

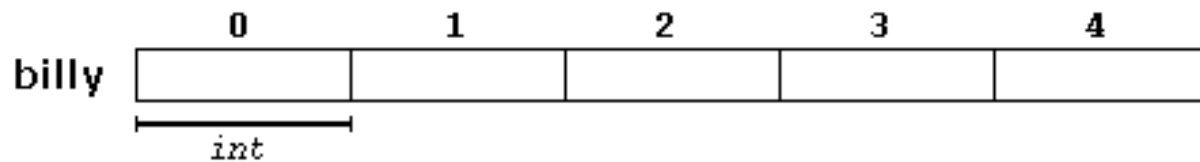
# III SEM- CSE DATA STRUCTURE-I

## UNIT : II TOPIC: ARRAYS

# Introduction to Arrays

- Up to now, we have used simple data types (int, char, etc.) except for strings.
- Now we look at the first data structure. A data structure is a **compound** data type; that is, it is a single entity made up of numerous parts.
- **Definition:** an array is a data structure consisting of an ordered set of data values of the same type. Think of it as a numbered list, but all elements on the list must be the same kind of thing (the same data type).
- An array is a series of elements of the same type placed in contiguous memory locations.

- Where each blank panel represents an element of the array, that in this case are integer values of type `int`.
- These elements are numbered from 0 to 4 since in arrays the first index is always 0, independently of its length.
- For example, an array to contain 5 integer values of type `int` called `bill` could be represented like this:



- Arrays are essentially a way to store many values under the same name.
- You can make an array out of any data-type including structures and classes.
- Think about arrays like this:
  - [ ][ ][ ][ ][ ][ ]
- Each of the bracket pairs is a slot (element) in the array, and you can put information into each one of them.
- It is almost like having a group of variables side by side.

# UNIT 2 CONTENTS

- Overview of arrays
- array based algorithms - searching and sorting:  
merge sort, quick sort,
- Sparse matrices.

# Arrays

- When defining an array in a program, we have to specify three things:
  - what kind of data it can hold (ints, chars, doubles, strings, etc.)
  - how many values it can hold (i.e. the maximum number it can hold)
  - a name for it
- To declare an array that can hold up to 10 integers:  
    int ar[10];
  - This reserves *space* for 10 integers. It does ***not*** place ***any*** values into the array.
  - Each position in the array is called an array element.
  - The elements in this array are numbered from 0 to 9. (**Not** 1 to 10.)

# Arrays

- You must use a literal number or a previously declared *const int* or #defined constant in an array declaration. No variables.
- In our code when we need to refer to a particular element in an array , we use *subscript* notation: [n]
- n can be:
  - an integer literal: [3]
  - an integer variable: [n]
  - an integer expression: [n+1]
  - any scalar expression: ['A'] (= to [65] // ASCII value of 'A')
  - any function that returns an integer value

# Arrays

- Note that you never use [n] by itself. You use it to specify **which** array **element** you refer to, but you need also to say **which array**:
- So to refer to the value of a particular array element, we write
- arrayname[array element];
- So, for example, to assign values to (i.e. store values into) an array:

ar[0] = 1;

ar[1] = 3;

ar[2] = 5;

...

ar[9] = 19;

Or:

for (i = 0; i < 10; i++)

ar[i] = (i\*2) + 1;

- Notice how the variable *i*, used as a subscript, runs from 0 to "less than" 10, i.e. from 0 to 9.



# Example

- To add up the 1st ten elements in an array:

```
sum = 0;
```

```
for (i = 0; i < 10; i++)
```

```
sum += ar[i];
```

- Remember:
  - [n] specifies which element you are referring to
  - ar[n] evaluates to the value stored in the array at element n
  - elements are numbered starting at 0
  - when you declare an array to hold n elements, its subscripts go from 0 to n-1

# Arrays

- You can initialize small arrays in the array declaration:
- `int ar[5] = {1, 3, 5, 7, 9}; // exactly enough values`
- `int ar[] = {1, 4, 9, 16}; // allocates 4 elements, 0..3`
- `char vowels[] = {'a', 'e', 'i', 'o', 'u'}; // 5 elements`
- `double sqrts[4] = {1.0, 1.414, 1.732, 2.0};`
- Also to initialize array elements, you can use executable program statements to assign computed values, or you can read data from keyboard or disk file.

# Arrays

- To increment the *i*th element:
  - `ar[i]++;`
  - `ar[i] += 1;`
  - `ar[i] = ar[i] + 1;`
- To add *n* to the *i*th element:
  - `ar[i] += n;`
  - `ar[i] = ar[i] + n;`
- To copy the contents of the *i*th element to the *k*th element:
  - `ar[k] = ar[i];`
- To exchange the values in `ar[i]` and `ar[k]`:
  - First, understand that you must declare a "temporary" variable to hold one value, and that it should be the same data type as the array elements being swapped:
  - `int temp; temp = ar[i];`      `//save a copy of value in I`
  - `ar[i] = ar[j];`      `//copy value from j to i`
  - `ar[j] = temp;`      `//copy saved value from i to j`

# One Dimensional Array

- Declaring 1 D Array

A typical declaration for an array in C is:

`type name [elements];`

Example : `int billy [5];`

- Initializing 1 D Array

`int billy[5] = { 16, 2, 77, 40, 12071 };`

This declaration would have created an array like this:

The amount of values between braces { } must not be larger than the number of elements that we declare for the array between square brackets [ ].

	0	1	2	3	4
billy	16	2	77	40	12071

# Insertion to Array

- **Example :**

```
for (i = 0; i <= 29; i++)  
{  
    printf ( "\n Enter Salary := " );  
    scanf ("%d", &salary[i]);  
}
```

The for loop causes the process of asking for and receiving a salary from the user to be repeated 30 times..

# Reading from an Array

- **Example :**

```
for (i = 0; i <= 29; i++)
```

```
{
```

```
    printf(" The salary of %d person is %d", I, salary[I]);
```

```
}
```

**OUTPUT :**

The salary of 0 person is 1000

The salary of 1 person is 5000

- The process is repeated 29 times till the last element of the array is displayed.

# Array Manipulation

- An individual array element can be used in a similar manner that a simple variable is used .

- **For example:**

**num[5] = 2 ;** this means assigns 2 to 6th element of num.

**p = (net[3] + amount[9]) / 2;**

assigns the average value of 2nd element of net and 10<sup>th</sup> element amount to P.

**--num[8];** means decrement the of 9<sup>th</sup> element of num by 1

**I = 5; p = num[++I]** assigns the value of **num[6]** to P.

# **Program to Generate First 15 Fibonacci Series Using Array**



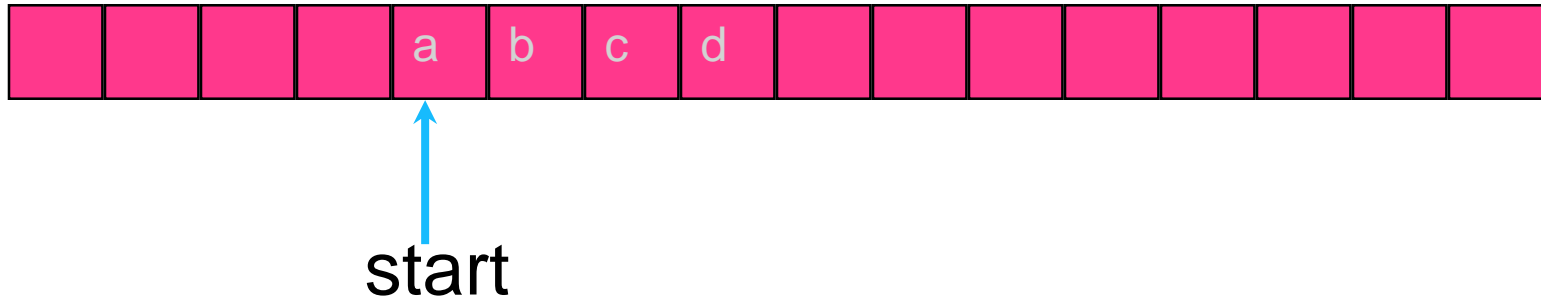
```
#include<stdio.h>
#include<conio.h>
void main( )
{
    int fibo[16] , i ;
    fibo[0]=0;
    fibo [1]=1;
    for(i=2;i<=15;i++)/* calculating fibo elements and storing in array fibo*/
    {
        fibo[i]= fibo[i-1] + fibo[i-2];
    }
    for(i=0;i<=15;i++)
        printf("%d\t", fibo[i]);
    getch( ) ;
}
```

**Output:-**

0 1 1 2 3 5 8 13 21 34 55 89 154 243 397

# 1D Array Representation In “C”.

Memory



- 1-dimensional array  $x = [a, b, c, d]$
- map into contiguous memory locations
- $\text{location}(x[i]) = \text{start} + i$

# 2D Arrays

- The elements of a 2-dimensional array `a` declared as:

```
int [][]a = new int[3][4];
```

