programming, and a typical example is the creation of a job queue by an operating system. If the operating system does not use priorities, then the jobs are processed in the order they enter the system. Write a program for simulating job queue. Write functions to add job and delete job from queue.
5. Department of Computer Engineering has student's club named 'Pinnacle Club'. Students of second, third and final year of department can be granted membership on request. Similarly, one may cancel the membership of club. First node is reserved for president of club and last node is reserved for the secretary of the club. Write C program to maintain club member ‘s information using singly linked list. Store student PRN and Name. Write functions to: a) Add and delete the members as well as president or even secretary. b) Compute total number of members of club c) Display members d) sorting of two linked list e) merging of two linked list f) Reversing using three pointers.

Data Structures Lab
List of Lab Assignments

6. Implement binary tree and perform following operations: Creation of binary tree and traversal recursive and non-recursive.
7. Implement a dictionary using a binary search tree where the dictionary stores keywords & its meanings. Perform following operations: Insert a keyword, Delete a keyword, Create mirror image and display level wise, Copy
8. Implement a threaded binary tree. Perform inorder traversal on the threaded binary tree.
9. Read the marks obtained by students of second year in an online examination of a particular subject. Find the maximum and minimum marks obtained in that subject. Use heap data structure and heap sort.

Data Structures (CSE2PM01A)

Evaluation Components-Theory(CCA)

DS Theory Components			
Components	**Planned Units**	**Planned Dates**	**Tentative Marks**
CCA-1 (Quiz and Problem Solving)	1 ,2,3	10-Aug-24	15
CCA- 2 (Active Learning)	4 ,5	23-Oct-24	15
Mid Term Exam	1,2,3	7-Oct-24	30
End Term Exam	1,2,3,4, 5	25-Nov	40

Unit wise Weight				
Unit 1	Unit 2	Unit 3	Unit 4	Unit 5
20 M	20 M	20 M	20 M	20 M

Evaluation Components- DS Lab

Components	Planned Assignments	Planned Dates	Tentative Marks
LCA 1	1,2,4	14-Aug-24	30
LCA 2	3,5,6,7	20-Sep-24	35
LCA 3	8, 9 + Practical Exam	17-Nov-24	35

Data Structures (CSE2PM01A)

- **Learning Resources:**
  - **Text Books:**
    1. Fundamentals of Data Structures, E. Horowitz, S. Sahni, S. A-Freed, Universities Press.
    2. Data Structures and Algorithms, A. V. Aho, J. E. Hopcroft, J. D. UIlman, Pearson.
  - **Reference Books:**
    1. The Art of Computer Programming: Volume 1: Fundamental Algorithms, Donald E. Knuth.
    2. Introduction to Algorithms, Thomas, H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, The MIT Press.
    3. Open Data Structures: An Introduction (Open Paths to Enriched Learning), (Thirty First Edition), Pat Morin, UBC Press.

Data Structures (CSE2PM01A)

- **Supplementary Readings:**
  1. Aaron Tanenbaum, “Data Structures using C”, Pearson Education.
  2. R. Gilberg, B. Forouzan, "Data Structures: A pseudo code approach with C", Cenage Learning, ISBN 9788131503140.
- **Web Resources:**
  - **Web Links:**
    - https://www.tutorialspoint.com/data_structures_algorithms/
  - **MOOCs:**
    - http://nptel.ac.in/courses/106102064/1
    - https://nptel.ac.in/courses/106103069/

Introduction to Data Structures

- Data, Data objects, Data Types
- Abstract Data types (ADT) and Data Structure
- Types of data structure
- Introduction to Algorithms
- Algorithm Design Tools: Pseudo code and flowchart
- Analysis of Algorithms- Space complexity, Time complexity,

  Asymptotic notations - Big-O, Theta and Omega,
  - Types of Sorting-Internal and External Sorting,

Introduction to Data Structures

- General Sort Concepts-Sort Order,
- Stability, Efficiency, and Number of Passes,
- Comparison Based Sorting Methods-Bubble Sort
- Insertion Sort, Selection Sort, Shell Sort
- Comparison of all sorting algorithm

Data, Data Objects and Data Types

- Computer Science is study of data
  - Machines that hold data
  - Languages for describing data manipulations
  - Foundations which describe what kinds of refined data can be produced from raw data (actual sensor data, Google form data)
  - Structures of refining data (sensor data w/o outliers)

Data, Data Objects and Data Types

Data is of two types

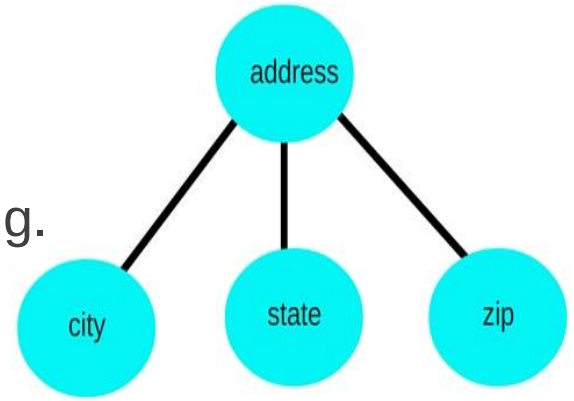
- **Atomic Data**

It consist of single piece of information. It cannot be divided into other meaningful pieces of data. e.g Name of Person, Name of Book

- **Composite Data**

It can be divided into subfields that have meaning.

e.g. Address, Telephone number

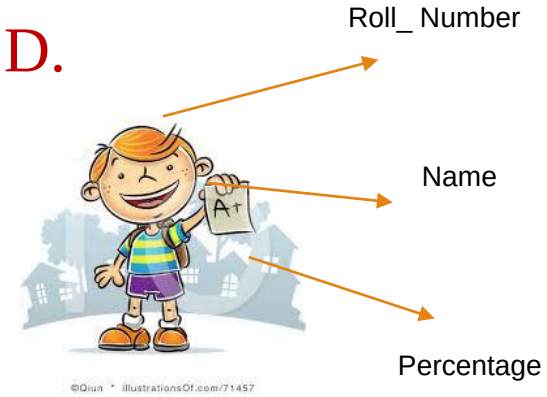


Data, Data Objects and Data Types

Data Objects

Data object is referring to **set of elements** say D.

For Example: Data Object **integers** refers to D={0,±1,±2,……}



Data Object represents an **object having a data**.

For Example: If student is one object then it will consist of different data like roll no, name, percentage, address etc.

Data, Data Objects and Data Types

Data Types

- A Data type is a term which refers to the kinds of data that variables may hold in a programming languages.
- For Example: In C programming languages, the data types are integer, float, character etc.
- Data type is a way to classify various types of data such as integer, string, etc. which determines the values that can be used with the corresponding type of data, the type of operations that can be performed on the corresponding type of data.
- There are two data types −
  - Built-in Data Type
  - Derived Data Type

Data, Data Objects and Data Types

Built-in Data Type

Those data types for which a language has built-in support are known as Built-in Data types.

For example, most of the languages provide the following built-in data types.

- Integers
- Boolean (True, False)
- Floating (Decimal numbers)
- Character and Strings

Derived Data Type

These data types are normally built by the combination of primary or built-in data types and associated operations on them.

For example −

- List (dynamic arrays)
- Array
- Structure
- Pointers in C

Abstract Data Type(ADT) and Data Structure

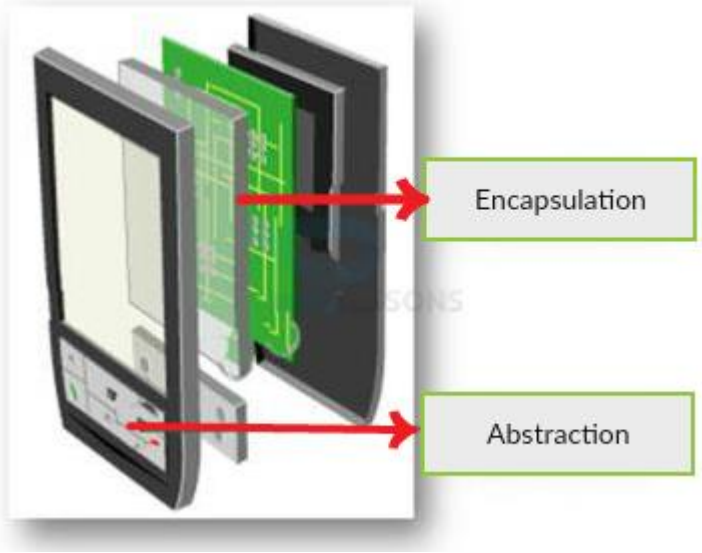
- **Abstract Data Type**
  - Concern about what can be done not how it can be done

Abstract Data Type consist of

- Declaration of Data
- Declaration of Operations
- Encapsulation of data and operations

Abstract data types

An abstract data type is a type with associated operations, but whose representation is hidden.



- The calculator explains it very well.
- One can use it different ways by giving various values and perform operations.
- But, mechanism how the operation is done is not shown.
- This process of hiding the information is called as *Abstraction*.

Abstract Data Types (ADT)

- An **ADT** is composed of
  - A collection of data
  - A set of operations on that data
- Specifications of an **ADT** indicate
  - What the ADT operations do, not how to implement them
- Implementation of an **ADT**
  - Includes choosing a particular data structure

Abstract Data Type(ADT) and Data Structure

Abstract Data Type Examples

- Array
- Tree
- Graph
- Linked List
- Matrix

```
structure ARRAY(value, index)
        declare CREATE()→array
                RETRIVE(array,index)→value
                STORE(array,index,value)→array;
        for all A ε array, i,j ε index ,x ε value let
                RETRIVE (CREATE,i) : : = error
                RETRIVE (STORE(A,i,x),j) : : =
                        if EQUAL(i,j) then x else
                                RETRIVE(A,j)
        end
end ARRAY
```

Data Structure

- A data structure is a set of domains D, a structured domain d ε D, a set of functions F and a set of axioms A.
- The triple(D,F,A) denotes the data structure d ε D and it will usually be abbreviated by writing d.
- The triple(D,F,A) is referred to as an abstract data type (ADT).
- It is called abstract precisely because the axioms do not imply a form of representations/implementation.

Data Structure

Example

Structure NATNO
Declare ZERO() → natno

ISZERO(NATNO) → Boolean
SUCC(natno) → natno
ADD(natno,natno) → natno

for all x,y ε natno let

ISZERO(ZERO) :: = true;
ADD(ZERO,y) :: =y;
ISZERO(SUCC(x))= false

D=NATNO

D ε D= {Boolean, natno}

F={ZERO,ISZERO,ADD,SUCC}

AXIOMS

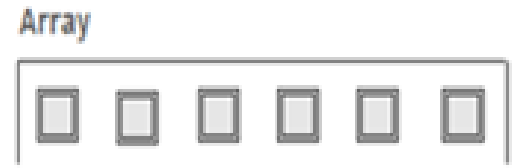
Types of Data Structures

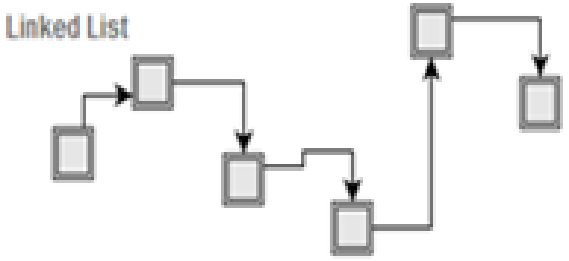
- **Linear data structure:**
  - The data structure where data items are organized sequentially or linearly where data elements attached one after another is called linear data structure. It has unique predecessor and Successor.
  - **Ex:** Arrays, Linked Lists

- **Non-Linear data structure:**
  - The data structure where data items are not organized sequentially is called non linear data structure. It don't have unique predecessor or Successor.
  - In other words, A data elements of the non linear data structure could be connected to more than one elements to reflect a special relationship among them.
  - **Ex:** Trees, Graphs

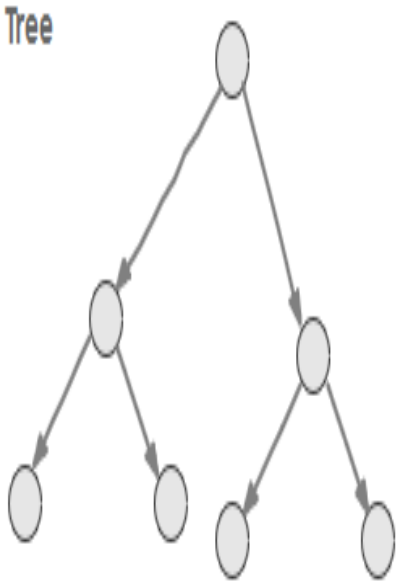
Types of Data Structures

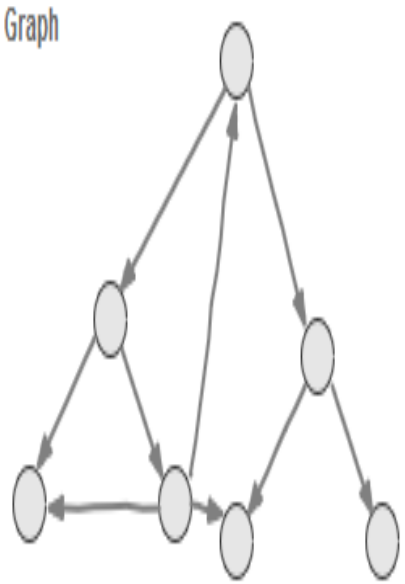
Linear Data Structure





Non Linear Data Structure





Types of Data Structures

- **Static data structure:**
  - Static Data structure has fixed memory size. It is the memory size allocated to data, which is static.
  - **Ex:** Arrays
- **Dynamic data structure:**
  - In Dynamic Data Structure, the size can be randomly updated during run time which may be considered efficient with respect to memory complexity of the code.
  - **Ex:** Linked List
- In comparison to dynamic data structures, static data structures provide easier access to elements. Dynamic data structures, as opposed to static data structures, are flexible.

Introduction to Algorithms

Algorithm

- Solution to a problem that is independent of any programming language.
- Sequence of steps required to solve the problem
- Algorithm is a finite set of instructions that if followed, accomplishes a particular task
- All algorithms must satisfy the following criteria:
  - **Input:** Zero or more Quantities are externally supplied
  - **Output**: At least one quantity is produced
  - **Definiteness**: Each instruction is clear and unambiguous
  - **Finiteness**: if we trace out the instructions of an algorithm then for all cases the algorithm terminates after a finite number of steps.
  - **Effectiveness**: Every instruction must be very basic so that it can be carried out in principle by a person using pencil and paper.

