# Data Structures

S. Durga Devi , CSE, CBIT

**Course Objectives**: The objectives of this course are

1. Basic linear and non-linear data structures.
2. Analyzing the performance of operations on data structures.
3. Different balanced binary trees, which provides efficient implementation for data structures.

**Course Outcomes:** On Successful completion of this course, student will be able to

1. Understand the basic concepts of data structures.
2. Analyze the performance of algorithms.
3. Distinguish between linear and non-linear data structures.
4. Identify the significance of balanced search trees.
5. Establish a suitable data structure for real world applications.

UNIT - I

Introduction: Data Types, Data structures, Types of Data Structures, Operations, ADTs, Algorithms, Comparison of Algorithms, Complexity, Time- space tradeoff. Recursion: Introduction, format of recursive functions, recursion Vs. Iteration, examples. Sorting: Quick sort, Merge Sort, Selection Sort

UNIT – II

Linked Lists: Introduction, Linked lists, Representation of linked list, operations on linked list, Comparison of Linked Lists with Arrays and Dynamic Arrays, Types of Linked Lists and operations-Circular Single Linked List, Double Linked List, Circular Double Linked List

UNIT - III

Stacks and Queues: Introduction to stacks, applications of stacks, implementation and comparison of stack implementations. Introduction to queues, applications of queues and implementations, Priority Queues and applications

UNIT - IV

Trees: Definitions and Concepts, Operations on Binary Trees, Representation of binary tree, Conversion of General Trees to Binary Trees, Representations of Binary Trees, Tree Traversal. Binary Search Trees: Representation and operations. Heap Tree: definition, representation, Heap Sort. Graphs: Introduction, Applications of graphs, Graph representations, graph traversals, Minimal Spanning Trees.

UNIT – V

Hashing: Introduction, Hashing Functions- Modulo, Middle of Square, Folding, Collision Techniques-Linear Probing, Quadratic Probing, Double Hashing, Balanced Search Trees: AVL Trees, Red-Black Trees, Splay Trees, B-Trees

## Text Books

1. Narasimha karumanchi, “Data Structures and Algorithms Made Easy”, Career Monk Publications, 2017

2. S. Sahni and Susan Anderson-Freed, “Fundamentals of Data structures in C”, E.Horowitz, Universities Press, 2nd Edition.

3. ReemaThareja, “Data Structures using C”, Oxford University Press.

4. Instructor material.

## Suggested Reading

1. D.S.Kushwaha and A.K.Misra, “Data structures A Programming Approach with C”, PHI.
2. 2. Seymour Lipschutz, “Data Structures with C”, Schaums Outlines, Kindle Edition.

# UNIT-1

- Introduction: Data Types, Data structures
- Types of Data Structures
- Operations, ADTs
- Algorithms, Comparison of Algorithms, Complexity, Time- space tradeoff.
- Recursion: Introduction, format of recursive functions, recursion Vs. Iteration, examples.
- Sorting: Quick sort, Merge Sort, Selection Sort

# Introduction to Data Structures

## What is a Data structure?

Data Structure is a set of algorithms which are used to structure the information while storing.

- Formal definition is data structure is a way of storing and organizing data in a computer so that it can be used( accessed) efficiently.
- These set of algorithms are implemented by using any programming languages like C,C++.
- Real time examples

1. English dictionary
2. City map
3. Cash Book

English Dictionary

*moo re ka le kwago ka moka ga rena.*

**boot** *noun* (*pl.* **boots**) 1 ■ putsu • I wear **boots** when I go walking in the mountains. *Ke rwala diputsu ge ke eya go sepela dithabeng.* 2 ■ putu • We put our suitcases in the **boot** of the car. *Re beile mekotla ya rena ya diaparo ka gare ga putu ya sefatanaga.*

**border** *noun* (*pl.* **borders**) 1 (*Geography*) ■ mollwane ❶ the boundary of a country *mollwane wa naga* • Namibia is on the **border** of South Africa. *Namibia e mollwaneng wa Afrika-Borwa.* • Two foreigners were arrested because they were trying to cross the **border** without permission. *Batšwantle ba babedi ba swerwe ka lebaka la gore ba be ba leka go tshela mollwane ntle le tumelelo.* 2 (*Art*) ■ morumo ❶ a strip around the edge of a picture or page or piece of cloth *leši leo le dikologago mafelelo a seswantšho goba letlakala goba seripa sa lešela* • Geometric patterns make interesting **borders**. *Dipatrone tša tšeometriki di dira merumo ya go kgahliša.*

**boredom** *noun* (**no plural**) ■ bodutu
◇ **slight boredom** ■ bodutwana

**borrow** *verb* (**borrows**, **borrowing**, **borrowed**) ■ adima • You can **borrow** money from the bank if you want to start a business. *O ka adima tšhelete pankeng ge o nyaka go thoma kgwebo.*

Don't confuse **borrow** and **lend**. You **borrow from** but **lend to**.

**boss** *noun, verb*
• *noun* (*pl.* **bosses**) ■ molaodi • I am going to

**bottle** * *noun, verb*
• *noun* (*pl.* **bottles**) ■ lebotlelo • Could I have a **bottle** of milk, please? *Na nka hwetša lebotlelo la maswi, hle?* • How much is a loaf of bread and a **bottle** of milk? *Lofo ya borotho le lebotlelo la maswi ke bokae?*
• *verb* (**bottles**, **bottling**, **bottled**) ■ bitiela • My mother **bottled** some peaches. *Mma o bitiela diperekisi ka lebotlelong.*

**bottom** * *noun, adjective*
• *noun* (*pl.* **bottoms**) 1 ■ bofase; botlase • If you don't stir your coffee, all the sugar will stay at the **bottom**. *Ge o sa hudue kofi ya gago, swikiri ka moka e tla dula bofase bja komiki.* • There are trees at the **bottom** of the garden. *Go na le mehlare bofase bja serapana.* 2 informal ■ marago • I sat on my **bottom**. *Ke ile ka dula ka marago.*
• *adjective* ■ [PC +] ka fase • You can put the book on the **bottom** shelf. *O ka bea puku boraleng bja ka fase.*

**bought** *verb* Past tense and past participle of BUY ■ rekile • Mpho **bought** school books at the beginning of the year. *Mpho o rekile dipuku tša sekolo mathomong a ngwaga.*

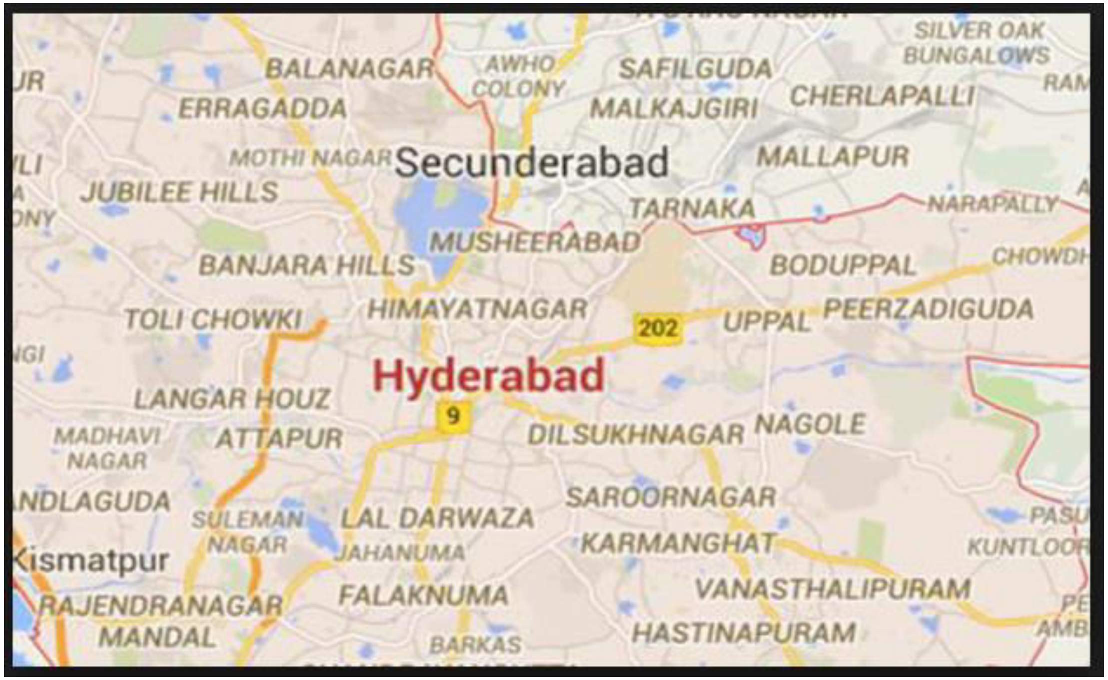
**bound** *verb* Past tense and past participle of BIND ■ tlemile

**boundary** *noun* (*pl.* **boundaries**) ■ mollwane

**bow** *noun* (*pl.* **bows**) ■ bora • He entered the courtyard armed with a spear, an arrow and a **bow**. *O tsene kgorong a itlhamile ka lerumo, mosebe le bora.*

This word rhymes with **no**.