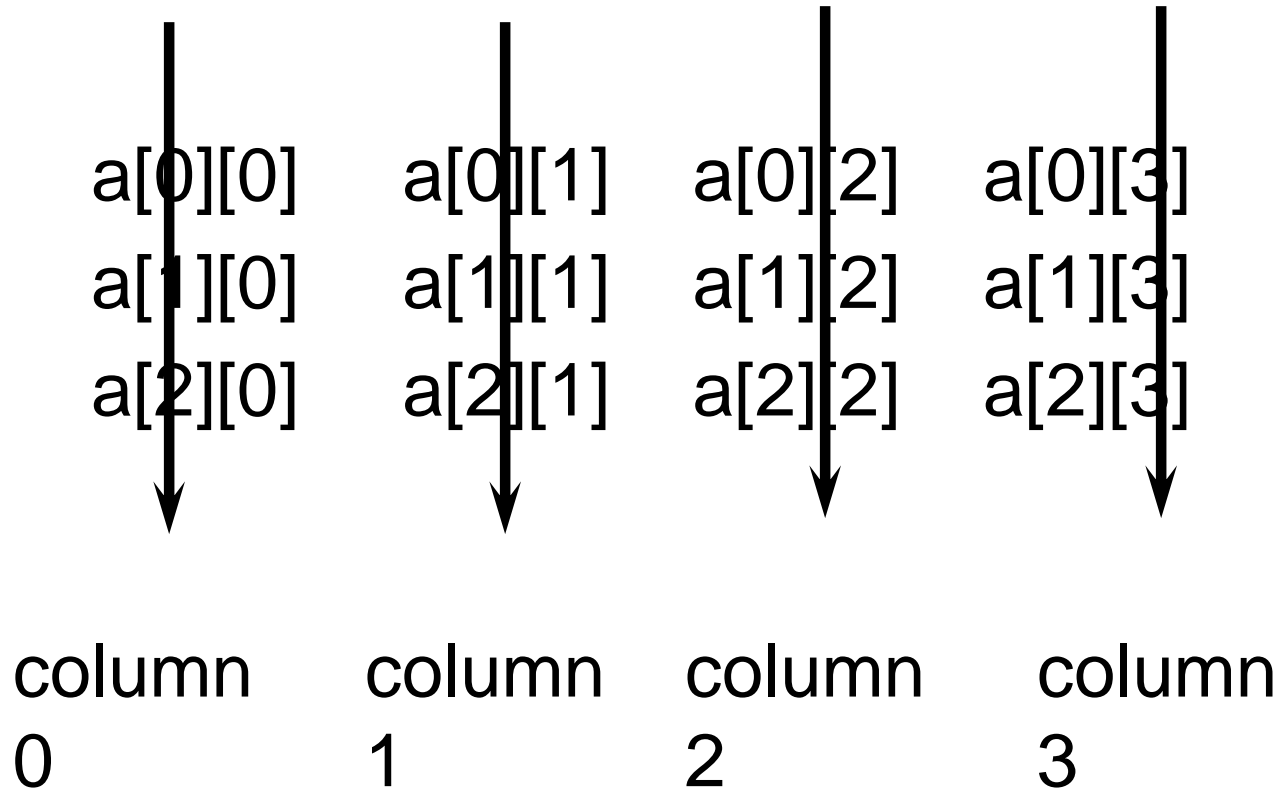
may be shown as a table

<code>a[0][0]</code>	<code>a[0][1]</code>	<code>a[0][2]</code>	<code>a[0][3]</code>
<code>a[1][0]</code>	<code>a[1][1]</code>	<code>a[1][2]</code>	<code>a[1][3]</code>
<code>a[2][0]</code>	<code>a[2][1]</code>	<code>a[2][2]</code>	<code>a[2][3]</code>

# Rows Of A 2D Array

<del>a[0][0]</del>	<del>a[0][1]</del>	<del>a[0][2]</del>	<del>a[0][3]</del>	→ row 0
<del>a[1][0]</del>	<del>a[1][1]</del>	<del>a[1][2]</del>	<del>a[1][3]</del>	→ row 1
<del>a[2][0]</del>	<del>a[2][1]</del>	<del>a[2][2]</del>	<del>a[2][3]</del>	→ row 2

# Columns Of A 2D Array



# 2D Array Representation In “C”.

- 2-dimensional array x

a, b, c, d

e, f, g, h

i, j, k, l

- view 2D array as a 1D array of rows

x = [row0, row1, row 2]

row 0 = [a,b, c, d]

row 1 = [e, f, g, h]

row 2 = [i, j, k, l]

and store as 4 1D arrays

# Row-Major Mapping

- Example 3 x 4 array:

a b c d  
e f g h  
i j k l

- Convert into 1D array  $y$  by collecting elements by rows.
- Within a row elements are collected from left to right.
- Rows are collected from top to bottom.
- We get  $y[] = \{a, b, c, d, e, f, g, h, i, j, k, l\}$



## row -Major Mapping

```
for (row=0;row<3;row++)  
{  
  for(col=0;col<4;col++)  
  {  
    printf("%d", a[ row][ col]) // 00,  
    01,02,03,10,11,12,13,20,21,22,23
```



# Column -Major Mapping

```
for(col=0;col<4;col++)  
{  
  for (row=0<;row<3;row++)  
  {  
    for (  
      a1[ a[ row][ col]) //00,10,20,01,11,21,02,12,22
```

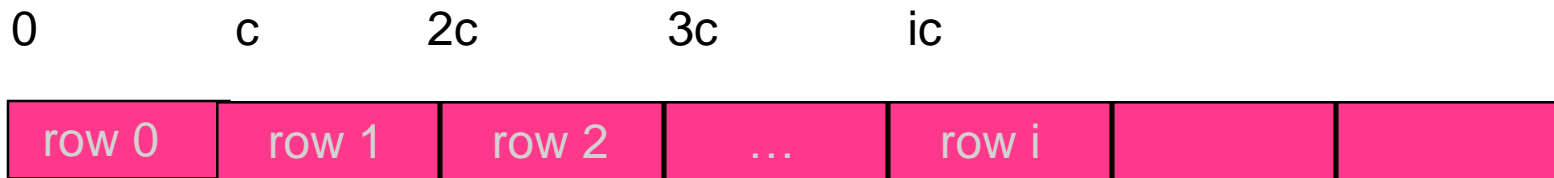
# Column -Major Mapping

<b>a[0][0]</b>
a[1][0]
a[2][0]
a[3]
a[4]
a[5]

# row -Major Mapping

<b>a[0][0]</b>
a[0][1]
a[0][2]
a[]
a[4]
a[5]

# Locating Element $x[i][j]$



- assume  $x$  has  $r$  rows and  $c$  columns
- each row has  $c$  elements
- $i$  rows to the left of row  $i$
- so  $ic$  elements to the left of  $x[i][0]$
- so  $x[i][j]$  is mapped to position  $ic + j$  of the 1D array

# Matrix

- Table of values. Has rows and columns, but numbering begins at 1 rather than 0.

a b c d      row 1

e f g h      row 2

i j k l      row 3

- Use notation  $x(i,j)$  rather than  $x[i][j]$ .
- May use a 2D array to represent a matrix.

# Diagonal Matrix

- DEFINITION :
- An  $n \times n$  matrix in which all nonzero terms are on the diagonal.

# Diagonal Matrix

1 0 0 0

0 2 0 0

0 0 3 0

0 0 0 4

- $x(i,j)$  is on diagonal iff  $i = j$
- number of diagonal elements in an  $n \times n$  matrix is  $n$
- non diagonal elements are zero
- store diagonal only vs  $n^2$  whole

# Lower Triangular Matrix

- An  $n \times n$  matrix in which all nonzero terms are either on or below the diagonal.

1	0	0	0
2	3	0	0
4	5	6	0
7	8	9	10

- $x(i,j)$  is part of lower triangle iff  $i \geq j$ .
- number of elements in lower triangle is  $1 + 2 + \dots + n = n(n+1)/2$ .
- store only the lower triangle



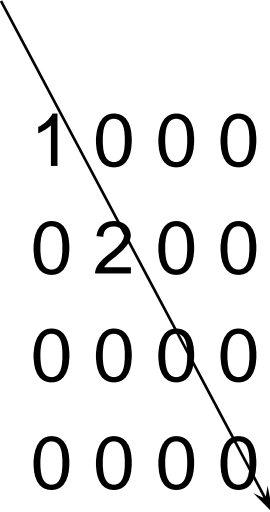
# Sparse Matrices

- Sparse ... many elements are zero.
- Dense ... few elements are zero.

# Example Of Sparse Matrices

- Diagonal.
  - upper triangular
  - lower triangular.
- 
- These are structured sparse matrices.
- 
- May be mapped into a 1D array so that a mapping function can be used to locate an element.

# sparse Matrix- 3 tuple form



1 0 0 0  
0 2 0 0  
0 0 0 0  
0 0 0 0

4 4 2

0 0 1

1 1 2

# Introduction to Sorting

## What is Sorting?

Sorting: an operation that segregates eles into groups according to specified criterion.

**$A = \{ 3 \ 1 \ 6 \ 2 \ 1 \ 3 \ 4 \ 5 \ 9 \ 0 \}$**

**$A = \{ 0 \ 1 \ 1 \ 2 \ 3 \ 3 \ 4 \ 5 \ 6 \ 9 \}$**

# Example: sorting numbers.

- Input: A sequence of  $n$  numbers  $\{a_1, a_2, \dots, a_n\}$
- Output: A reordered sequence of the input  $\{a_1, a_2, a_3, \dots, a_n\}$  such that  $a_1 \leq a_2 \leq a_3 \dots \leq a_n$ .
- Input instance:  $\{5, 2, 4, 1, 6, 3\}$ .
- Output :  $\{1, 2, 3, 4, 5, 6\}$ .
- An instance of a problem consists of the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem.

# Why Sort and Examples

Consider:

- Sorting Books in Library (Dewey system)
- Sorting Individuals by Height (Feet and Inches)
- Sorting Movies in Blockbuster (Alphabetical)
- Sorting Numbers (Sequential)

# 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
- Heap Sort
- Quick Sort
- Radix Sort
- Swap Sort

# Bubble Sort.

- Bubble sort
  - Strategy
    - Compare adjacent elements and exchange them if they are out of order
    - Comparing the first two elements, the second and third elements, and so on, will move the largest (or smallest) elements to the end of the array
    - Repeating this process will eventually sort the array into ascending (or descending) order.



# Bubble Sort.

(a) Pass 1

Initial array:

29	10	14	37	13
10	29	14	37	13
10	14	29	37	13
10	14	29	37	13
10	14	29	13	<b>37</b>

(b) Pass 2

10	14	29	13	<b>37</b>
10	14	29	13	<b>37</b>
10	14	29	13	<b>37</b>
10	14	13	<b>29</b>	<b>37</b>

```

i j
0 0  a[0]&a[1]
    1  a[1]&a[2]
    2  a[2]&a[3]
    3
1  0  a[0]&a[1]
    1  a[1]&a[2]
    2  a[2]&a[3]
    3
2  0  a[0]&a[1]
    1  a[1]&a[2]
    2  a[2]&a[3]
    3

```

```

\\bubble sort n=4
for(i=0;i<n-1;i++)
{
for(j=0;j<n-i-1;j++)
{
if(a[j]>a[j+1])
{ t= a[j];
a[j]=a[j+1];
a[j+1]=t;
}
}
}
}

```

# time complexity

main() ---1

{ ----1

int i=0, n=5;-----1

while(i<n) n {--n-1

printf("hello");

i++;----3

}----3

}----1

$O(n+n-1+n-1+n-1+4)$   $O(4n+1) = O(n)$

012345

## time complexity

n=5

i=0,1,2,3,4,5

```
main()
{
for(i=0;i<n;i++) -- n+1
    for(j=0;j<n;j++) ---
n(n+1)
printf(“%d%d”, i,j)--n*n
}
}
```

$n+1+n(n+1)+n^2$

$2n^2 == O(n^2)$

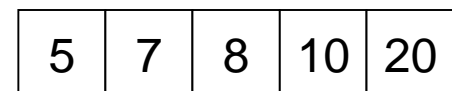
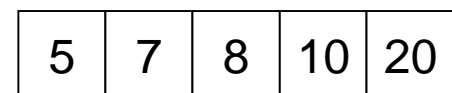
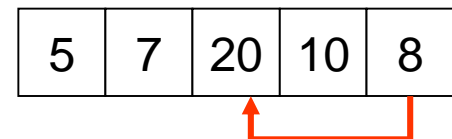
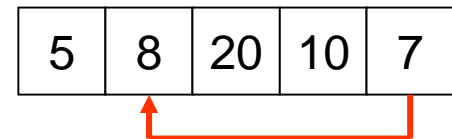
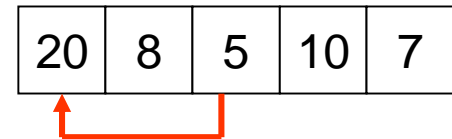
# Bubble Sort.

