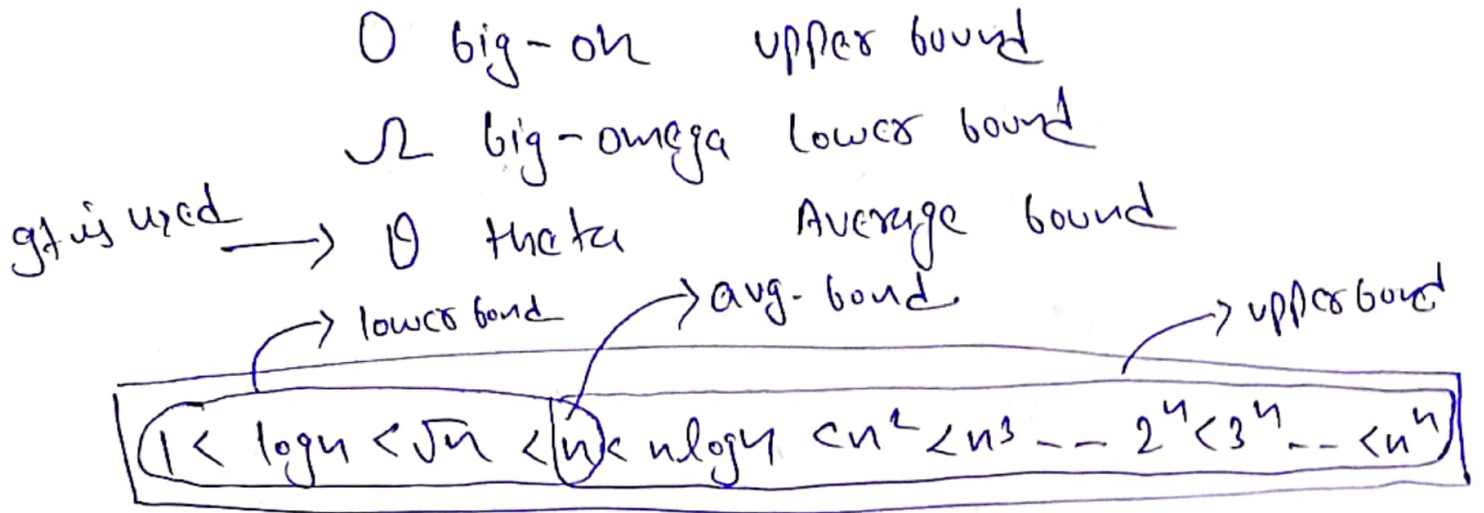


# Asymptotic Notation



## $\Rightarrow$ Big-oh

The function  $f(n) = O(g(n))$  iff  $\exists$  +ve constants  $c$  and  $n_0$

such that  $f(n) \leq c * g(n) \forall n \geq n_0$

eg.  $f(n) = 2n + 3$

$$2n + 3 \leq \underbrace{10}_{c} \underbrace{n}_{g(n)} \quad n \geq 1$$

This is  
 $\uparrow$  useful.

$\therefore f(n) = O(n)$

$\rightarrow$   $c$  can be any no. but greater than  $2n+3$ .

or

$$2n + 3 \not\leq ?$$

$$2n + 3 \not\leq 2n + 3n$$

$$2n + 3 \leq \underbrace{5n}_{g(n)} \quad n \geq 1$$

$$f(n) = O(n)$$

or

eg -  $f(n) = 2n + 3$

$$2n + 3 \leq 2n^2 + 3n^2$$

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

$$f(n) \leftarrow 2n \leq 5n^2 \quad n \geq 1 \quad \leftarrow c g(n)$$

## Omega

The function  $f(n) = \Omega(g(n))$  iff  $\exists$  +ve constants  $C$  and  $n_0$ .

such that  $f(n) \geq C * g(n) \forall n \geq n_0$ .

eg.  $f(n) = 2n + 3$

$$2n + 3 \geq 1 * \log n \quad \forall n \geq 1$$

$\downarrow \quad \uparrow \quad \uparrow$   
 $f(n) \quad C \quad g(n)$

$\therefore f(n) = \Omega(n)$   
 $f(n) = \Omega(\log n)$   
 $\rightarrow$  This is useful.

Note : If we write other variable it may be true but not useful.

---

## Theta Notation

The function  $f(n) = \Theta(g(n))$  iff  $\exists$  +ve constants  $C_1, C_2$  and  $n_0$ .

such that  $C_1 * g(n) \leq f(n) \leq C_2 * g(n)$

eg.  $f(n) = 2n + 3$

$$1 * n \leq 2n + 3 \leq 5 * n$$

$C_1 g(n) \quad f(n) \quad C_2 g(n)$

$\therefore f(n) = \Theta(n)$   
 $\downarrow$

Note : It is mostly recommended.

This only says that is possible.

Que 2.1  $f(n) = 2n^2 + 3n + 4$ .

(i)  $2n^2 + 3n + 4 \leq 2n^2 + 3n^2 + 4n^2$   
 $2n^2 + 3n + 4 \leq 9n^2$       $n \geq 1$   
 $\uparrow \quad \uparrow$   
 $c \quad g(n)$

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

(ii)  $f(n) \geq 1 \times n^2$   
 $2n^2 + 3n + 4 \geq 1 \times n^2$   
 $\Omega(n^2)$

(iii)  $1 \times n^2 \leq 2n^2 + 3n + 4 \leq 9n^2$       $\Theta(n^2)$

Que 2.2      $f(n) = n^2 \log n + n$ .

$1 \times n^2 \log n \leq n^2 \log n + n \leq 10n^2 \log n$ .

$O(n^2 \log n)$       $\Omega(n^2 \log n)$       $\Theta(n^2 \log n)$

Que 2.3

$f(n) = n! = n \times (n-1) \times (n-2) \dots 3 \times 2 \times 1$

$1 \times 1 \times \dots \times 1 \leq 1 \times 2 \times 3 \dots \times n \leq n \times n \times n \dots \times n$

$1 \leq n! \leq n^n$

$\Omega(1)$       $O(n^n)$

↳ Here we can not find any  
 ↳ for factorial we function we  
 go for lower bound and upper bound.

# DATA STRUCTURES

## □ Introduction:

- Basic Concepts and Notations
- Complexity analysis: time space tradeoff
- Algorithmic notations, Big O notation
- Introduction to omega, theta and little o notation

# Basic Concepts and Notations

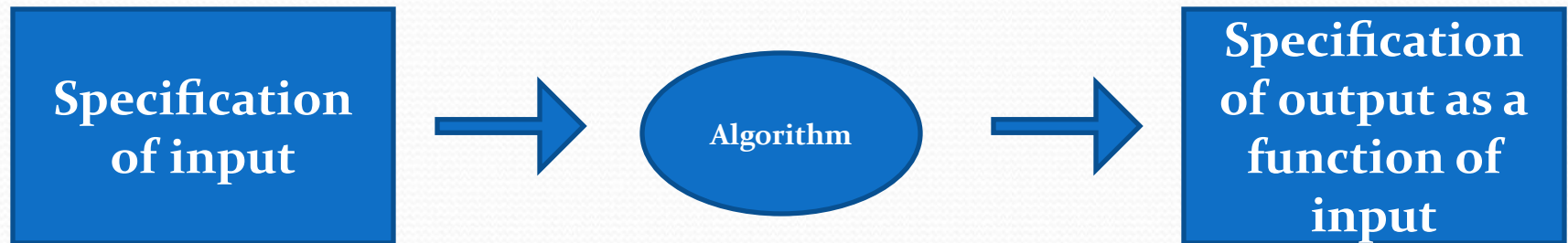
- Algorithm: Outline, the essence of a computational procedure, step-by-step instructions
- Program: an implementation of an algorithm in some programming language
- Data Structure: **Organization** of data needed to solve the problem

# Algorithmic Problem



- Infinite number of input instances satisfying the specification. For example: A sorted, non-decreasing sequence of natural numbers of non-zero, finite length:
  - 1, 20, 908, 909, 100000, 10000000000.
  - 3.

# Algorithmic Solution



- Algorithm describes actions on the input instance
- Infinitely many correct algorithms for the same algorithmic problem

# What is a Good Algorithm?

- Efficient:
  - Running time
  - Space used
- Efficiency as a function of input size:
  - The number of bits in an input number
  - Number of data elements(numbers, points)



# Complexity Analysis and Time Space Trade-off

# Complexity

- A measure of the performance of an algorithm
- An algorithm's performance depends on
  - *internal* factors
  - *external* factors

# External Factors

- Speed of the computer on which it is run
- Quality of the compiler
- Size of the input to the algorithm

# Internal Factor

- The algorithm's efficiency, in terms of:
  - Time required to run
  - Space (memory storage) required to run

• Note:

- Complexity measures the *internal* factors (usually more interested in time than space)

# Two ways of finding complexity

- Experimental study
- Theoretical Analysis

# Experimental study

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Get an accurate measure of the actual running time  
Use a method like `System.currentTimeMillis()`
- Plot the results