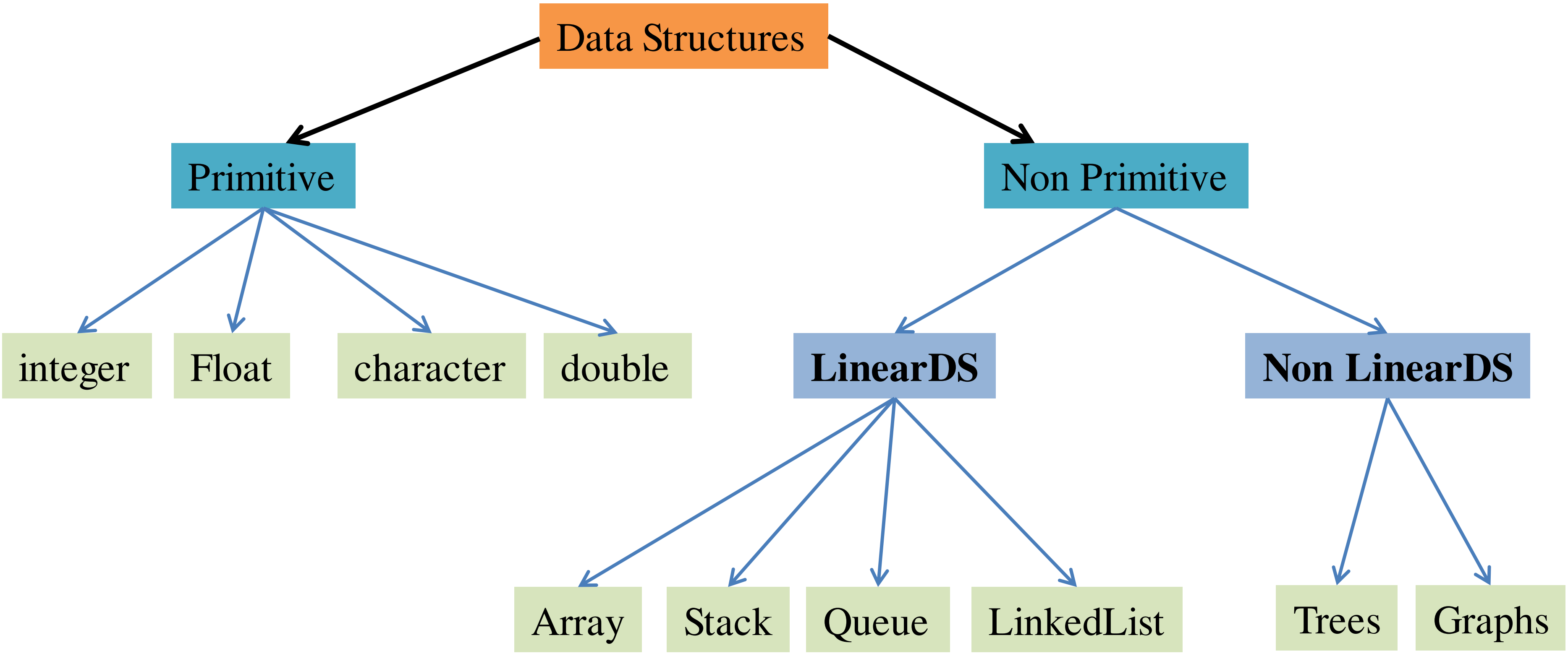
City Map



# Types of Data Structures

Several data structures are available in the computer science and used based on their applications.

Data structures are classified into two types.



1.Linear data structure

All the elements are accessed in sequential order or linear order means that all the elements are adjacent to each other. Each element has exactly two neighbors. Predecessor and successor.

- the first element does not have predecessor
- The last element does not have successor.

2. Non linear data structure no such sequence in elements, if one element is connected to a more than two adjacent element then it is a non linear data structure.

# Linear vs Non Linear Ds

| Linear | Non Linear |
|---|---|
| 1. Data items arranged in sequence | 1. Not in sequence |
| 2. All elements are adjacent to each other | 2. One element adjacent to more than one element |
| 3. Easy implementation | 3. Difficult to implement |
| 4. All elements can access in single run | 4. Not possible with single run |
| 5. array, stack, queues, linked list | 5. Trees and graphs |

# Applications of data structures

- Implementing data bases of different organizations (ds used is B-Trees)
- Implement compilers for different languages (ds used is hash tables).
- Used in every program and software system.

# Abstract Data Type (ADT)

- For user defined data types we need to define operations
- Data structure is implemented around the concept of an abstract data type that defines the data together with the operations.
- ADT contains
  - data
  - operations
  - no implementation details.
- ADT is also called as user defined data type.
- Only tells about what are the data values required and what operations performed on those objects.
- Example stack, array, linked list, etc.

# Algorithm specifications

## Algorithm:

a finite set of instructions to perform a specific task.

## Properties of Algorithm

1. Input: finite set of inputs to be given ( zero/more)
2. Output : at least one output should be produced
3. Definiteness : all the instructions are clear and un ambiguous
4. Finiteness: the algorithm should be terminated in a finite number of steps
5. Effectiveness: ability to produce desired results by an algorithm.

# What is a good algorithm?

- Efficient
  - small running time
  - less memory space.
- Performance analysis and measurement

- performance analysis helps to select the best algorithm among multiple algorithms to solve a problem.

- performance of an algorithm is a process of making evaluating judgement about algorithms

What measures are taken to select best algorithms?

1. Correctness/ effectiveness- whether algorithm producing correct results or not
2. Easy to understand
3. Easy to implement
4. How much memory occupied by an algorithm (space)
5. Execution speed of an algorithm (time)

- To analyse an algorithm we will consider only space and time

# Performance analysis

- Performance analysis is a process of calculating space and time required by that algorithm
- Performance analysis of algorithm is performed by considering following measures.

1. Space required to complete the task of that algorithm ***(Space Complexity)***. It includes program space and data space
2. Time required to complete the task of that algorithm ***(Time Complexity)***

# Space complexity

- Total amount of computer memory required by an algorithm to complete its execution is called space complexity of that algorithm.
- When a program is under execution it uses computer memory for three reasons

1. Instructional space: stores compiled version of instructions
2. Environmental stack: store information about function calls
3. Data space: stores variables, constants and structures etc…

To compute space complexity data space is to be considered.

- Example-1

```
int add(int a, int b)
{
   return a+b;
}
```

total space is 2bytes for each variable a and b.

2 bytes for return value

Total space required is 6 bytes.

Example -2

```
int sum(int A[], int n)
{
int sum = 0, i;
for(i = 0; i < n; i++)
sum = sum + A[i];
 return sum;
}
```

total space is

A[] - n*2

n= 2

i=2

sum=2

return value=2

total space is 2n+8

# Time Complexity

- Time complexity of an algorithm is amount of time required to complete its execution.
- The running time of an algorithm depends on following ..

1. Processor speed
2. Single or multiple processors
3. 32 bit or 64 bit of a machine
4. Depends on operations
5. Input size

to compute time complexity only input size is to be considered.

- It requires 1 unit of time for Arithmetic and Logical operations
- It requires 1 unit of time for Assignment and Return value
- It requires 1 unit of time for Read and Write operations

If amount of time required by an algorithm is fixed for all input values is called constant time complexity.

```
int add(int a, int b)
  {
    return a+b;
}
```

a+b 1 unit

Return value 1 unit

Total 2 units of time is taken by above code.

Example-2

| | Cost<br>Time require for line (Units) | Repeatation<br>No. of Times Executed | Total<br>Total Time required in worst case |
|---|---|---|---|
| `int sumOfList( int A[ ], int n)` | | | |
| `{` | | | |
| `int sum = 0, i;` | 1 | 1 | 1 |
| `for(i = 0; i < n; i++)` | 1 + 1 + 1 | 1 + (n+1) + n | 2n + 2 |
| `sum = sum + A[i];` | 2 | n | 2n |
| `return sum;` | 1 | 1 | 1 |
| `}` | | | |
| | | | **4n + 4** |

If amount of time required by an algorithm is increasing with the increase of input values is said to be linear time complexity.

# Asymptotic notations

- We have to analyse an algorithm how running time increases as input size increases.
- When running time of an algorithm grows up with increasing input size is order of growth or rate of growth of an algorithm.
- Asymptotic notation of an algorithm is a mathematical representation of its complexity.
- When we want to represent a complexity of an algorithm we use only most significant terms only least significant terms are ignored.
- Example
- **Algorihtm 1 : $5n^2 + 2n + 1$**
- **Algorihtm 2 : $10n^2 + 8n + 3$**

**Three types of asymptotic notations**

**1. Big oh(O)**

**2. Omega (($\Omega$)**

**3.Theta ($\Theta$)**

# 1. Big Oh notation (O)

- Big - Oh notation is used to define the **upper bound** of an algorithm in terms of Time Complexity.
- That means Big - Oh notation always indicates the maximum time required by an algorithm for all input values.
- Big - Oh notation describes the worst case of an algorithm time complexity.
- **Consider function f(n) the time complexity of an algorithm and g(n) is the most significant term. If f(n) <= C g(n) for all n >= $n_0$, C > 0 and $n_0$ >= 1. Then we can represent f(n) as O(g(n)).**
- **f(n) = O(g(n))**
- **Example**
- Consider the following f(n) and g(n)...
  **f(n) = 3n + 2**
  **g(n) = n**
  If we want to represent **f(n)** as **O(g(n))** then it must satisfy **f(n) <= C x g(n)** for all values of **C > 0** and **$n_0$>= 1**
- f(n) <= C g(n)
  ⇒3n + 2 <= C n

  Above condition is always TRUE for all values of **C = 4** and **n >= 2**.
  By using Big - Oh notation we can represent the time complexity as follows...
  **3n + 2 = O(n)**