- Analysis
  - Worst case:  $O(n^2)$
  - Best case:  $O(n)$

# Selection Sorting

Step:

1. select the smallest element  
among  $data[i] \sim data[data.length-1]$ ;
2. swap it with  $data[i]$ ;
3. if not finishing, repeat 1&2



# Pseudo-code for Insertion Sorting

- Place ith ele in proper position:
  - `temp = data[i]`
  - shift those elements `data[j]` which greater than `temp` to right by one position.
  - place `temp` in its proper position.

# Insert Action: $i=1$

temp

8
---

20	8	5	10	7
----	---	---	----	---

$i = 1$ , first iteration

8
---

20	20	5	10	7
----	----	---	----	---

---
-----

8	20	5	10	7
---	----	---	----	---



# Insert Action: $i=2$

temp

5

8	20	5	10	7
---	----	---	----	---

$i = 2$ , second iteration

5

8	20	20	10	7
---	----	----	----	---

5

8	8	20	10	7
---	---	----	----	---

---

5	8	20	10	7
---	---	----	----	---

# Insert Action: $i=3$

temp

10
----

5	8	20	10	7
---	---	----	----	---

$i = 3$ , third iteration

10
----

5	8	20	20	7
---	---	----	----	---

---
-----

5	8	10	20	7
---	---	----	----	---

# Insert Action: $i=4$

temp

7

5 8 10 20 7

$i = 4$ , forth  
iteration

7

5 8 10 20 20

7

5 8 10 10 20

7

5 8 8 10 20

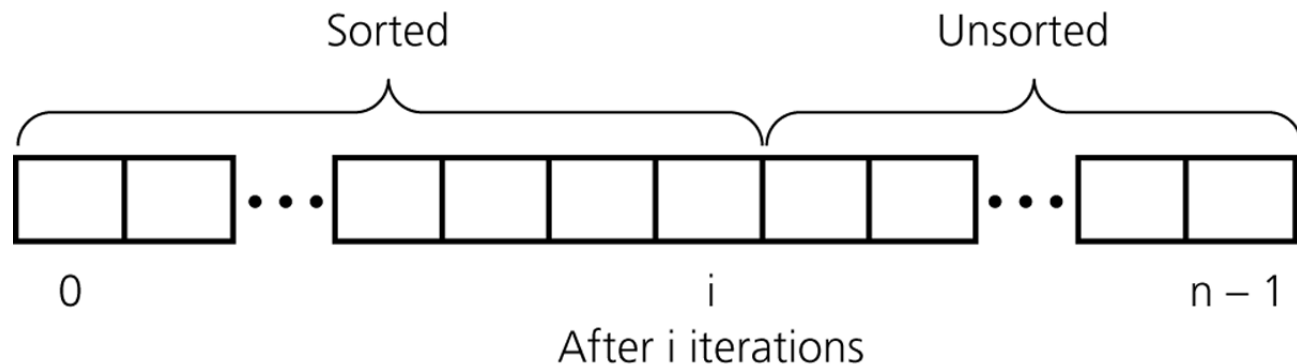
---

5 7 8 10 20

# Insertion Sort

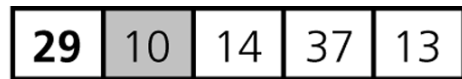
## Insertion sort

- Strategy
  - Partition the array into two regions: sorted and unsorted.
  - Take each ele from the unsorted region and insert it into its correct order in the sorted region.

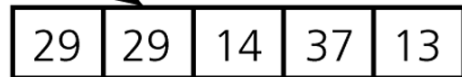


# Insertion Sort

Initial array:



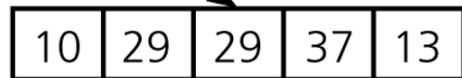
Copy 10



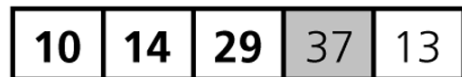
Shift 29



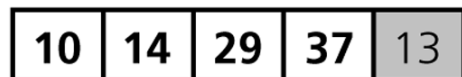
Insert 10; copy 14



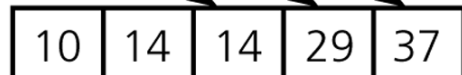
Shift 29



Insert 14; copy 37, insert 37 on top of itself



Copy 13



Shift 37, 29, 14

Sorted array:



Insert 13

Figure :

An insertion sort of an array of five integers.

# Insertion Sort

## ■ Analysis

- Worst case:  $O(n^2)$ .
- For small arrays.
- Insertion sort is appropriate due to its simplicity.
- For large arrays.
- Insertion sort is prohibitively inefficient.

# Quick sort.

- Quick sort
  - A divide-and-conquer algorithm
  - Strategy
    - Partition an array into elements that are less than the pivot and those that are greater than or equal to the pivot
    - Sort the left section
    - Sort the right section

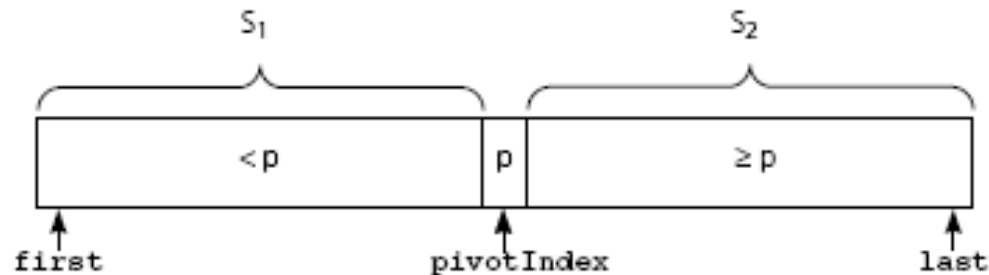


Figure :

A partition about a pivot

# Quick sort.

- Using an invariant to develop a partition algorithm
  - Invariant for the partition algorithm
  - The elements in region  $S_1$  are all less than the pivot, and those in  $S_2$  are all greater than or equal to the pivot.

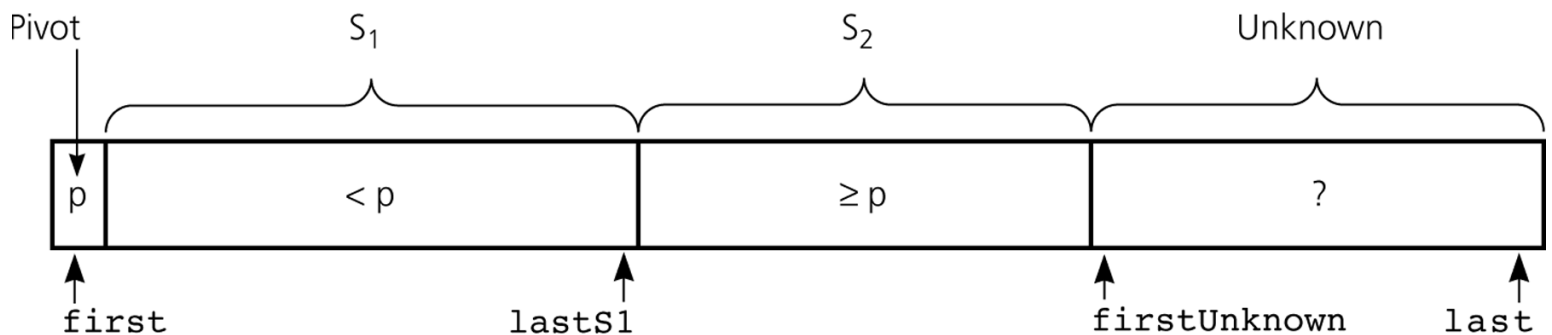


Figure :  
Invariant for the partition algorithm



# Quick sort.

- Analysis
- Worst case
  - quick sort is  $O(n^2)$  when the array is already sorted and the smallest ele is chosen as the pivot

Original array:

5	6	7	8	9
---	---	---	---	---

Pivot | Unknown

5	6	7	8	9
---	---	---	---	---

Pivot |  $S_2$  | Unknown

5	6	7	8	9
---	---	---	---	---

$S_1$  is empty

Pivot |  $S_2$  | Unknown

5	6	7	8	9
---	---	---	---	---

$S_1$  is empty

Pivot |  $S_2$  | Unknown

5	6	7	8	9
---	---	---	---	---

$S_1$  is empty

Pivot |  $S_2$

First partition:

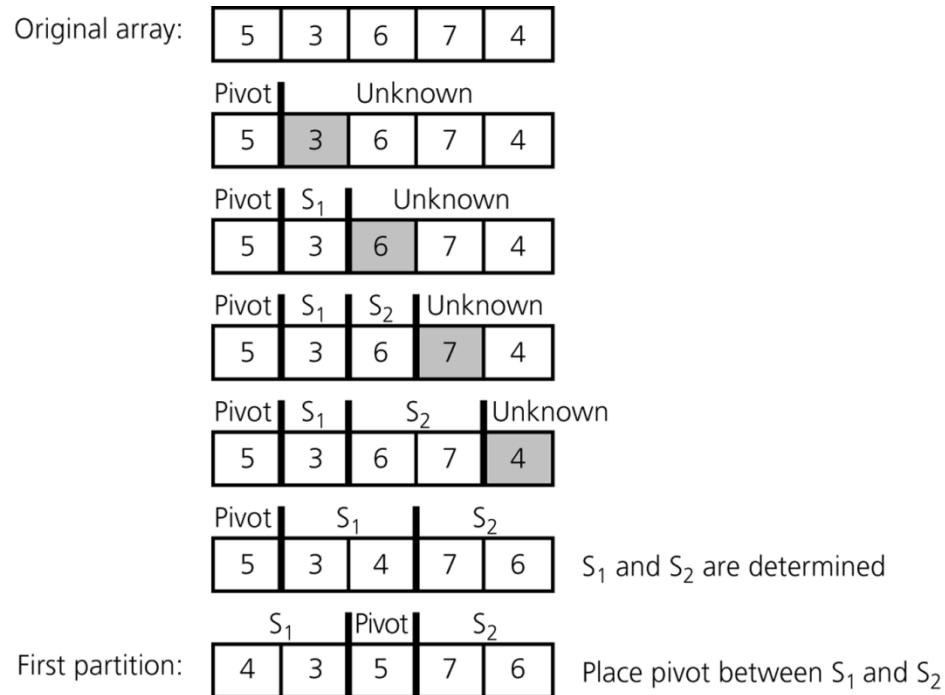
5	6	7	8	9
---	---	---	---	---

$S_1$  is empty

4 comparisons, 0 exchanges

# Quick sort.

- Analysis
- Average case
  - quick sort is  $O(n * \log_2 n)$  when  $S_1$  and  $S_2$  contain the same – or nearly the same – number of elements arranged at random



# Quick sort

Procedure 6.7: QUICK(A, N, BEG, END, LOC)

Here A is an array with N elements. Parameters BEG and END contain the boundary values of the sublist of A to which this procedure applies. LOC keeps track of the position of the first element A[BEG] of the sublist during the procedure. The local variables LEFT and RIGHT will contain the boundary values of the list of elements that have not been scanned.

1. [Initialize.] Set  $LEFT := BEG$ ,  $RIGHT := END$  and  $LOC := BEG$ .
2. [Scan from right to left.]
  - (a) Repeat while  $A[LOC] \leq A[RIGHT]$  and  $LOC \neq RIGHT$ :  
 $RIGHT := RIGHT - 1$ .  
[End of loop.]

# Quick sort contd...

- (b) If  $LOC = RIGHT$ , then: Return.
  - (c) If  $A[LOC] > A[RIGHT]$ , then:
    - (i) [Interchange  $A[LOC]$  and  $A[RIGHT]$ .]  
     $TEMP := A[LOC]$ ,  $A[LOC] := A[RIGHT]$ ,  
     $A[RIGHT] := TEMP$ .
    - (ii) Set  $LOC := RIGHT$ .
    - (iii) Go to Step 3.
- [End of If structure.]
3. [Scan from left to right.]
- (a) Repeat while  $A[LEFT] \leq A[LOC]$  and  $LEFT \neq LOC$ :  
     $LEFT := LEFT + 1$ .  
[End of loop.]
  - (b) If  $LOC = LEFT$ , then: Return.
  - (c) If  $A[LEFT] > A[LOC]$ , then
    - (i) [Interchange  $A[LEFT]$  and  $A[LOC]$ .]  
     $TEMP := A[LOC]$ ,  $A[LOC] := A[LEFT]$ ,  
     $A[LEFT] := TEMP$ .
    - (ii) Set  $LOC := LEFT$ .
    - (iii) Go to Step 2.
- [End of If structure.]

Algorithm 6.8: (Quicksort) This algorithm sorts an array  $A$  with  $N$  elements.

```

#include <stdio.h>
void quicksort (int [], int, int);
int main()
{
    int list[50];
    int size, i;
    printf("Enter the number of elements: ");
    scanf("%d", &size);
    printf("Enter the elements to be sorted:\n");
    for (i = 0; i < size; i++)
    {
        scanf("%d", &list[i]);
    }
    quicksort(list, 0, size - 1);
    printf("After applying quick sort\n");
    for (i = 0; i < size; i++)
    {
        printf("%d ", list[i]);
    }
    printf("\n");
    return 0;
}

```

```

void quicksort(int list[], int low, int high)
{
    int pivot, i, j, temp;
    if (low < high)
    {
        pivot = low;    i = low;    j = high;
        while (i < j)
        {
            while (list[i] <= list[pivot] && i <= high)
            {
                i++;
            }
            while (list[j] > list[pivot] && j >= low)
            {
                j--;
            }
            if (i < j)
            {
                temp = list[i];
                list[i] = list[j];
                list[j] = temp;
            }
        }
    }
}

```

```
temp = list[j];  
    list[j] = list[pivot];  
    list[pivot] = temp;  
    quicksort(list, low, j - 1);  
    quicksort(list, j + 1, high);  
}  
}
```

# Quick sort.

- Analysis

- Quick sort is usually extremely fast in practice.
- Even if the worst case occurs, quicksort's performance is acceptable for moderately large arrays.



# Merge sort.

- Important divide-and-conquer sorting algorithms
- Merge sort
  - A recursive sorting algorithm
  - Gives the same performance, regardless of the initial order of the array elements
  - **Strategy:**
  - Divide an array into halves
  - Sort each half
  - Merge the sorted halves into one sorted array

# Merge sort

theArray: 

8	1	4	3	2
---	---	---	---	---

Divide the array in half

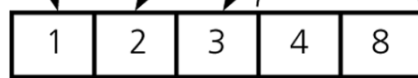


Sort the halves

Merge the halves:

- a.  $1 < 2$ , so move 1 from left half to tempArray
- b.  $4 > 2$ , so move 2 from right half to tempArray
- c.  $4 > 3$ , so move 3 from right half to tempArray
- d. Right half is finished, so move rest of left half to tempArray

Temporary array  
tempArray:



Copy temporary array back into  
original array

theArray:



Figure :

A merge sort with an auxiliary temporary array.

# Merge sort.

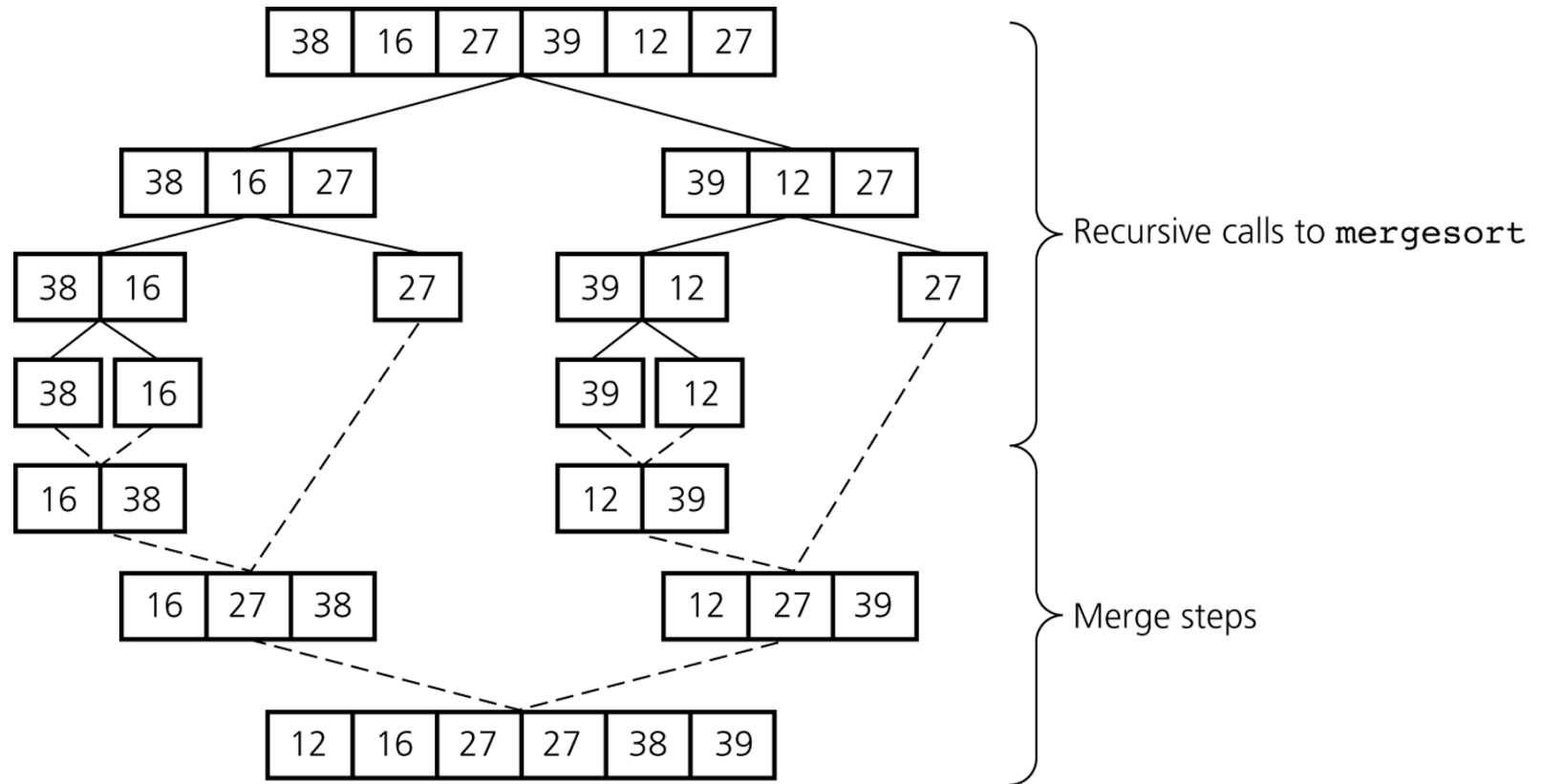


Figure :

A merge sort of an array of six integers.