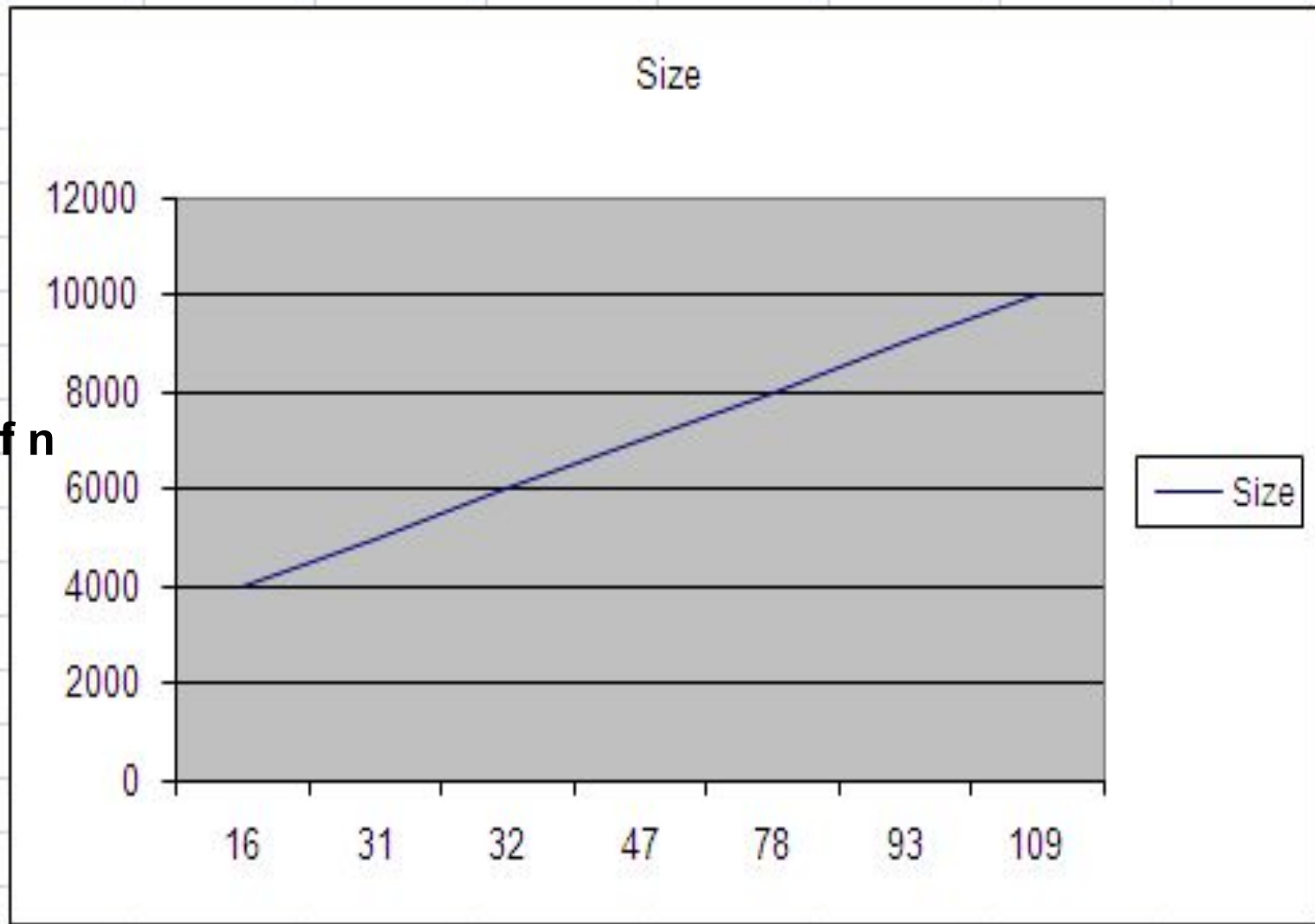
# Example

- a. 

```
Sum=0;
for(i=0;i<N;i++)
  for(j=0;j<i;j++)
    Sum++;
```

# Example graph

Size of n



• Time in millisec



# Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used
- Experimental data though important is not sufficient

# Theoretical Analysis

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size,  $n$ .
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

# Complexity analysis

- Why we should analyze algorithms?
  - Predict the resources that the algorithm requires
    - Computational time (CPU consumption)
    - Memory space (RAM consumption)
    - Communication bandwidth consumption
  - The **running time** of an algorithm is:
    - The total number of primitive operations executed (machine independent steps)
    - Also known as **algorithm complexity**

## Factors

- To determine resource consumption
  - CPU time
  - Memory space
- Compare different methods for solving the same problem before actually implementing them and running the programs.
- To find an efficient algorithm

# Space Complexity

- The space needed by an algorithm is the sum of a fixed part and a variable part
- The fixed part includes space for
  - Instructions
  - Simple variables
  - Fixed size component variables
  - Space for constants
  - Etc..

# Cont...

- The variable part includes space for
  - Component variables whose size is dependant on the particular problem instance being solved
  - Recursion stack space
  - Etc..

# Time Complexity

- The time complexity of a problem is
  - the number of steps that it takes to solve an instance of the problem as a function of the size of the input (usually measured in bits), using the most efficient algorithm.
- The exact number of steps will depend on exactly what machine or language is being used.
- To avoid that problem, the Asymptotic notation is generally used.

# Time Complexity

- Worst-case
  - An upper bound on the running time for any input of given size
- Average-case
  - Assume all inputs of a given size are equally likely
- Best-case
  - The lower bound on the running time



# Time Complexity – Example

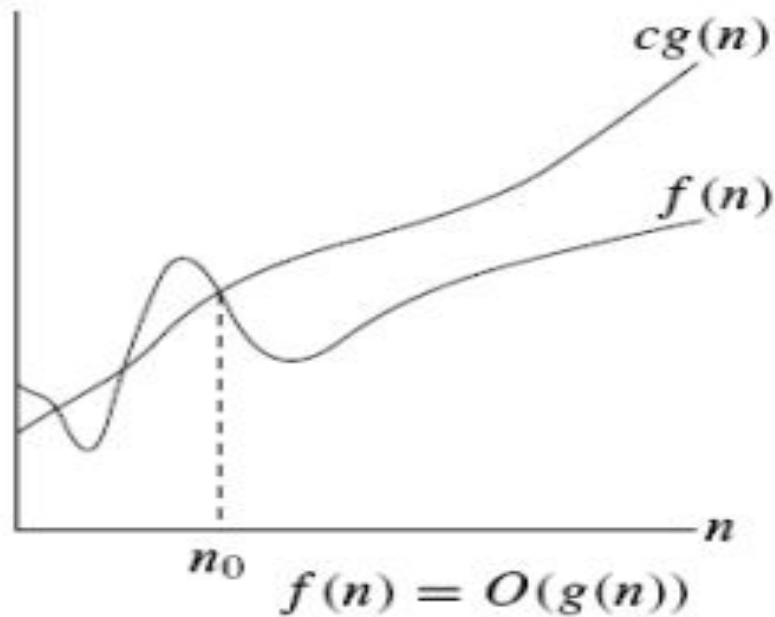
- Sequential search in a list of size  $n$ 
  - Worst-case:
    - $n$  comparisons
  - Best-case:
    - 1 comparison
  - Average-case:
    - $n/2$  comparisons

# Asymptotic notations

- **Algorithm complexity** is rough estimation of the number of steps performed by given computation depending on the size of the input data
  - Measured through **asymptotic notation**
    - $O(g)$  where  $g$  is a function of the input data size
  - Examples:
    - Linear complexity  $O(n)$  – all elements are processed once (or constant number of times)
    - Quadratic complexity  $O(n^2)$  – each of the elements is processed  $n$  times

# O-notation

## Asymptotic upper bound



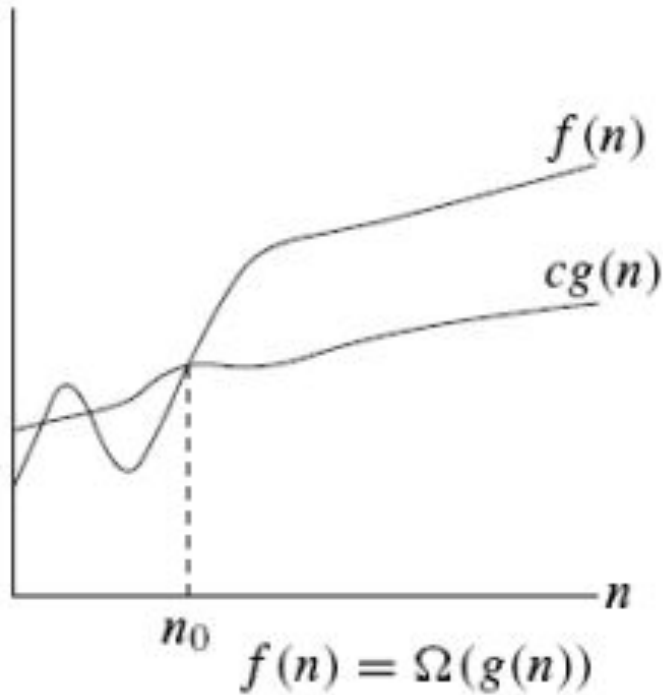
$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that}$   
 $0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}.$

# Example

- The running time is  $O(n^2)$  means there is a function  $f(n)$  that is  $O(n^2)$  such that for any value of  $n$ , no matter what particular input of size  $n$  is chosen, the running time of that input is bounded from above by the value  $f(n)$ .
  - $3 * n^2 + n/2 + 12 \in O(n^2)$
  - $4 * n * \log_2(3 * n + 1) + 2 * n - 1 \in O(n * \log n)$

# $\Omega$ notation

## Asymptotic lower bound



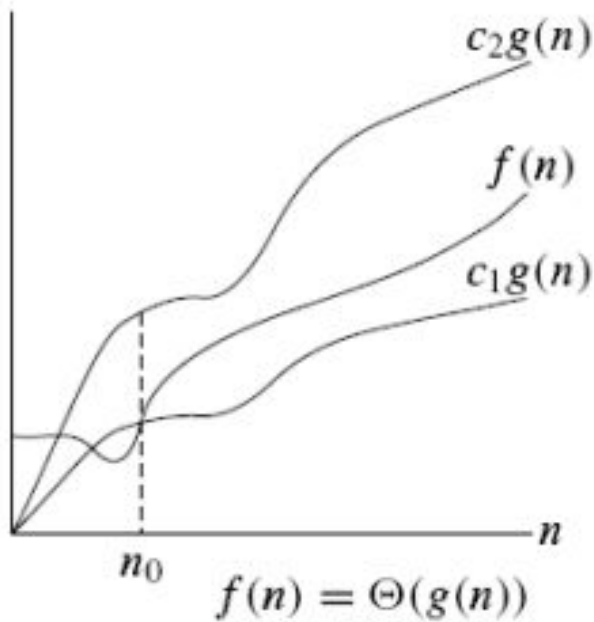
$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\} .$

# Example

- When we say that the running time (no modifier) of an algorithm is  $\Omega(g(n))$ .
- we mean that no matter what particular input of size  $n$  is chosen for each value of  $n$ , the running time on that input is at least a constant times  $g(n)$ , for sufficiently large  $n$ .
- $n^3 + 20n \in \Omega(n^2)$

# $\Theta$ notation

$g(n)$  is an asymptotically tight bound of  $f(n)$



$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that}$   
 $0 \leq c_1g(n) \leq f(n) \leq c_2g(n) \text{ for all } n \geq n_0\} .^1$

# Example

$$\frac{1}{2}n^2 - 3n = \Theta(n^2)$$

$$c_1n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2n^2$$

for all  $n \geq n_0$ . Dividing by  $n^2$  yields

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2.$$

$$n \geq 1, c_2 \geq 1/2$$

$$n \geq 7, c_1 \leq 1/14$$

choose  $c_1 = 1/14, c_2 = 1/2, n_0 = 7$ .




