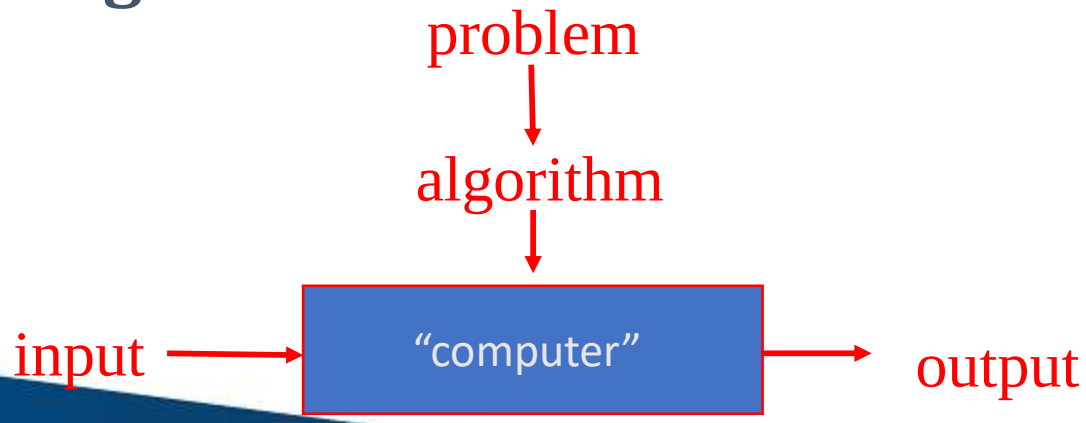
Data Structures and Algorithms (CSE2001)

MODULE 4
INTRODUCTION TO ALGORITHMS

What is an algorithm?

An *algorithm* is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.

Notion of Algorithm



Features/ Properties of Algorithm

- Input: Should accept 0 or more inputs

- Output: Should produce at least one output

- Effectiveness: Every instruction should transform the given I/P to desired output

- Finiteness/termination: Algorithm should terminate after finite number of steps

- Definiteness: Each Instruction should be clear and unambiguous

Why study algorithms?

- Theoretical importance
  - the core of computer science

- Practical importance
  - A practitioner's toolkit of known algorithms
  - Framework for designing and analyzing algorithms for new problems

Example: Google's PageRank Technology

Analysis of algorithms

- How good is the algorithm?
  - time efficiency
  - space efficiency

- Does there exist a better algorithm?
  - lower bounds
  - optimality

Algorithm design strategies

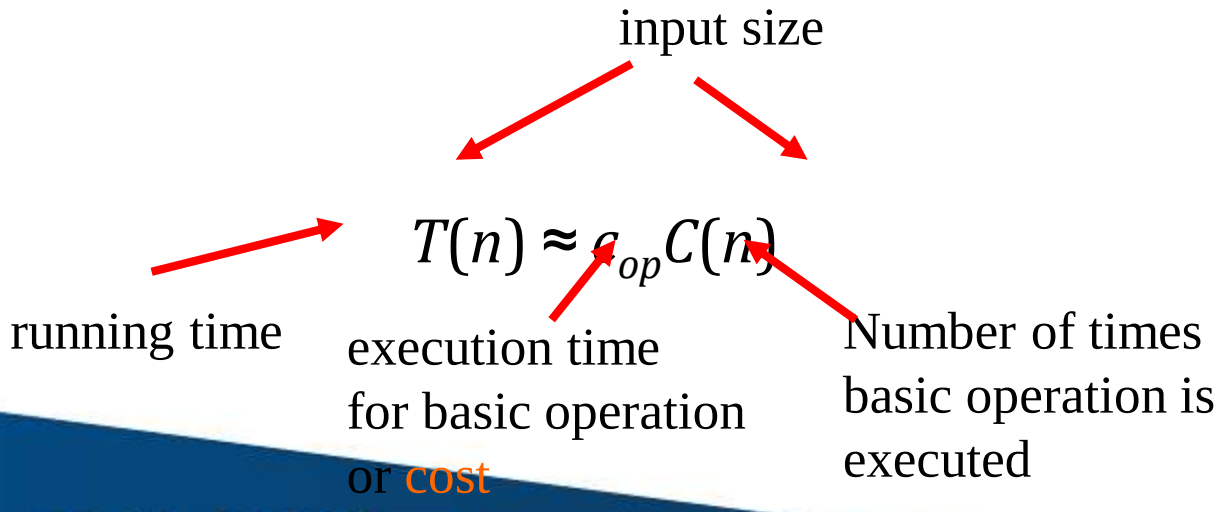
- **Brute force**
- **Divide and conquer**
- **Decrease and conquer**
- **Transform and conquer**

- **Greedy approach**
- **Dynamic programming**
- **Backtracking and branch-and-bound**
- **Space and time tradeoffs**

Theoretical analysis of time efficiency

Time efficiency is analyzed by determining the number of repetitions of the *basic operation* as a function of *input size*

- *Basic operation*: the operation that contributes the most towards the running time of the algorithm



Note: Different basic operations may cost differently!

Input size and basic operation examples

Problem	*Input size measure*	*Basic operation*
Searching for key in a list of *n* items	**Number of list's items, i.e. *n***	**Key comparison**
Multiplication of two matrices	**Matrix dimensions or total number of elements**	**Multiplication of two numbers**
Checking primality of a given integer *n*	***n*'size = number of digits (in binary representation)**	**Division**
Typical graph problem	**#vertices and/or edges**	**Visiting a vertex or traversing an edge**

Best-case, average-case, worst-case

For some algorithms, efficiency depends on form of input:

- Worst case: $C_{worst}(n)$ – maximum over inputs of size n
- Best case: $C_{best}(n)$ – minimum over inputs of size n
- Average case: $C_{avg}(n)$ – "average" over inputs of size n

  - Number of times the basic operation will be executed on typical input
  - NOT the average of worst and best case

Order of growth

- As the value of n(input size) increases, time required for execution also increases i.e., behavior of algorithm changes with the increase in the value of n. This change in the behavior is called **orders of growth**
- Most important: Order of growth within a constant multiple as $n \to \infty$
- Example:
  - How much faster will algorithm run on computer that is twice as fast?
  - How much longer does it take to solve problem of double input size?

Values of some important functions as $n \to \infty$

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Table 2.1 Values (some approximate) of several functions important for analysis of algorithms

Basic asymptotic efficiency classes

1	**constant**
log *n*	**logarithmic**
n	**linear**
n* log *n	***n*-log-*n***
n^2	**quadratic**
n^3	**cubic**
2^n	**exponential**
***n*!**	**factorial**

Asymptotic order of growth

A way of comparing functions that ignores constant factors and small input sizes (because?)