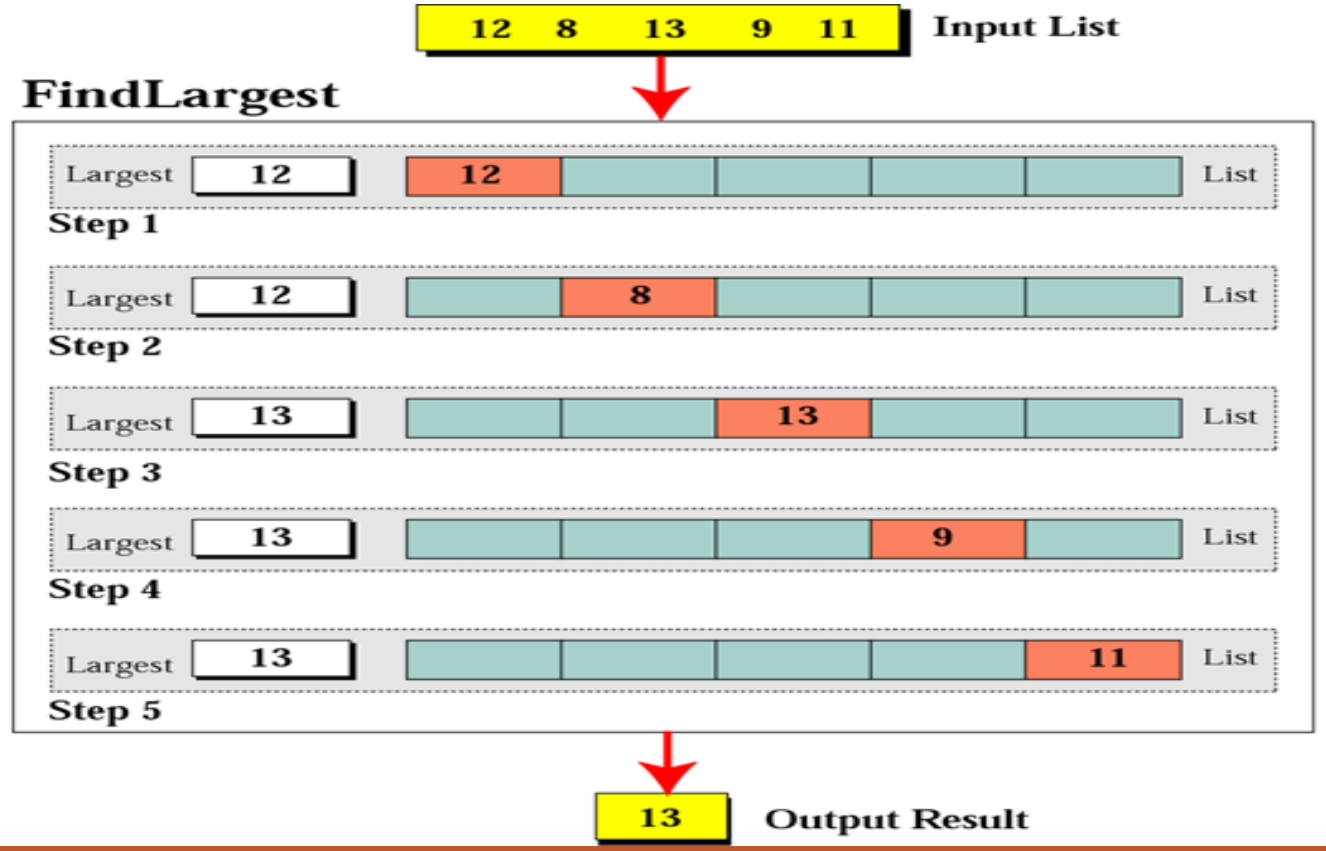
Introduction to Algorithms

Program vs Algorithm

- A program is a written out set of statements in a language that can be executed by the machine.
- An algorithm is simply an idea or a solution to a problem that is often procedurally written.

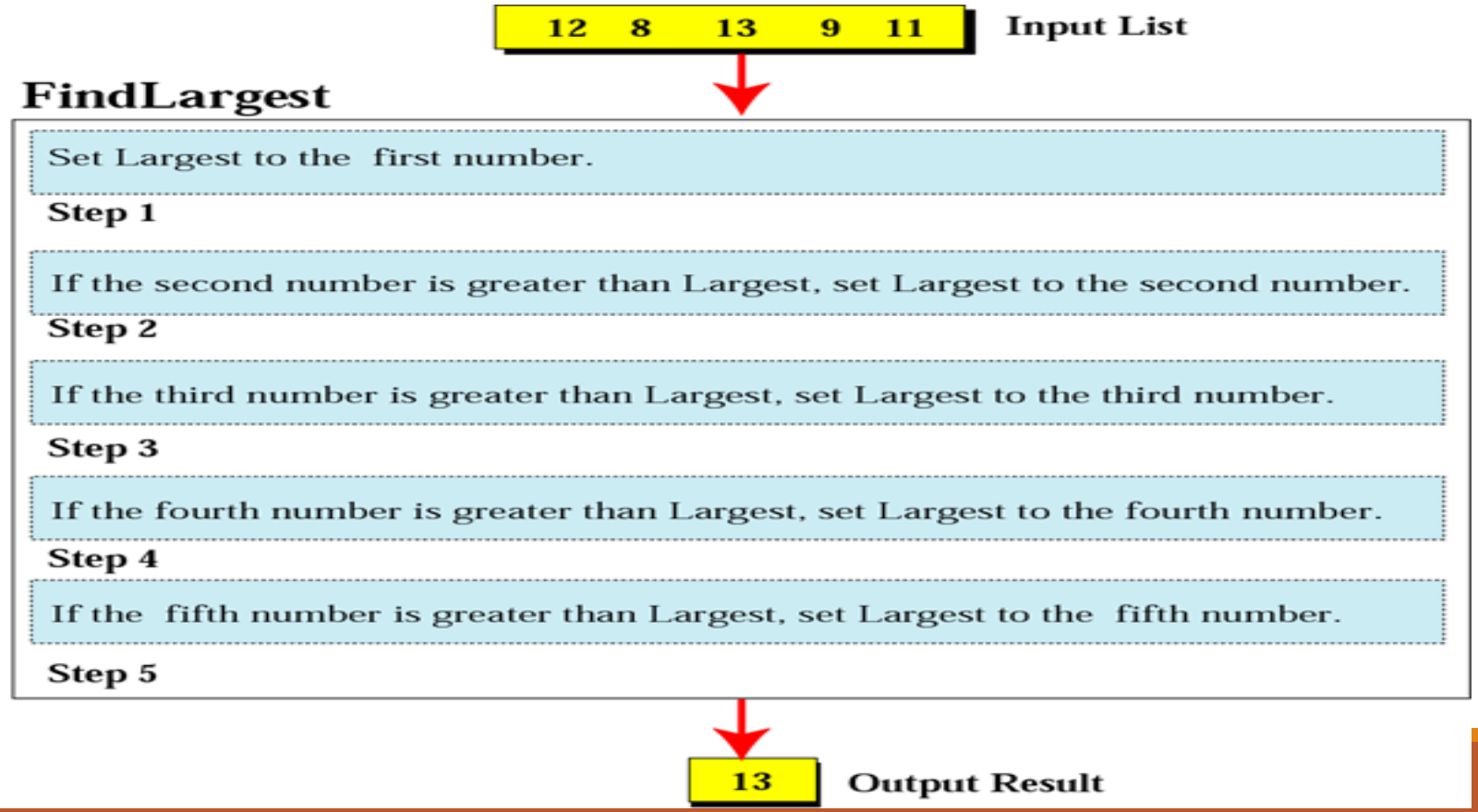
Introduction to Algorithms

Example : **Finding the largest integer among five integers**



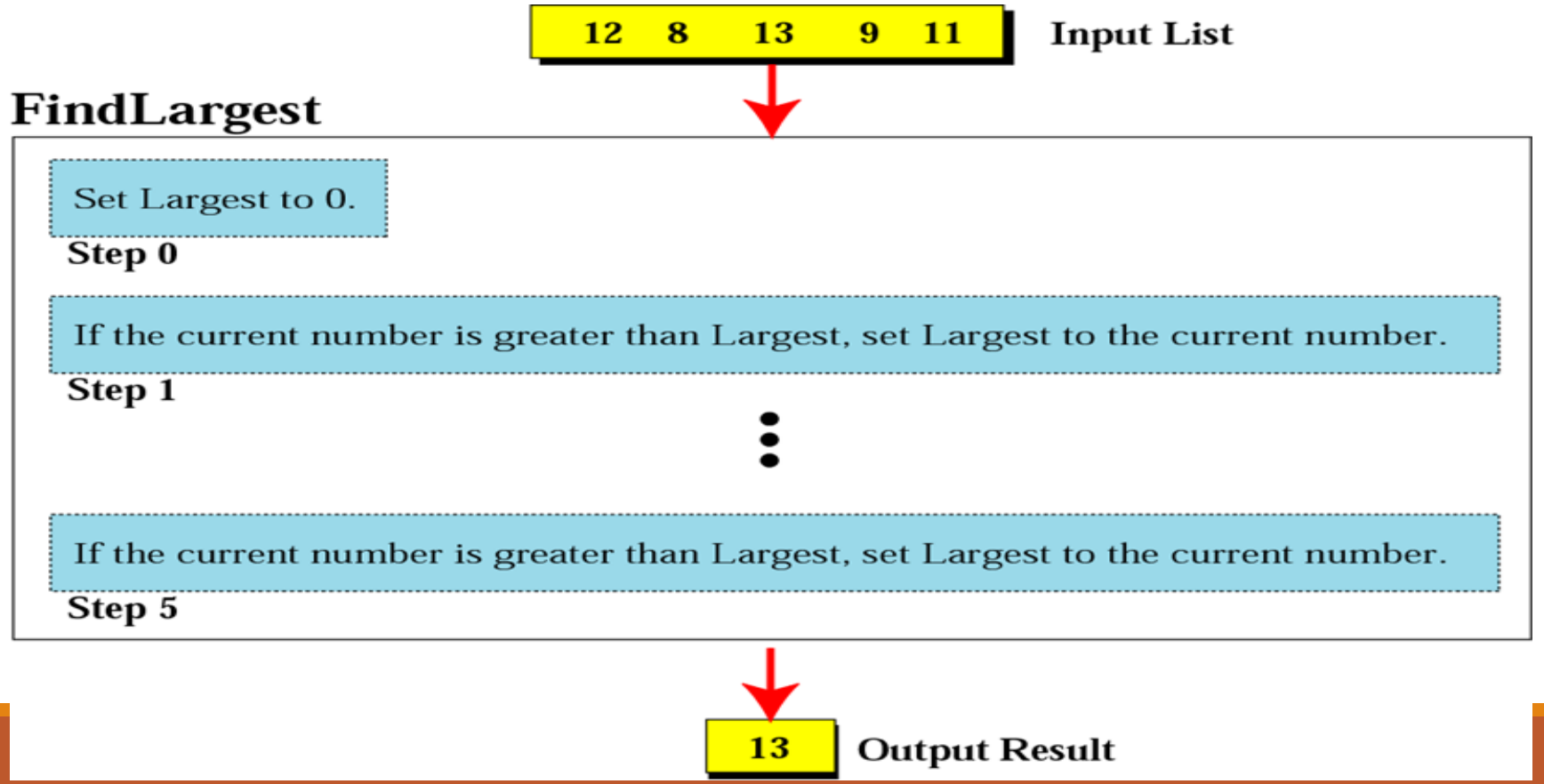
Introduction to Algorithms

Defining actions in Find Largest algorithm

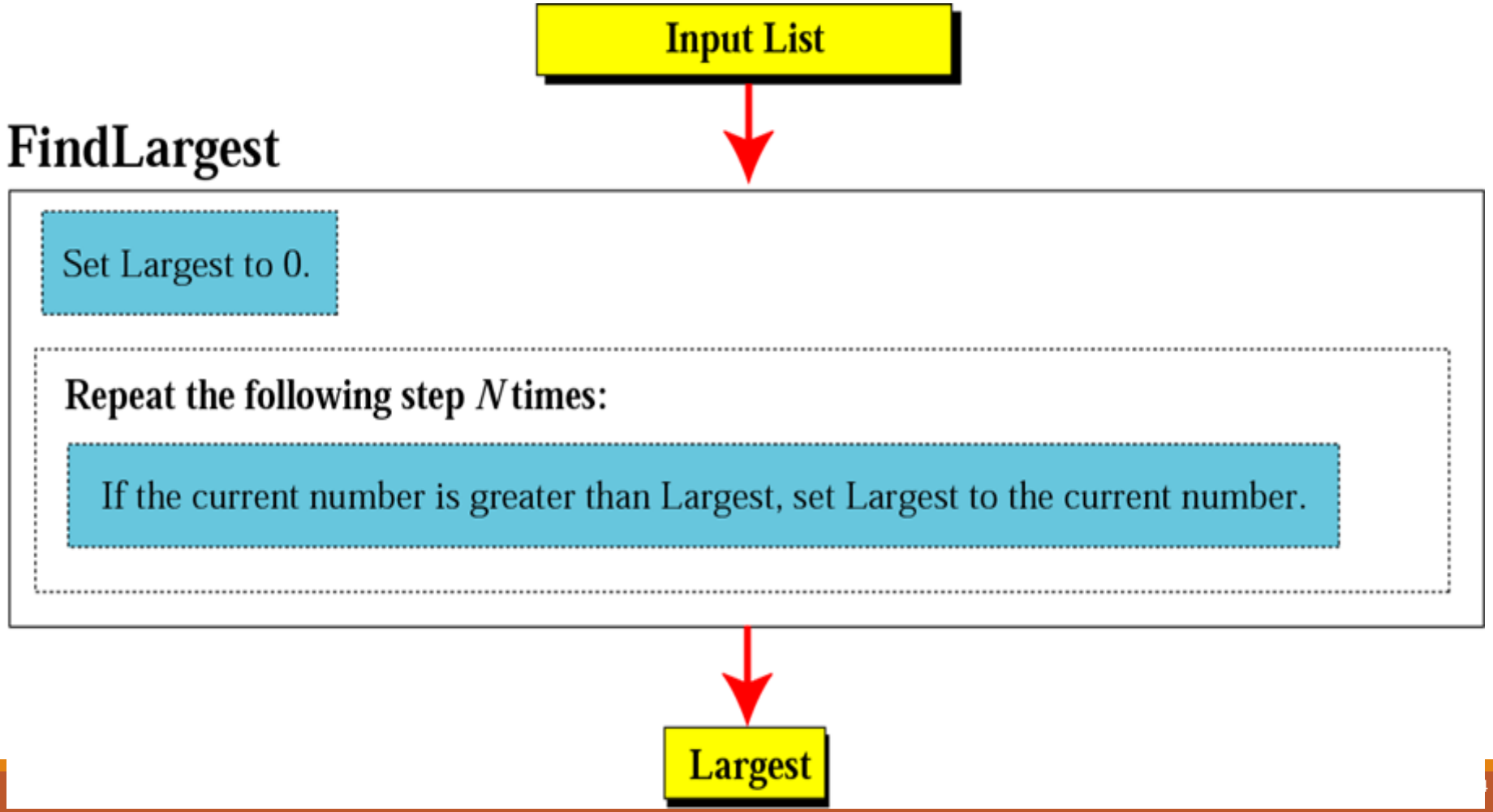


Introduction to Algorithms

Find Largest refined

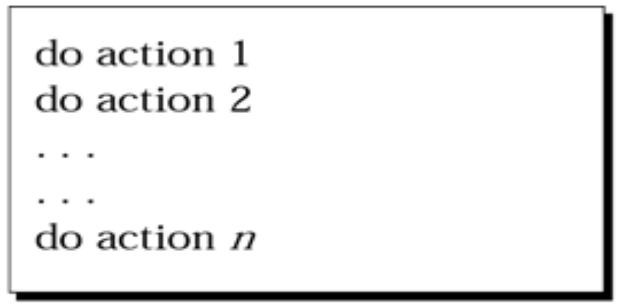


Introduction to Algorithms

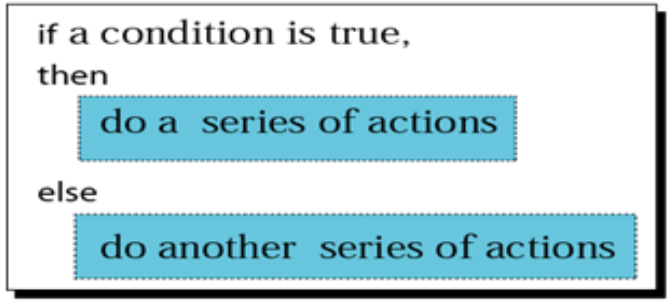


Introduction to Algorithms

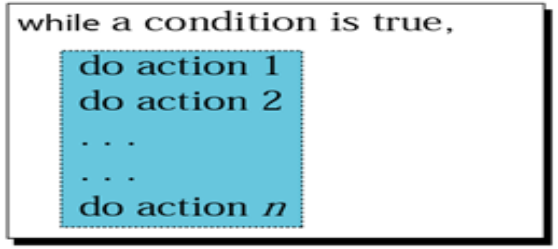
Three constructs



a. Sequence



b. Decision



c. Repetition

Algorithm Design Tools

➢ Pseudo Code

- is an artificial and informal language that helps programmers develop algorithms.
- Uses English-like phrases with some Visual Basic terms to outline the program

➢ Flowchart

- Graphical representation of an algorithm.
- Graphically depicts the logical steps to carry out a task and shows how the steps relate to each other.