# Big O notation

- $f(n)=O(g(n))$  iff there exist a positive constant  $c$  and non-negative integer  $n_0$  such that
$$f(n) \leq cg(n) \text{ for all } n \geq n_0.$$
- $g(n)$  is said to be an upper bound of  $f(n)$ .

# Basic rules

1. Nested loops are multiplied together.
2. Sequential loops are added.
3. Only the largest term is kept, all others are dropped.
4. Constants are dropped.
5. Conditional checks are constant (i.e. 1).



# Example of complexity

# Linear loop

1)

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

```
{
```

```
    cout << i << endl;
```

```
}
```

```
//time taken = ?
```

2)

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

```
{
```

```
    cout << i << endl;
```

```
}
```

**//time taken = ?**



- Ans:  $O(n)$

# Quadratic Loops

```
1) for(int i = 0; i < 100; i++)  
  {  
    for(int j = 0; j < 100; j++)  
      {  
        //do swap stuff, constant time  
      }  
  }    //Time Taken =?
```

```
2) for(int i = 0; i < n; i++)
    {
        for(int j = 0; j < n; j++)
            {
                //do swap stuff, constant time
            }
    }
    //Time Taken =?
```



- Ans  $O(n^2)$

# Complex condition

```
1) for(int i = 0; i < 2*100; i++)  
    {  
        cout << i << endl;  
    }
```

**//Time Taken =?**

```
2) for(int i = 0; i < 2*n; i++)  
    {  
        cout << i << endl;  
    }
```

**//Time Taken =?**

- At first you might say that the upper bound is  $O(2n)$ ; however, we drop constants so it becomes  $O(n)$

# More loops in one program

```
1) for(int i = 0; i < 10 ; i++)  
    {  
        cout << i << endl;  
    }
```

```
for(int i = 0; i < 100; i++)  
    {  
        for(int j = 0; j < 100; j++)  
            {  
                //do constant time stuff  
            }  
    } //Time Taken =?
```

2)

```
for(int i = 0; i < n; i++)  
{  
    cout << i << endl;  
}
```

```
for(int i = 0; i < n; i++)  
{  
    for(int j = 0; j < n; j++)  
    {  
        //do constant time stuff  
    }  
} //Time Taken =?
```

- Ans : In this case we add each loop's Big O, in this case  $n+n^2$ .  $O(n^2+n)$  is not an acceptable answer since we must drop the lowest term. The upper bound is  $O(n^2)$ . Why? Because it has the largest growth rate

# Quadratic loop

```
1) for(int i = 0; i < 100; i++)  
  {  
    for(int j = 0; j < 2; j++)  
    {  
      //do stuff  
    }  
  } //Time Taken =?
```



```
2) for(int i = 0; i < n; i++)  
  {  
    for(int j = 0; j < 2; j++)  
    {  
      //do stuff  
    }  
  }
```

**//Time Taken =?**

- Ans: Outer loop is 'n', inner loop is 2, this we have  $2n$ , dropped constant gives up  $O(n)$

# Complex iteration

```
1) for(int i = 1; i < n; i = i* 2)
    {
        cout << i << endl;
    }
```

**//Time Taken =?**

```
2) for(int i = 1; i < 100; i =i* 2)
{
    cout << i << endl;
}
```


**//Time Taken =?**

- There are  $n$  iterations, however, instead of simply incrementing, 'i' is increased by  $2 \times \text{itself}$  each run. Thus the loop is  $\log(n)$ .

# Quadratic loop

```
1) for(int i = 0; i < n; i++)  
  {  
    for(int j = 1; j < n; j *= 2)  
      {  
        //do constant time stuff  
      }  
  }
```

**//Time Taken =?**



- Ans:  $n \cdot \log(n)$

```
While (n>=1)
```

```
{
```

```
    n=n/2;
```

```
}
```

```
2) While (n>=1)
```

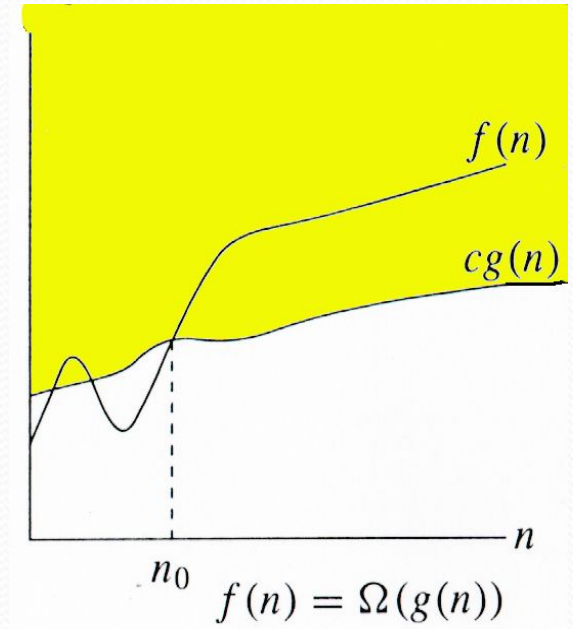
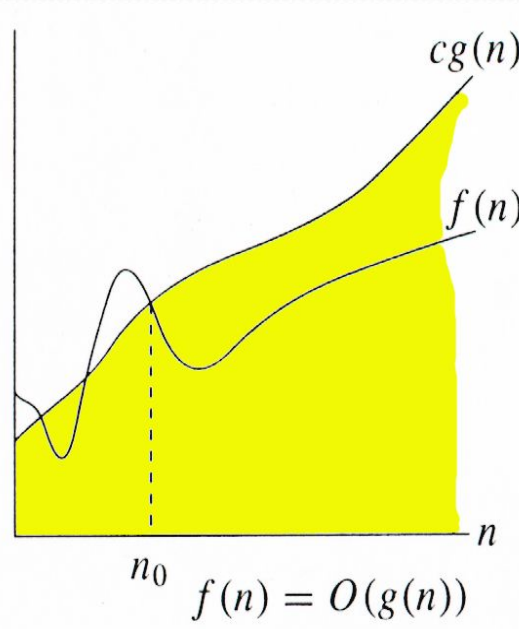
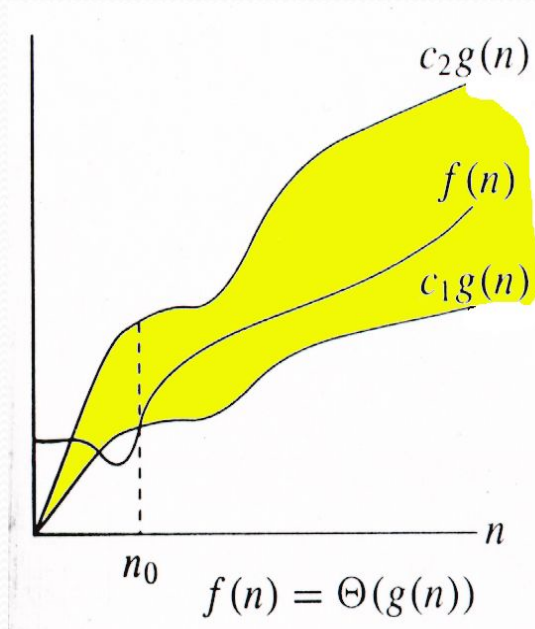
```
{
```

```
    n=n/2;
```

```
}
```



# Relations Between $\Theta$ , $O$ , $\Omega$



# time space tradeoff

- A time space tradeoff is a situation where the memory use can be reduced at the cost of slower program execution (and, conversely, the computation time can be reduced at the cost of increased memory use).
- As the relative costs of CPU cycles, RAM space, and hard drive space change—hard drive space has for some time been getting cheaper at a much faster rate than other components of computers[citation needed]—the appropriate choices for time space tradeoff have changed radically.
- Often, by exploiting a time space tradeoff, a program can be made to run much faster.

# Time Space Trade-off

- In computer science, a **space-time** or **time-memory trade off** is a situation where the memory use can be reduced at the cost of slower program execution (or, vice versa, the computation time can be reduced at the cost of increased memory use). As the relative costs of CPU cycles, RAM space, and hard drive space change — hard drive space has for some time been getting cheaper at a much faster rate than other components of computers-the appropriate choices for space-time tradeoffs have changed radically. Often, by exploiting a space-time tradeoff, a program can be made to run much faster.

# Types of Time Space Trade-off

- **Lookup tables v. recalculation**

The most common situation is an algorithm involving a lookup table: an implementation can include the entire table, which reduces computing time, but increases the amount of memory needed, or it can compute table entries as needed, increasing computing time, but reducing memory requirements.

- **Compressed v. uncompressed data**

A space-time trade off can be applied to the problem of data storage. If data is stored uncompressed, it takes more space but less time than if the data were stored compressed (since compressing the data reduces the amount of space it takes, but it takes time to run the decompression algorithm). Depending on the particular instance of the problem, either way is practical.