# Merging 2 sorted arrays

```
#include<stdio.h>
int main()
{
    int n1, n2, n3, arr1[100],arr2[100] arr3[200];
    scanf("%d", &n1);
    for(int i=0; i<n1; i++)
        scanf("%d", &arr1[i]);
    scanf("%d", &n2);
    for(int i=0; i<n2; i++)
        scanf("%d", &arr2[i]);
    merge(arr1, arr2, n1, n2, arr3);
    printf("Array after merging\n");
    for (int i=0; i < n1+n2; i++)
        printf("%d ", arr3[i]);
    return 0;
}
```

# Merging 2 sorted arrays

```
void merge(int arr1[], int arr2[], int n1, int n2, int arr3[])
{
    int i = 0, j = 0, k = 0;

    while (i < n1 && j < n2)
    {
        if (arr1[i] < arr2[j])
            arr3[k++] = arr1[i++];
        else
            arr3[k++] = arr2[j++];
    }
    while (i < n1)
        arr3[k++] = arr1[i++];

    while (j < n2)
        arr3[k++] = arr2[j++];
}
```

# Output

4

2 4 5 7

3

1 3 6

Array after merging :

1 2 3 4 5 6 7

# “Divide and Conquer”

- Very important strategy in computer science:
  - Divide problem into smaller parts
  - Independently solve the parts
  - Combine these solutions to get overall solution
- **Idea 1**: Divide array into two halves, *recursively* sort left and right halves, then *merge* two halves → **Mergesort**
- **Idea 2** : Partition array into items that are “small” and items that are “large”, then recursively sort the two sets → **Quicksort**

# Mergesort

- **A divide-and-conquer algorithm:**
- **Divide the unsorted array into 2 halves until the sub-arrays only contain one element**
- **Merge the sub-problem solutions together:**
  - **Compare the sub-array's first elements**
  - **Remove the smallest element and put it into the result array**
  - **Continue the process until all elements have been put into the result array**



98	23	45	14	6	67	33	42
----	----	----	----	---	----	----	----

98	23	45	14	6	67	33	42
----	----	----	----	---	----	----	----

98	23	45	14
----	----	----	----

6	67	33	42
---	----	----	----

98	23	45	14	6	67	33	42
----	----	----	----	---	----	----	----

98	23	45	14
----	----	----	----

6	67	33	42
---	----	----	----

98	23
----	----

45	14
----	----

98	23	45	14	6	67	33	42
----	----	----	----	---	----	----	----

98	23	45	14
----	----	----	----

6	67	33	42
---	----	----	----

98	23
----	----

45	14
----	----

98
----

23
----

98	23	45	14	6	67	33	42
----	----	----	----	---	----	----	----

98	23	45	14
----	----	----	----

6	67	33	42
---	----	----	----

98	23
----	----

45	14
----	----

98	23
----	----

[Merge]

98	23	45	14	6	67	33	42
----	----	----	----	---	----	----	----

98	23	45	14
----	----	----	----

6	67	33	42
---	----	----	----

98	23
----	----

45	14
----	----

98	23
----	----

23
----

Merge

98	23	45	14	6	67	33	42
----	----	----	----	---	----	----	----

98	23	45	14
----	----	----	----

6	67	33	42
---	----	----	----

98	23
----	----

45	14
----	----

98	23
----	----

23	98
----	----

Merge

98	23	45	14	6	67	33	42
----	----	----	----	---	----	----	----

98	23	45	14
----	----	----	----

6	67	33	42
---	----	----	----

98	23
----	----

45	14
----	----

98
----

23
----

45
----

14
----

23	98
----	----



98	23	45	14	6	67	33	42
----	----	----	----	---	----	----	----

98	23	45	14
----	----	----	----

6	67	33	42
---	----	----	----

98	23
----	----

45	14
----	----

98
----

23
----

45
----

14
----

23	98
----	----

[Merge]

98	23	45	14	6	67	33	42
----	----	----	----	---	----	----	----

98	23	45	14
----	----	----	----

6	67	33	42
---	----	----	----

98	23
----	----

45	14
----	----

98
----

23
----

45
----

14
----

23	98
----	----

14
----

[Merge]

98	23	45	14	6	67	33	42
----	----	----	----	---	----	----	----

98	23	45	14
----	----	----	----

6	67	33	42
---	----	----	----

98	23
----	----

45	14
----	----

98
----

23
----

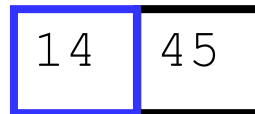
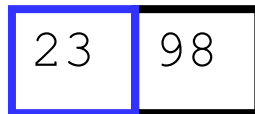
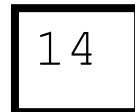
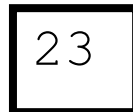
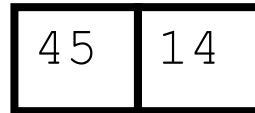
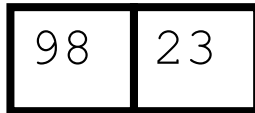
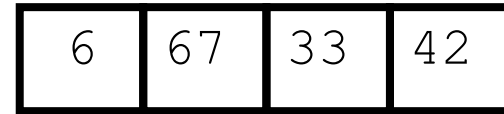
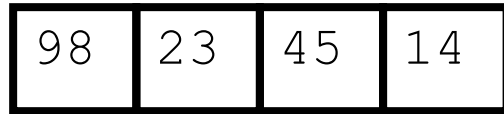
45
----

14
----

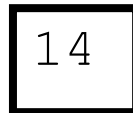
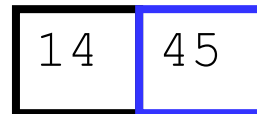
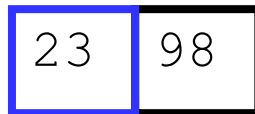
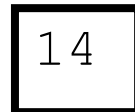
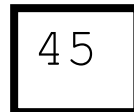
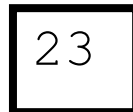
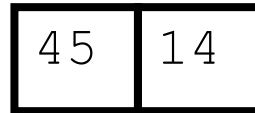
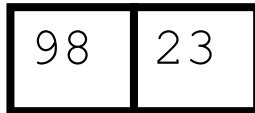
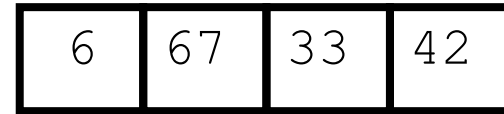
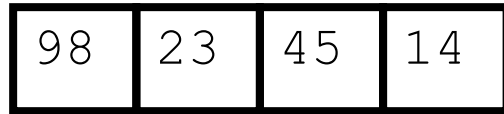
23	98
----	----

14	45
----	----

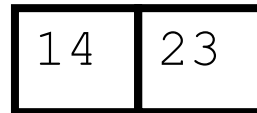
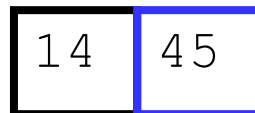
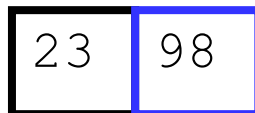
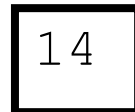
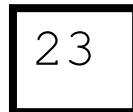
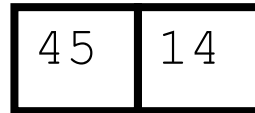
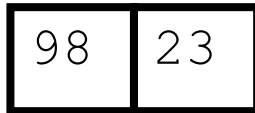
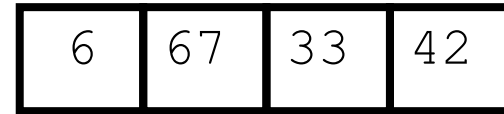
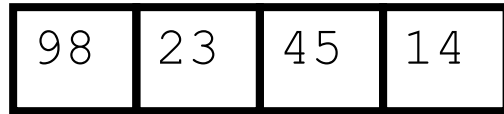
[Merge]



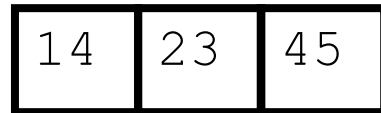
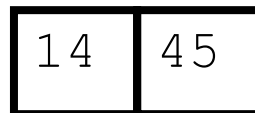
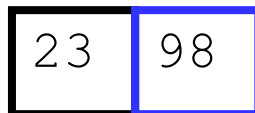
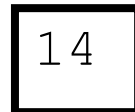
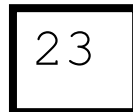
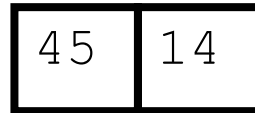
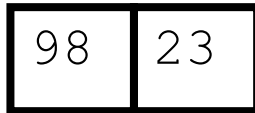
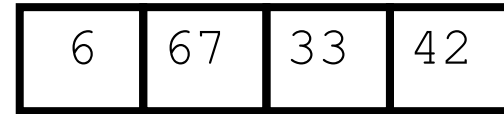
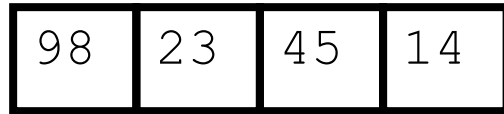
Merge



Merge



Merge



Merge

98	23	45	14	6	67	33	42
----	----	----	----	---	----	----	----

98	23	45	14
----	----	----	----

6	67	33	42
---	----	----	----

98	23
----	----

45	14
----	----

98
----

23
----

45
----

14
----

23	98
----	----

14	45
----	----

14	23	45	98
----	----	----	----

Merge



98	23	45	14	6	67	33	42
----	----	----	----	---	----	----	----

98	23	45	14
----	----	----	----

6	67	33	42
---	----	----	----

98	23
----	----

45	14
----	----

6	67
---	----

33	42
----	----

98
----

23
----

45
----

14
----

23	98
----	----

14	45
----	----

14	23	45	98
----	----	----	----

98	23	45	14	6	67	33	42
----	----	----	----	---	----	----	----

98	23	45	14
----	----	----	----

6	67	33	42
---	----	----	----

98	23
----	----

45	14
----	----

6	67
---	----

33	42
----	----

98
----

23
----

45
----

14
----

6
---

67
----

23	98
----	----

14	45
----	----

14	23	45	98
----	----	----	----

98	23	45	14	6	67	33	42
----	----	----	----	---	----	----	----

98	23	45	14
----	----	----	----

6	67	33	42
---	----	----	----

98	23
----	----

45	14
----	----

6	67
---	----

33	42
----	----

98
----

23
----

45
----

14
----

6
---

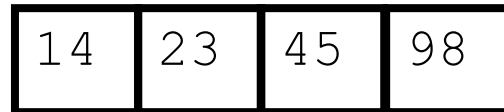
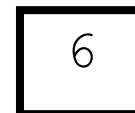
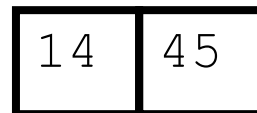
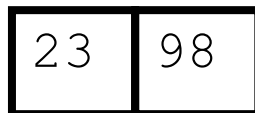
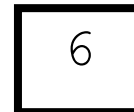
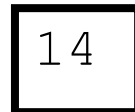
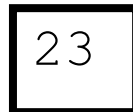
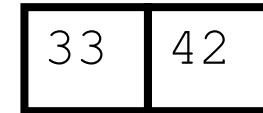
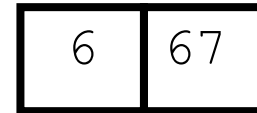
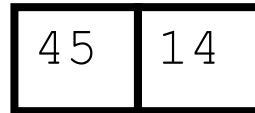
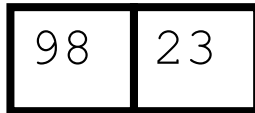
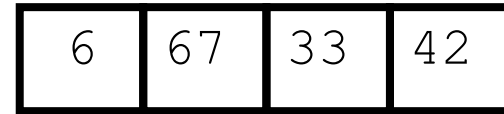
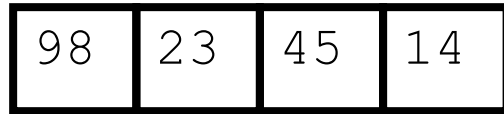
67
----