- $O(g(n))$: class of functions $f(n)$ that grow no faster than $g(n)$
- $\Theta(g(n))$: class of functions $f(n)$ that grow at same rate as $g(n)$
- $\Omega(g(n))$: class of functions $f(n)$ that grow at least as fast as $g(n)$

Big-oh

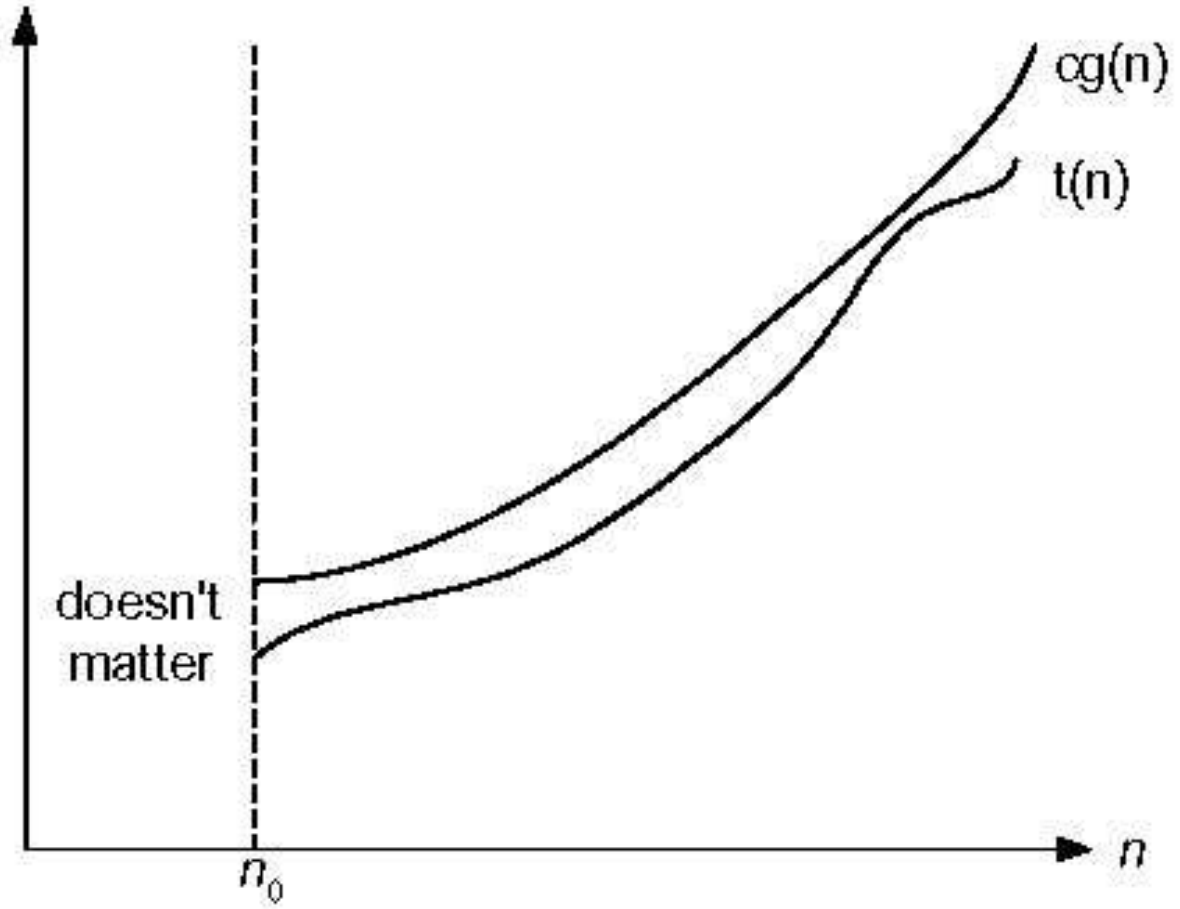


Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$

Big-omega

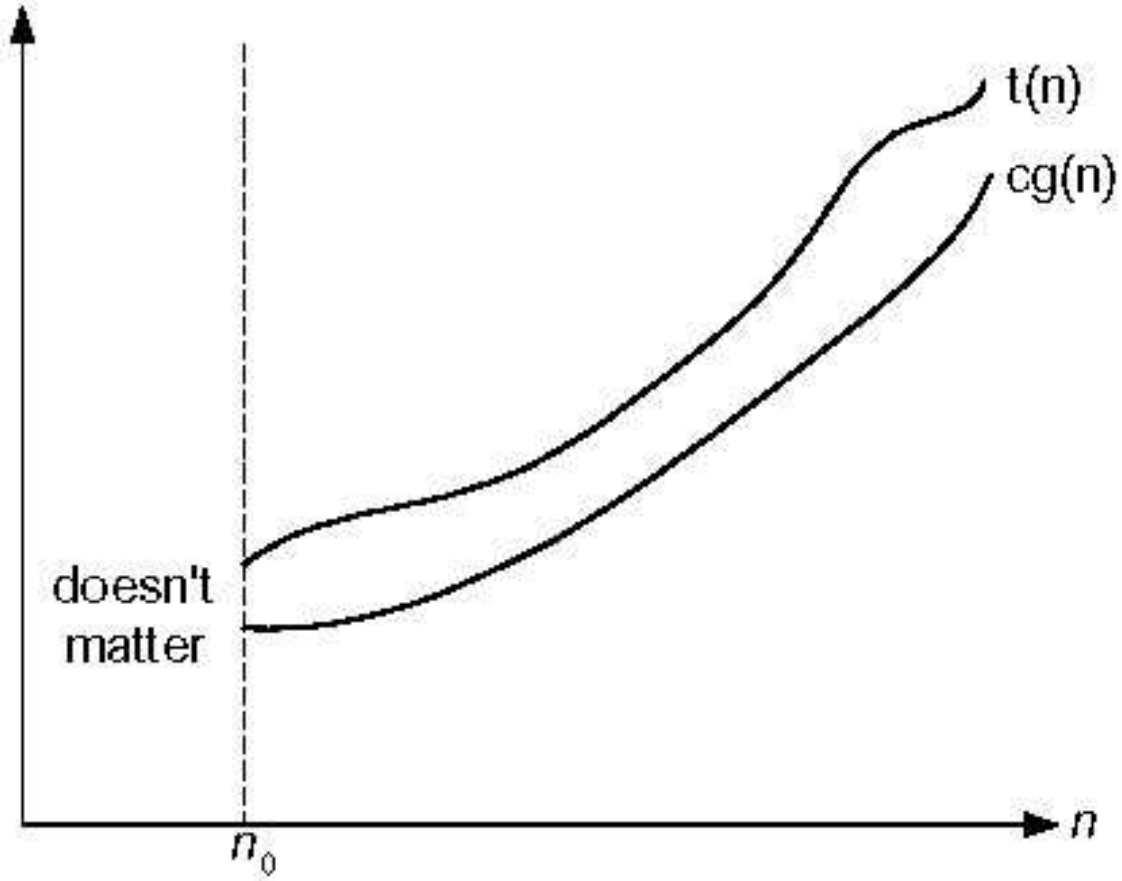


Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

Big-theta

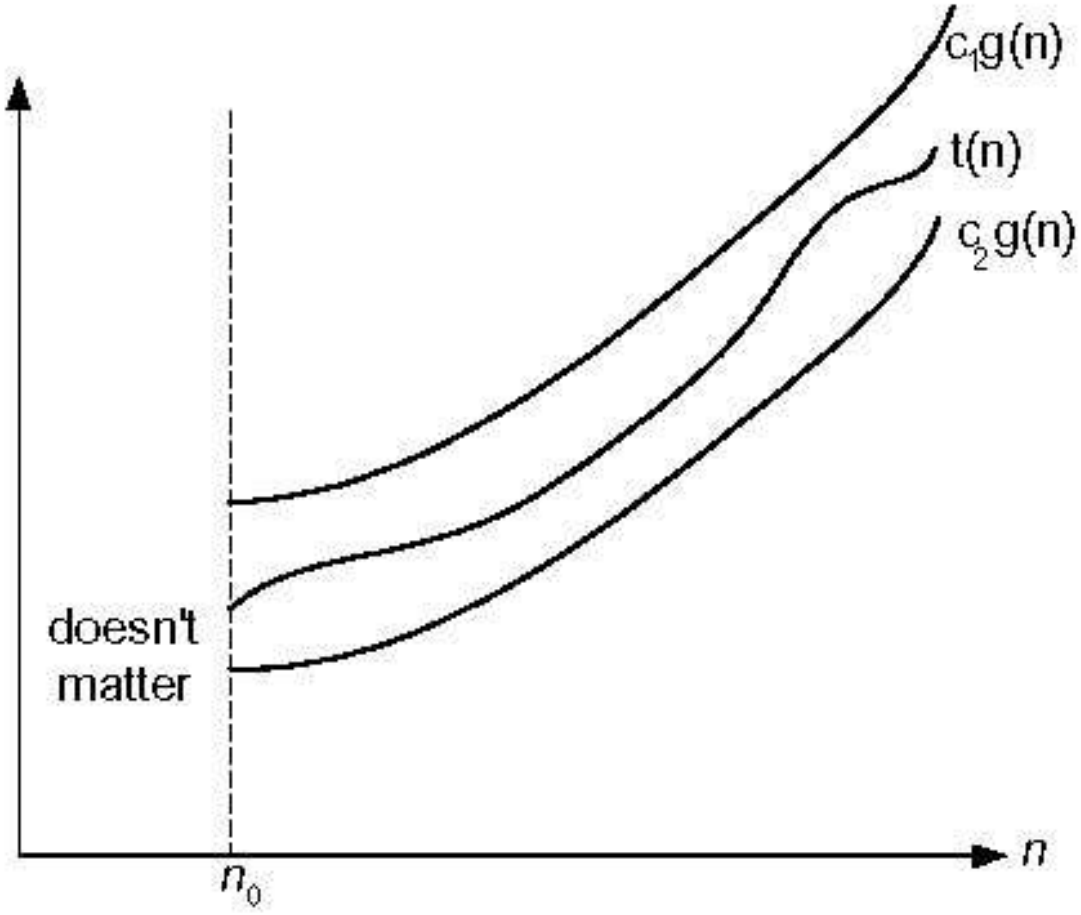


Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

Establishing order of growth using the definition

Definition: $f(n)$ is in $O(g(n))$, denoted $f(n) \in O(g(n))$, if order of growth of $f(n) \leq$ order of growth of $g(n)$ (within constant multiple), i.e., there exist positive constant c and non-negative integer n_0 such that

$$f(n) \leq c\, g(n) \text{ for every } n \geq n_0$$

Examples:

- $10n$ is in $O(n^2)$
- $5n+20$ is in $O(25n)$

O-notation

- Formal definition
  - A function *t(n)* is said to be in O*(g(n))*, denoted *t(n)* $\in$ O*(g(n))*, if *t(n)* is bounded above by some constant multiple of *g(n)* for all large *n*, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

    t(n) <= cg(n) for all $n \geq n_0$
- Exercises: prove the following using the above definition
  - $10n^2 \in \Omega(n^2)$
  - $0.1n^2 \in \Omega(n^3)$