Thank You

# ARRAYS

- Linear arrays: Memory representation
- Traversal
- Insertion
- Deletion
- Linear Search
- Binary Search
- Merging
- 2D Array : Memory representation

# CONTENTS

2.1 Introductions

2.2 Linear Array

2.2.1 Linear Array Representations in Memory

2.2.2 Traversing Algorithm

2.2.3 Insert Algorithms

2.2.4 Delete Algorithms

2.2.5 Sequential and Binary Search Algorithm

2.2.6 Merging Algorithm

2.3 Multidimensional Array

2.3.1 2-D Array

2.3.2 Representations in Memory

# 2.1 Introduction

- Data Structure can be classified as:
  - ✓ linear
  - ✓ non-linear
- Linear (elements arranged in sequential in memory location) i.e. array & linear link-list
- Non-linear such as a tree and graph.
- Operations:
  - ✓ Traversing, Searching, Inserting, Deleting, Sorting, Merging
- Array is used to store a fix size for data and a link-list the data can be varies in size.



# 2.1 Introduction

- Advantages of an Array:
  - Very simple
  - Economy – if full use of memory
  - Random accessed at the same time
- Disadvantage of an Array:
  - wasting memory if not fully used
  - Stores same data types' elements

## 2.2 Linear Array

- Homogeneous data:
  - a) Elements are represented through indexes.
  - b) Elements are saved in sequential in memory locations.
- Number of elements,  $N \rightarrow$  length or size of an array.

If:

UB : upper bound ( the largest index)

LB : lower bound (the smallest index)

Then:  $N = UB - LB + 1$

Length =  $N = UB$  when  $LB = 1$

## 2.2 Linear Array

- All elements in A are written symbolically as, 1 .. n is the subscript.  
A<sub>1</sub>, A<sub>2</sub>, A<sub>3</sub>, .... , A<sub>n</sub>
- In FORTRAN and BASIC □ A(1), A(2), ..., A(N)
- In Pascal, C/C++ and Java □ A[0], A[1], ..., A[N-1]
- subscript starts from 0  
LB = 0, UB = N-1

## 2.2.1 Representation of Array in a Memory

- The process to determine the address in a memory:
  - a) First address – base address.
  - b) Relative address to base address through index function.

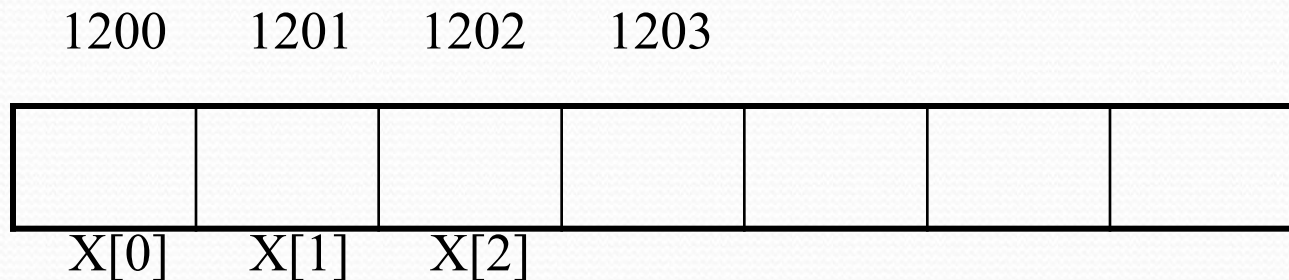
Example: char X[100];

Let *char* uses 1 location storage.

If the base address is 1200 then the next element is in 1201.

Index Function is written as:

**Loc (X[i]) = Loc(X[0]) + i** , *i* is subscript and LB = 0



## 2.2.1 Representation of Array in a Memory

- In general, index function:

$$\text{Loc}(X[i]) = \text{Loc}(X[\text{LB}]) + w * (i - \text{LB});$$

where  $w$  is length of memory location required.

For real number: 4 byte, integer: 2 byte and character: 1 byte.

- Example:

If  $\text{LB} = 5$ ,  $\text{Loc}(X[\text{LB}]) = 1200$ , and  $w = 4$ , find  $\text{Loc}(X[8])$  ?

$$\begin{aligned}\text{Loc}(X[8]) &= \text{Loc}(X[5]) + 4 * (8 - 5) \\ &= 1212\end{aligned}$$

## 2.2.2 Traversing Algorithm (While loop)

- Traversing operation means visit every element once, whether processed or not.

Traversal(LA,N): This is an algorithm to traverse N elements of an array LA ..

1. [Assign counter] Set  $K:=0$ .
  2. Repeat step 2.1 and 2.2 while  $K \leq UB$ 
    - 2.1 [visit element]  
do PROCESS on LA[K].
    - 2.2 [add counter]  
Set  $K:=K+1$
- [end of while loop]
4. exit.

# 2.2.2 Traversing Algorithm (for loop)

Traversal(LA,N): This is an algorithm to traverse  
N elements of an array LA .

1.Repeat step 2 for K =0 to N-1

2. [visit element]

do PROCESS on LA[K].

[end of for loop]

3. exit.





## 2.2.3 Insertion Algorithm(while loop)

- Insertion is to insert some element in the array at user defined location

INSERT(LA, N, K, ITEM):LA is a linear array with N element, K is integer positive where  $K < N$  and  $LB = 0$ , Insert an element, ITEM in index K.

1. [Assign counter], Set  $J := N - 1$  ; [LB = 0]
2. Repeat step 2.1 and 2.2 while  $J \geq K$ 
  - 2.1 [shift to the right all elements from J]  
Set  $LA[J+1] := LA[J]$
  - 2.2 [decrement counter] Set  $J := J - 1$
- [End of while loop]
4. [Insert element] Set  $LA[K] := ITEM$
5. [Reset N] Set  $N := N + 1$
6. Exit

## 2.2.3 Insertion Algorithm(for loop)

- INSERT(LA, N, K, ITEM):LA is a linear array with N element, K is integer positive where  $K < N$  and  $LB = 0$ ,Insert an element, ITEM in index K.
- 1. Read: K and ITEM [index to perform insertion and value to be inserted] (optional step as mentioned in the declaration)
- 2. Repeat step for  $j=N-1$  to K
- 3. [shift to the right all elements from J]
- Set  $LA[J+1] := LA[J]$   
[end of for loop]
- 4. [Insert element] Set  $LA[K] := ITEM$
- 5. [Reset N] Set  $N := N + 1$
- 6. Exit

## 2.2.4 Deletion Algorithm

- Delete item.