# 2. Omega Notation (Ω)

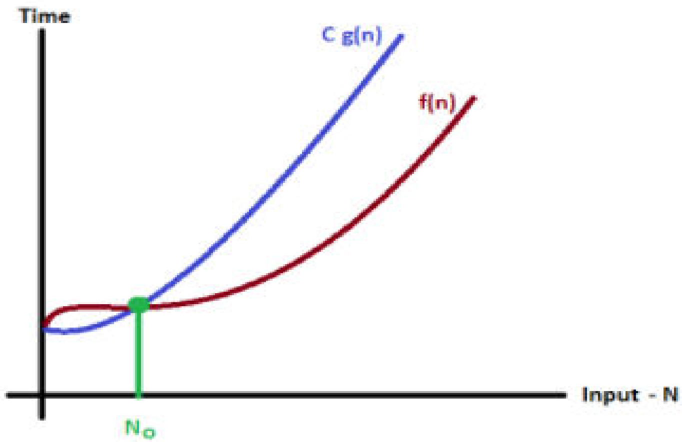
- Big - Omega notation is used to define the **lower bound** of an algorithm in terms of Time Complexity.
- That means Big - Omega notation always indicates the minimum time required by an algorithm for all input values.
- Big - Omega notation describes the best case of an algorithm time complexity.
- **Consider function f(n) the time complexity of an algorithm and g(n) is the most significant term. If $f(n) >= C \times g(n)$ for all $n >= n_0$, $C > 0$ and $n_0 >= 1$. Then we can represent f(n) as $\Omega(g(n))$.**
- **Example**
- Consider the following f(n) and g(n)...
  **$f(n) = 3n + 2$**
  **$g(n) = n$**
  If we want to represent **f(n)** as **$\Omega(g(n))$** then it must satisfy **$f(n) >= C\, g(n)$** for all values of **$C > 0$** and **$n_0 >= 1$**
- $f(n) >= C\, g(n)$
  $\Rightarrow 3n + 2 <= C\, n$

  Above condition is always TRUE for all values of **$C = 1$** and **$n >= 1$**.
  By using Big - Omega notation we can represent the time complexity as follows...
  **$3n + 2 = \Omega(n)$**

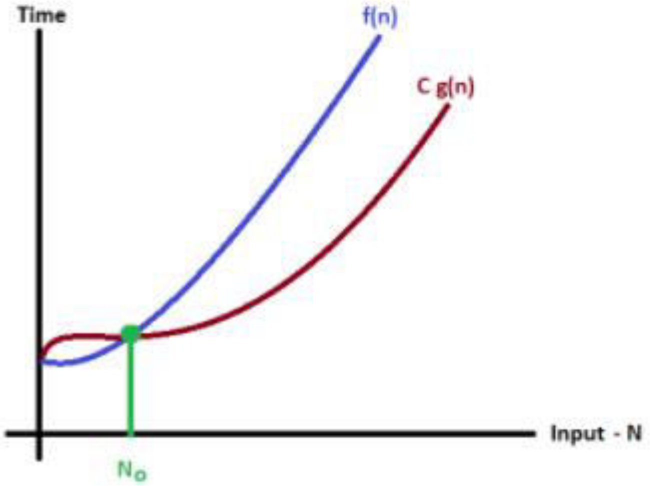
# 3. Theta notation($\Theta$)

- Big - Theta notation is used to define the **average bound** of an algorithm in terms of Time Complexity.
- Theta notation always indicates the average time required by an algorithm for all input values.
- Big - Theta notation describes the average case of an algorithm time complexity.
- **Consider function f(n) the time complexity of an algorithm and g(n) is the most significant term. If $C_1$ g(n) <= f(n) <= $C_2$ g(n) for all n >= $n_0$, $C_1$, $C_2$ > 0 and $n_0$ >= 1. Then we can represent f(n) as $\Theta$(g(n)).**
- **Example**
- Consider the following f(n) and g(n)...
  **f(n) = 3n + 2**
  **g(n) = n**
  If we want to represent **f(n)** as **$\Theta$(g(n))** then it must satisfy **$C_1$ g(n) <= f(n) >= $C_2$ g(n)** for all values of **$C_1$, $C_2$ > 0** and **$n_0$>= 1**
- $C_1$ g(n) <= f(n) >= $\Rightarrow C_2$ g(n)
  $C_1$ n <= 3n + 2 >= $C_2$ n
  Above condition is always TRUE for all values of **$C_1$ = 1, $C_2$ = 4** and **n >= 1**.
  By using Big - Theta notation we can represent the time compexity as follows...
  **3n + 2 = $\Theta$(n)**

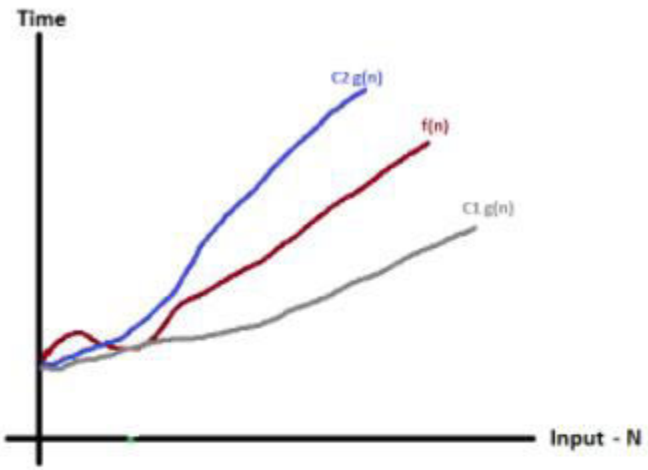
# Big Oh notation




## Omege Notation (Ω)




## Theta notation(Θ)




S. Durga Devi,CSE,CBIT

# Recursion

- A function which calls itself is a recursive.
- Recursion solves a smaller to tasks by calling itself.
- Ensure recursion should terminate at some condition.
- Recursion uses stack memory to execute.
- Recursive functions has two cases.

1. Base case : where recursion calling should terminate.
2. Recursive case: function calls itself to perform a sub task.

Why recursion is needed?

- Recursion reduces the code
- There are some problems which are difficult with iterative method can be solved with recursive method.
- Ex- Towers of Hanoi, Binary search, Divide and Conquer problems.

# Example

- To find factorial of a given number recursive function is

  n!=1, if n=0 //base case

  n!=n*(n-1)!, if n>0 // recursive case.

```
factorial(int number)
  {                                    number=4
    int fact;
     if(number==0)// base case
            return 1;
     else
    fact=number*factorial(number-1);//recursive  case.
     return fact;
  }//factorial()
```

factorial (4) =4*factorial (3)

factorial (3) =3*factorial (2)

factorial (2) =2*factorial (1)

factorial (1) =1*factorial (0)

factorial (0) =1

factorial (1) =1*1=1

factorial (2) =2*1=2

factorial (3) =3*2=6

factorial (4) =4*6=24

Factorial(4) returns 24 to main because main is calling function to factorial(n)

# Differences between recursive and iterative methods.

| Recursive method | Iterative method |
|---|---|
| Reduces the code | Length of the code is more |
| Speed of recursive methods are slow | Faster |
| terminated when it meets base condition | terminated when the condition is false |
| Each recursive call requires extra space to execute. Ex stack | does not require any extra space |
| If recursion goes into infinite , the program run out of memory and results in stack overflow | Goes to infinite loop |
| solution to some problems are easier with recursion | solution to problem may not always easy |

# Examples of recursion

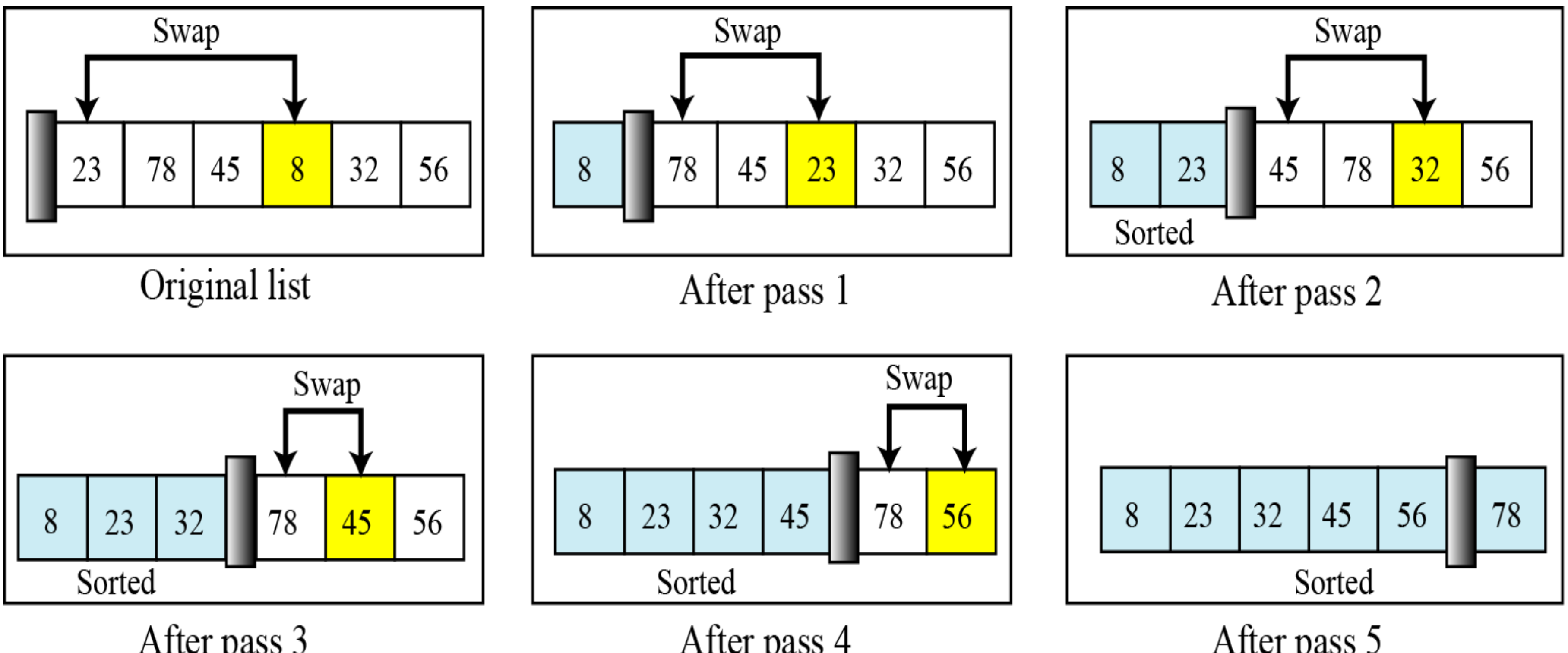
1. Factorial of a number
2. Fibonacci series
3. Tree traversals- Preorder, Inorder, Postorder.
4. Binary search
5. Merge sort
6. Quick sort
7. Towers of Hanoi
8. Divide and conquer problems
9. Dynamic programming