Ω-notation

- Formal definition
  - A function *t(n)* is said to be in Ω*(g(n))*, denoted *t(n)* ∈ Ω*(g(n))*, if *t(n)* is bounded below by some constant multiple of *g(n)* for all large *n*, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

    $t(n) \geq cg(n)$ for all $n \geq n_0$
- Exercises: prove the following using the above definition
  - $10n^2 \in \Omega(n^2)$
  - $0.3n^2 - 2n \in \Omega(n^2)$
  - $0.1n^3 \in \Omega(n^2)$

Θ-notation

- Formal definition
  - A function *t(n)* is said to be in Θ*(g(n))*, denoted *t(n)* ∈ Θ*(g(n))*, if *t(n)* is bounded both above and below by some positive constant multiples of *g(n)* for all large *n*, i.e., if there exist some positive constant c_1 and c_2 and some nonnegative integer n_0 such that

    $c_2\, g(n) \leq t(n) \leq c_1\, g(n)$ for all $n \geq n_0$

- Exercises: prove the following using the above definition
  - $10n^2 \in \Theta(n^2)$
  - $(1/2)n(n+1) \in \Theta(n^2)$

Time efficiency of nonrecursive algorithms

General Plan for Analysis

- Decide on parameter *n* indicating *input size*
- Identify algorithm's *basic operation*
- Determine *worst*, *average*, and *best* cases for input of size *n*
- Set up a sum for the number of times the basic operation is executed. Apply rules/formulas to deduce efficiency and represent the same using asymptotic notations