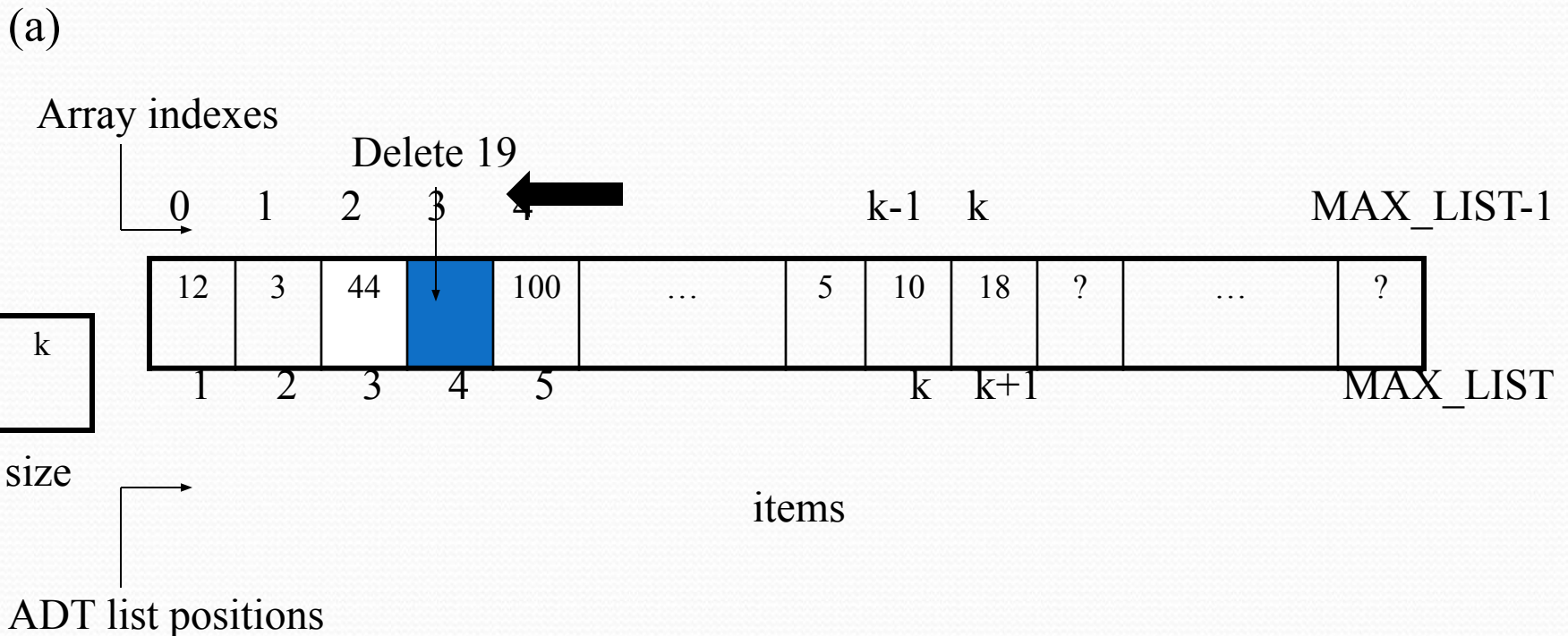


Figure 2: Deletion causes a gap

## 2.2.4 Deletion Algorithm

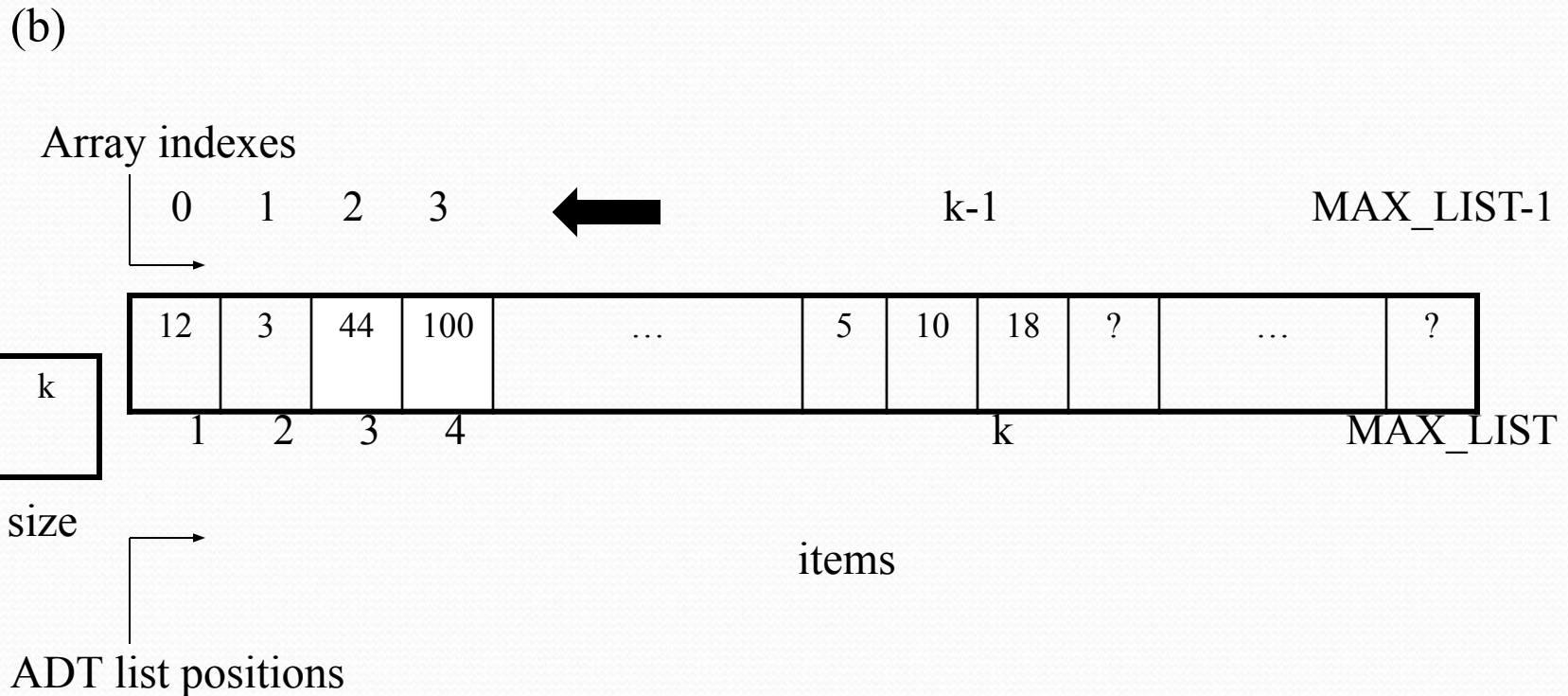


Figure 3: Fill gap by shifting

## 2.2.4 Deletion Algorithm(for loop)

- DELETE(LA, N, K, ITEM):In an array LA this
- Algo will delete the element ITEM at location
- K. Where LA has N elements.
- 1. Read :K. [index element you want to delete]
- 2. Set ITEM:=LA[K]. [storing value before deleting]
- 3. Repeat step 3.1 for I = K to N-2 [If LB = 0]
- 3.1 [Shift element, forward]
- Set LA[I] := LA[I+1]
- [end of for loop]
- 4. [Reset N in LA]
- Set N := N – 1
- 5. Exit

## 2.2.4 Deletion Algorithm(while loop)

- DELETE(LA, N, K, ITEM):In an array LA this Algorithm will delete the element ITEM at location K. Where LA has N elements.
- 1. Read :K. [index element you want to delete]
- 2. Set ITEM:=LA[K] and Set I:=K. [storing value before deleting]
- 3. Repeat step 3.1 while I< N-1 [If LB = 0]
- 3.1 [Shift element, forward]
- Set LA[I] := LA[I+1]
- Set I:=I+1
- [end of while loop]
- 4. [Reset N in LA]
- Set N := N – 1
- 5. Exit

# 2.2.5 Sequential Search(while loop)

Compare successive elements of a given list with a search ITEM until

1. either a match is encountered
2. or the list is exhausted without a match.



**SequentialSearch(LA, N, ITEM, LOC):** Algorithm to search element ITEM in an array LA having N elements and storing its location in LOC

1. Set  $I := 0$
2. Repeat step 2.1 while  $i < N \ \&\& \ LA[I] \neq \text{ITEM}$ 
  - 2.1 Set  $I := I + 1$
3. If  $LA[I] = \text{ITEM}$ , then:  
write: found at  $LOC = I$   
else:  
write: not found.
4. Exit

# Other algorithm to do linear search/sequential search (for loop)

**SequentialSearch(LA, N, ITEM, LOC,TEMP):** Algorithm to search element ITEM in an array LA having N elements and storing its location in LOC. Where TEMP is a flag variable.

1. Set TEMP :=0
2. Repeat step 2.1 for I=0 to N-1
  - 2.1 if LB[I]=ITEM,then:  
    set TEMP:=1 and LOC:=I  
    and EXIT for loop
3. If Temp=1,then:  
    write:found at LOC  
    else:  
    write:not found.
4. Exit



## 2.2.5 Binary Search Algorithm

- Binary search algorithm is efficient if the array is sorted.
- A binary search is used whenever the list starts to become large.
- Consider to use binary searches whenever the list contains more than 16 elements.
- The binary search starts by testing the data in the element at the middle of the array to determine if the target is in the first or second half of the list.
- If it is in the first half, we do not need to check the second half. If it is in the second half, we do not need to test the first half. In other words we eliminate half the list from further consideration. We repeat this process until we find the target or determine that it is not in the list.

## 2.2.5 Binary Search Algorithm

- To find the middle of the list, we need three variables, one to identify the beginning of the list, one to identify the middle of the list, and one to identify the end of the list.
- We analyze two cases here: the target is in the list (target found) and the target is not in the list (target not found).

## 2.2.5 Binary Search Algorithm

- **Target found case:** Assume we want to find 22 in a sorted list as follows:

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]
4	7	8	10	14	21	22	36	62	77	81	91

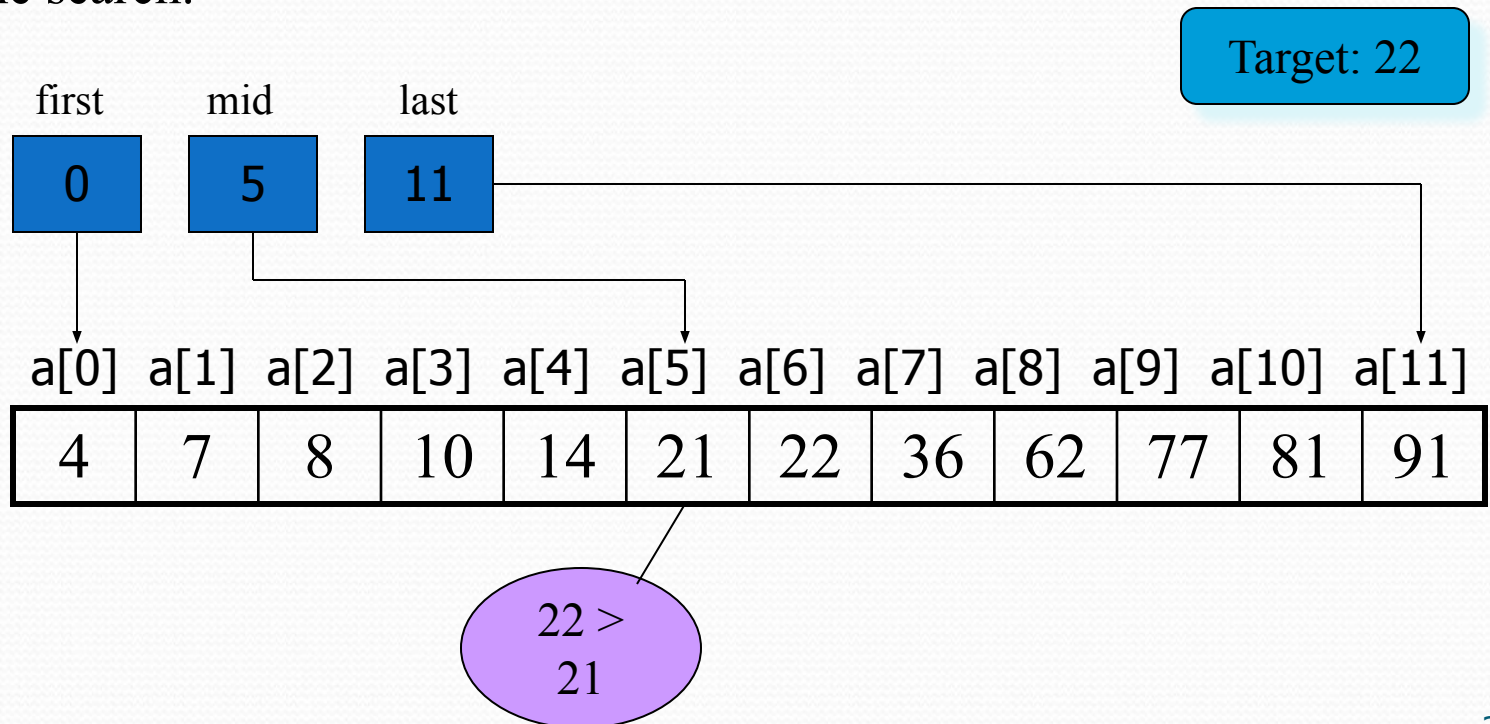
- The three indexes are first, mid and last. Given first as 0 and last as 11, mid is calculated as follows:

$$\text{mid} = (\text{first} + \text{last}) / 2$$

$$\text{mid} = (0 + 11) / 2 = 11 / 2 = 5$$

## 2.2.5 Binary Search Algorithm

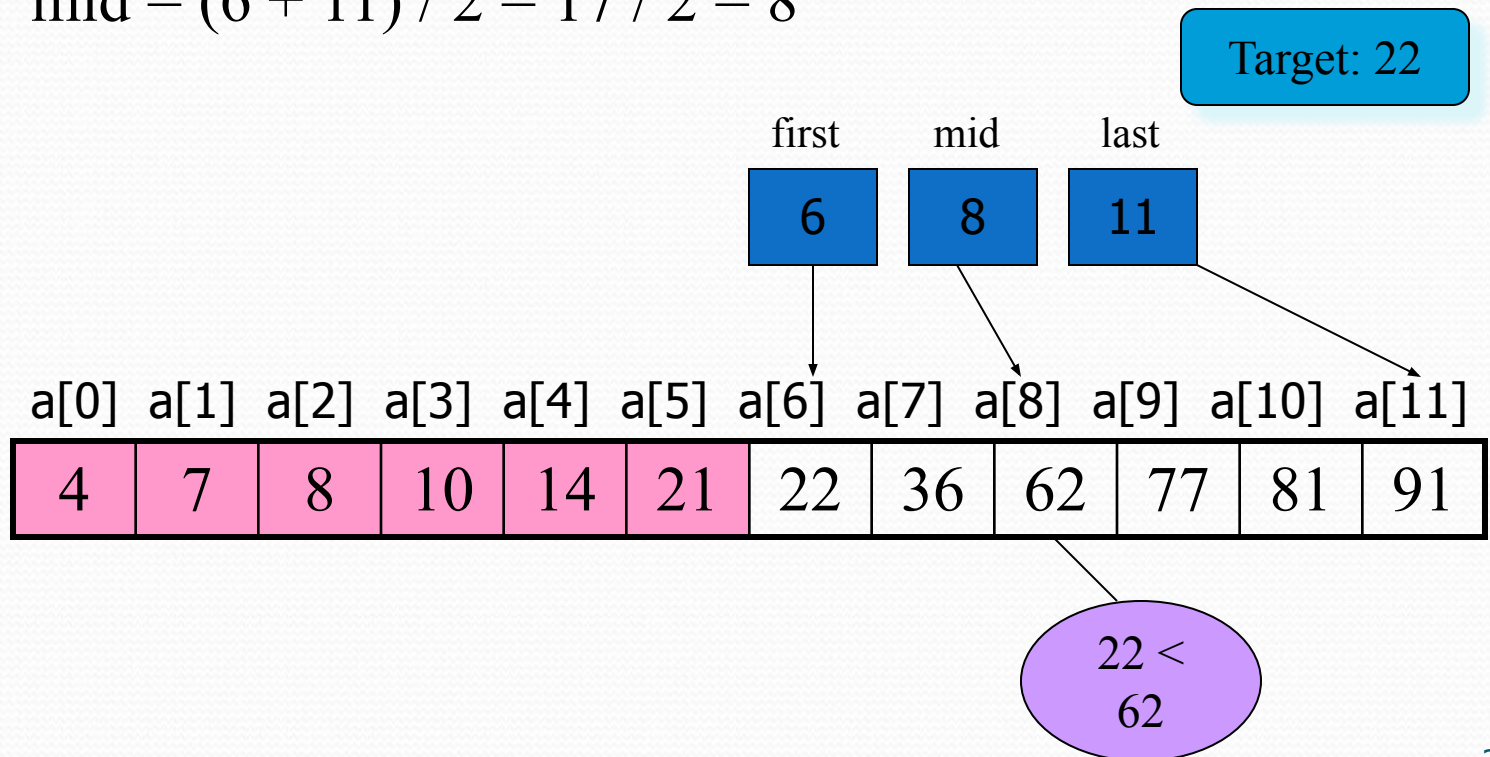
- At index location 5, the target is greater than the list value ( $22 > 21$ ). Therefore, eliminate the array locations 0 through 5 (mid is automatically eliminated). To narrow our search, we assign  $\text{mid} + 1$  to first and repeat the search.



## 2.2.5 Binary Search Algorithm

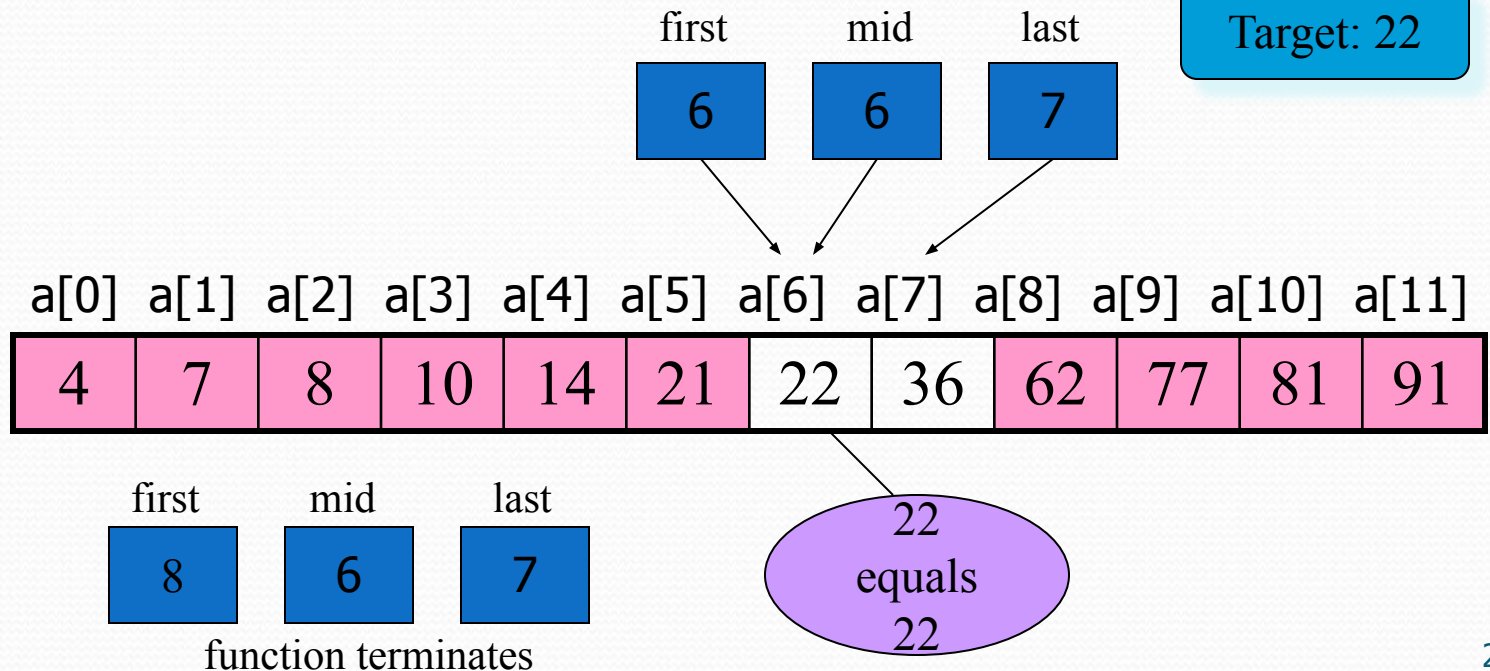
- The next loop calculates mid with the new value for first and determines that the midpoint is now 8 as follows:

$$\text{mid} = (6 + 11) / 2 = 17 / 2 = 8$$



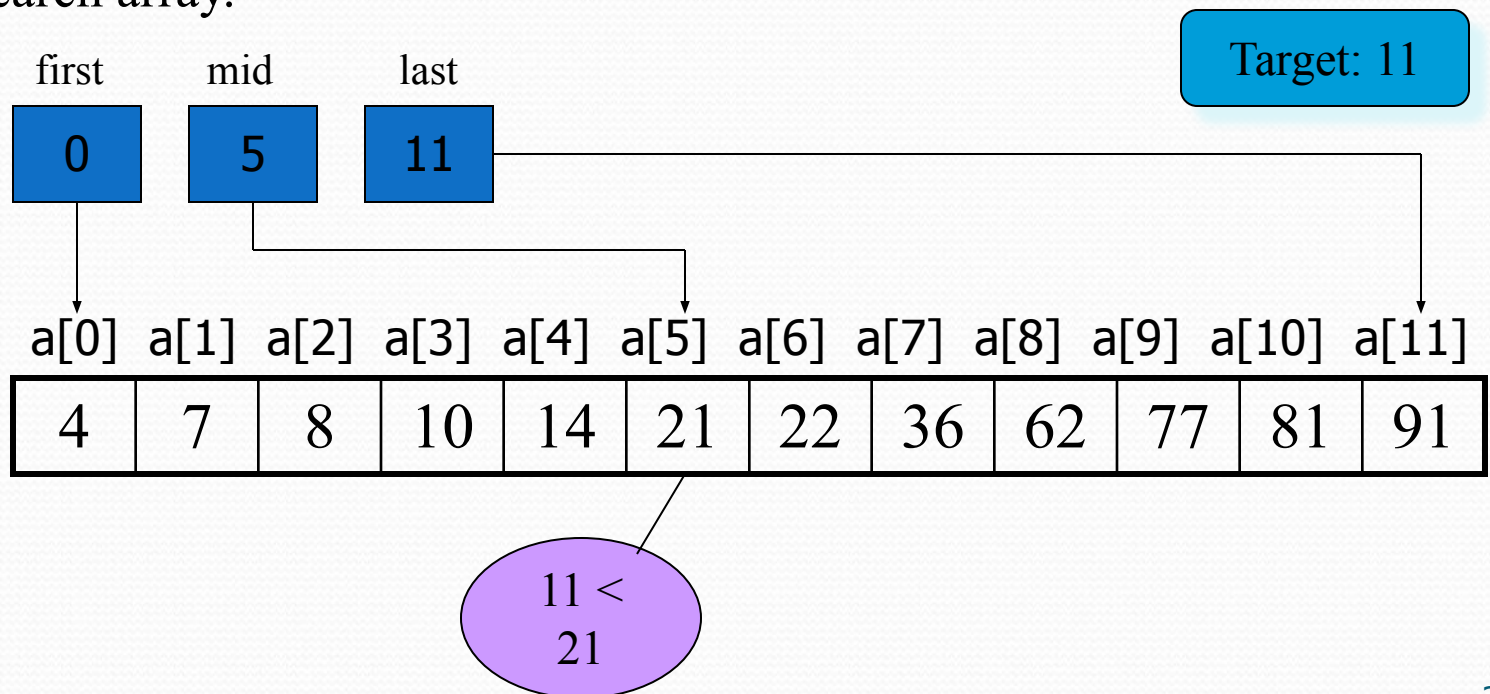
## 2.2.5 Binary Search Algorithm

- When we test the target to the value at mid a second time, we discover that the target is less than the list value ( $22 < 62$ ). This time we adjust the end of the list by setting last to mid - 1 and recalculate mid. This step effectively eliminates elements 8 through 11 from consideration. We have now arrived at index location 6, whose value matches our target. This stops the search.