# Selection Sort

## Procedure:

- Selection sort involved scanning through the list to select the smallest element and swap it with the first element.
- The rest of the list is then search for the next smallest and swap it with the second element.
- This process is repeated until the rest of the list reduces to one element, by which time the list is sorted.

**The following table shows how selection sort works.**



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 77 | 33 | 44 | 55 | 88 | 66 | 22 | 11 |
| Pass-1 | 77 | 33 | 44 | 55 | 88 | 66 | 22 | 11 |
| Pass-2 | 11 | 33 | 44 | 55 | 88 | 66 | 22 | 77 |
| Pass-3 | 11 | 22 | 44 | 55 | 88 | 66 | 33 | 77 |
| Pass-4 | 11 | 22 | 33 | 55 | 88 | 66 | 44 | 77 |
| Pass-5 | 11 | 22 | 33 | 44 | 88 | 66 | 55 | 77 |
| Pass-6 | 11 | 22 | 33 | 44 | 55 | 66 | 88 | 77 |
| Pass-7 | 11 | 22 | 33 | 44 | 55 | 66 | 88 | 77 |

Write a C program to implement Selection sort

```
#include<stdio.h>
void selectionsort(int a[],int n);
void main()
{
int a[100],i,n;
printf("\n enter the size of the array");
scanf("%d",&n);
printf("enter array elements");
for(i=0;i<n;i++)
scanf("%d",&a[i]);
selectionsort(a,n);
}//main
```

```
void selectionsort(int a[],int n)
{
int i,j,k,loc,min,temp;
for(i=0;i<n-1;i++)
{
printf("\n");
printf("\nstep-%d",i+1);
min=a[i];
loc=i;
for(j=i+1;j<=n-1;j++)
{
        if(min>a[j])
        {
                min=a[j];
        loc=j;}
}//j
temp=a[i];
a[i]=a[loc];
a[loc]=temp;
for(k=0;k<n;k++)
printf("%3d",a[k]);
}//i
printf("\n\n sorted array is ");
for(k=0;k<n;k++)
printf("%3d",a[k]);
}//selectionsort
```

# Merge Sort

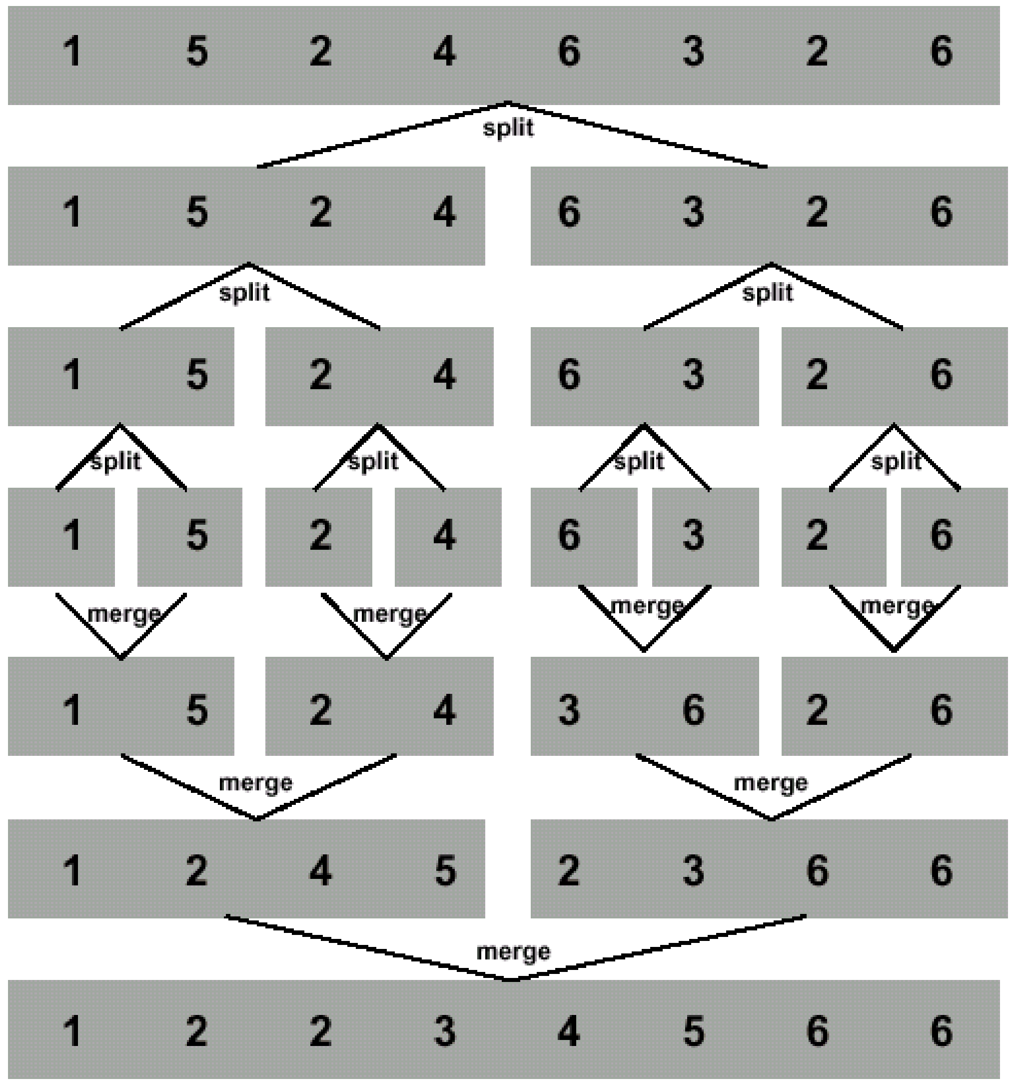
- This algorithm uses the divide- and – conquer method.
- Split array A[0..$n$-1] into about equal halves and make copies of each half in arrays B and C.
- Sort arrays B and C recursively.