Algorithm Design Tools

Example 1: Print 1 to 20:

Algorithm

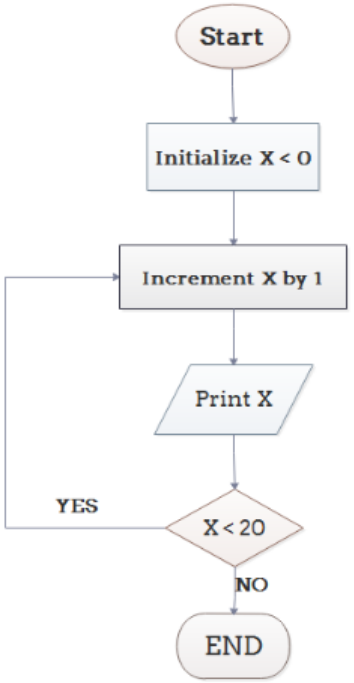
Step 1: Initialize X as 0,

Step 2: Increment X by 1,

Step 3: Print X,

Step 4: If X is less than 20 then go back to step 2.

Flowchart



Pseudo Code

- **Pseudocode** is an informal high-level description of the operating principle of a computer program or other algorithm.
- It uses the structural conventions of a normal programming language, but is intended for human reading rather than machine reading.

```
Algorithm SORT(A, n)
{
    for (i =1;i<n; i++)
            {
            j = i ;
            for (k = j+1;k<n; k++)
            {
               if A[k] <  A[j]
                        j=k;
          }
          t = A[i];
            A[i]= A[j];
           A[j]=t
           }
}
```

Pseudo Code

Examples

```
Algorithm grade_assignment( )
{
        if (student_grade >= 60)
                print "passed"
        else
                print "failed"
 }
```

Examples

```
Algorithm grade_count()
  {
  total=0
  grade_counter =1

   while (grade_counter<10)
   {
     read next grade
     total=total + grade
     grade_counter=grade_counter + 1

    }
  class_average=total/10
  print class_average.
  }
```

Pseudo Code

Some Keywords That Should be Used And Additional Points:

- Algorithm Keyword is used
- Curly brackets are used instead of begin-end
- Directly programming syntaxes are used
- Easy to convert into program
- Semicolons used

Pseudo Code

Some Keywords That Should be Used And Additional Points:

- Words such as set, reset, increment, compute, calculate, add, sum, multiply, ... print, display, input, output, edit, test , etc. with careful indentation tend to foster desirable pseudocode.

- Also, using words such as Set and Initialize, when assigning values to variables is also desirable.

Pseudo Code

Formatting and Conventions in Pseudo code

- INDENTATION in pseudocode should be identical to its implementation in a programming language.
- Use curly brackets for indentation
- No flower boxes (discussed ahead) in your pseudocode.
- Do not include data declarations in your pseudocode.
- But do cite variables that are initialized as part of their declarations. E.g. "initialize count to zero" is a good entry.

Pseudo Code

Calls to Functions should appear as:

Call FunctionName (arguments: field1, field2, etc.)

Returns in functions should appear as:

Return (field1)

Function headers should appear as:

FunctionName (parameters: field1, field2, etc.)

Functions called with addresses should be written as:

Call FunctionName (arguments: pointer to fieldn, pointer to field1, etc.)

Function headers containing pointers should be indicated as:

FunctionName (parameters: pointer to field1, pointer to field2, ...)

Returns in functions where a pointer is returned:

Return (pointer to fieldn)

Pseudo Code

1. Function Call

EVERY function should have a flowerbox PRECEDING IT.

This flower box is to include the functions name, the main purpose of the function, parameters it is expecting (number and type), and the type of the data it returns.

All of these listed items are to be on separate lines with spaces in between each explanatory item.

FORMAT of flowerbox should be

```
**************************************************
Function:   ( cryptic text describing single function
                        ..... (indented like this)
                        .....
            Calls:      Start listing functions "this" function calls
                        Show these functions:  one per line,
indented

            Called by:  List of functions that calls "this" function
                        Show these functions:  one per line,
indented.

            Input Parameters:  list, if appropriate; else None

            Returns:    List, if appropriate.
**************************************************
```

Pseudo Code

➢ **Advantages and Disadvantages**

Pseudocode Disadvantages

- It's not visual
- There is no accepted standard, so it varies widely from company to company

Pseudocode Advantages

- Can be done easily on a word processor
- Easily modified
- Implements structured concepts well

Analysis of Algorithms

- Finding Efficiency of an algorithm in terms of
  - Time Complexity
  - Space Complexity

Analysis of Algorithms

- **What is time complexity**
  - Finding amount of time required for executing set of instructions or functions
  - It is represented in terms of frequency count
  - Frequency count is number of time every instruction of a code is to be executed.

- **What is space complexity**
  - Finding amount of memory space the program is going to consume.
  - It is calculated in terms of variables used in program.

Common Rates of Growth

Let n be the size of input to an algorithm, and k some constant. The following are common rates of growth.

- Constant: O(k), for example O(1)
- Linear: O(n)
- Logarithmic: $O(\log_k n)$
- Linear : n O(n) or n log n: $O(n \log_k n)$
- Quadratic: $O(n^2)$
- Polynomial: $O(n^k)$
- Exponential: $O(k^n)$

Example

```
void fun()
{
  int a;
   a=5;
   printf(“%d”,a);
}
```

```
void fun()
{
  int a;
   a=0;
   for(i=0;i<n;i++)
    {
        a=a + i;
     }
   printf(“%d”,a);
}
```

Solving Problems

Find Frequency Count and Time Complexity

```
for(i=1;i<=n;i++)
{
        for(j=1;j<=n;j++)
        {
                C[j][j]=0;
              for(k=1;k<=n;k++)

C[i][j]=c[i][j]+a[i][k]*b[k][j];
        }
}
```

```
i=1;
do
{
   a++;
   if(i==5)
     break;
   i++;
}
while(i<=n);
```

```
i=1;
while(i<=n)
{
   a++;
   if(i==5)
     break;
     i++;
}
```

Find Frequency Count and Time Complexity

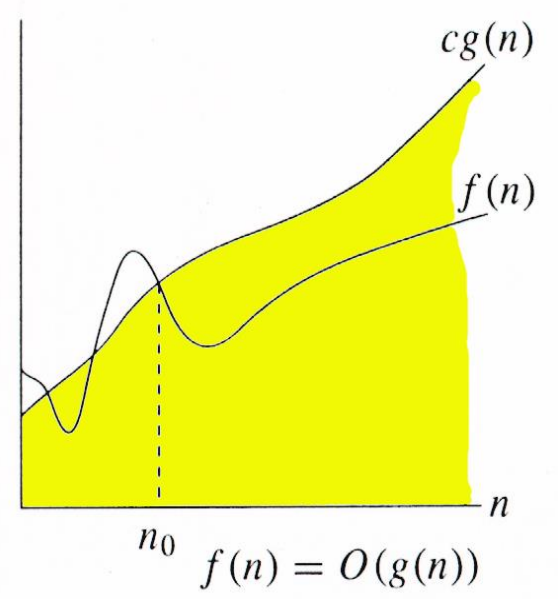
```
i=n;
while(i>=1){
        i--;
}
```

```
i=10;
for(i=10;i<=n;i++)
                for(j=1;j<i;j++)
                                x=x+1;
```

```
i=0;

for(i=0;i<=n;i++)

for(j=1;j<i;j++)

x=x+1;
```

O-notation Example

For a given function g(n), we denote O(g(n)) as the set of
functions:
O(g(n)) = { f(n)| there exists positive constants c and n0 such that $0 \leq f(n) \leq c\, g(n)$ for all $n \geq n0$ }
It is used to represent the worst case growth of an algorithm in time or a data structure in space when they are dependent on n, where n is big enough.



Big Oh - Example

$f(n) = n^2 + 5n = O(n^2)$

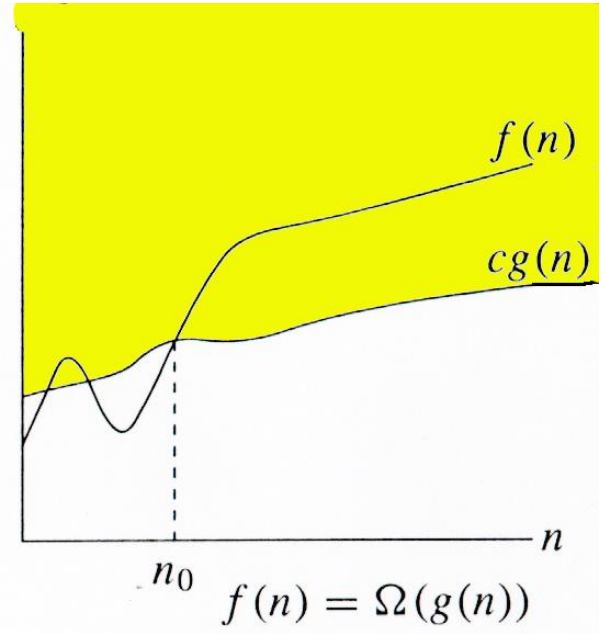
$g(n) = n^2$ ……. c = 2

n	$n^2 + 5n$	$2n^2$
1	5	2
2	14	8
5	50	50

$f(n) <= c\ g(n)$ for all $n> = n_0$ where c=2 & n_0 =5

Ω -notation

Ω (g(n)) represents a set of functions such that:
Ω(g(n)) = {f(n): there exist positive constants c and n0 such that 0 ≤ c g(n) ≤ f(n) for all n≥ n0}



Big Omega - Example

Example 1 :

$f(n) = n^2 + 5n$

$g(n) = n^2$ ……. c = 1

n	$n^2 + 5n$	$c*n^2$
1	5	1
2	14	4
5	50	25

$f(n) >= c\ g(n)$ for all $n> = n_0$
where $c=1$ & $n_0 =1$

Example 1 :

Prove that if $T(n) = 15n^3 + n^2 + 4$, $T(n) = \Omega\ (n^3)$.

Proof.

Let $c = 15$ and $n_0 = 1$.

Must show that $0 \leq cg(n)$ and $cg(n) \leq f\ (n)$.

$0 \leq 15n^3$ for all $n \geq n_0 = 1$.

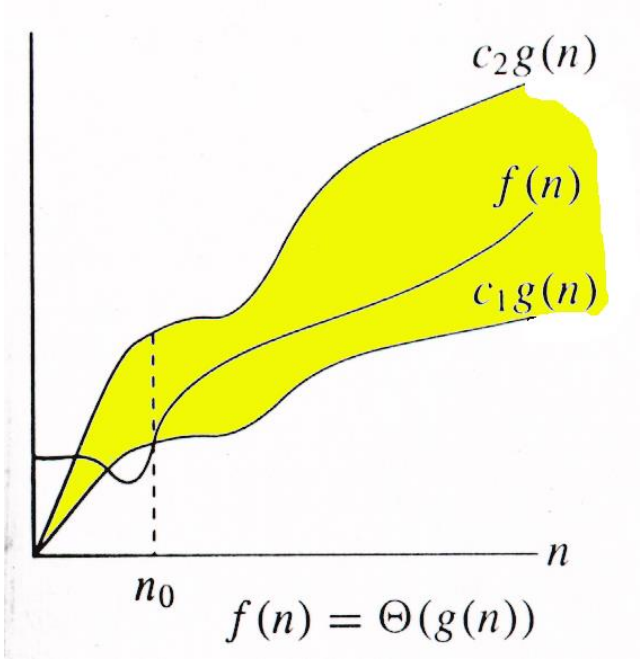
$cg(n) = 15n^3 \leq 15n^3 + n^2 + 4 = f\ (n)$

Θ-notation

Asymptotic tight bound

Θ (g(n)) represents a set of functions such that:

Θ (g(n)) = {f(n): there exist positive constants c1, c2, and n0such that $0 \leq c1g(n) \leq f(n) \leq c2g(n)$ for all $n \geq n0$ }



Theta Example

$f(N) = \Theta(g(N))$ iff $f(N) = O(g(N))$ and $f(N) = \Omega(g(N))$
It can be read as "f(N) has order exactly g(N)".

The growth rate of f(N) equals the growth rate of g(N). The growth rate of f(N) is the same as the growth rate of g(N) for large N.

Theta means the bound is the tightest possible.

If T(N) is a polynomial of degree k, $T(N) = \Theta(N^{\mathbf{k}})$.

For logarithmic functions, $T(\log_{\mathbf{m}} N) = \Theta(\log N)$.

Analysis of Algorithms

- Algorithm analysis is done in following three cases

  - Best Case

    The amount of time a program might be expected to take on best possible input data

  - Worst Case

    The amount of time a program would take on worst possible input configuration.

  - Average case

    The amount of time a program might be expected to take on typical(or average) input data

Example: Sorting Algorithms

Practice Assignments

1. Write a pseudo code and draw flowchart to input any alphabet and check whether it is vowel or consonant.
2. Write a pseudo code to check whether a number is even or odd
3. Write a pseudo code to check whether a year is leap year or not.
4. Write a pseudo code to check whether a number is negative, positive or zero
5. Write a pseudo code to input basic salary of an employee and calculate its Gross salary according to following:

   Basic Salary <= 10000 : HRA = 20%, DA = 80%
   Basic Salary <= 20000 : HRA = 25%, DA = 90%
   Basic Salary > 20000 : HRA = 30%, DA = 95%

Practice Problems

Q.1 Determine frequency count of following statements? Analyze time complexity of the following code:

```
i) for (i=1;i<=n;i++)
            for (j=1;j<=m;j++)
      sum=sum+i;
ii) i=n;
while(i>=1)
{
i--;
}
```

Practice Problems

Problems on frequency count & time complexity

```
for(i=1;i<=n;i++)
{
For(j=1;j<=n;j++)
{
C[j][j]=0;
For(k=1;k<=n;k++)
C[i][j]=c[i][j]+a[i][k]*b[k][j];
}
}
```

```
double IterPow(double X,int N)
{
double Result=1;
while(N>0)
{
Result=Result* X
N--;
}
return result;
```

Practice Problems

Q.3 What is the frequency count of a statement? Analyze time complexity of following code?

```
for(i=1;i<=n;i++)
    for(j=1;j<=m;j++)
        for(k=1;k<=p;k++)
{
            Sum=sum+i
}
```

Takeaway

- Data Structures plays major role in problem solving.
- Pseudo code and flowcharts are the tools used to represent the solution of a problem in effective way.
- Analysis of algorithms is done in terms of time complexity and space complexity.