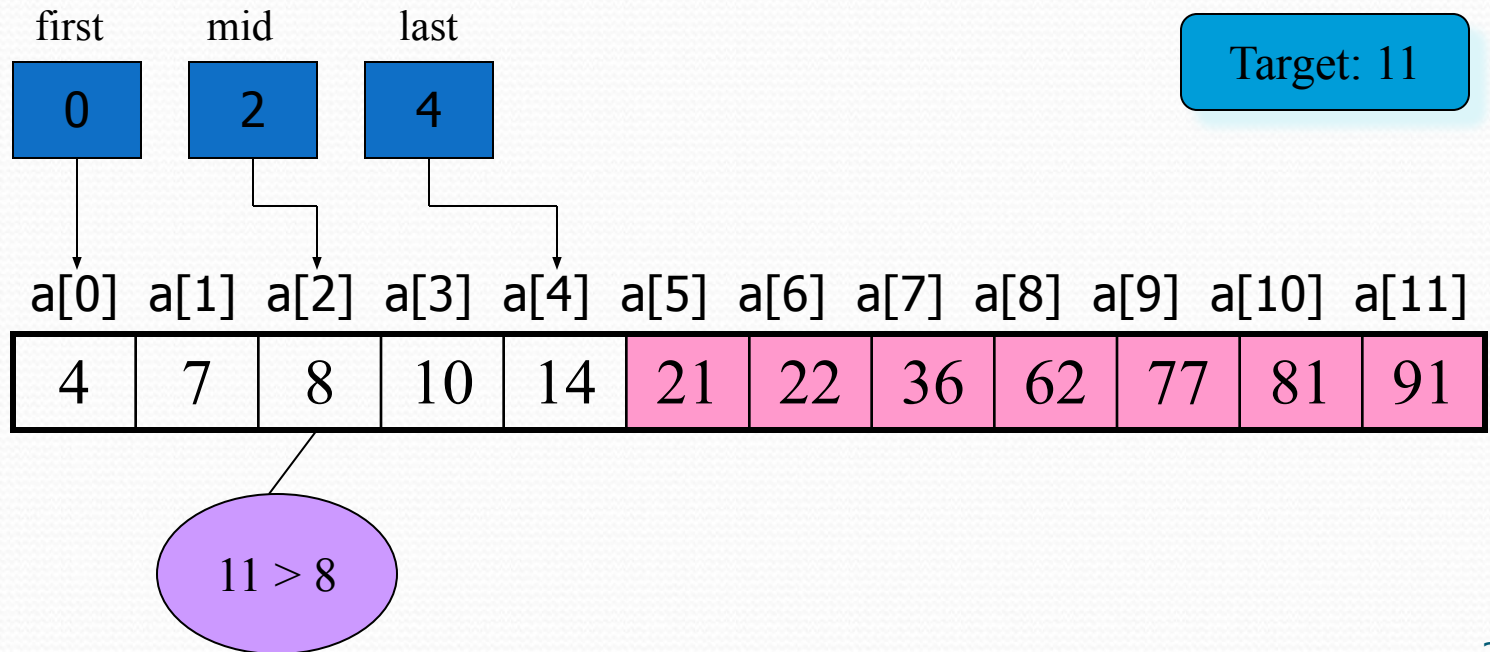
## 2.2.5 Binary Search Algorithm

- Target not found case: This is done by testing for first and last crossing: that is, we are done when first becomes greater than last. Two conditions terminate the binary search algorithm when (a) the target is found or (b) first becomes larger than last. Assume we want to find 11 in our binary search array.



## 2.2.5 Binary Search Algorithm

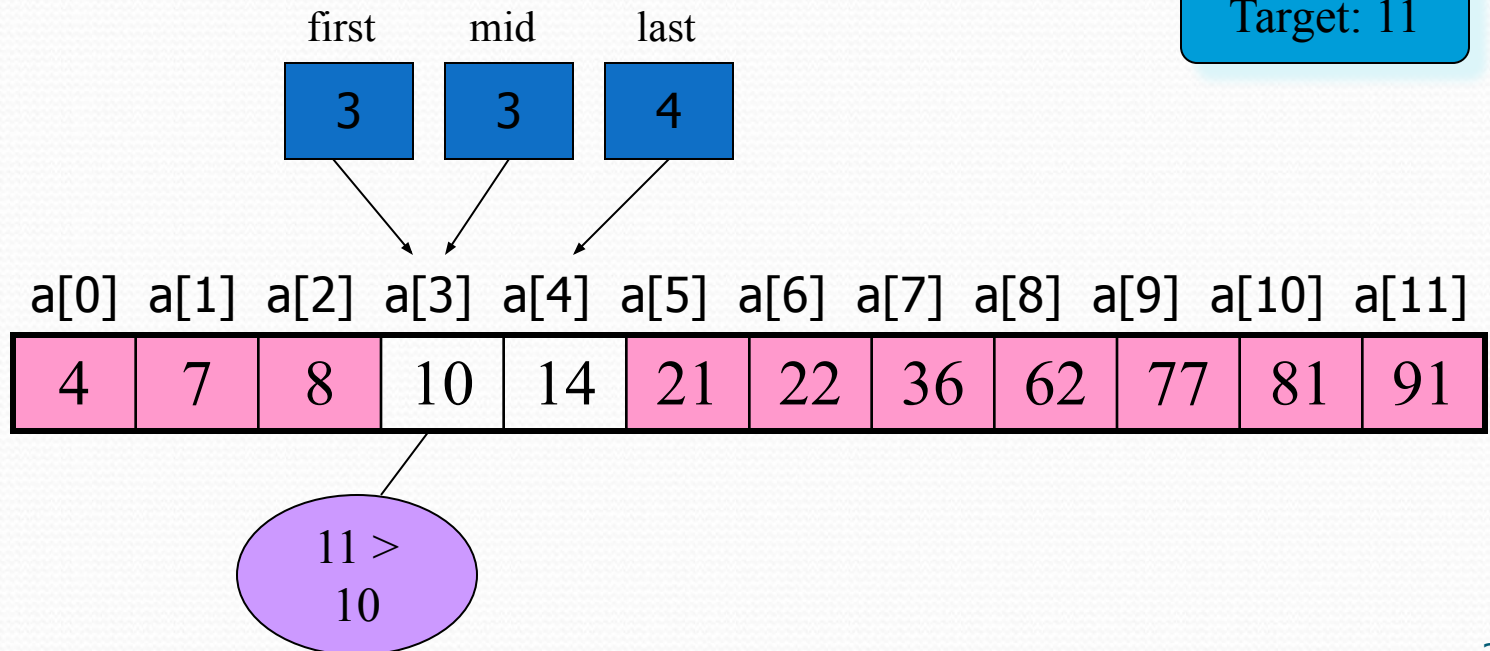
- The loop continues to narrow the range as we saw in the successful search until we are examining the data at index locations 3 and 4.





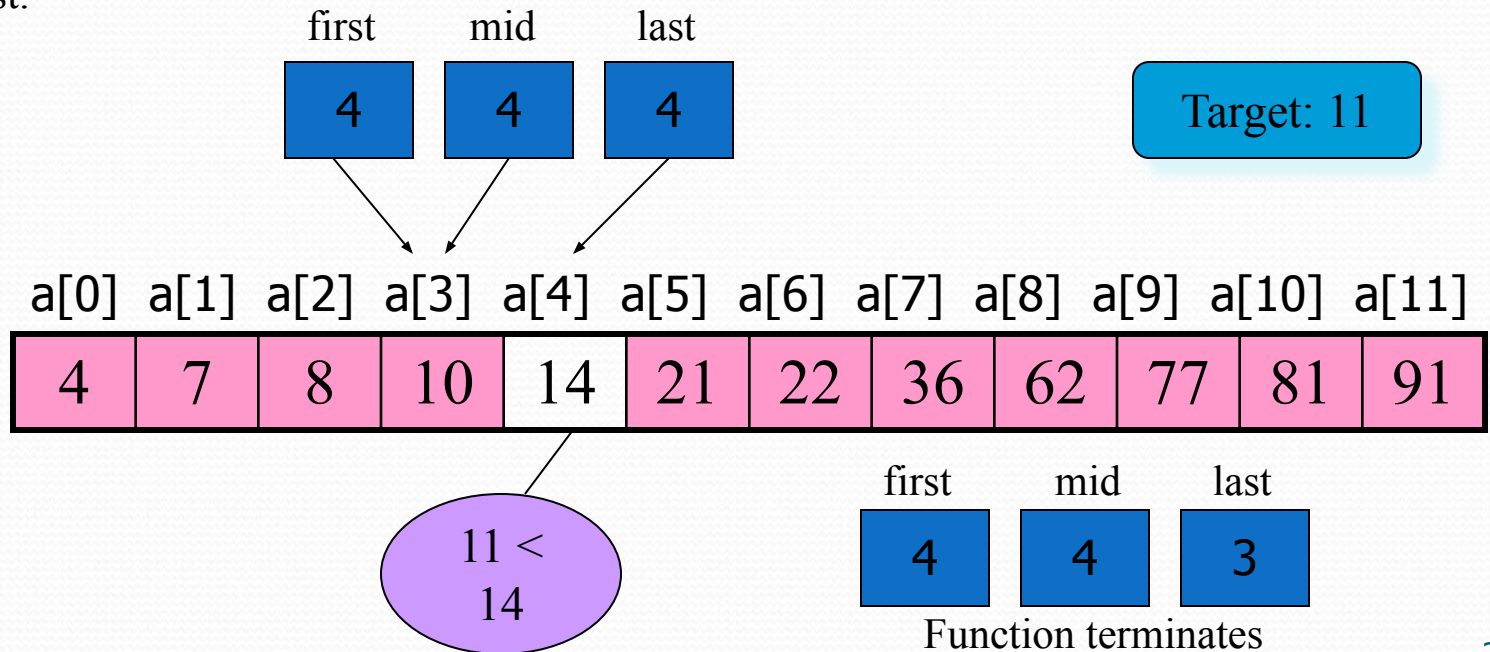
## 2.2.5 Binary Search Algorithm

- These settings of first and last set the mid index to 3 as follows:  
$$\text{mid} = (3 + 4) / 2 = 7 / 2 = 3$$



## 2.2.5 Binary Search Algorithm

- The test at index 3 indicates that the target is greater than the list value, so we set first to mid + 1, or 4. We now test the data at location 4 and discover that  $11 < 14$ . The mid is as calculated as follows:
- At this point, we have discovered that the target should be between two adjacent values; in other words, it is not in the list. We see this algorithmically because last is set to mid - 1, which makes first greater than last, the signal that the value we are looking for is not in the list.