**Merge sorted** arrays **B and C** into **array A** as follows:

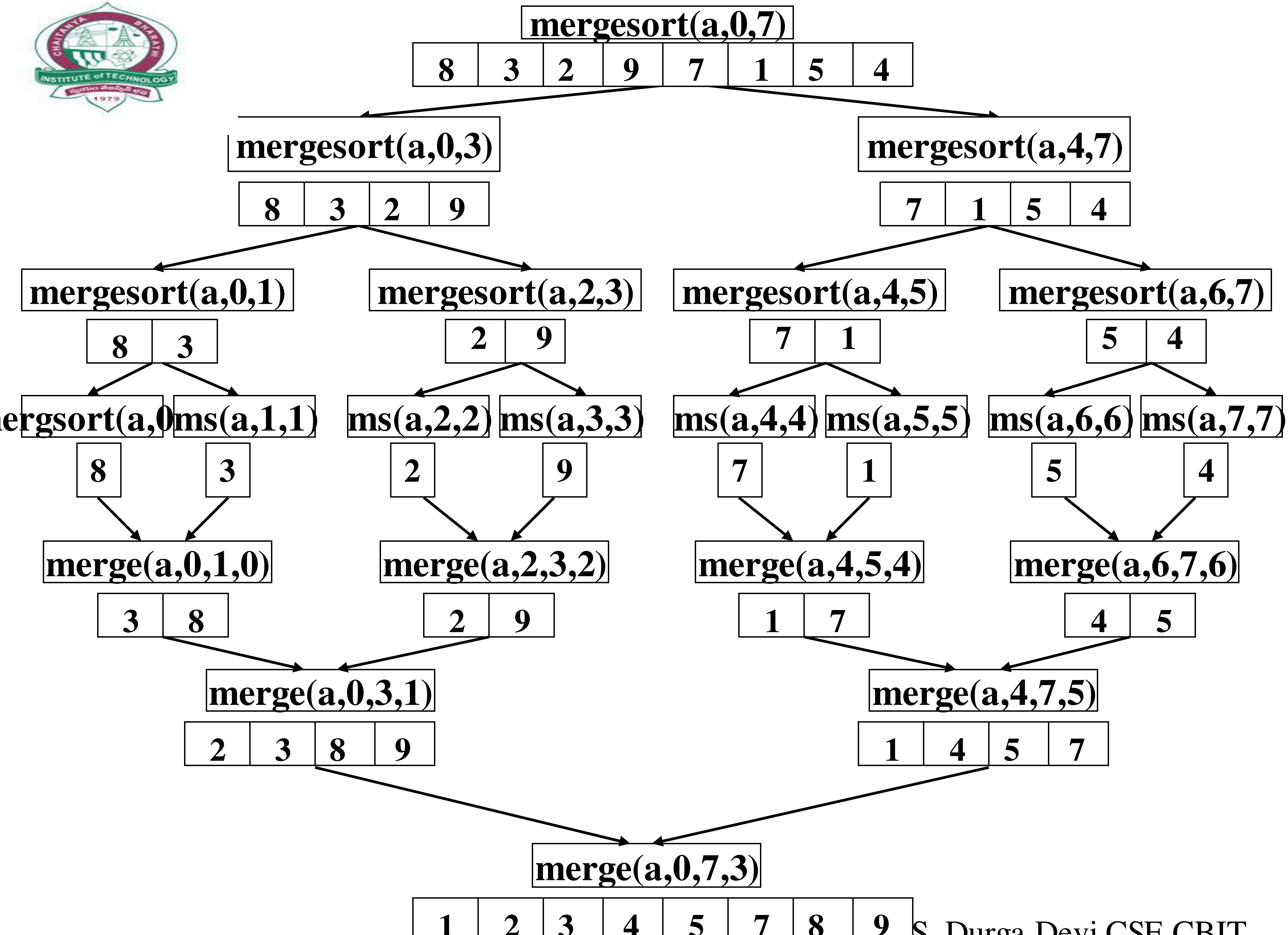
- Repeat the following until no elements remain in one of the arrays:
  - Compare the first elements in the remaining unprocessed portions of the arrays.
  - Copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array.
- Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into A.

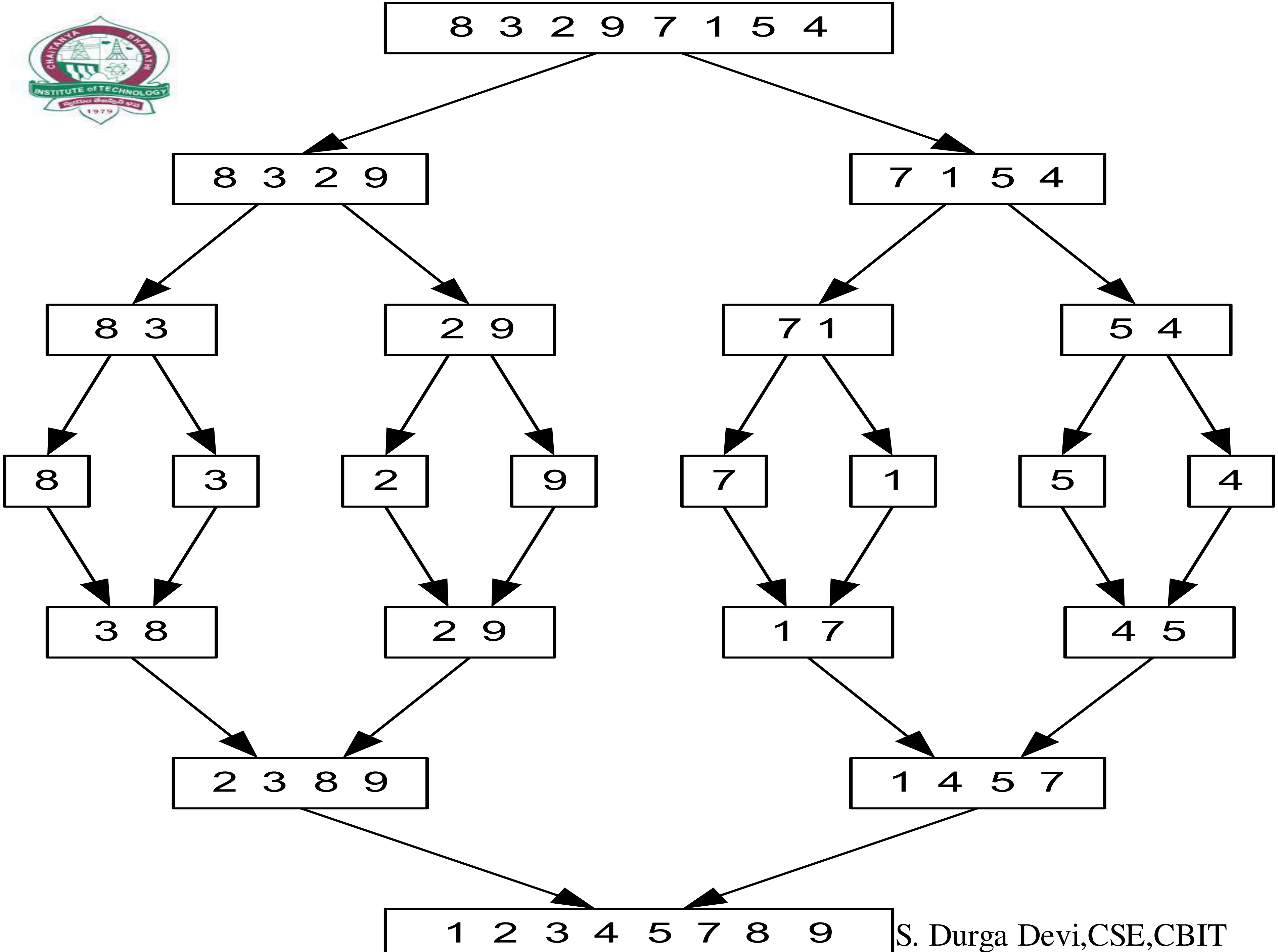
# Merge Sort

Input:




Output.


mergesort(a,0,7)
8 3 2 9 7 1 5 4
mergesort(a,0,3)
8 3 2 9
mergesort(a,4,7)
7 1 5 4
mergesort(a,0,1)
8 3
mergesort(a,2,3)
2 9
mergesort(a,4,5)
7 1
mergesort(a,6,7)
5 4
ergsort(a,0
ms(a,1,1)
ms(a,2,2)
ms(a,3,3)
ms(a,4,4)
ms(a,5,5)
ms(a,6,6)
ms(a,7,7)
8
3
2
9
7
1
5
4
merge(a,0,1,0)
3 8
merge(a,2,3,2)
2 9
merge(a,4,5,4)
1 7
merge(a,6,7,6)
4 5
merge(a,0,3,1)
2 3 8 9
merge(a,4,7,5)
1 4 5 7
merge(a,0,7,3)
1 2 3 4 5 7 8 9


8 3 2 9 7 1 5 4
8 3 2 9
7 1 5 4
8 3
2 9
7 1
5 4
8
3
2
9
7
1
5
4
3 8
2 9
1 7
4 5
2 3 8 9
1 4 5 7
1 2 3 4 5 7 8 9

# Mergsort program

```
#include "stdio.h"
void merge(int [],int,int,int);
void mergesort(int a[],int low,int high)
{
int mid;
  if(low < high)
               {
  mid = (low + high)/2;
 mergesort(a,low,mid);
 mergesort(a,mid+1,high);
 merge(a,low,high,mid);
               } //if
}//mergesort
```

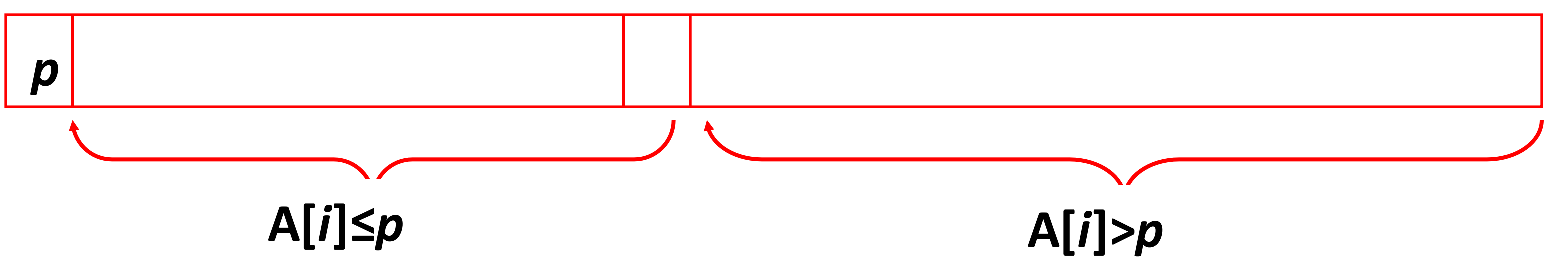
```
void merge(int a[],int l,int h,int m){
   int c[100],i,j,k;
  i = l;  j = m + 1;   k = l;

  while (i <= m && j <= h)
      {
  if(a[i] < a[j])
{
  c[k] = a[i]; i++;  k++;
}//if
else
{
 c[k] = a[j]; j++;  k++;
} //else
}//while
 while(i <= m)      c[k++] = a[i++];
 while(j <= h)      c[k++] = a[j++];
 for(i = l; i < k; i++) a[i] = c[i];
}//merge
```

```
void main()
{
    int i,n,a[100];
        printf("\n Enter the size of the array :");
    scanf("%d",&n);
    printf("\n Enter the elements :\n");
    for(i = 0; i < n; i++)
        scanf("%d",&a[i]);
    mergesort(a,0,n-1);
    printf("\n Elements in sorted order :\n");
    for(i = 0; i < n; i++)
        printf("%5d",a[i]);
}
```