Useful summation formulas and rules

$\Sigma_{l \le i \le n} 1 = 1+1+\ldots+1 = n - l + 1$

In particular, $\Sigma_{l \le i \le n} 1 = n - 1 + 1 = n \in \Theta(n)$

$\Sigma_{1 \le i \le n} i = 1+2+\ldots+n = n(n+1)/2 \approx n^2/2 \in \Theta(n^2)$

$\Sigma_{1 \le i \le n} i^2 = 1^2+2^2+\ldots+n^2 = n(n+1)(2n+1)/6 \approx n^3/3 \in \Theta(n^3)$

$\Sigma_{0 \le i \le n} a^i = 1 + a + \ldots + a^n = (a^{n+1} - 1)/(a - 1)$ for any $a \neq 1$

In particular, $\Sigma_{0 \le i \le n} 2^i = 2^0 + 2^1 + \ldots + 2^n = 2^{n+1} - 1 \in \Theta(2^n)$

$\Sigma(a_i \pm b_i) = \Sigma a_i \pm \Sigma b_i \quad \Sigma c a_i = c \Sigma a_i \quad \Sigma_{l \le i \le u} a_i = \Sigma_{l \le i \le m} a_i + \Sigma_{m+1 \le i \le u} a_i$

Example 1: Maximum element

ALGORITHM *MaxElement*$(A[0..n-1])$

```
//Determines the value of the largest element in a given array
//Input: An array A[0..n − 1] of real numbers
//Output: The value of the largest element in A
maxval ← A[0]
for i ← 1 to n − 1 do
    if A[i] > maxval
        maxval ← A[i]
return maxval
```

T(n) = $\Sigma_{1 \le i \le n-1} 1 = n-1 = \Theta(n)$ comparisons

Plan for Analysis of Recursive Algorithms

- Decide on a parameter indicating an input's size.
- Identify the algorithm's basic operation.
- Check whether the number of times the basic op. is executed may vary on different inputs of the same size. (If it may, the worst, average, and best cases must be investigated separately.)
- Set up a recurrence relation with an appropriate initial condition expressing the number of times the basic op. is executed.

- Solve the recurrence (or, at the very least, establish its solution's order of growth) by backward substitutions or another method.

Example 1: Recursive evaluation of $n!$

Definition: $n! = 1 * 2 * \ldots * (n-1) * n$ for $n \geq 1$ and $0! = 1$

Recursive definition of $n!$: $F(n) = F(n-1) * n$ for $n \geq 1$ and $F(0) = 1$

Size: n

Basic operation: multiplication

ALGORITHM $F(n)$

//Computes $n!$ recursively
//Input: A nonnegative integer n
//Output: The value of $n!$
if $n = 0$ **return** 1
else return $F(n-1) * n$

Recurrence relation:

$M(n) = M(n-1) + 1$

$M(0) = 0$

Solving the recurrence for M(n)

$$M(n) = M(n\text{-}1) + 1,\ M(0) = 0$$

M(n) = M(n-1) + 1

= (M(n-2) + 1) + 1 = M(n-2) + 2

= (M(n-3) + 1) + 2 = M(n-3) + 3

…

= M(n-i) + i

= M(0) + n

= n

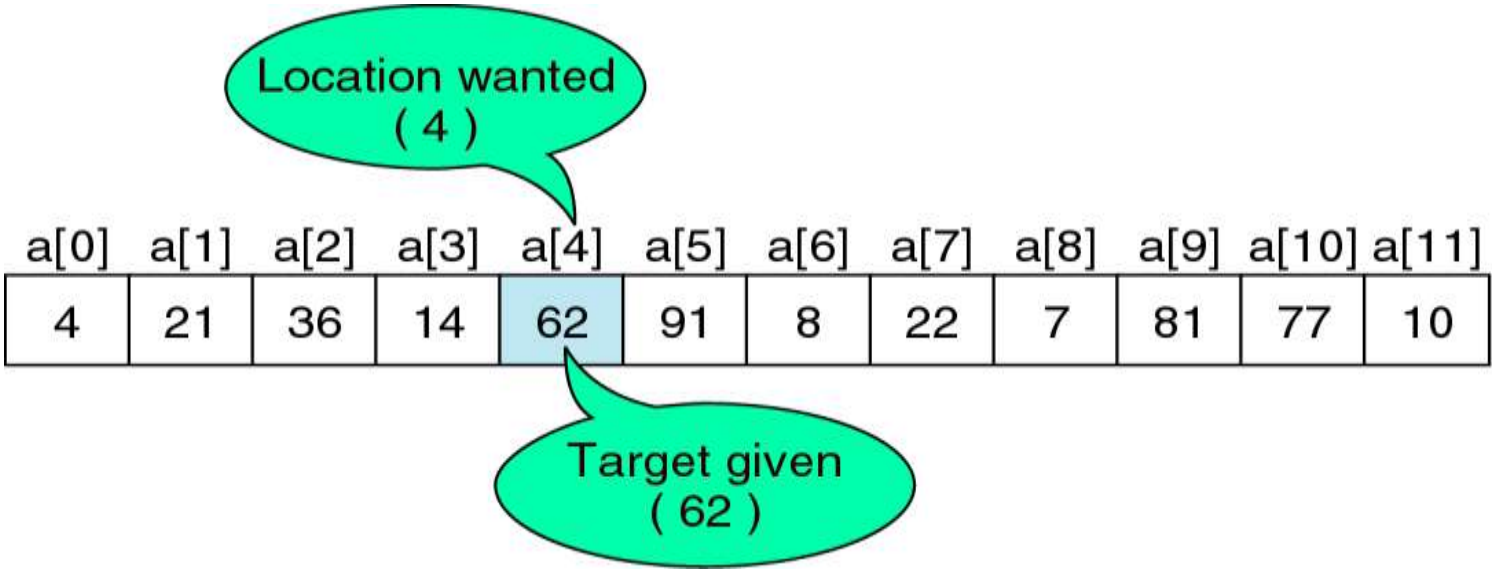
The method is called backward substitution.

Searching

The process uses to find the location of a target among the list of objects. In array, find the location (index) of the first element in the array given the value.