23	98
----	----

14	45
----	----

14	23	45	98
----	----	----	----

Merge



Merge

98	23	45	14	6	67	33	42
----	----	----	----	---	----	----	----

98	23	45	14
----	----	----	----

6	67	33	42
---	----	----	----

98	23
----	----

45	14
----	----

6	67
---	----

33	42
----	----

98
----

23
----

45
----

14
----

6
---

67
----

23	98
----	----

14	45
----	----

6	67
---	----

14	23	45	98
----	----	----	----

Merge

98	23	45	14	6	67	33	42
----	----	----	----	---	----	----	----

98	23	45	14
----	----	----	----

6	67	33	42
---	----	----	----

98	23
----	----

45	14
----	----

6	67
---	----

33	42
----	----

98
----

23
----

45
----

14
----

6
---

67
----

33
----

42
----

23	98
----	----

14	45
----	----

6	67
---	----

14	23	45	98
----	----	----	----

98	23	45	14	6	67	33	42
----	----	----	----	---	----	----	----

98	23	45	14
----	----	----	----

6	67	33	42
---	----	----	----

98	23
----	----

45	14
----	----

6	67
---	----

33	42
----	----

98
----

23
----

45
----

14
----

6
---

67
----

33
----

42
----

23	98
----	----

14	45
----	----

6	67
---	----

14	23	45	98
----	----	----	----

Merge

98	23	45	14	6	67	33	42
----	----	----	----	---	----	----	----

98	23	45	14
----	----	----	----

6	67	33	42
---	----	----	----

98	23
----	----

45	14
----	----

6	67
---	----

33	42
----	----

98
----

23
----

45
----

14
----

6
---

67
----

33
----

42
----

23	98
----	----

14	45
----	----

6	67
---	----

33
----

14	23	45	98
----	----	----	----

Merge



98	23	45	14	6	67	33	42
----	----	----	----	---	----	----	----

98	23	45	14
----	----	----	----

6	67	33	42
---	----	----	----

98	23
----	----

45	14
----	----

6	67
---	----

33	42
----	----

98
----

23
----

45
----

14
----

6
---

67
----

33
----

42
----

23	98
----	----

14	45
----	----

6	67
---	----

33	42
----	----

14	23	45	98
----	----	----	----

Merge

98	23	45	14	6	67	33	42
----	----	----	----	---	----	----	----

98	23	45	14
----	----	----	----

6	67	33	42
---	----	----	----

98	23
----	----

45	14
----	----

6	67
---	----

33	42
----	----

98
----

23
----

45
----

14
----

6
---

67
----

33
----

42
----

23	98
----	----

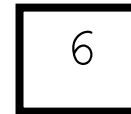
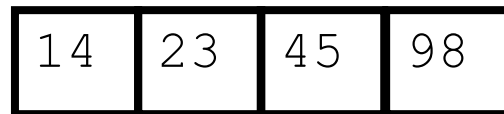
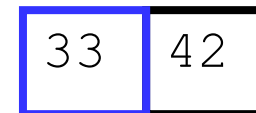
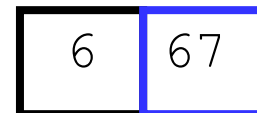
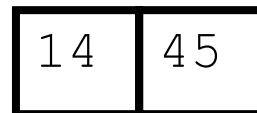
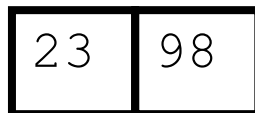
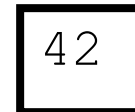
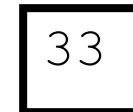
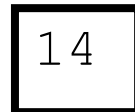
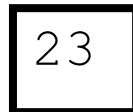
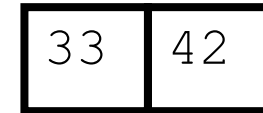
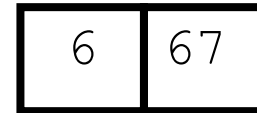
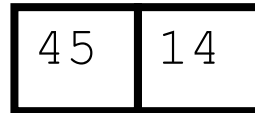
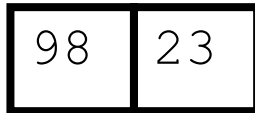
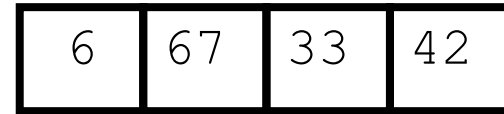
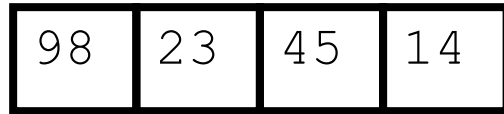
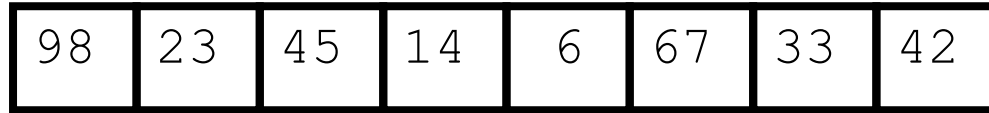
14	45
----	----

6	67
---	----

33	42
----	----

14	23	45	98
----	----	----	----

Merge



Merge

98	23	45	14	6	67	33	42
----	----	----	----	---	----	----	----

98	23	45	14
----	----	----	----

6	67	33	42
---	----	----	----

98	23
----	----

45	14
----	----

6	67
---	----

33	42
----	----

98
----

23
----

45
----

14
----

6
---

67
----

33
----

42
----

23	98
----	----

14	45
----	----

6	67
---	----

33	42
----	----

14	23	45	98
----	----	----	----

6	33
---	----

Merge

98	23	45	14	6	67	33	42
----	----	----	----	---	----	----	----

98	23	45	14
----	----	----	----

6	67	33	42
---	----	----	----

98	23
----	----

45	14
----	----

6	67
---	----

33	42
----	----

98
----

23
----

45
----

14
----

6
---

67
----

33
----

42
----

23	98
----	----

14	45
----	----

6	67
---	----

33	42
----	----

14	23	45	98
----	----	----	----

6	33	42
---	----	----

Merge

98	23	45	14	6	67	33	42
----	----	----	----	---	----	----	----

98	23	45	14
----	----	----	----

6	67	33	42
---	----	----	----

98	23
----	----

45	14
----	----

6	67
---	----

33	42
----	----

98
----

23
----

45
----

14
----

6
---

67
----

33
----

42
----

23	98
----	----

14	45
----	----

6	67
---	----

33	42
----	----

14	23	45	98
----	----	----	----

6	33	42	67
---	----	----	----

Merge

98	23	45	14	6	67	33	42
----	----	----	----	---	----	----	----

98	23	45	14
----	----	----	----

6	67	33	42
---	----	----	----

98	23
----	----

45	14
----	----

6	67
---	----

33	42
----	----

98
----

23
----

45
----

14
----

6
---

67
----

33
----

42
----

23	98
----	----

14	45
----	----

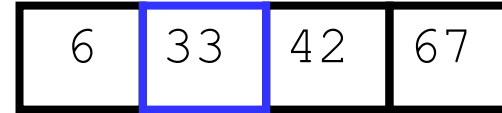
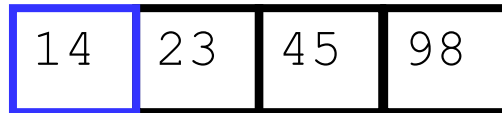
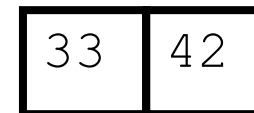
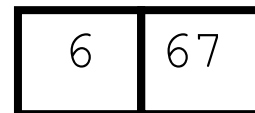
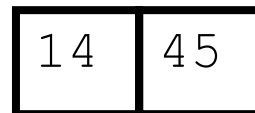
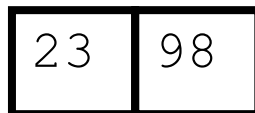
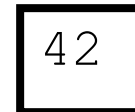
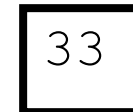
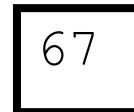
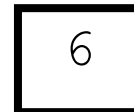
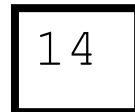
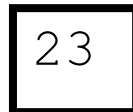
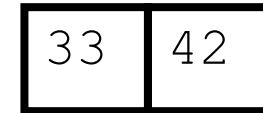
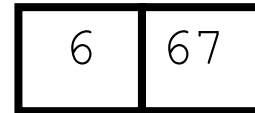
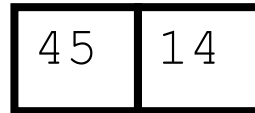
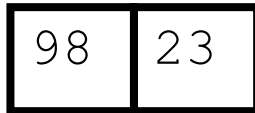
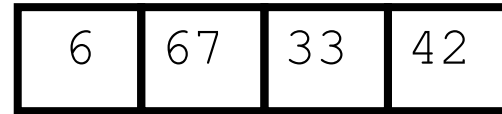
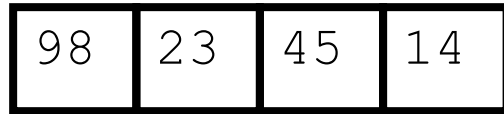
6	67
---	----