# Quick Sort

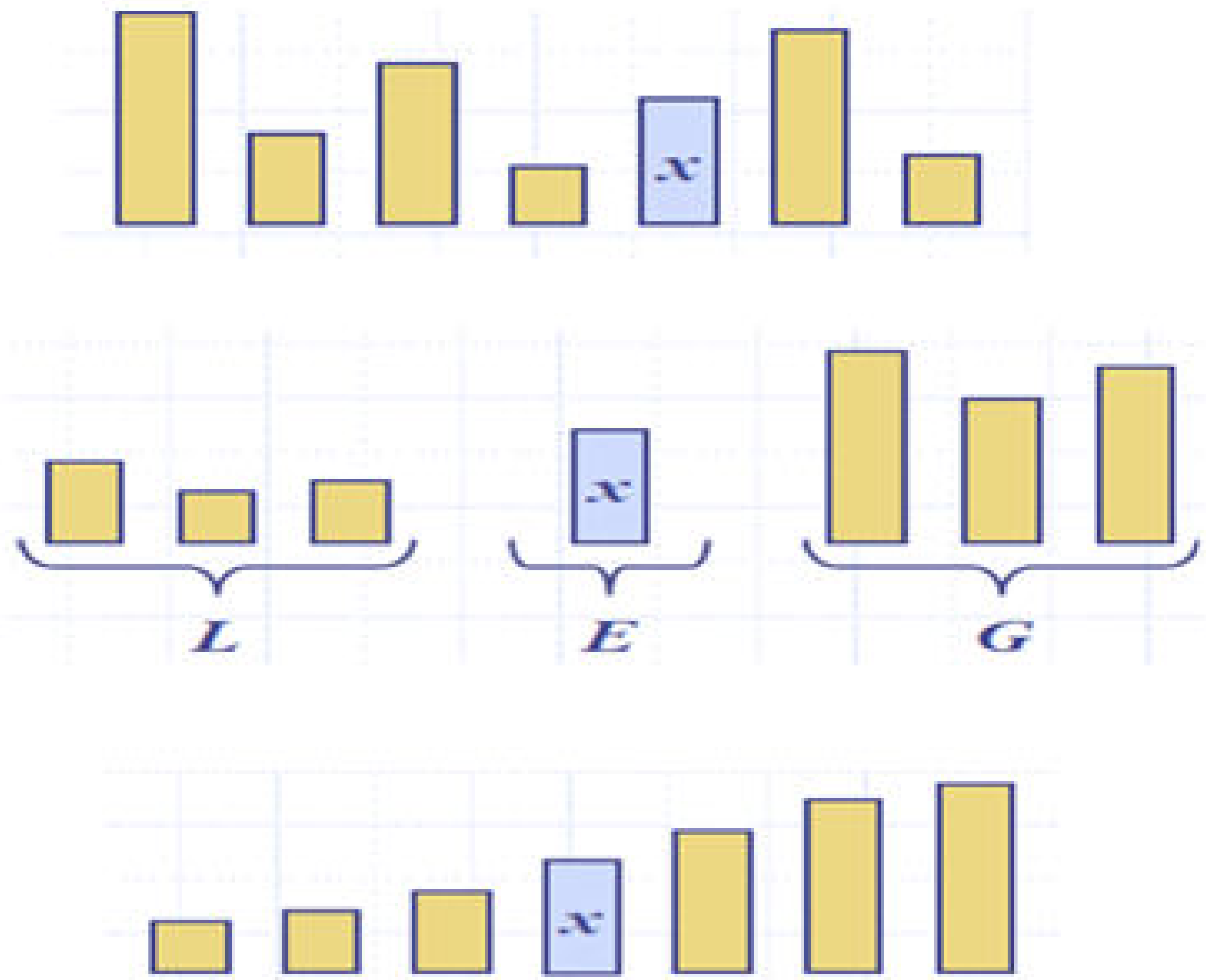
- This algorithm uses the divide- and – conquer method.
- Select a *pivot* element from the array of elements.
- Rearrange the list so that all the elements before the pivot are smaller than or equal to the pivot and those after the pivot are larger than the pivot .
- After such a partitioning, the pivot is placed in its final position.
- Sort the two sublists recursively.



# Quicksort Algorithm

Given an array of *n* elements (e.g., integers):

- If array only contains one element, return
- Else
  - pick one element to use as *pivot.*
  - Partition elements into two sub-arrays:
    - Elements less than or equal to pivot
    - Elements greater than pivot
  - Quicksort two sub-arrays
  - Return results


x
x
L
E
G
x

Recursive implementation with the left most array entry selected as the pivot element.

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 15 | 12 | 3 | 21 | 25 | 3 | 9 | 8 | 18 | 28 | 5 |
| 1 | 9 | 12 | 3 | 5 | 8 | 3 | 15 | 25 | 18 | 28 | 21 |
| 2 | 8 | 3 | 3 | 5 | 9 | 12 | 15 | 21 | 18 | 25 | 28 |
| 3 | 5 | 3 | 3 | 8 | 9 | 12 | 15 | 18 | 21 | 25 | 28 |
| 4 | 3 | 3 | 5 | 8 | 9 | 12 | 15 | 18 | 21 | 25 | 28 |
| 5 | 3 | 3 | 5 | 8 | 9 | 12 | 15 | 18 | 21 | 25 | 28 |
| 6 | 3 | 3 | 5 | 8 | 9 | 12 | 15 | 18 | 21 | 25 | 28 |

## /*Write a c program to implement the quick sort*/

```
#include<stdio.h>
void quicksort(int [10],int,int);

int main(){
        int x[20],size,i;
        printf("\nEnter size of the array: ");
        scanf("%d",&size);
        printf("\nEnter %d elements: ",size);
        for(i=0;i<size;i++)
                scanf("%d",&x[i]);
        quicksort(x,0,size-1);
        printf("\nSorted elements: ");
        for(i=0;i<size;i++)
                printf(" %d",x[i]);
        getch();
        return 0;
}
```

```
void quicksort(int x[10],int first,int last){
        int pivot,j,temp,i;
        if(first<last){
                pivot=first;
                i=first;
        j=last;
                while(i<j){
                        while(x[i]<=x[pivot]&&i<last)
                                i++;
                        while(x[j]>x[pivot])
                                j--;
                if(i<j){
                                temp=x[i];
        x[i]=x[j];              x[j]=temp;
                        }
                }
                temp=x[pivot];
                x[pivot]=x[j];
                x[j]=temp;
                quicksort(x,first,j-1);
                quicksort(x,j+1,last);
        }
}
```

pivot_index = 0

| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] |
|---|---|---|---|---|---|---|---|---|
| 40 | 20 | 10 | 80 | 60 | 50 | 7 | 30 | 100 |
| | i | | | | | | | j |

1. While x[i] <= x[pivot]
   ++i

pivot_index = 0

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 40 | 20 | 10 | 80 | 60 | 50 | 7 | 30 | 100 |
| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] |
| | i | | | | | | | j |

1. While x[i] <= x[pivot]
   ++i

pivot_index = 0

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 40 | 20 | 10 | 80 | 60 | 50 | 7 | 30 | 100 |
| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] |
| | | i | | | | | | j |

1. While x[i] <= x[pivot]
   ++i

pivot_index = 0

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 40 | 20 | 10 | 80 | 60 | 50 | 7 | 30 | 100 |
| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] |

i j

1. While x[i] <= x[pivot]
   ++i
2. While x[j] > x[pivot]
   --j

pivot_index = 0

| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] |
|---|---|---|---|---|---|---|---|---|
| 40 | 20 | 10 | 80 | 60 | 50 | 7 | 30 | 100 |
|  |  |  | i |  |  |  |  | j |

1. While x[i] <= x[pivot]

    ++i

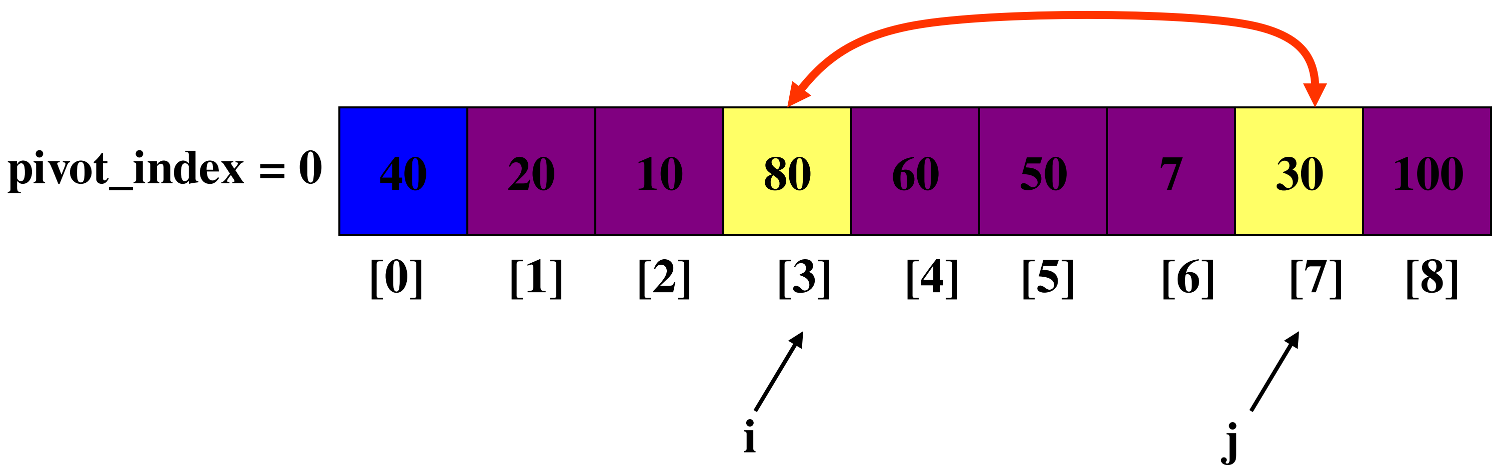
2. While x[j] > x[pivot]