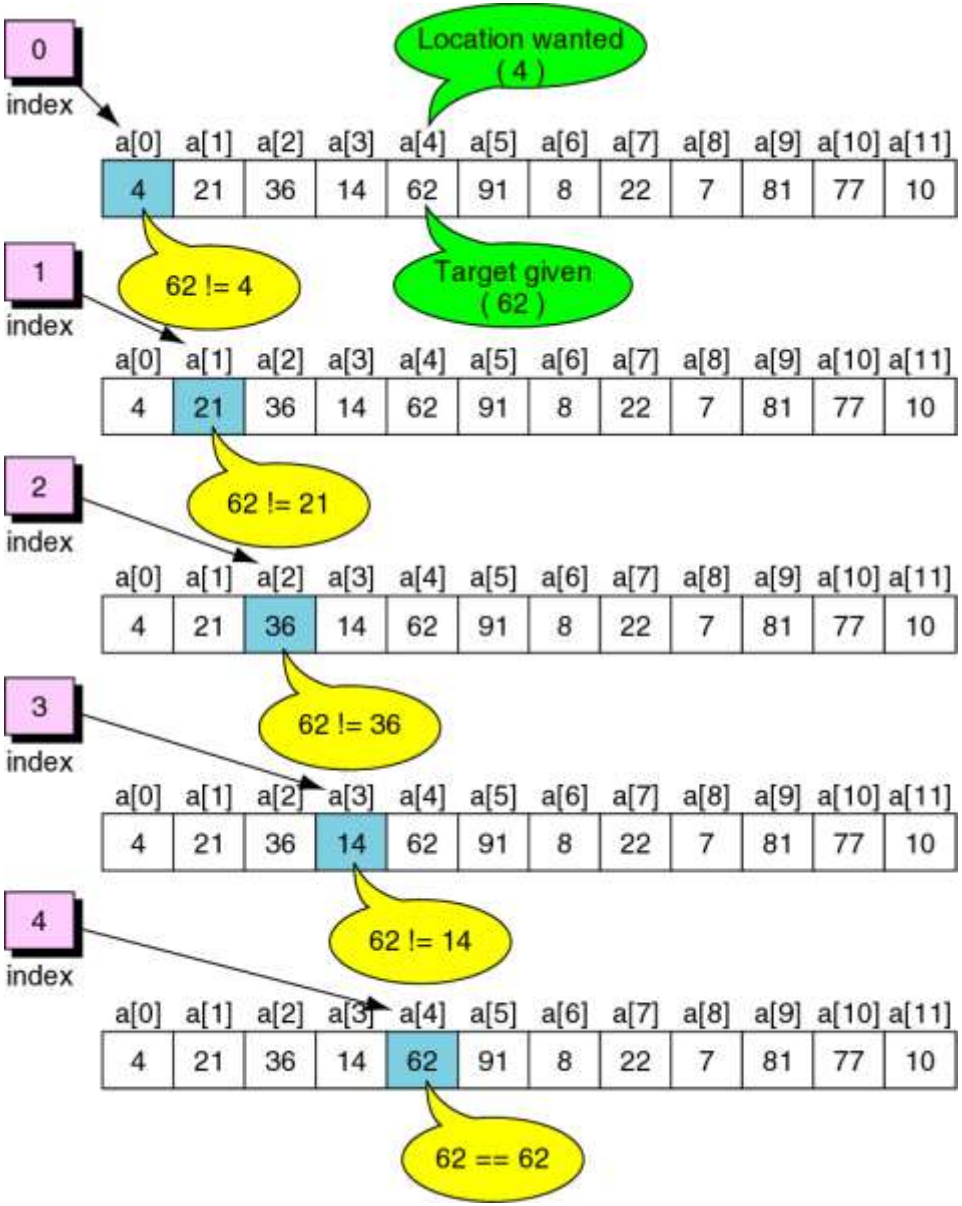
Searching algorithms:

1. **Sequential Search –** used when list is unordered
2. **Binary Search –** list is ordered

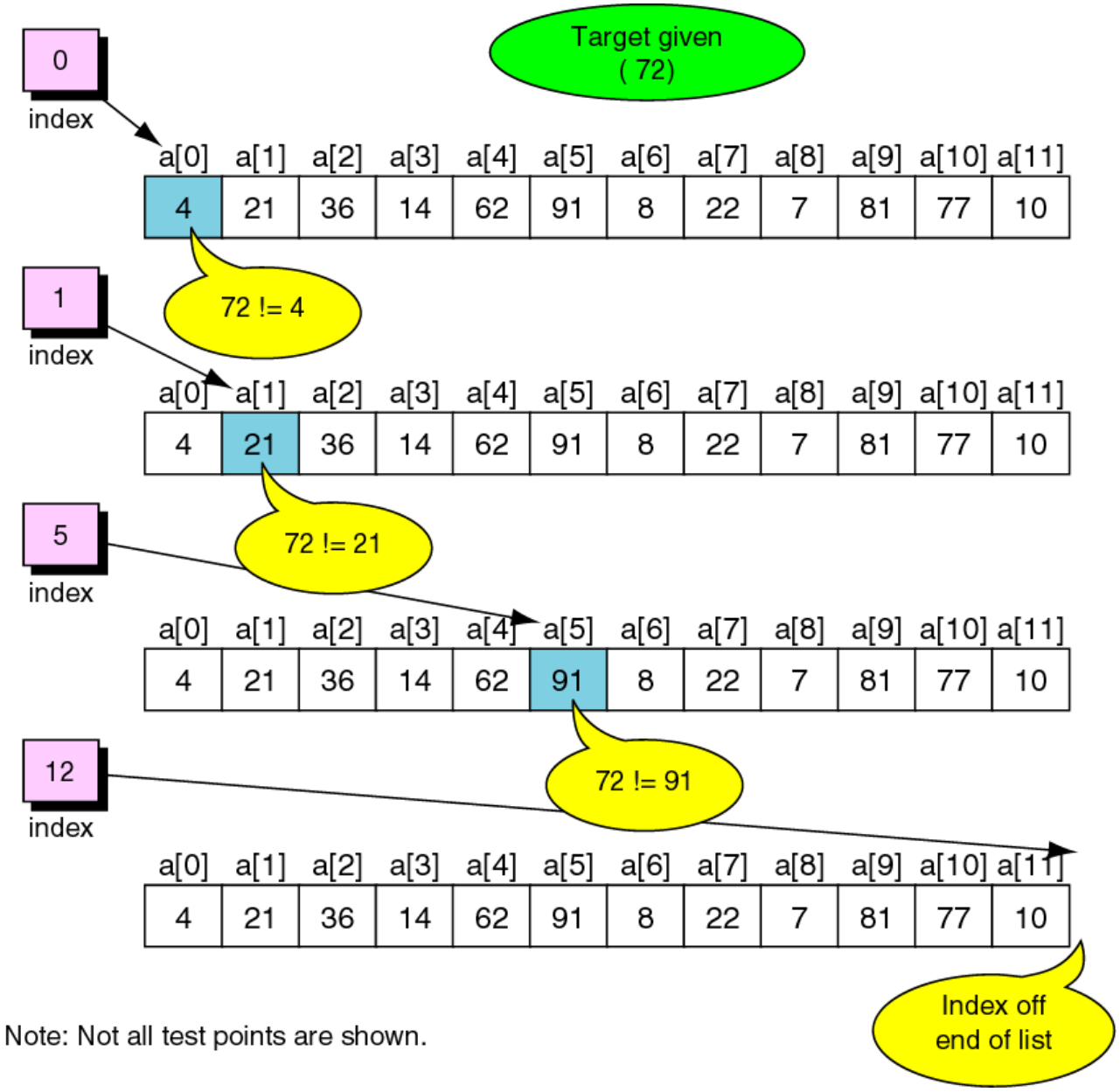


Search Concept

Sequential Search concept – Locating data in unordered list



Sequential Search concept – Unsuccessful search in unordered list



Note: Not all test points are shown.