## 2.2.5 Binary Search Algorithm

- Example algorithm:

DATA – sorted array

ITEM – Info

LB – lower bound

UB – upper bound

BEG – start Location

MID – middle Location

END – last Location

## 2.2.5 Binary Search Algorithm

**BINARY SEARCH(DATA,END,ITEM,MID,BEG,N,LOC):**binary search is applied on array DATA having N elements. Where BEG represents lower bound LB ,END represents upper bound UB and MID represents middle index of the array.

1. [Define variables]

Set  $BEG := LB$ ,  $END := UB$

Set  $MID := (BEG + END) / 2$

2. Repeat steps 3 and 4 While  $BEG \leq END$  &&  $DATA[MID] \neq ITEM$

3. If  $ITEM < DATA[MID]$ , then:

$END = MID - 1$

else:

$BEG := MID + 1$

4. Set  $MID := (BEG + END) / 2$

[end of while loop]

5. If  $DATA[MID] = ITEM$ , then:

Set  $LOC := MID$ .

write :element found at MID

else:

write: not found.

6. Exit

# Another way to do binary search algorithm

- `BINARYSEARCH(DATA,END,ITEM,MID,BEG,N,LOC)`: binary search is applied on array `DATA` having `N` elements. Where `BEG` represents lower bound `LB`, `End` represents upper bound `UB` and `MID` represents middle index of the array.
- 1. [Define variables]
  - Set `BEG := LB, LAST := UB`
  - Set `MID := (BEG+END)/2`
- 2. Repeat steps 3 and 4 While `BEG <= END`

- 3. If  $ITEM < DATA[MID]$ , then:
- $END = MID - 1$
- else if  $ITEM = DATA[MID]$ , then:
- write :Element found at MID and EXIT from while loop
- else:
- $BEG := MID + 1$ .
- 4. Set  $MID := (BEG + END) / 2$
- [end of while loop]
- 5. Exit

## 2.2.6 Merging Algorithm

- Suppose  $A$  is a sorted list with  $r$  elements and  $B$  is a sorted list with  $s$  elements. The operation that combines the elements of  $A$  and  $B$  into a single sorted list  $C$  with  $n=r + s$  elements is called merging.

## 2.2.6 Merging Algorithm

- Algorithm: Merging (A, R,B,S,C,N)  
Here A and B be sorted arrays with R and S elements respectively. This algorithm merges A and B into an array C with  $N=R+ S$  elements
- Step 1: Set  $NA:=0$ ,  $NB:=0$  and  $NC:=0$
- Step 2: Repeat while  $NA < R$  and  $NB < S$ :
  - if  $A[NA] \leq B[NB]$ , then:
    - Set  $C[NC] := A[NA]$
    - Set  $NA: = NA +1$
  - else
    - Set  $C[NC] := B[NB]$
    - Set  $NB: = NB +1$
  - [End of if-else structure]
  - Set  $NC:= NC +1$
  - [End of while Loop]



## 2.2.6 Merging Algorithm

- Step 3: If  $NA > R$ , then:
  - Repeat while  $NB < S$ :
    - Set  $C[NC] := B[NB]$
    - Set  $NB := NB + 1$
    - Set  $NC := NC + 1$
    - [End of while Loop]
  - else
    - Repeat while  $NA < R$ :
      - Set  $C[NC] := A[NA]$
      - Set  $NC := NC + 1$
      - Set  $NA := NA + 1$
      - [End of while loop]
    - [End of if-else structure]
- Step 4: Exit

## 2.2.6 Merging Algorithm

- Complexity of merging: The input consists of the total number  $n=r+s$  elements in A and B. Each comparison assigns an element to the array C, which eventually has  $n$  elements. Accordingly, the number  $f(n)$  of comparisons cannot exceed  $n$ :

$$f(n) \leq n = O(n)$$

# Exercises

- Find where the indicated elements of an array `a` are stored, if the base address of `a` is  $200^*$  and  $LB = 0$ 
  - a) `double a[10]; a[3]`?
  - b) `int a[26]; a[2]`?

\* (assume that `int(s)` are stored in 4 bytes and `double(s)` in 8 bytes).

## 2.3 MULTIDIMENSIONAL ARRAY

- Two or more subscripts.

# 2-D ARRAY

- A 2-D array, A with  $m \times n$  elements.
- In math application it is called *matrix*.
- In business application – table.
- Example:

Assume 25 students had taken 4 tests.

The marks are stored in 25 X 4 array locations:

	U0	U1	U2	U3
Stud 0	88	78	66	89
Stud 1	60	70	88	90
Stud 2	62	45	78	88
..	..	..	..	..
..	..	..	..	..
Stud 24	78	88	98	67

n

m

# 2-D ARRAY

- Multidimensional array declaration in C++:-

```
int StudentMarks [25][4];
```

```
StudentMarks[0][0] = 88;
```

```
StudentMarks[0][1] = 78;.....
```

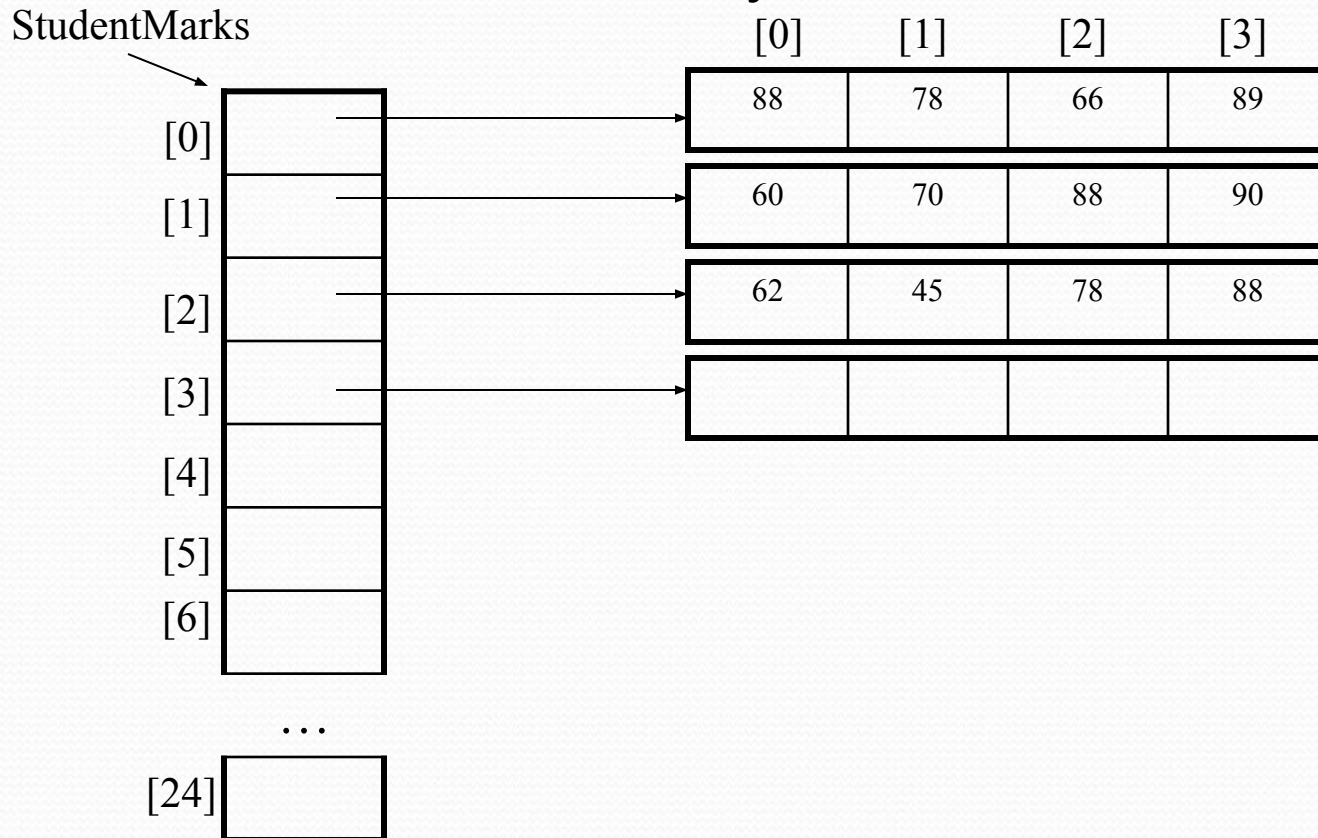
OR

```
int StudentMarks [25][4] = {{88, 78, 66, 89},
```

```
    {60, 70, 88, 90},...}
```

## 2.3.1 2-D ARRAY

- In C++ the 2-D array is visualized as follows:



## 2.3.2 Representation of 2D arrays in Memory

Column Major Order:

$$\text{LOC}(A[j, k]) = \text{Base}(A) + w[m*k + j]$$

Row Major order:

$$\text{LOC}(A[j, k]) = \text{Base}(A) + w[n*j + k]$$

Given: A 2-D array,  $A$  with  $m \times n$  elements.





Thank You