    --j

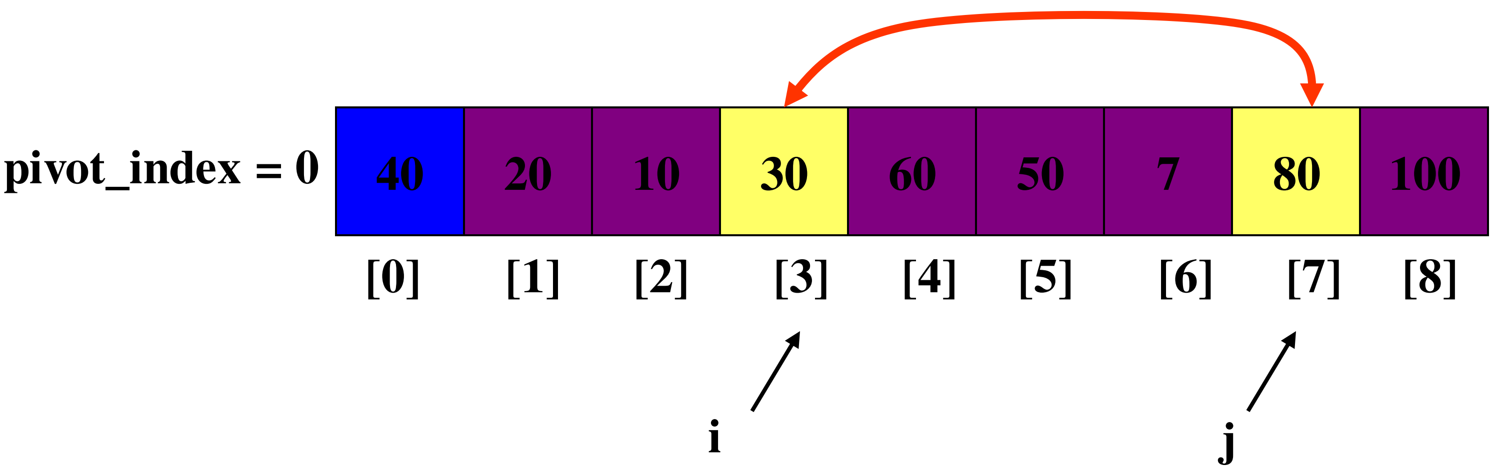
pivot_index = 0

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 40 | 20 | 10 | 80 | 60 | 50 | 7 | 30 | 100 |
| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] |
| | | | i | | | | j | |

1. While x[i] <= x[pivot]
    ++i
2. While x[j] > x[pivot]
    --j
3. If i < j
    swap x[i] and x[j]



1. While x[i] <= x[pivot]
   ++i
2. While x[j] > x[pivot]
   --j
3. If i < j
   swap x[i] and x[j]



1. While x[i] <= x[pivot]
   ++i
2. While x[j] > x[pivot]
   --j
3. If i < j
   swap x[i] and x[j]
4. While j > i, go to 1.

pivot_index = 0

| 40 | 20 | 10 | 30 | 60 | 50 | 7 | 80 | 100 |
|---|---|---|---|---|---|---|---|---|
| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] |
|  |  |  | i |  |  |  | j |  |

1. **While x[i] <= x[pivot]**
   **++i**
2. **While x[j] > x[pivot]**
   **--j**
3. **If i < j**
   **swap x[i] and x[j]**
4. **While j > i, go to 1.**

**pivot_index = 0**

| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] |
|---|---|---|---|---|---|---|---|---|
| 40 | 20 | 10 | 30 | 60 | 50 | 7 | 80 | 100 |

**i** **j**

1. While x[i] <= x[pivot]
   ++i
2. While x[j] > x[pivot]
   --j
3. If i < j
   swap x[i] and x[j]
4. While j > i, go to 1.

pivot_index = 0

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 40 | 20 | 10 | 30 | 60 | 50 | 7 | 80 | 100 |
| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] |

i j

1. While x[i] <= x[pivot]
   ++i
2. While x[j] > x[pivot]
   --j
3. If i < j
   swap x[i] and x[j]
4. While j > i, go to 1.

pivot_index = 0

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 40 | 20 | 10 | 30 | 60 | 50 | 7 | 80 | 100 |
| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] |

i j

1. While data[i] <= data[pivot]
   ++i
2. While data[j] > data[pivot]
   --j
3. If i < j
   swap data[i] and data[j]
4. While j > i, go to 1.

pivot_index = 0

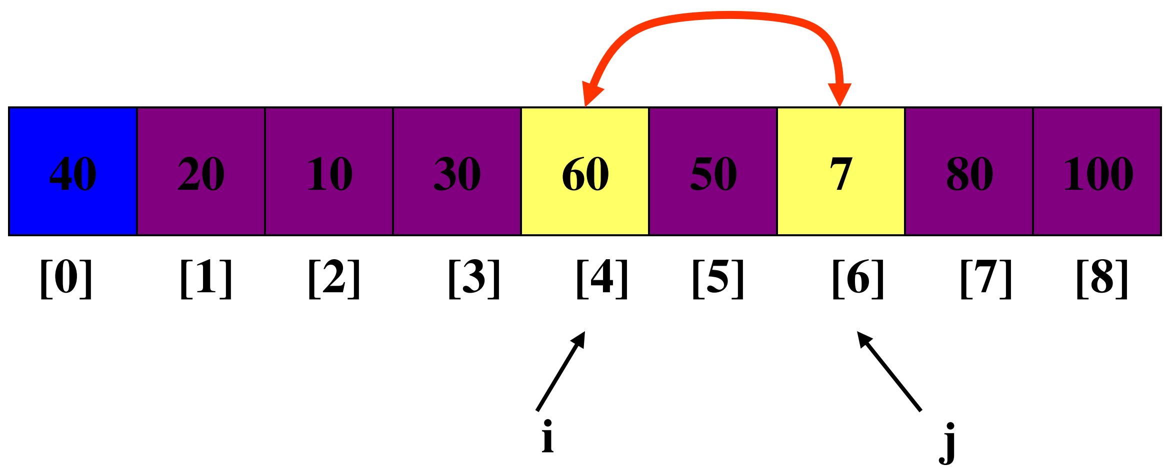
| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] |
|---|---|---|---|---|---|---|---|---|
| 40 | 20 | 10 | 30 | 60 | 50 | 7 | 80 | 100 |

i → [4]

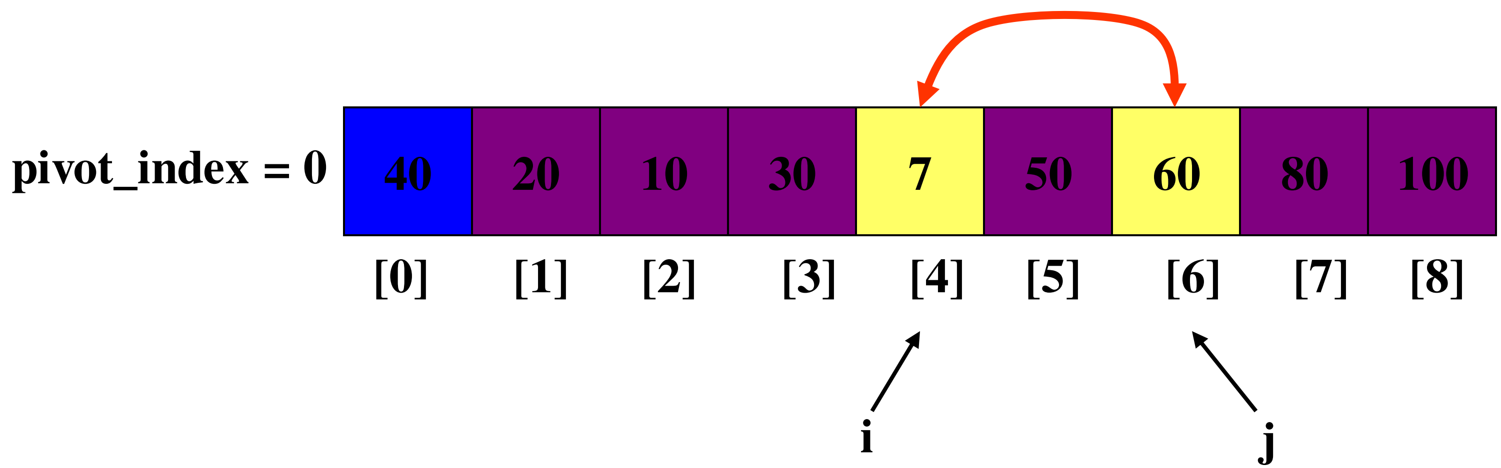
j → [6]

1. While x[i] <= x[pivot]
   ++i
2. While x[j] >x[pivot]
   --j
3. If i < j
   swap x[i] and x[j]
4. While j > i, go to 1.

pivot_index = 0



1. While x[i] <= x[pivot]
   ++i
2. While x[j] > x[pivot]
   --j
3. If i < j
   swap x[i] and x[j]
4. While j > i, go to 1.