Sequential Search Algorithm and Efficiency

ALGORITHM *SequentialSearch*$(A[0..n-1], K)$

```
//Searches for a given value in a given array by sequential search
//Input: An array A[0..n − 1] and a search key K
//Output: The index of the first element of A that matches K
//        or −1 if there are no matching elements
i ← 0
while i < n and A[i] ≠ K do
    i ← i + 1
if i < n return i
else return −1
```

Efficiency:

Worst case : n key comparisons. O(n)

Best case : 1 key comparison. Ω(1)

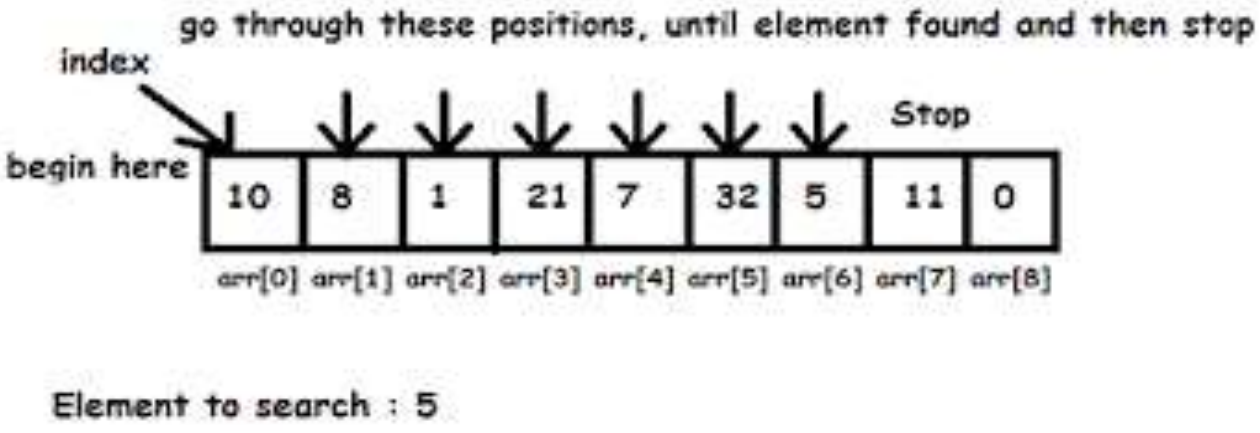
Average case: (n+1)/2, key comparisions. Θ(n/2)

(assuming Search is successful)

Sequential Search (Uses Brute Force Technique)

- Algorithm efficiency depends on the input size n
- For some algorithms efficiency depends on type of input.
  - Example: Sequential Search
    - *Problem:* Given a list of *n* elements and a search key *K*, find an element equal to *K*, if any.
    - *Algorithm:* Scan the list and compare its successive elements with *K* until either a matching element is found (*successful search*) or the list is exhausted (*unsuccessful search*)

- Example



Binary Search

Very efficient algorithm for searching in sorted array:

$$K$$

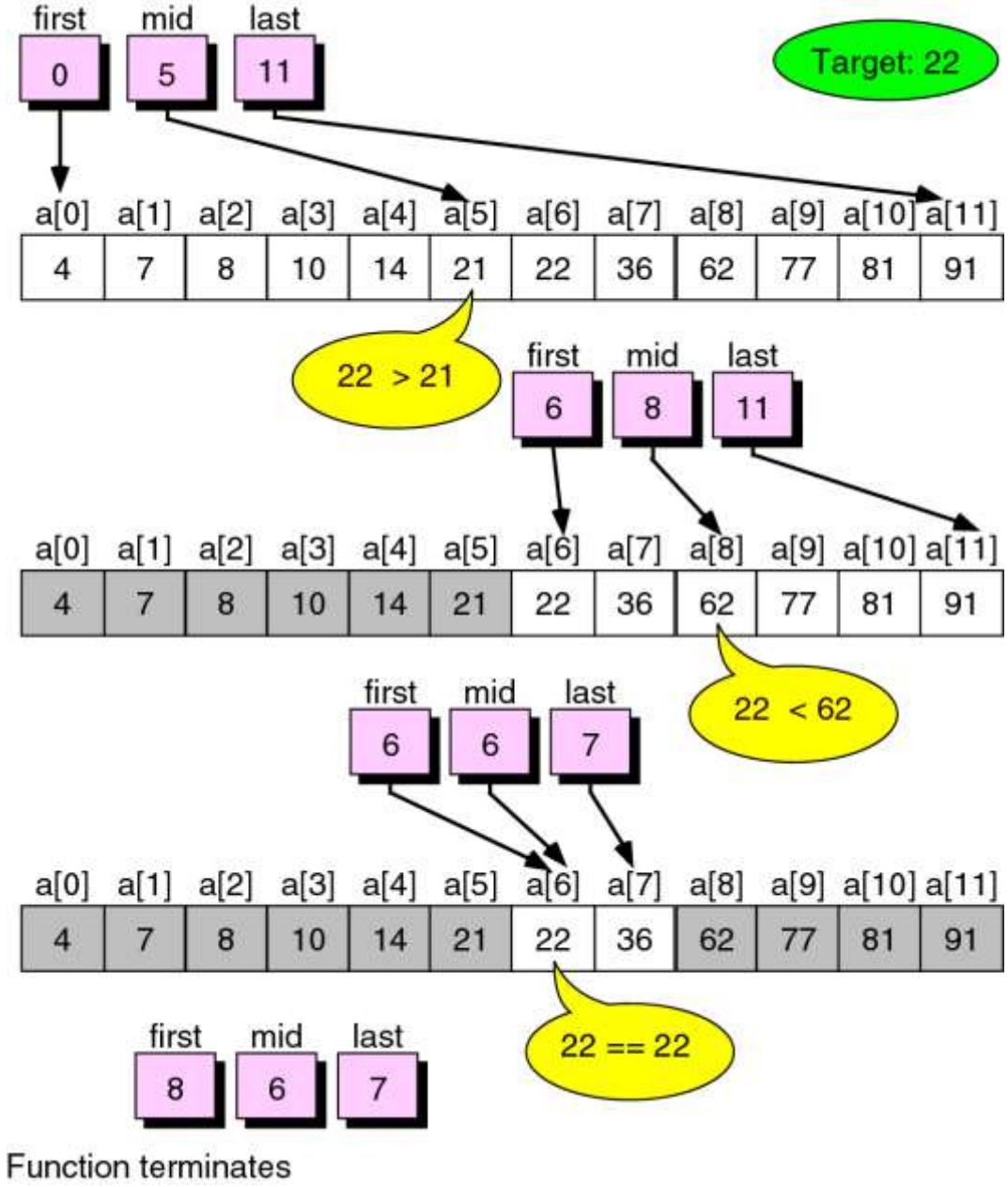
vs

$$A[0] \ . \ . \ . \ A[m] \ . \ . \ . \ A[n-1]$$

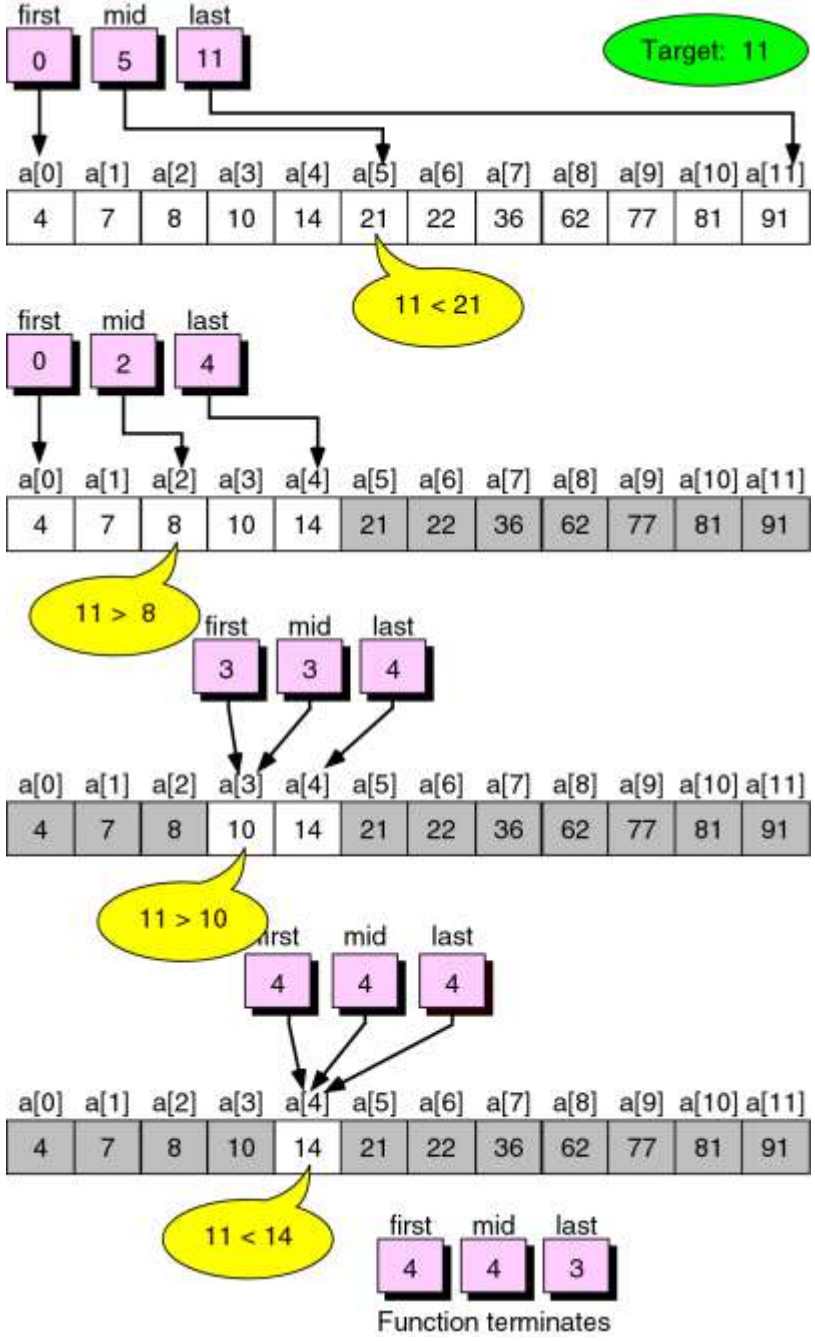
If $K = A[m]$, stop (successful search); otherwise, continue searching by the same method in $A[0..m-1]$ if $K < A[m]$ and in $A[m+1..n-1]$ if $K > A[m]$

$l \leftarrow 0$; $r \leftarrow n-1$
while $l \leq r$ do
  $m \leftarrow \lfloor (l+r)/2 \rfloor$
  if $K = A[m]$ return m
  else if $K < A[m]$ $r \leftarrow m-1$
  else $l \leftarrow m+1$
return -1

Binary Search Example



Unsuccessful Binary Search Example



Analysis of Binary Search

- Time efficiency
  - **worst-case using recurrence equation:**

T(n)= 1 if n=1

T(n/2) +1 otherwise

Where,

T(n/2) : time required to search either left or right part of the array
1 represents time required to compare middle element
By using recurrence equation:

T(n)= T(n/2)+1……(1)

Solve Eq (1) using Master's theorem

a=1, b=2 and d= 0 (Because 1 in eq (1) can be written as n^0=1

a=b=1 So case If $a = b^d$,

$T(n) \in O(n^d \log n)$

$T(n) \in O(n^0 \log n)$

$\boldsymbol{T(n) \in O(\log n)}$

Average case is: Θ(log n)

Best case is: Ω(1) the best case occurs when the element to be searched is present in the middle of the array.