Subject : Data Structures

Unit 1

Part 2: Sorting

S. Y. B. Tech CSE

Semester – III

SCHOOL OF COMPUTER SCIENCE & ENGINEERING
DEPARTMENT OF COMPUTER ENGINEERING & TECHNOLOGY

Contents

- Sorting: Types of Sorting-Internal and External Sorting
- General Sort Concepts-Sort Order, Stability, Efficiency and Number of Passes
- Comparison Based Sorting Methods-Bubble Sort
- Insertion Sort
- Selection Sort
- Shell Sort
- Comparison of all sorting Algorithm

‘

Sorting

General Sort Concepts

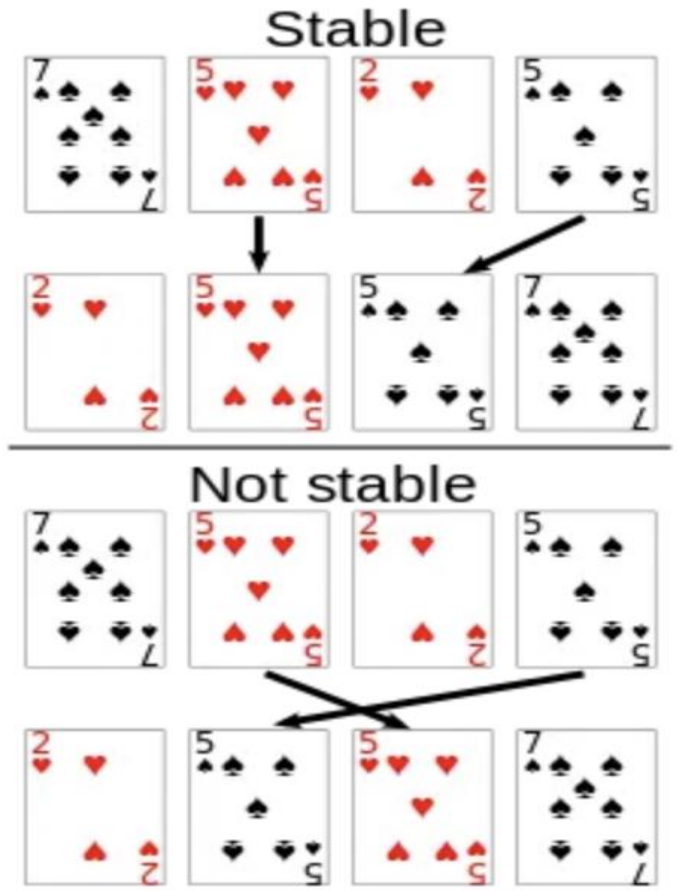
Sort Order :

- Data can be ordered either in ascending order or in descending order
- The order in which the data is organized, either ascending order or descending order, is called sort order

Sort Stability

- **A sorting method** is said to be stable if at the end of the method, identical elements occur in the same relative order as in the original unsorted set
- Sort Efficiency
- Sort efficiency is a measure of the relative efficiency of a sort

Stable and Not stable

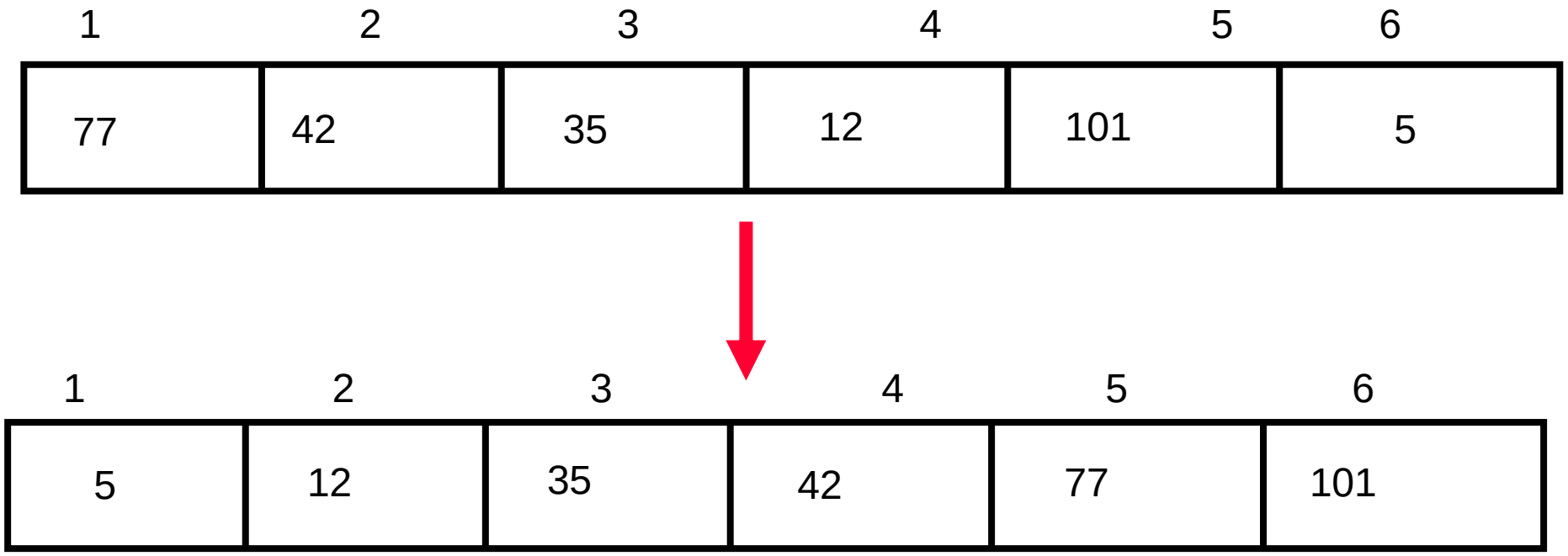


Passes

- During the sorted process, the data is traversed many times
- Each traversal of the data is referred to as a sort pass
- In addition, the characteristic of a sort pass is the placement of one or more elements in a sorted list

Sorting

Sorting takes an unordered collection and makes it an ordered one.



Sorting

- ***Sorting*** is a process that organizes a collection of data into either ascending or descending order.
- An ***internal sort*** requires that the collection of data fit entirely in the computer's main memory.
- We can use an ***external sort*** when the collection of data cannot fit in the computer's main memory all at once but must reside in secondary storage such as on a disk.

In Place Sort

- The amount of extra space required to sort the data is constant with the input size.

Stable sort algorithms

- A stable sort keeps equal elements in the same order
- This may matter when you are sorting data according to some characteristic

  Example: sorting students by test scores

Ann	98
Bob	90
Dan	75
Joe	98
Pat	86
Sam	90
Zöe	86

original array

Ann	98
Joe	98
Bob	90
Sam	90
Pat	86
Zöe	86
Dan	75

stably sorted

Unstable sort algorithms

- An unstable sort may or may not keep equal elements in the same order
- Stability is usually not important, but sometimes it is important

Ann	98
Bob	90
Dan	75
Joe	98
Pat	86
Sam	90
Zöe	86

original array

Joe	98
Ann	98
Bob	90
Sam	90
Zöe	86
Pat	86
Dan	75

unstably sorted

Internal and External Sorting

- If data is sorted in particular order then searching becomes easier .
- Sorting is organizing data in ascending or descending order .
- If computer needs to sort the data , it uses two techniques
- Internal and External Sorting

Comparison

Sr No	Internal Sorting	External Sorting
1	In internal sorting , all the data to sort is stored in main memory at all times while sorting is in progress	In External sorting , the data is stored outside memory(like on disk) and only loaded into memory in small chunks
2	It is performed when the data to be sorted is small enough to fit in main memory	It is used when data cannot fit into main memory entirely
3	In other words, it is used when the size of input is small	It is used when size of input is large
4	In this method, the storage devices is main memory (RAM) Ex: Bubble sort, insertion sort, quick sort, heapsort,etc..	In this method, storage device are secondary memory (hard disk) and main memory Ex: External Merge sort ,etc…

Internal Sorting

Rollno	Name	Marks
1	Piyush	90
2	Chaitanya	97
3	Aditya	98
4	Jayesh	96
5	Jitendra	99

Main Memory

5 Records can be stored in Memory

Internal Sorting :Records are fetched into main memory and then can perform the sorting

External Sorting

Rollno	Name	Marks
11	Piyush	99
24	Chaitanya	97
3	Aditya	98
4	Jayesh	96
55	Jitendra	95
6	Ayesha	94
77	Prathemesh	90
8	Sushmita	89
9	Neha	88
10	Shubham	85
.		
.		
.		
.		
200	Danish	70

Main Memory

5 Records can be stored in Memory

Block 1
5 Records in each block

Block 2
5 Records in each block

Block 3
5 Records in each block

Block 4
5 Records in each block

Types of Sorting Algorithms

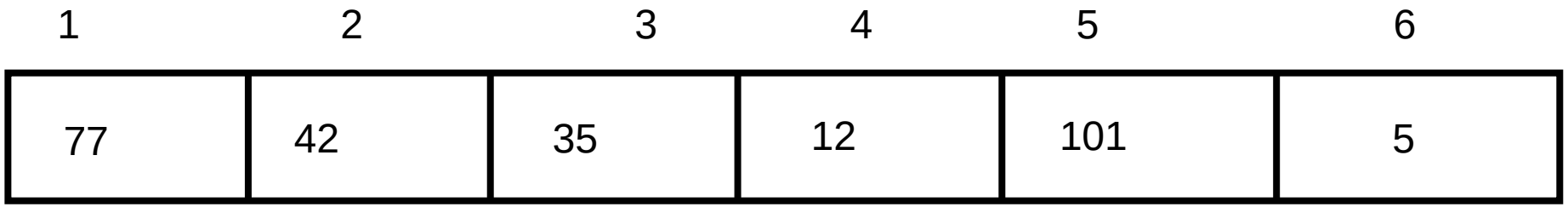
There are many, many different types of sorting algorithms, but the primary ones are:

- Bubble Sort
- Selection Sort
- Insertion Sort
- Merge Sort
- Shell Sort
- Quick Sort
- Radix Sort

Bubble sort("Bubbling Up" the Largest Element)

Traverse a collection of elements

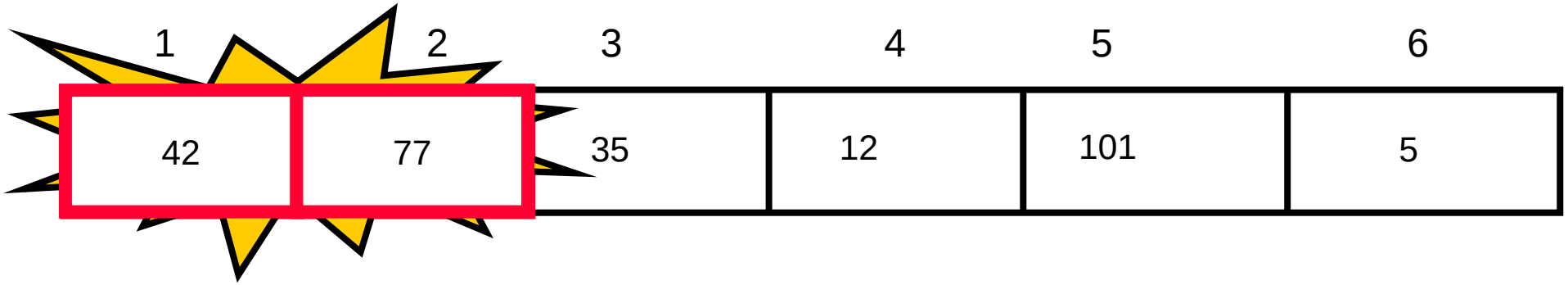
- **Move from the front to the end**
- **“Bubble” the largest value to the end using pair-wise comparisons and swapping**



"Bubbling Up" the Largest Element

Traverse a collection of elements

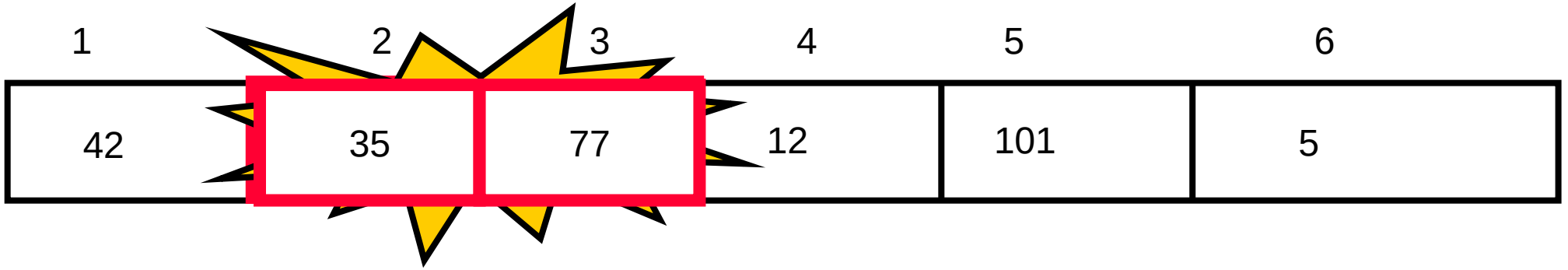
- **Move from the front to the end**
- **“Bubble” the largest value to the end using pair-wise comparisons and swapping**



"Bubbling Up" the Largest Element

Traverse a collection of elements

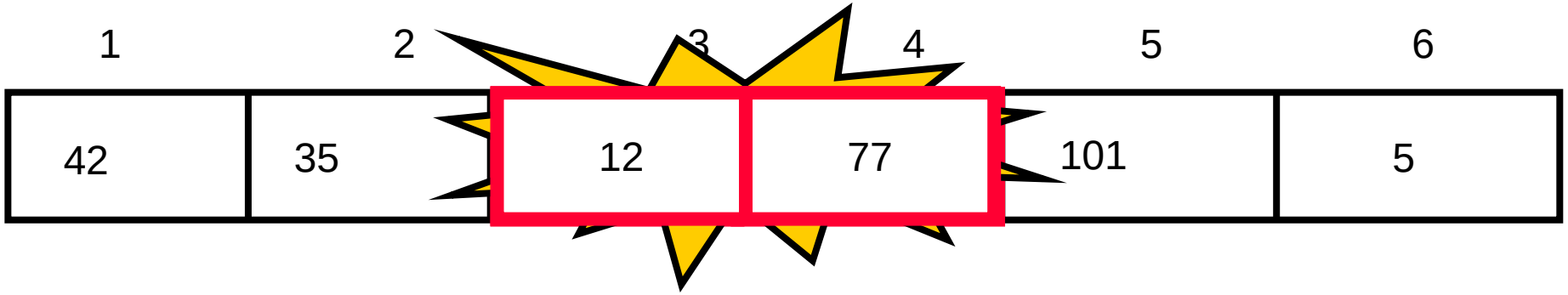
- **Move from the front to the end**
- **“Bubble” the largest value to the end using pair-wise comparisons and swapping**