33	42
----	----

14	23	45	98
----	----	----	----

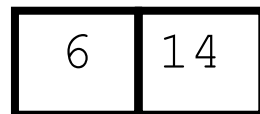
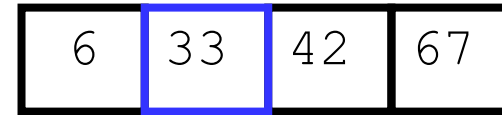
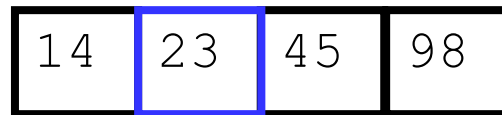
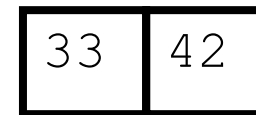
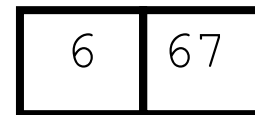
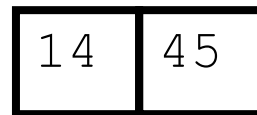
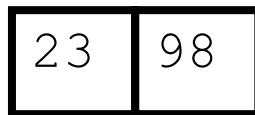
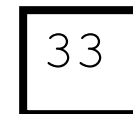
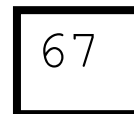
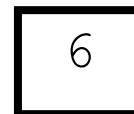
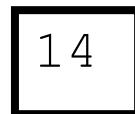
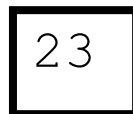
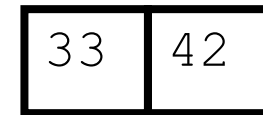
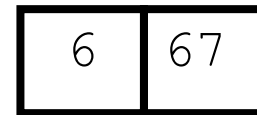
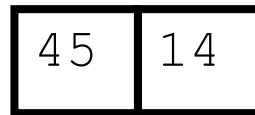
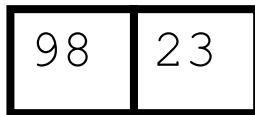
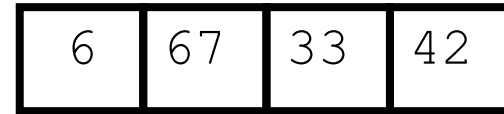
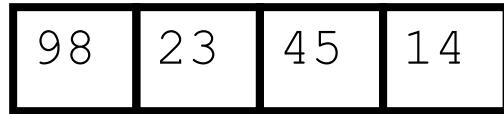
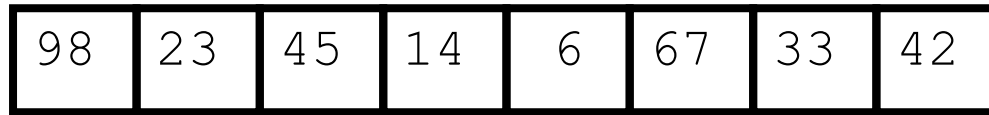
6	33	42	67
---	----	----	----

Merge



Merge





Merge

98	23	45	14	6	67	33	42
----	----	----	----	---	----	----	----

98	23	45	14
----	----	----	----

6	67	33	42
---	----	----	----

98	23
----	----

45	14
----	----

6	67
---	----

33	42
----	----

98
----

23
----

45
----

14
----

6
---

67
----

33
----

42
----

23	98
----	----

14	45
----	----

6	67
---	----

33	42
----	----

14	23	45	98
----	----	----	----

6	33	42	67
---	----	----	----

6	14	23
---	----	----

Merge

98	23	45	14	6	67	33	42
----	----	----	----	---	----	----	----

98	23	45	14
----	----	----	----

6	67	33	42
---	----	----	----

98	23
----	----

45	14
----	----

6	67
---	----

33	42
----	----

98
----

23
----

45
----

14
----

6
---

67
----

33
----

42
----

23	98
----	----

14	45
----	----

6	67
---	----

33	42
----	----

14	23	45	98
----	----	----	----

6	33	42	67
---	----	----	----

6	14	23	33
---	----	----	----

Merge

98	23	45	14	6	67	33	42
----	----	----	----	---	----	----	----

98	23	45	14
----	----	----	----

6	67	33	42
---	----	----	----

98	23
----	----

45	14
----	----

6	67
---	----

33	42
----	----

98
----

23
----

45
----

14
----

6
---

67
----

33
----

42
----

23	98
----	----

14	45
----	----

6	67
---	----

33	42
----	----

14	23	45	98
----	----	----	----

6	33	42	67
---	----	----	----

6	14	23	33	42
---	----	----	----	----

Merge

98	23	45	14	6	67	33	42
----	----	----	----	---	----	----	----

98	23	45	14
----	----	----	----

6	67	33	42
---	----	----	----

98	23
----	----

45	14
----	----

6	67
---	----

33	42
----	----

98
----

23
----

45
----

14
----

6
---

67
----

33
----

42
----

23	98
----	----

14	45
----	----

6	67
---	----

33	42
----	----

14	23	45	98
----	----	----	----

6	33	42	67
---	----	----	----

6	14	23	33	42	45
---	----	----	----	----	----

Merge

98	23	45	14	6	67	33	42
----	----	----	----	---	----	----	----

98	23	45	14
----	----	----	----

6	67	33	42
---	----	----	----

98	23
----	----

45	14
----	----

6	67
---	----

33	42
----	----

98
----

23
----

45
----

14
----

6
---

67
----

33
----

42
----

23	98
----	----

14	45
----	----

6	67
---	----

33	42
----	----

14	23	45	98
----	----	----	----

6	33	42	67
---	----	----	----

6	14	23	33	42	45	67
---	----	----	----	----	----	----

Merge

98	23	45	14	6	67	33	42
----	----	----	----	---	----	----	----

98	23	45	14
----	----	----	----

6	67	33	42
---	----	----	----

98	23
----	----

45	14
----	----

6	67
---	----

33	42
----	----

98
----

23
----

45
----

14
----

6
---

67
----

33
----

42
----

23	98
----	----

14	45
----	----

6	67
---	----

33	42
----	----

14	23	45	98
----	----	----	----

6	33	42	67
---	----	----	----

6	14	23	33	42	45	67	98
---	----	----	----	----	----	----	----

Merge

98	23	45	14	6	67	33	42
----	----	----	----	---	----	----	----

98	23	45	14
----	----	----	----

6	67	33	42
---	----	----	----

98	23
----	----

45	14
----	----

6	67
---	----

33	42
----	----

98
----

23
----

45
----

14
----

6
---

67
----

33
----

42
----

23	98
----	----

14	45
----	----

6	67
---	----

33	42
----	----

14	23	45	98
----	----	----	----

6	33	42	67
---	----	----	----

6	14	23	33	42	45	67	98
---	----	----	----	----	----	----	----



98	23	45	14	6	67	33	42
----	----	----	----	---	----	----	----



6	14	23	33	42	45	67	98
---	----	----	----	----	----	----	----

# Algorithm

Mergesort(Passed an array)

if array size  $> 1$

Divide array in half

Call Mergesort on first half.

Call Mergesort on second half.

Merge two halves.

Merge(Passed two arrays)

Compare leading element in each array

Select lower and place in new array.

(If one input array is empty then place  
remainder of other array in output array)

# Recursive Mergesort

```
Mergesort(A[], T[] : integer array, left, right : integer) : {  
  if left < right then  
    mid := (left + right)/2;  
    Mergesort(A,T,left,mid);  
    Mergesort(A,T,mid+1,right);  
    Merge(A,T,left,right);  
}
```

```
MainMergesort(A[1..n]: integer array, n : integer) : {  
  T[1..n]: integer array;  
  Mergesort[A,T,1,n];  
}
```

```

/* C Program to implement Merge Sort using Recursion */
#include <stdio.h>

void mergeSort(int [], int, int, int);
void partition(int [],int, int);
int main()
{
    int list[50];
    int i, size;
    printf("How Many elements u want to Sort :: ");
    scanf("%d", &size);
    printf("\nEnter [ %d ] elements below to be Sorted :: \n",size);
    for(i = 0; i < size; i++)
    {
        printf("\nEnter [ %d ] Element :: ",i+1);
        scanf("%d", &list[i]);
    }
    partition(list, 0, size - 1);
    printf("\n\nAfter implementing Merge sort, Sorted List is :: \n\n");
    for(i = 0;i < size; i++)
    {
        printf("%d  ",list[i]);
    }
    printf("\n");
    return 0;
}