1. While x[i] <= x[pivot]
   ++i
2. While x[j] > x[pivot]
   --j
3. If i < j
   swap x[i] and x[j]
4. While j > i, go to 1.

pivot_index = 0

| 40 | 20 | 10 | 30 | 7 | 50 | 60 | 80 | 100 |
|---|---|---|---|---|---|---|---|---|
| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] |

i j

1. While x[i] <= x[pivot]
   ++i
2. While x[j] > x[pivot]
   --j
3. If i < j
   swap x[i] and x[j]
4. While j > i, go to 1.

pivot_index = 0

| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] |
|---|---|---|---|---|---|---|---|---|
| 40 | 20 | 10 | 30 | 7 | 50 | 60 | 80 | 100 |
| | | | | i | | j | | |

1. While x[i] <= x[pivot]
   ++i
2. While x[j] > x[pivot]
   --j
3. If i < j
   swap x[i] and x[j]
4. While j > i, go to 1.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| pivot_index = 0 | 40 | 20 | 10 | 30 | 7 | 50 | 60 | 80 | 100 |
| | [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] |

i j

1. While data[i] <= data[pivot]
   ++i
2. While data[j] > data[pivot]
   --j
3. If i < j
   swap data[i] and data[j]
4. While j > i, go to 1.

pivot_index = 0

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 40 | 20 | 10 | 30 | 7 | 50 | 60 | 80 | 100 |
| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] |
| | | | | | i | j | | |

1. While data[i] <= data[pivot]
   ++i
2. While data[j] > data[pivot]
   --j
3. If i < j
   swap data[i] and data[j]
4. While j > i, go to 1.

pivot_index = 0

| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] |
|---|---|---|---|---|---|---|---|---|
| 40 | 20 | 10 | 30 | 7 | 50 | 60 | 80 | 100 |

i j

1. While x[i] <= x[pivot]
   ++i
2. While x[j] > x[pivot]
   --j
3. If i < j
   swap x[i] and x[j]
4. While j > i, go to 1.

pivot_index = 0

| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] |
|---|---|---|---|---|---|---|---|---|
| 40 | 20 | 10 | 30 | 7 | 50 | 60 | 80 | 100 |

i j

1. While x[i] <= x[pivot]
    ++i
2. While x[j] > x[pivot]
    --j
3. If i < j
    swap x[i] and x[j]
4. While j > i, go to 1.

pivot_index = 0

| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] |
|---|---|---|---|---|---|---|---|---|
| 40 | 20 | 10 | 30 | 7 | 50 | 60 | 80 | 100 |

i j

1. While x[i] <= x[pivot]
   ++i
2. While x[j] > x[pivot]
   --j
3. If i < j
   swap x[i] and x[j]
4. While j > i, go to 1.

pivot_index = 0

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 40 | 20 | 10 | 30 | 7 | 50 | 60 | 80 | 100 |
| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] |

i

j

1. While x[i] <= x[pivot]
   ++i
2. While x[j] > x[pivot]
   --j
3. If i < j
   swap x[i] and x[j]
4. While j > i, go to 1.
5. Swap x[j] and x[pivot_index]

pivot_index = 0

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 40 | 20 | 10 | 30 | 7 | 50 | 60 | 80 | 100 |
| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] |

i

j

1. While x[i] <= x[pivot]
   ++i
2. While x[j] > x[pivot]
   --j
3. If i < j
   swap x[i] and x[j]
4. While j > i, go to 1.
5. Swap x[j] and x[pivot_index]

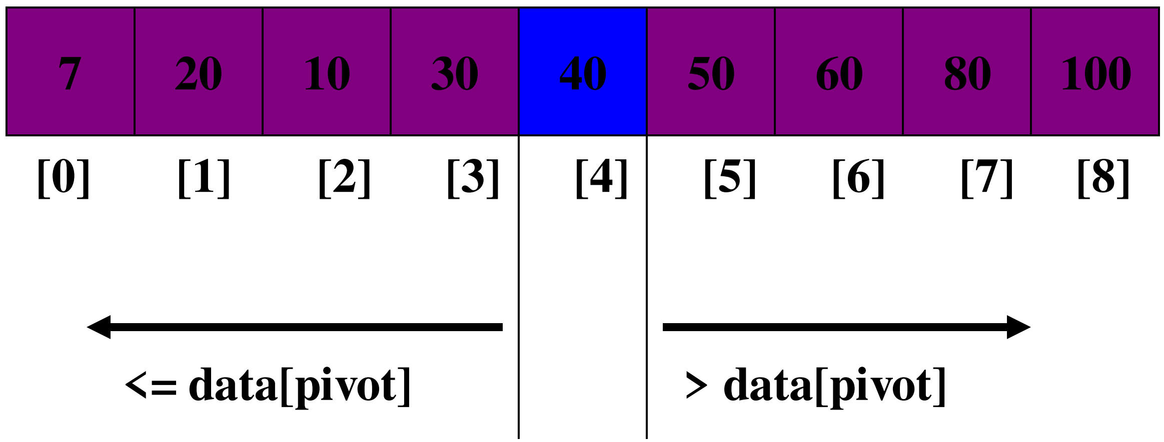
pivot_index = 4

| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] |
|---|---|---|---|---|---|---|---|---|
| 7 | 20 | 10 | 30 | 40 | 50 | 60 | 80 | 100 |

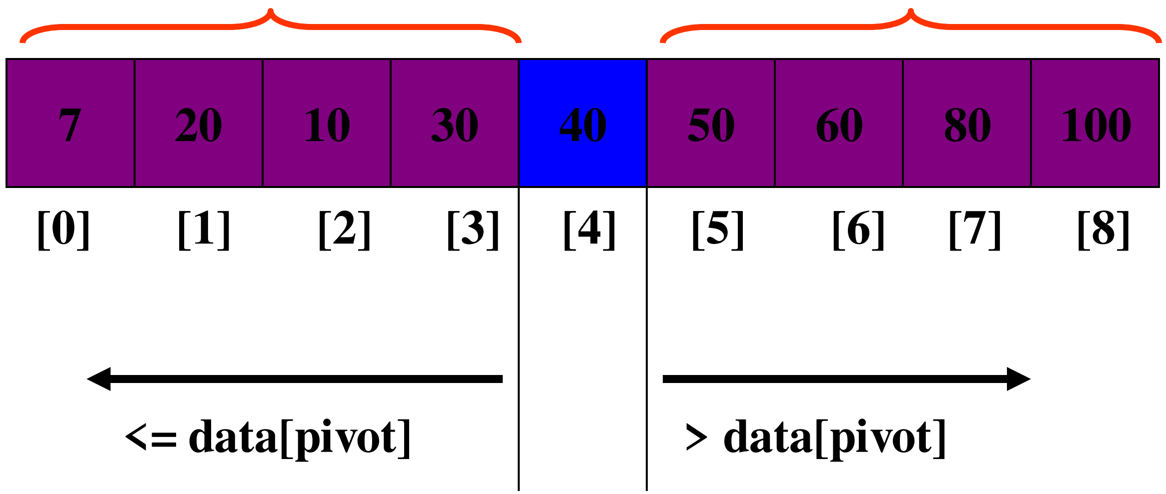
i

j

# Partition Result



# Recursion: Quicksort Sub-arrays



- Example

44,33,11,55,77,90,40,60,99,22,66

- **Big O Notation**
  - Indicates the worst-case run time for an algorithm.
  - In other words, how hard an algorithm has to work to solve a problem.

**Time Complexities of Searching & Sorting Algorithms:**

| | Average Case | Worst Case |
|---|---|---|
| Linear Search | - | O(n) |
| Binary Search | O(log n) | O(n) |
| Bubble Sort | $O(n^2)$ | $O(n^2)$ |
| Selection Sort | - | $O(n^2)$ |
| Insertion Sort | - | $O(n^2)$ |
| Merge Sort | O(n log n) | O(n log n) |
| Quick Sort | O(n log n) | $O(n^2)$ |

## Review of Algorithms

- Selection Sort
  - An algorithm which orders items by repeatedly looking through remaining items to find the least one and moving it to a final location
- Bubble Sort
  - Sort by comparing each adjacent pair of items in a list in turn, swapping the items if necessary, and repeating the pass through the list until no swaps are done
- Insertion Sort
  - Sort by repeatedly taking the next item and inserting it into the final data structure in its proper order with respect to items already inserted.
- Merge Sort
  - An algorithm which splits the items to be sorted into two groups, recursively sorts each group, and merges them into a final, sorted sequence
- Quick Sort
  - An in-place sort algorithm that uses the divide and conquer paradigm. It picks an element from the array (the pivot), partitions the remaining elements into those greater than and less than this pivot, and recursively sorts the partitions.