"Bubbling Up" the Largest Element

Traverse a collection of elements

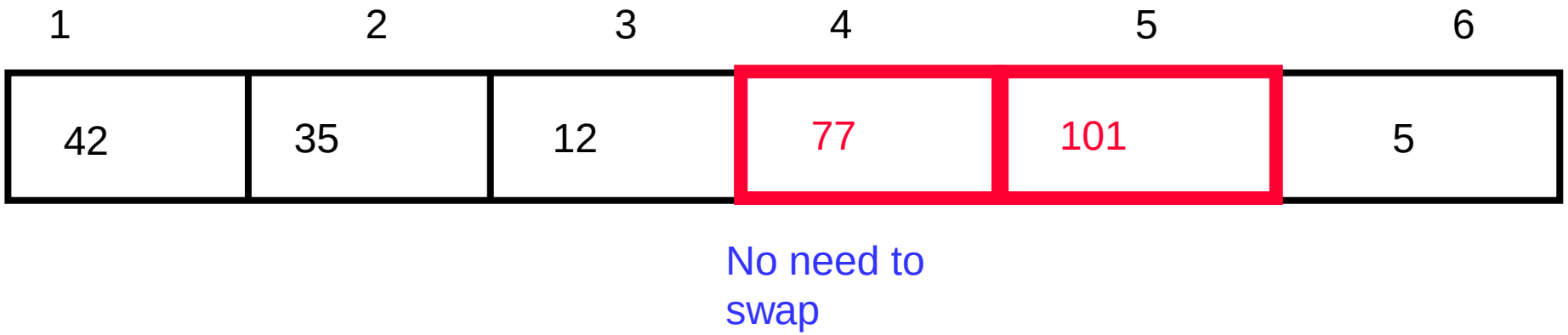
- **Move from the front to the end**
- **“Bubble” the largest value to the end using pair-wise comparisons and swapping**



"Bubbling Up" the Largest Element

Traverse a collection of elements

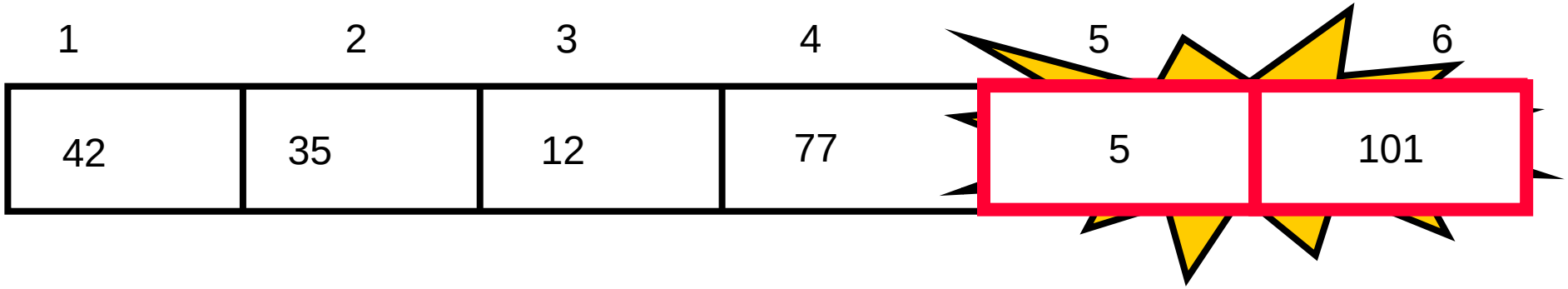
- **Move from the front to the end**
- **“Bubble” the largest value to the end using pair-wise comparisons and swapping**



"Bubbling Up" the Largest Element

Traverse a collection of elements

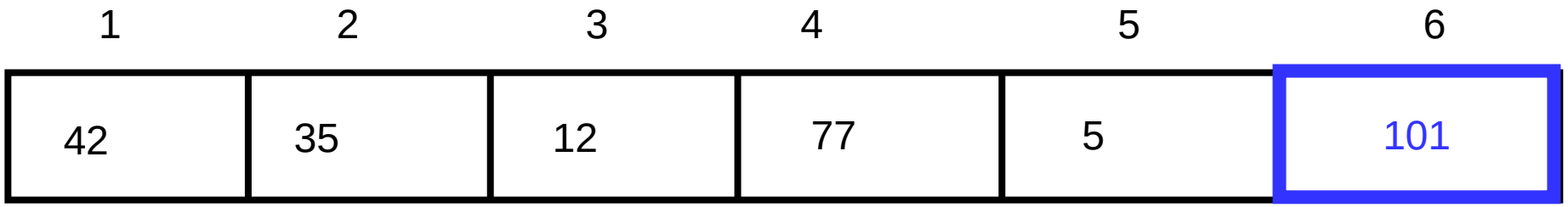
- **Move from the front to the end**
- **“Bubble” the largest value to the end using pair-wise comparisons and swapping**



"Bubbling Up" the Largest Element

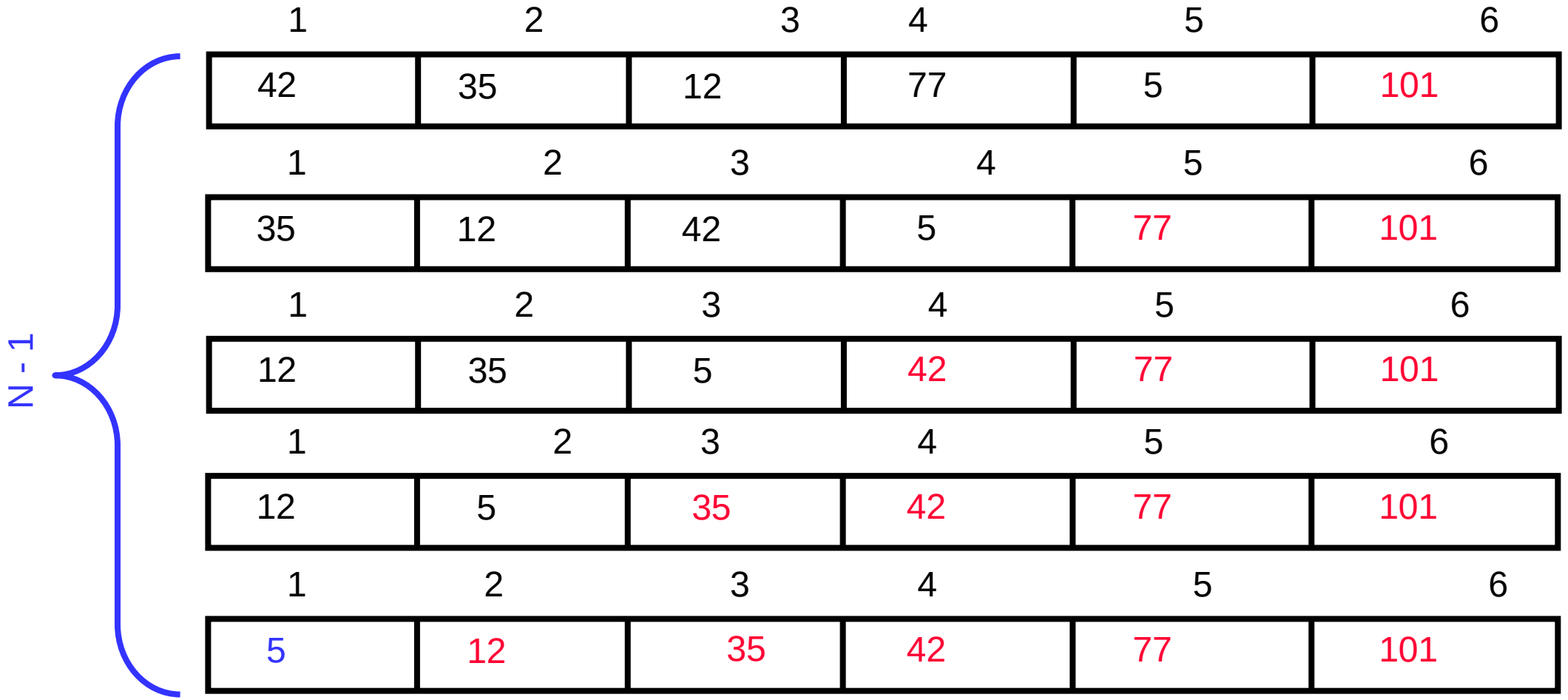
Traverse a collection of elements

- **Move from the front to the end**
- **“Bubble” the largest value to the end using pair-wise comparisons and swapping**

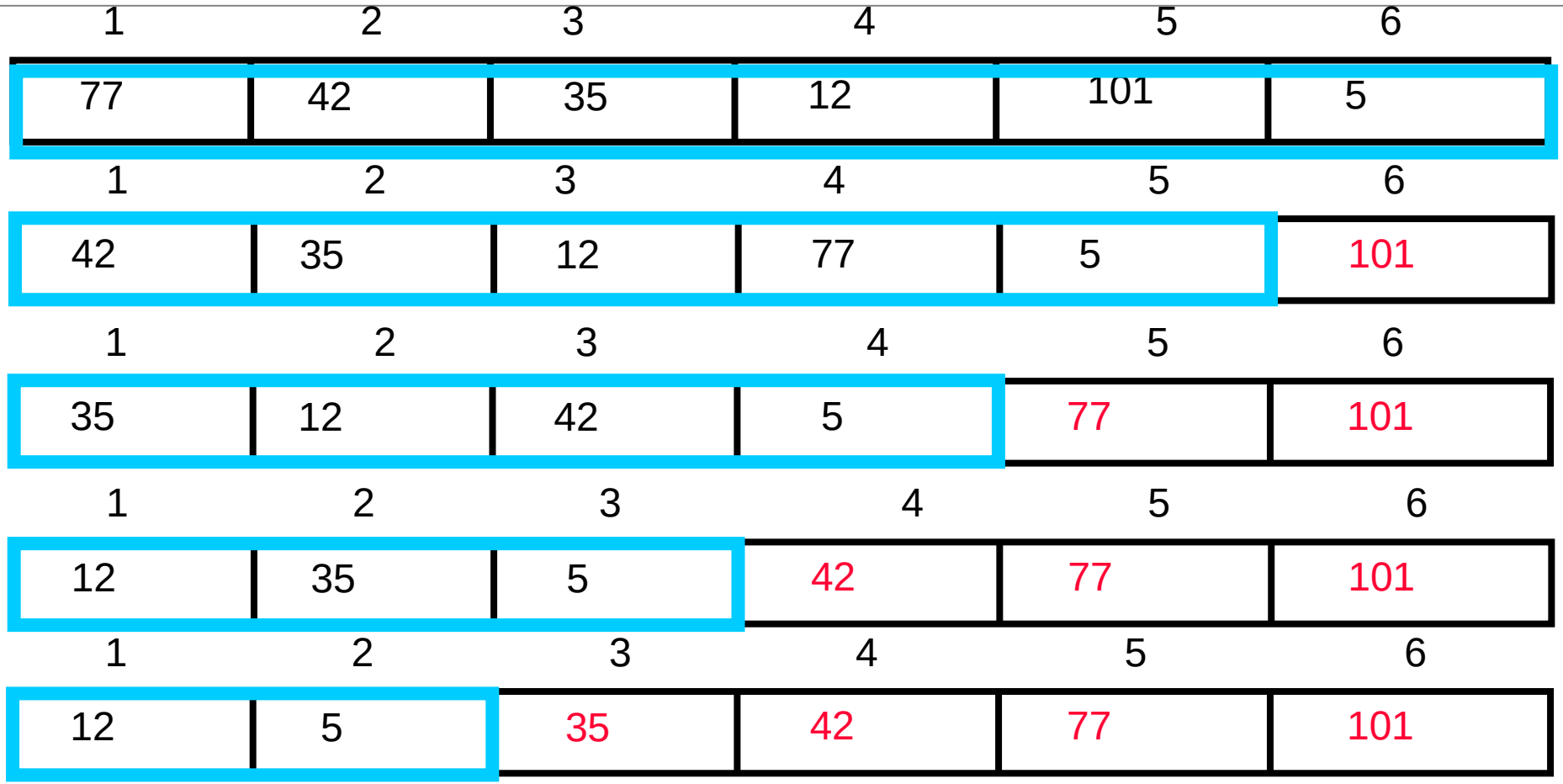


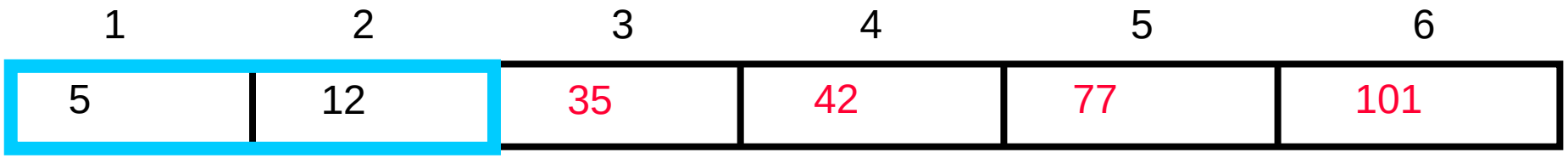
Largest value correctly placed

“Bubbling” All the Elements



Reducing the Number of Comparisons




1
2
3
4
5
6
5
12
35
42
77
101

Pass 1 : Total Comparisons : 4

0	1	2	3	4	Compare
55	44	33	22	11	A[0]>A[1]
44	55	33	22	11	A[1]>A[2]
44	33	55	22	11	A[2]>A[3]
44	33	22	55	11	A[3]>A[4]
44	33	22	11	55	

- Pass 2: Total Comparisons :3

0	1	2	3	4	Compare
44	33	22	11	55	A[0]>A[1]
33	44	22	11	55	A[1]>A[2]
33	22	44	11	55	A[2]>A[3]
33	22	11	44	55	

- Pass 3 : Total Comparisons : 2

0	1	2	3	4	Compare
33	22	11	44	55	A[0]>A[1]
22	33	11	44	55	A[1]>A[2]
22	11	33	44	55	

- Pass4: Total Comparison : 1

0	1	2	3	4	Compare
22	11	33	44	55	A[0]>A[1]
11	22	33	44	55	

Bubble sort

```
  Algorithm  bubble()
{
        for  i =1 to n
         {
          for   j=0   to n-i-1
          {
                if  a[j]>a[j+1]
                {
                  swap(a[j],a[j+1])
             }
          }
      display(a,n); }
```

MIT-WPU
।। विश्वशान्तिर्ध्रुवं ध्रुवा ।।

Analysis of Bubble Sort

The time complexity for each of the cases is given by the following:

Average-case complexity = O(n^2)

Best-case complexity = O(n^2)

Worst-case complexity = O(n^2)

Selection Sort

Idea:

- Find the smallest element in the array
- Exchange it with the element in the first position
- Find the second smallest element and exchange it with the element in the second position
- Continue until the array is sorted

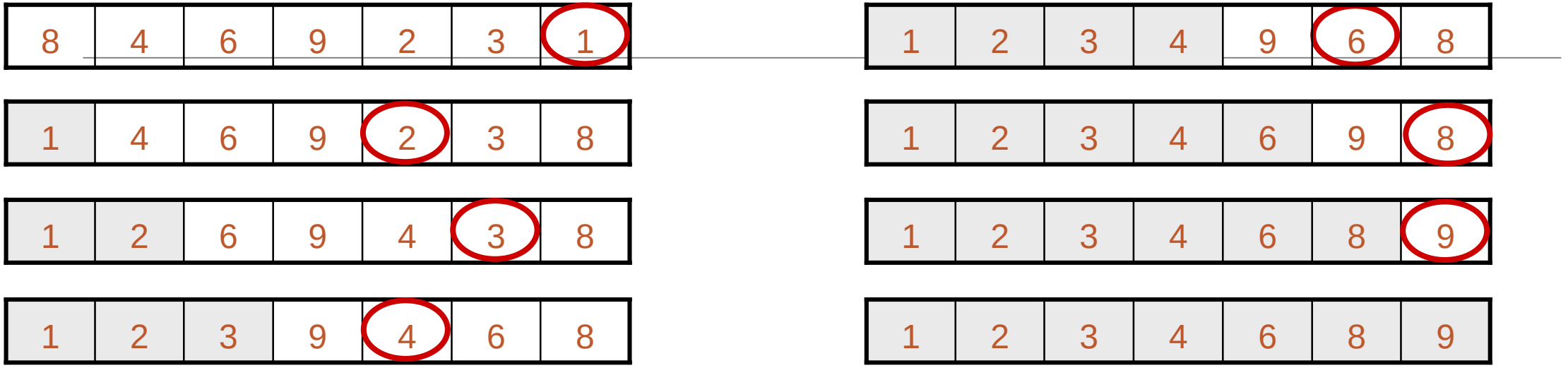
Disadvantage:

- Running time depends only slightly on the amount of order in the file

Selection Sort

```
Void selectionsort()
{
    for i= 0 to n-2
     {
          minpos=i;
         for  j=i+1 to n-1
        {
              if a[j]<a[minpos]
               { minpos = j; }
}
        if (minpos!=i )
{
     temp=a[i];  a[i]=a[minpos];a[minpos]=temp;
}}
```

Example


8 4 6 9 2 3 1
1 4 6 9 2 3 8
1 2 6 9 4 3 8
1 2 3 9 4 6 8
1 2 3 4 9 6 8
1 2 3 4 6 9 8
1 2 3 4 6 8 9
1 2 3 4 6 8 9

Analysis of Selection Sort

Best Case: O(n2)

Worst Case:O(n2)

Average case :O(n2)

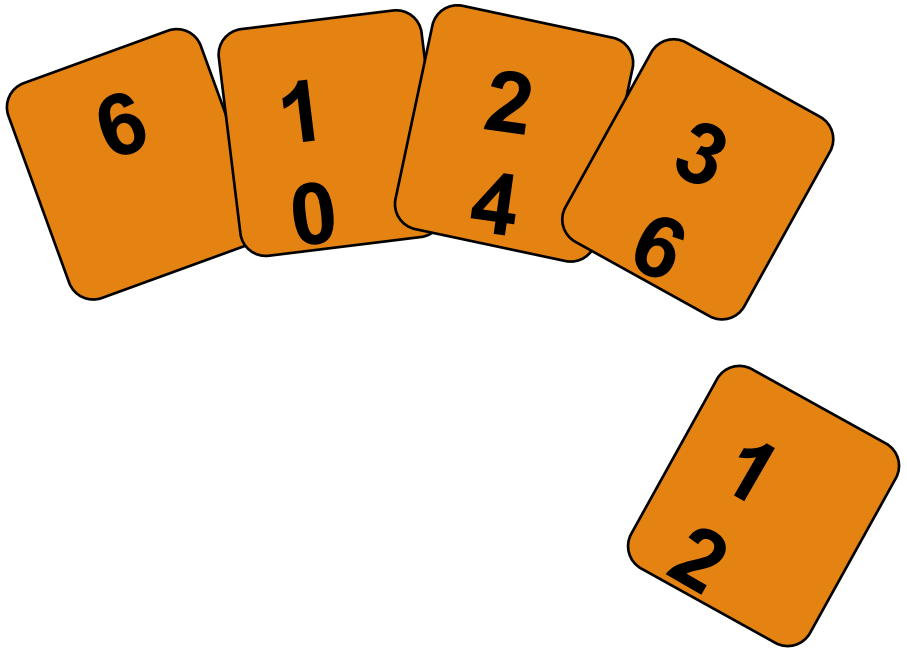
- **not stable**
- **In place sort**

Insertion Sort

Idea: like sorting a hand of playing cards

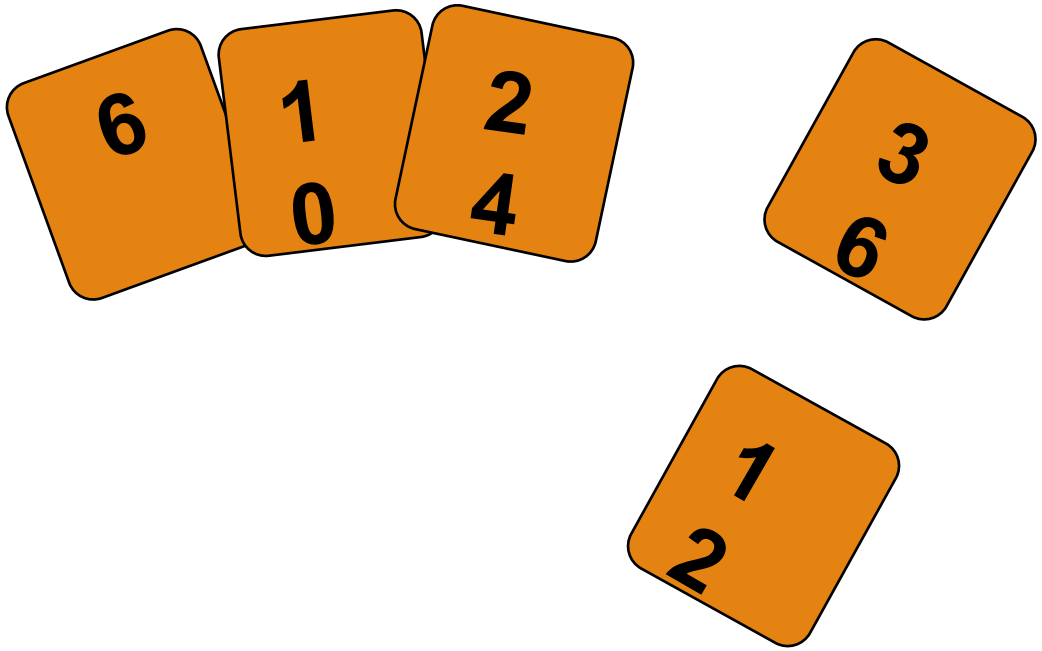
- Start with an empty left hand and the cards facing down on the table.
- Remove one card at a time from the table, and insert it into the correct position in the left hand
  - compare it with each of the cards already in the hand, from right to left
- The cards held in the left hand are sorted
  - these cards were originally the top cards of the pile on the table

Insertion Sort

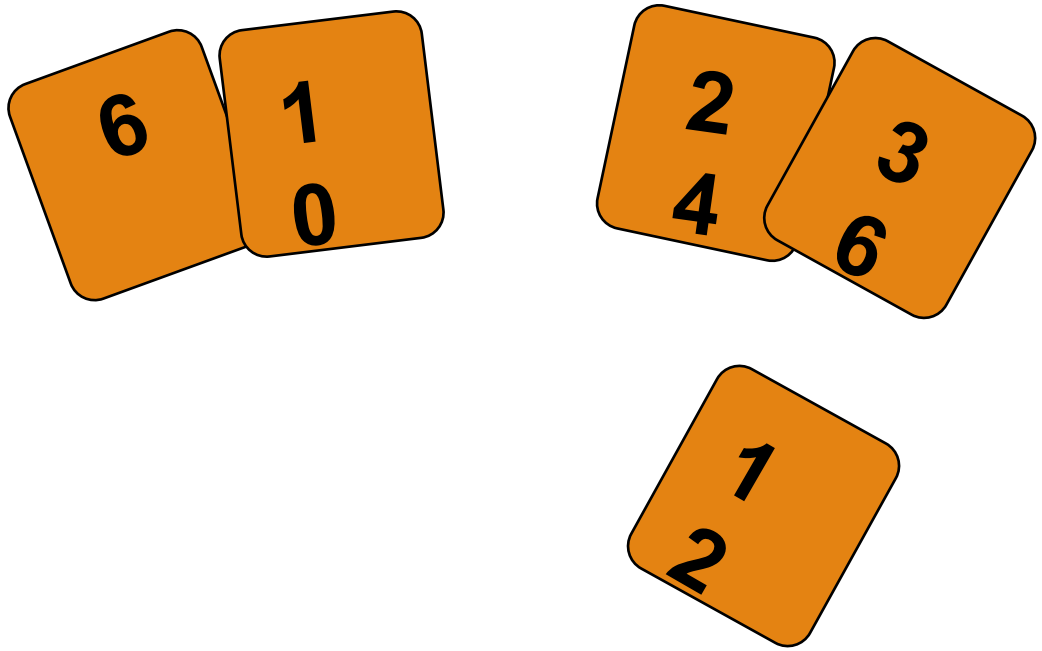


To insert 12, we need to make room for it by moving first 36 and then 24.

Insertion Sort



Insertion Sort

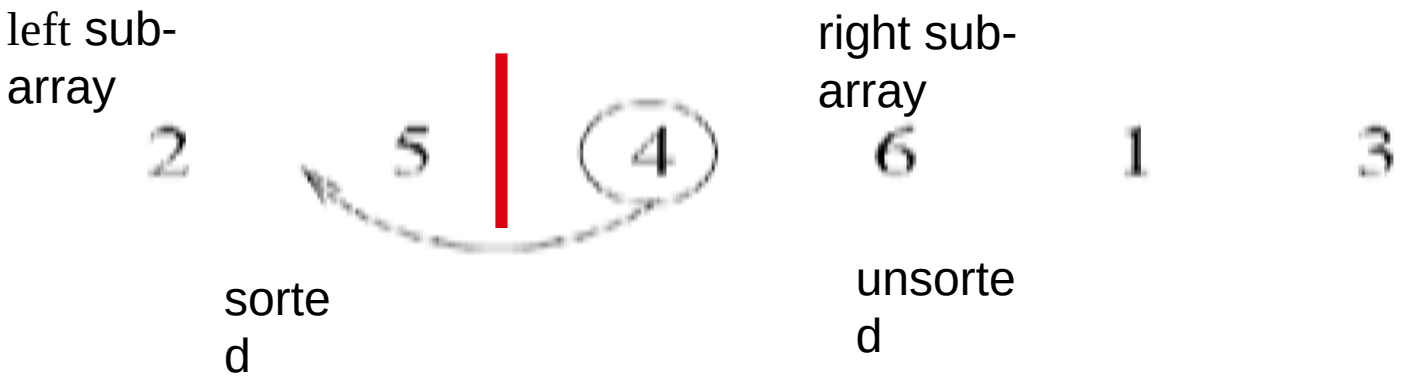


Insertion Sort

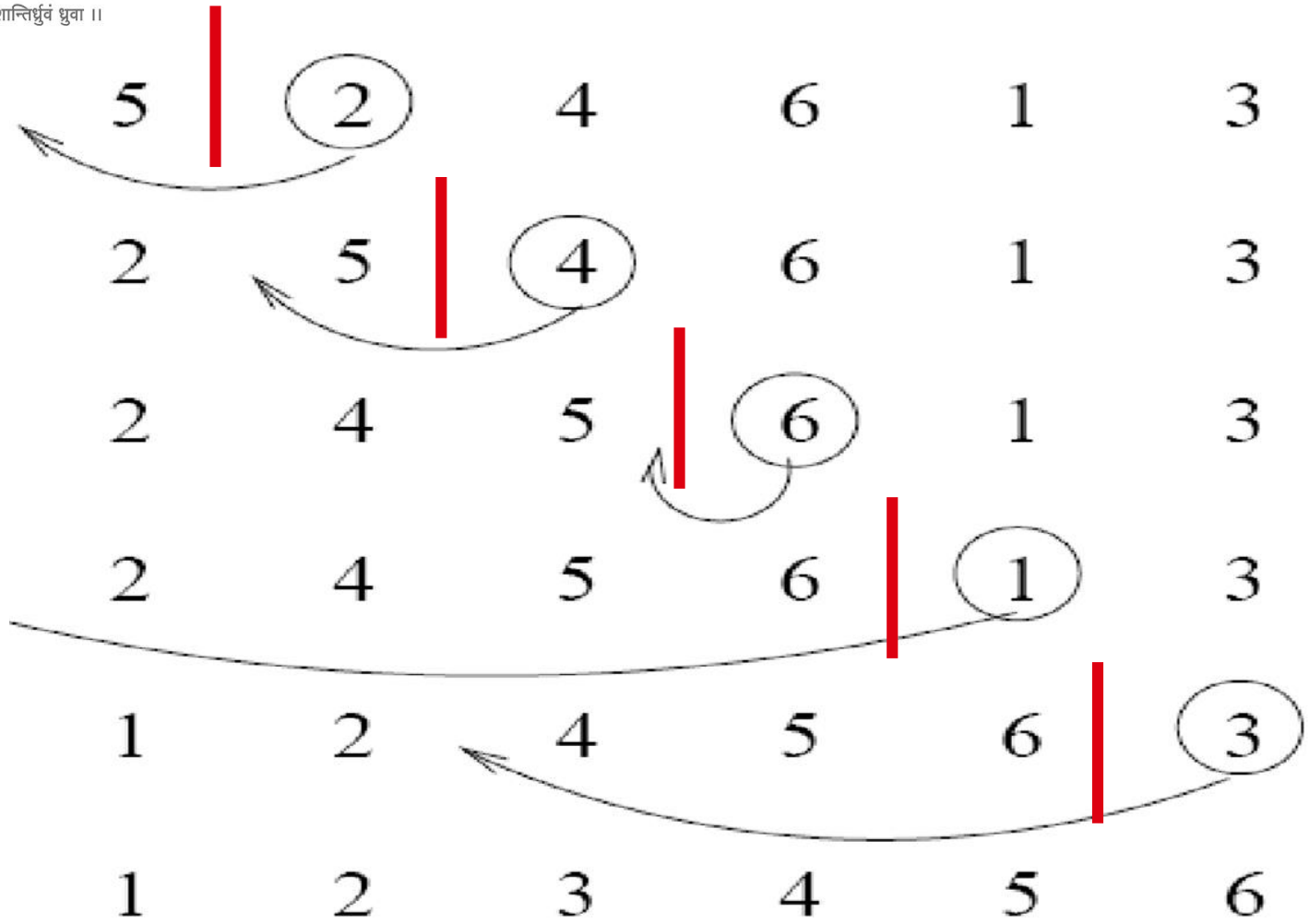
input array

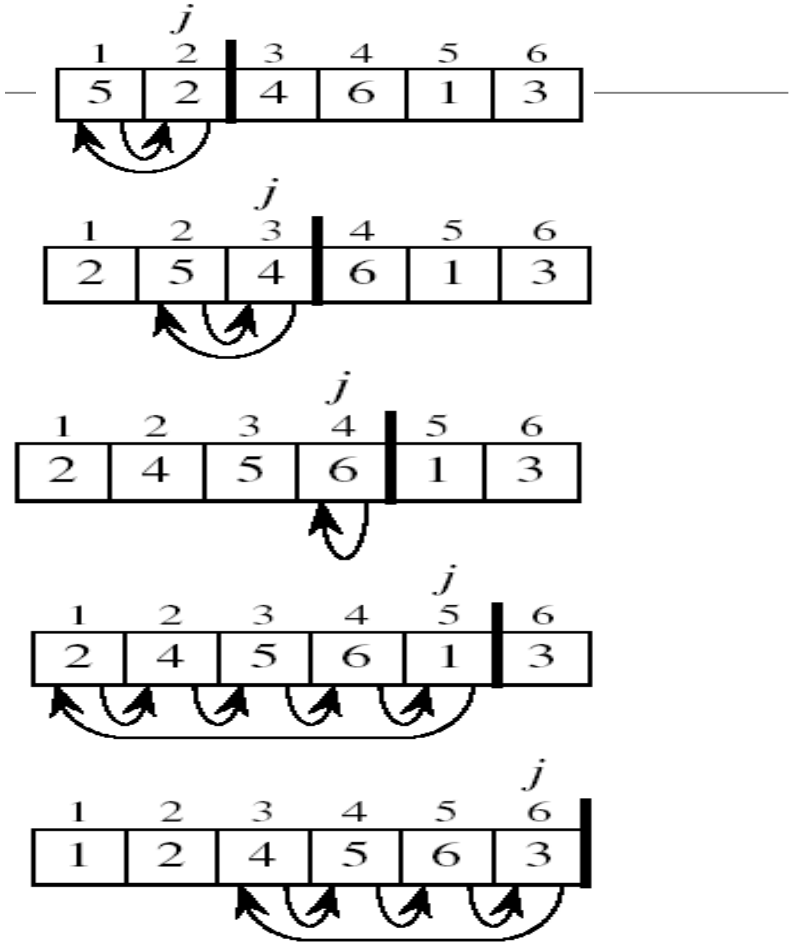
5 2 4 6 1 3

at each iteration, the array is divided in two sub-arrays:



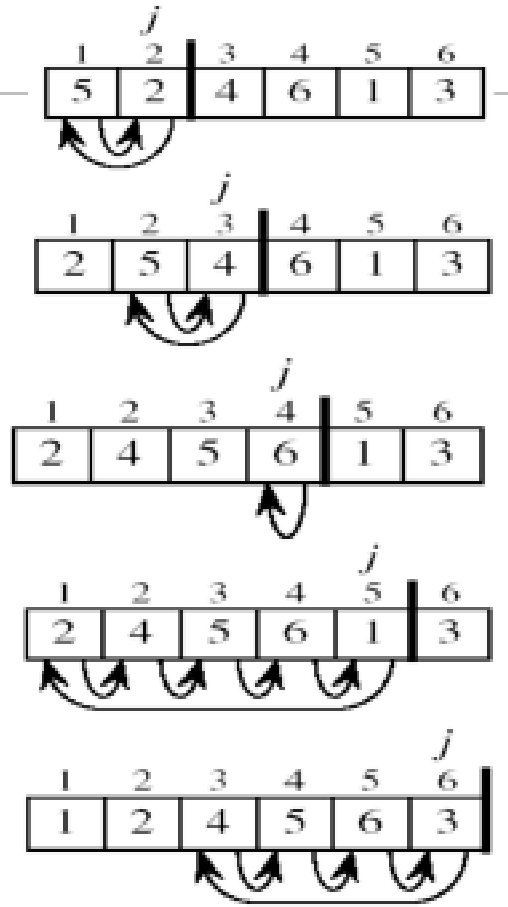
Insertion Sort





Insertion Sort

```
void insertionSort( arr[10], n)
{
   for (i = 1; i < n; i++)   {
     key = arr[i];
     j = i-1;
     while (j >= 0 && arr[j] > key)
     {
       arr[j+1] = arr[j];
       j = j-1;
     }
    arr[j+1] = key;
   }
}
```

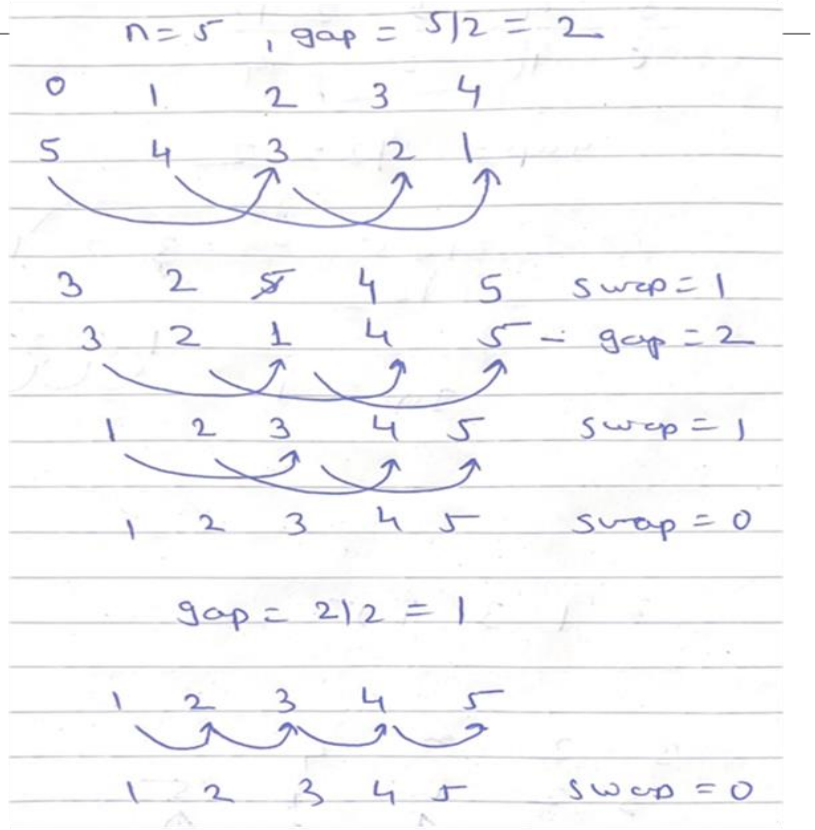


Analysis of Insertion Sort

- Insertion sort performs two operations:
- It scans through the list, comparing each pair of elements,
- and it swaps elements if they are out of order.
- If the input array is already in sorted order, insertion sort compares $O(n)$ elements and performs no swaps
- Best case, insertion sort runs in O(n) time
- Worst and average case $O(n^2)$
- Sable sort
- In-place sort
- Insertion sort is used ,when number of elements is small.
- input array is almost sorted, only few elements are misplaced in complete big array.

Shell sort

```
void shell_sort(int A[],int n]
{
    gap=n/2;
    do
    {
        do
         {
            swapped=0;
             for(i = 0; i < n- gap ; i++)
                 if(A[ I ] > A[ i + gap ])
                  {
                    swap();
                    swapped=1;
                  }
         } while(swapped == 1);
    }while((gap=gap/2) >= 1);
}
```



Example of Shell Sort

https://www.w3resource.com/ODSA/AV/Sorting/shellsortAV.html

- Unstable Sort

Gap=5

0	1	2	3	4	5	6	7	8	9
33	77	55	88	11	99	22	44	66	01

a[0]>a[5]
No swap

0	1	2	3	4	5	6	7	8	9
33	77	55	88	11	99	22	44	66	01

a[1]>a[6]
swap

0	1	2	3	4	5	6	7	8	9
33	77	55	88	11	99	22	44	66	01

a[2]>a[7]
swap

0	1	2	3	4	5	6	7	8	9
33	22	55	88	11	99	77	44	66	01

a[3]>a[8]
swap

0	1	2	3	4	5	6	7	8	9
33	22	44	88	11	99	77	55	66	01

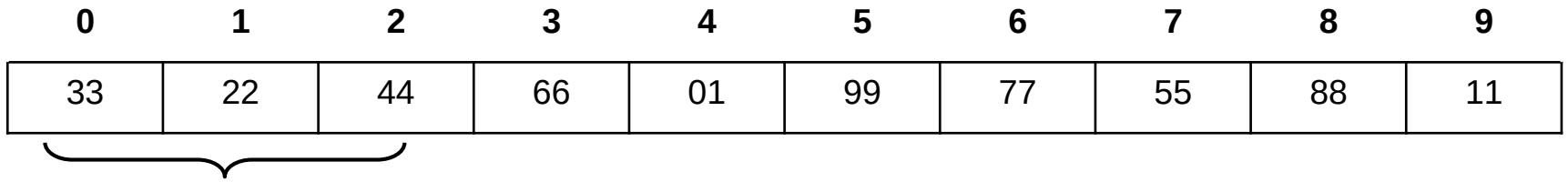
a[4]>a[9]
swap

0	1	2	3	4	5	6	7	8	9
33	22	44	66	11	99	77	55	88	01

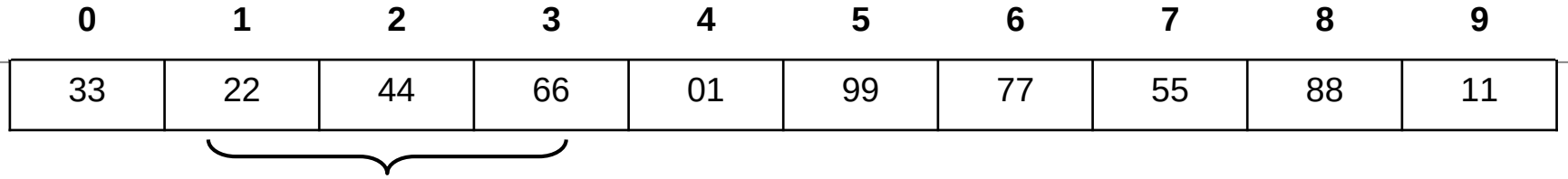
0	1	2	3	4	5	6	7	8	9
33	22	44	66	01	99	77	55	88	11