```

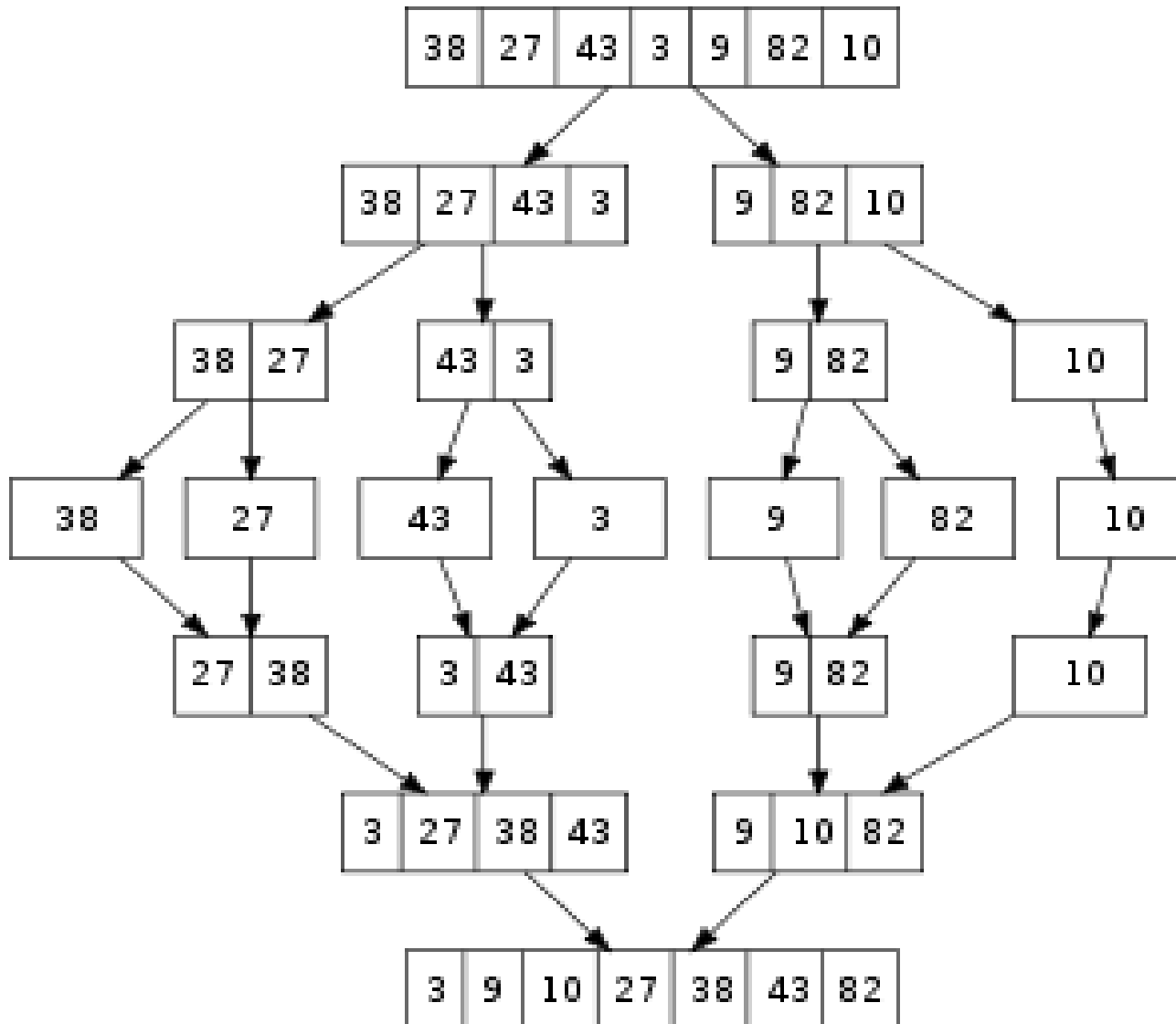
```
void partition(int list[],int low,int high)
{
    int mid;
    if(low < high)
    {
        mid = (low + high) / 2;
        partition(list, low, mid);
        partition(list, mid + 1, high);
        mergeSort(list, low, mid, high);
    }
}
```

```

void mergeSort(int list[],int low,int mid,int high)
{
    int i, mi, k, lo, temp[50];
    lo = low;    i = low;    mi = mid + 1;
    while ((lo <= mid) && (mi <= high))
    {
        if (list[lo] <= list[mi])
        {
            temp[i] = list[lo];
            lo++;
        }
        else
        {
            temp[i] = list[mi];
            mi++;
        }
        i++;
    }
    if (lo > mid)
    {
        for (k = mi; k <= high; k++)
        {
            temp[i] = list[k];
            i++;
        }
    }
    else
    {
        for (k = lo; k <= mid; k++)
        {
            temp[i] = list[k];
            i++;
        }
    }
    for (k = low; k <= high; k++)
    {
        list[k] = temp[k];
    }
}

```

# Merge Sort



# Merge sort.

- Analysis

Worst case:  $O(n * \log_2 n)$

Average case:  $O(n * \log_2 n)$

- Advantage

- It is an extremely efficient algorithm with respect to time

- Drawback

- It requires a second array as large as the original array



# A Comparison of Sorting Algorithms

	<u>Worst case</u>	<u>Average case</u>
Selection sort	$n^2$	$n^2$
Bubble sort	$n^2$	$n^2$
Insertion sort	$n^2$	$n^2$
Mergesort	$n * \log n$	$n * \log n$
Quicksort	$n^2$	$n * \log n$
Radix sort	$n$	$n$
Treesort	$n^2$	$n * \log n$
Heapsort	$n * \log n$	$n * \log n$

Approximate growth rates of time required for eight sorting algorithms

# Search Algorithms

- Sequential Search (Linear Search).
- Binary Search.

# Sequential Search

$O(n)$

- **A sequential search**
- Begins at the beginning of the list/array and continues until the ele is found or the entire list/array has been searched

```
#include<stdio.h>

void main()
{
int a[100],n,i,ele,loc=-1;
printf("\nEnter the number of element:");
scanf("%d",&n);
printf("Enter the number:\n");
for(i=0;i<=n-1;i++)
{
    scanf("%d",&a[i]);
}
//traversal
for(i=0;i<=n-1;i++)
{
    printf("%d",a[i]);
}
```

```
//linear search
```

```
printf("Enter the no. to be search\n"); 12,34,56,7,9 ele=17
```

```
scanf("%d",&ele);
```

```
for(i=0;i<=n-1;i++)
```

```
{
```

```
if(ele==a[i])
```

```
{
```

```
loc=i; //loc=3
```

```
break;
```

```
}
```

```
}
```

```
if(loc>=0)
```

```
printf("\n%ele found in position%d",ele,loc+1);
```

```
else
```

```
printf("\nele does not exists");
```

```
getch();
```

```
}
```

# Search Algorithms

Suppose that there are  $n$  elements in the array. The following expression gives the average number of comparisons:

$$\frac{1+2+\dots+n}{n}$$

It is known that

$$1+2+\dots+n = \frac{n(n+1)}{2}$$

# Cont...

- Therefore, the following expression gives the average number of comparisons made by the sequential search in the successful case:

$$\frac{1+2+\dots+n}{n} = \frac{1}{n} \frac{n(n+1)}{2} = \frac{n+1}{2}$$

# Linear search

- Advantages:

- Easy algorithm to understand.
- Array can be in any order.

- Disadvantages:

- Inefficient (Slow), for array of  $N$  elements.
- Examines  $N/2$  elements on average for value in array,  $N$  elements for value not in array.



# Binary Search: $O(\log_2 n)$

- A **binary search** looks for an ele in a list using a divide-and-conquer strategy.

# Binary Search

- Binary search algorithm assumes that the elements in the array being searched are sorted
- The algorithm begins at the middle of the array in a binary search
- If the element for which we are searching is less than the element in the middle, we know that the element won't be in the second half of the array
- Once again we examine the “middle” element

# Cont...

- The process continues with each comparison cutting in half the portion of the array .
- Binary Search: middle element

$$\text{mid} = \frac{\text{left} + \text{right}}{2}$$

- $\text{mid} = (\text{left} + \text{right}) / 2$

# Binary Search: Example

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
list	4	8	19	25	34	39	45	48	66	75	89	95

Values of first, last, and middle and the Number of Comparisons for Search Item 89

Iteration	first	last	mid	list[mid]
1	0	11	5	39
2	6	11	8	66
3	9	11	10	89

# Binary Search

[0]	ant
[1]	cat
[2]	chicken
[3]	cow
[4]	deer
[5]	dog
[6]	fish
[7]	goat
[8]	horse
[9]	camel
[10]	snake

## Searching for cat

BinarySearch(0, 10)	middle: 5	cat < dog
BinarySearch(0, 4)	middle: 2	cat < chicken
BinarySearch(0, 1)	middle: 0	cat > ant
BinarySearch(1, 1)	middle: 1	cat = cat <b>Return: true</b>

## Searching for zebra

BinarySearch(0, 10)	middle: 5	zebra > dog
BinarySearch(6, 10)	middle: 8	zebra > horse
BinarySearch(9, 10)	middle: 9	zebra > camel
BinarySearch(10, 10)	middle: 10	zebra > snake
BinarySearch(11, 10)		last > first <b>Return: false</b>

## Searching for fish

BinarySearch(0, 10)	middle: 5	fish > dog
BinarySearch(6, 10)	middle: 8	fish < horse
BinarySearch(6, 7)	middle: 6	fish = fish <b>Return: true</b>

# Binary Search

- Advantages:

- Much more efficient than Linear Search.
- For array of  $N$  elements, perform at most  $\log_2 N$  comparisons.

- Disadvantages:

- Requires that array elements be sorted.

```

#include<stdio.h>
#include<conio.h>
void main()
{
    int a[100],i,loc,mid,beg,end,n,flag=0,ele;
    clrscr();
    printf("How many elements");
    scanf("%d",&n);
    printf("Enter the element of the array\n");
    for(i=0;i<=n-1;i++)
    {
        scanf("%d",&a[i]);
    }
    printf("Enter the element to be
    searched\n");
    scanf("%d",&ele);
    loc=0;
    beg=0;
    end=n-1;

```

```

while((beg<=end)&&(ele!=a[mid]))
{
    mid=((beg+end)/2);
    if(ele==a[mid])
    {
        printf("search is successfull\n");
        loc=mid;
        printf("position of the ele%d\n",loc+1);
        flag=flag+1;
    }
    if(ele<a[mid])
        end=mid-1;
    else
        beg=mid+1;
}
if(flag==0)
{
    printf("search is not successfull\n");
}
getch();
}

```

# Binary Search

```
bool BinSearch(double list[ ], int n, double ele,
    int&index){
    int left=0;
    int right=n-1;
    int mid;
    while(left<=right){
        mid=(left+right)/2;
        if(ele> list [mid]){ left=mid+1; }
        else if(ele< list [mid]){right=mid-1;}
        else{
            ele= list [mid];
```



# Cont...

```
index=mid;  
    return true; }  
    }// while  
    return false;  
}
```

# SUMMARY

- Algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.
- Algorithms can be used for sorting and
- Other practical examples like Electronic commerce.
- Algorithms involves encryption/decryption techniques.
- Divide and Conquer is a method of algorithm design.

# Summary

- **SORTING**
- Order-of-magnitude analysis and Big O notation measure an algorithm's time requirement as a function of the problem size by using a growth-rate function
- To compare the inherent efficiency of algorithms
  - Examine their growth-rate functions when the problems are large
  - Consider only significant differences in growth-rate functions

# Summary

- SORTING
- Worst-case and average-case analyses
  - Worst-case analysis considers the maximum amount of work an algorithm requires on a problem of a given size
  - Average-case analysis considers the expected amount of work an algorithm requires on a problem of a given size

# Summary

- SORTING
- Order-of-magnitude analysis can be used to choose an implementation for an abstract data type
- Selection sort, bubble sort, and insertion sort are all  $O(n^2)$  algorithms
- Quick sort and merge sort are two very efficient sorting algorithms

# Summary

- **SEARCHING**
- Searching is the fundamental operation in computer science.
- Searching refers to the operation of finding the location of a given ele in a collection of eles.
- The particular algorithm one chooses depends on the properties of the data.
- And the operation may perform on the data.

# Summary

- **SEARCHING**
- Accordingly, we will want to know the complexity of each algorithm.
- Sometimes we will also discuss the space requirement of our algorithm.

# References

- Data Structures, Schaum Series by Seymour Lipschutz, G.A.V. Pai.
- C & Data Structure by P.G. Deshpande and O.G. Kakde.
- An Introduction to data Structures with applications by Tremblay and Sorenson.