Gap=2

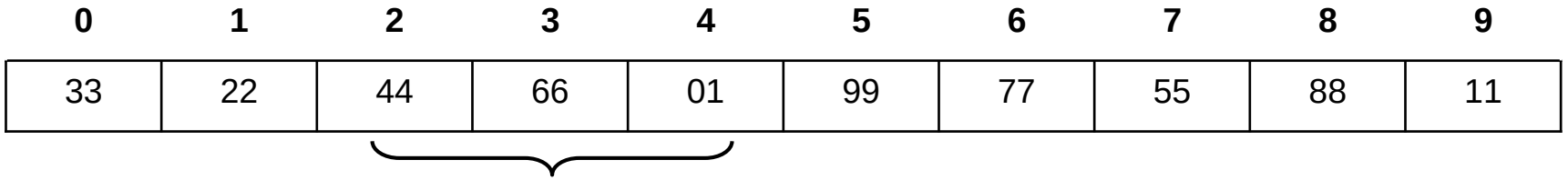
a[0]>a[2]
No swap



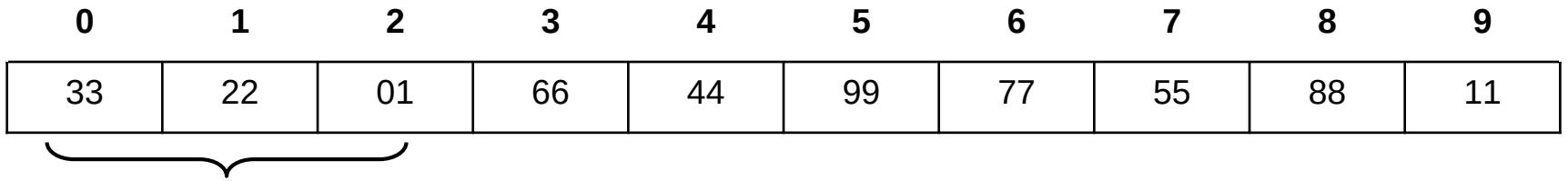
a[1]>a[3]
No swap



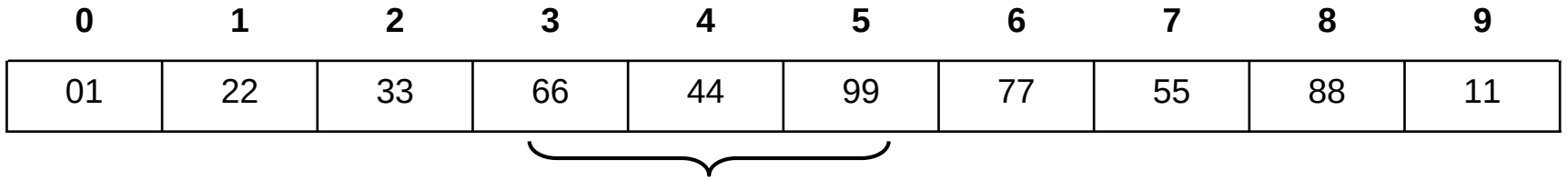
a[2]>a[4]
swap



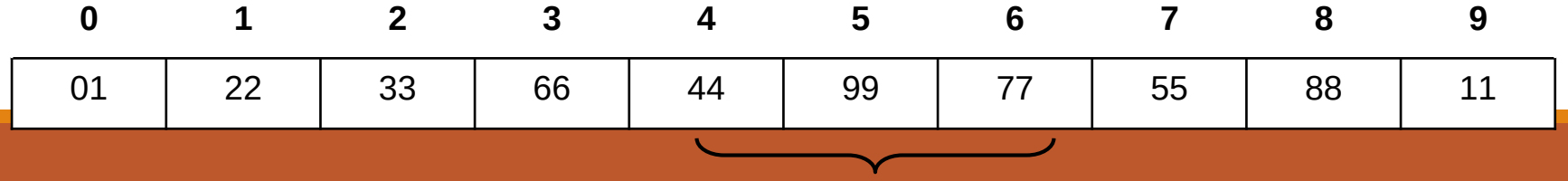
a[0]>a[2]
swap

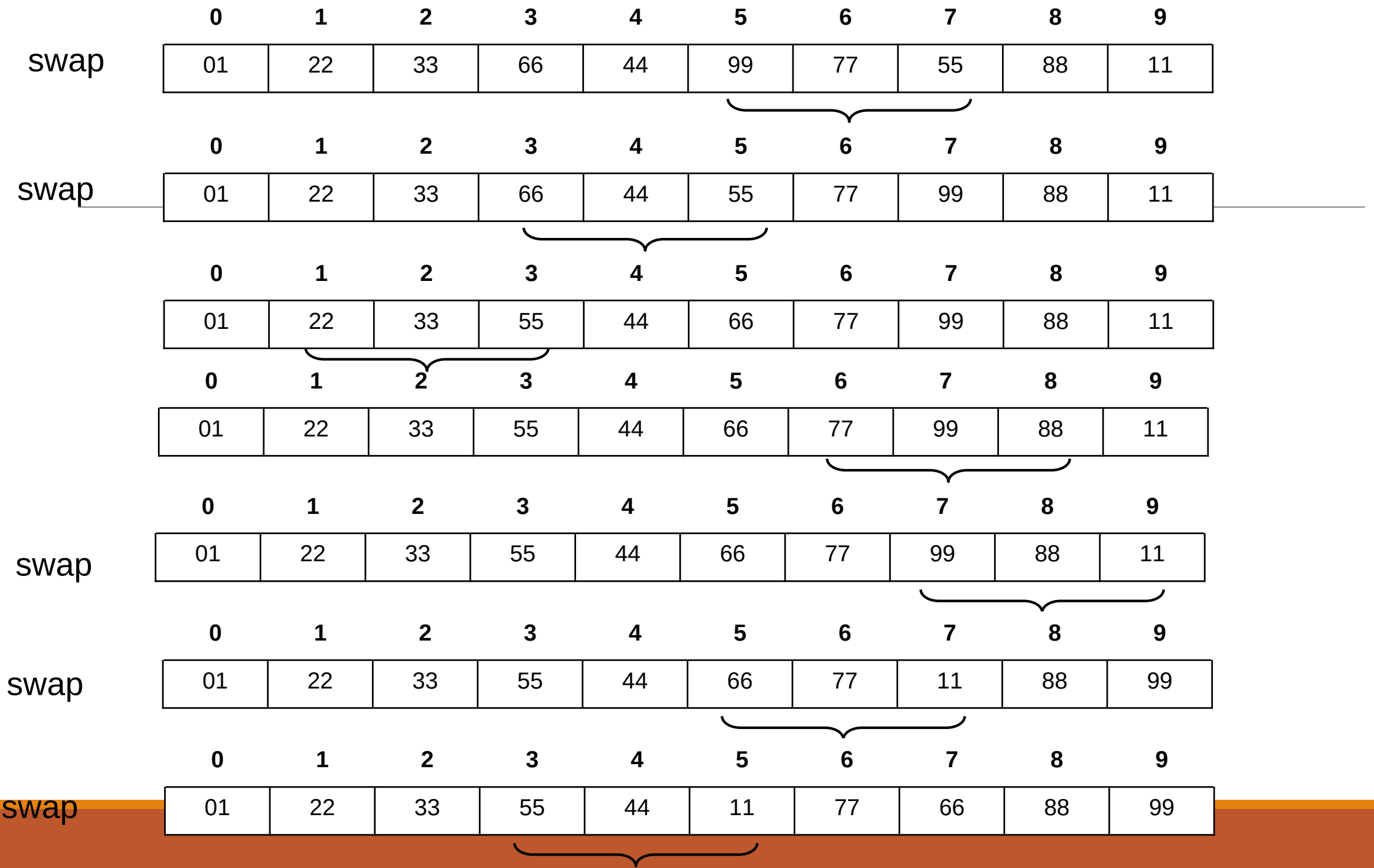


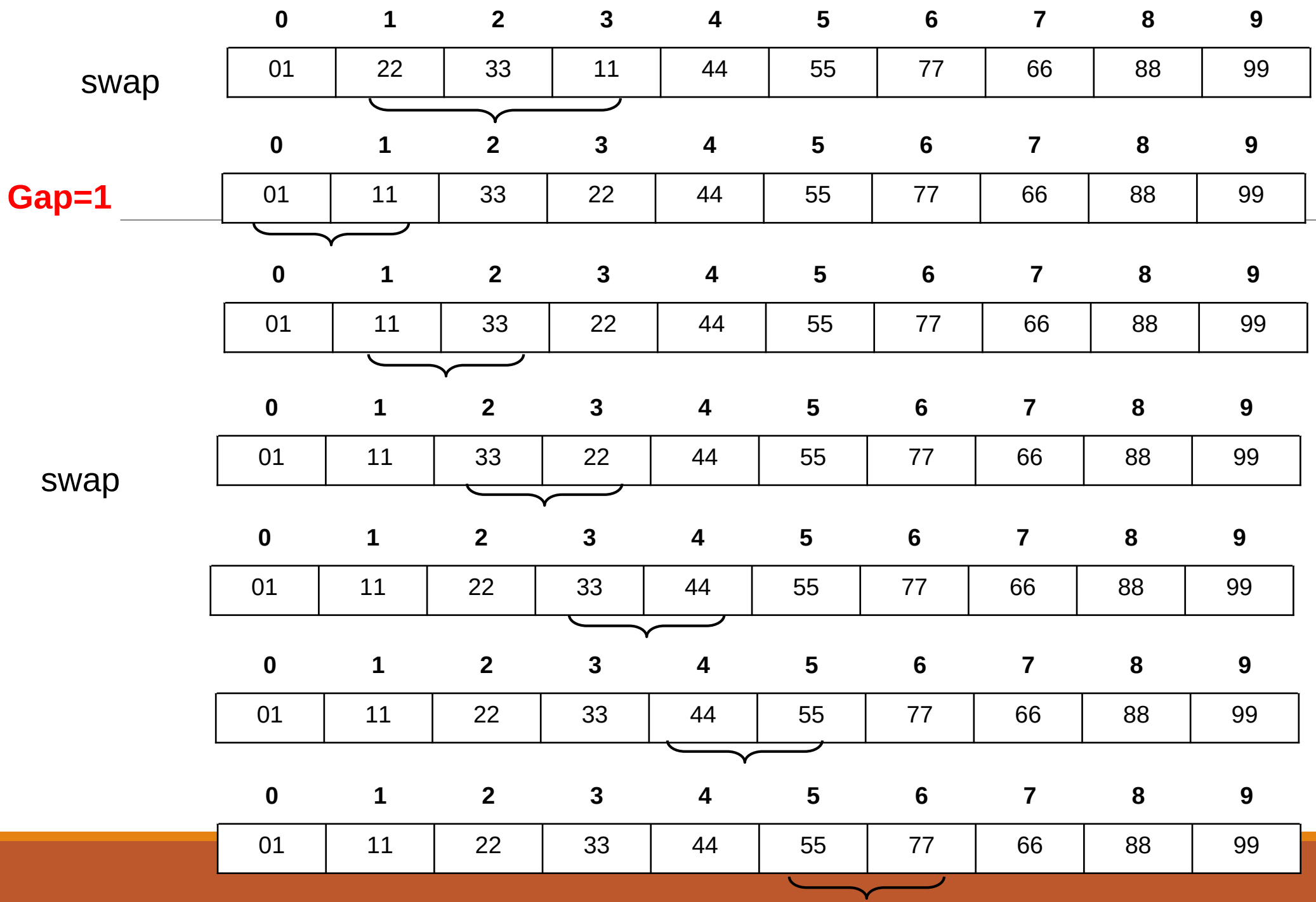
a[3]>a[5]
No swap

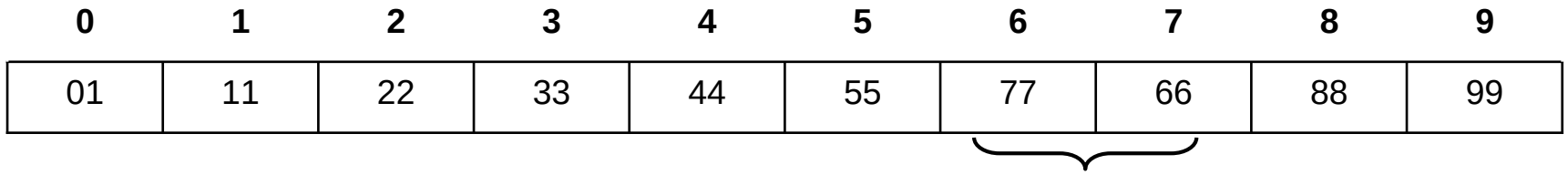


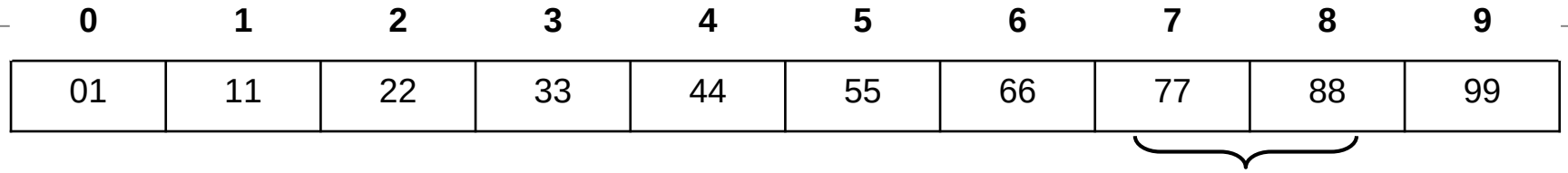
a[4]>a[6]
No swap

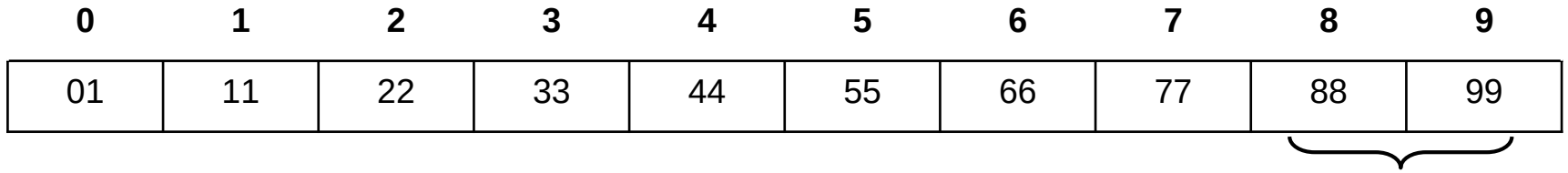



0 1 2 3 4 5 6 7 8 9
swap
01 22 33 66 44 99 77 55 88 11
0 1 2 3 4 5 6 7 8 9
swap
01 22 33 66 44 55 77 99 88 11
0 1 2 3 4 5 6 7 8 9
01 22 33 55 44 66 77 99 88 11
0 1 2 3 4 5 6 7 8 9
01 22 33 55 44 66 77 99 88 11
0 1 2 3 4 5 6 7 8 9
swap
01 22 33 55 44 66 77 99 88 11
0 1 2 3 4 5 6 7 8 9
swap
01 22 33 55 44 66 77 11 88 99
0 1 2 3 4 5 6 7 8 9
swap
01 22 33 55 44 11 77 66 88 99


swap
0 1 2 3 4 5 6 7 8 9
01 22 33 11 44 55 77 66 88 99
Gap=1
0 1 2 3 4 5 6 7 8 9
01 11 33 22 44 55 77 66 88 99
0 1 2 3 4 5 6 7 8 9
01 11 33 22 44 55 77 66 88 99
swap
0 1 2 3 4 5 6 7 8 9
01 11 33 22 44 55 77 66 88 99
0 1 2 3 4 5 6 7 8 9
01 11 22 33 44 55 77 66 88 99
0 1 2 3 4 5 6 7 8 9
01 11 22 33 44 55 77 66 88 99
0 1 2 3 4 5 6 7 8 9
01 11 22 33 44 55 77 66 88 99


0 1 2 3 4 5 6 7 8 9
01 11 22 33 44 55 77 66 88 99


0 1 2 3 4 5 6 7 8 9
01 11 22 33 44 55 66 77 88 99


0 1 2 3 4 5 6 7 8 9
01 11 22 33 44 55 66 77 88 99

Shell sort- Complexity Analysis

1. Complexity in the **Best Case: O(n*Log n)**

   The total number of comparisons for each interval (or increment) is equal to the size of the array when it is already sorted.

1. Complexity in the **Average Case: O(n*log n)**

   It's somewhere around O. (n1.5)

1. Complexity in the **Worst-Case** Scenario: Less Than or Equal to **O (n2)**

   Shell sort's worst-case complexity is always less than or equal to **O. (n2)**

The degree of complexity is determined by the interval picked. The above complexity varies depending on the increment sequences used. The best increment sequence has yet to be discovered.

Non-comparison sorting algorithm

- The sorting algorithms that sort the data without comparing the elements are called non-comparison sorting algorithms.
- The examples for non-comparison sorting algorithms are counting sort, bucket sort, radix sort, and others.

COMPARISON OF ALL SORTING METHODS

sort	Best	Average	Worst	stable	In place	Comparison based sorting
Bubble	**$\Omega(N^2)$**	**$\Theta(N^2)$**	**$O(N^2)$**	✓	✓	✓
Insertion	**$\Omega(N)$**	**$\Theta(N^2)$**	**$O(N^2)$**	✓	✓	✓
Selection	**$\Omega(N^2)$**	**$\Theta(N^2)$**	**$O(N^2)$**	X	✓	✓
Shell	*$\Omega(n \log(n))$*	*$O(n*\log n)$*	**$O(N^2)$**	X	✓	✓
Radix	**$\Omega(N\ k)$**	**$\Theta(N\ k)$**	**$O(N\ k)$**	X	X	X
Bucket	**$\Omega(N + k)$**	**$\Theta(N + k)$**	**$O(N^2)$**	✓	X	X
Merge	**$\Omega(N \log N)$**	**$\Theta(N \log N)$**	**$O(N \log N)$**	✓	X	✓
Quick	**$\Omega(N \log N)$**	**$\Theta(N \log N)$**	**$O(N^2)$**	X	✓	✓

COMPARISON OF ALL SORTING METHODS

Sorting method	Technique in brief	Best case	Worst case	Memory requirement	Is stable	Pros	Cons
Bubble sort	Repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order	$O(n^2)$	$O(n^2)$	No extra space needed	Yes	1. A simple and easy method 2. Efficient for small lists $n > 100$	Highly inefficient for large data

Selection sort	Finds the minimum value in the list and then swaps it with the value in the first position, repeats these steps for the remainder of the list (starting at the second position and advancing each time)	$O(n^2)$	$O(n^2)$	No extra space needed	Yes	1. Recommended for small files 2. Good for partially sorted data	Inefficient for large lists

Insertion sort	Every repetition of insertion sort removes an element from the input data, inserts it into the correct position in the already sorted list until no input elements remain. The choice of which element to remove from the input is arbitrary	$O(n)$	$O(n^2)$	No extra space needed	Yes	1. Relatively simple and easy to implement 2. Good for almost sorted data	Inefficient for large lists

Merge sort	Conceptually, a merge sort works as follows: If the list is of length 0 or 1, then it is already sorted. Otherwise, the algorithm divides the unsorted list into two sub-lists of about half the size Then, it sorts each sub-list recursively by reapplying the merge sort and then merges the two sub-lists back into one sorted list	$O(n\log_2 n)$	$O(n\log_2 n)$	Extra space proportional to n is needed	Yes	1. Good for external file sorting 2.Can be applied to files of any size	1. It requires twice the memory of the heap sort because of the second array used to store the sorted list. 2. It is recursive, which can make it a bad choice for applications that run on machines with limited memory

Practice Problem Statement

1.Given a sorted array **Arr[]**(0-index based) consisting of **N** distinct integers and an integer **k**, the task is to find the index of k, if it's present in the array **Arr[]**. Otherwise, find the index where **k** must be inserted to keep the array sorted.

2.There are n tree in a forest. Heights of the trees is stored in array tree[ ]. If ith tree is cut at height h, the wood obtained is tree[i]-h, given that tree[i]>h. If total wood needed is k (not less, neither more) find the height at which every tree should be cut (all trees have to be cut at the same height).

3.Given an integer **K** and an array **height[]** where **height[i]** denotes the height of the **i^{th}** tree in a forest.The task is to make a cut of height **X** from the ground such that exactly **K** unit wood is collected.If it is not possible then print **-1** else print **X**.

Practice Problem Statement

4. Given a sorted array with possibly duplicate elements, the task is to find indexes of first and last occurrences of an element x in the given array.

5.Given an array of integers. Find a peak element in it. An array element is a peak if it is NOT smaller than its neighbours. For corner elements, we need to consider only one neighbour.

6.Given a sorted and rotated array **A** of N distinct elements which is rotated at some point, and given an element **K**.●The task is to find the index of the given element K in the array A.

Practice Problem Statement

7.Given two arrays **A** and **B** contains integers of size **N** and **M**, the task is to find numbers which are present in the first array, but not present in the second array.

8.Given an integer **x,** find the square root of x. If **x** is not a perfect square, then return floor(√x).

Practice Problem Statement

Given an array of distinct integers. Sort the array into a wave-like array and return it. In other words, arrange the elements into a sequence such that a1 >= a2 <= a3 >= a4 <= a5..... (considering the increasing lexicographical order).

Input: n = 5
arr[] = {1,2,3, 4,5 }
Output: 2 1 4 3 5
Explanation: Array elements after sorting it in wave form are 2 1 4 3 5.

Practice Problem Statement

Given two integer arrays **A1[ ]** and **A2[ ]** of size **N** and **M** respectively. Sort the first array **A1[ ]** such that all the relative positions of the elements in the first array are the same as the elements in the second array **A2[ ]**.

See example for better understanding.

Note: If elements are repeated in the second array, consider their first occurrence only.

Example 1:

Input:

N = 11

M = 4

A1[] = {2, 1, 2, 5, 7, 1, 9, 3, 6, 8, 8}

A2[] = {2, 1, 8, 3}

Output:

2 2 1 1 8 8 3 5 6 7 9

Explanation: Array elements of A1[] are sorted according to A2[]. So 2 comes first then 1 comes, then comes 8, then finally 3 comes, now we append remaining elements in sorted order.

Given an array Arr[] of N distinct integers and a range from L to R, the task is to count the number of triplets having a sum in the range [L, R].

Input:
N = 4
Arr = {8 , 3, 5, 2}
L = 7, R = 11
Output: 1
Explanation: There is only one triplet {2, 3, 5} having sum 10 in range [7, 11].