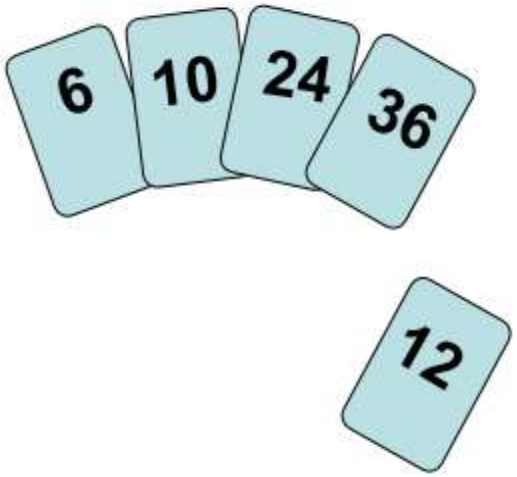
•Optimal for searching a sorted array

•Limitations: must be a sorted array (not linked list)

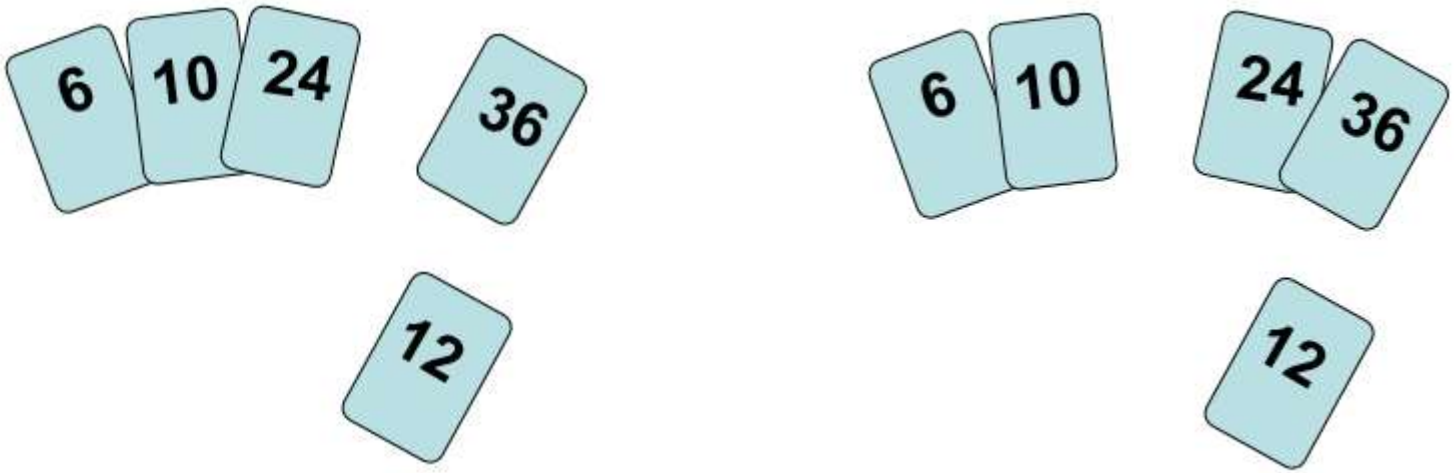
•Bad (degenerate) example of divide-and-conquer because only one of the sub-instances is solved

Insertion Sort

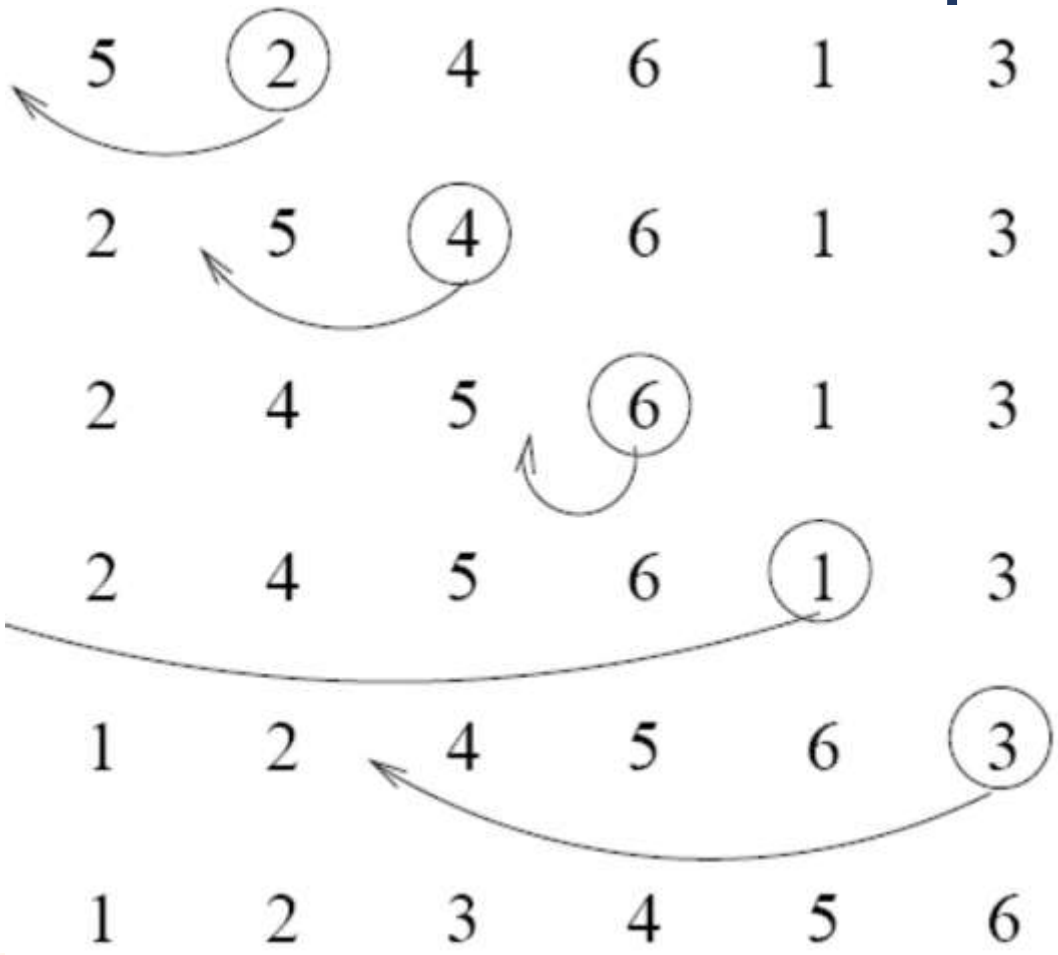
- Idea: like sorting a hand of playing cards
- Start with an empty left hand and the cards facing down on the table.
- Remove one card at a time from the table, and insert it into the correct position in the left hand
- compare it with each of the cards already in the hand, from right to left
- The cards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

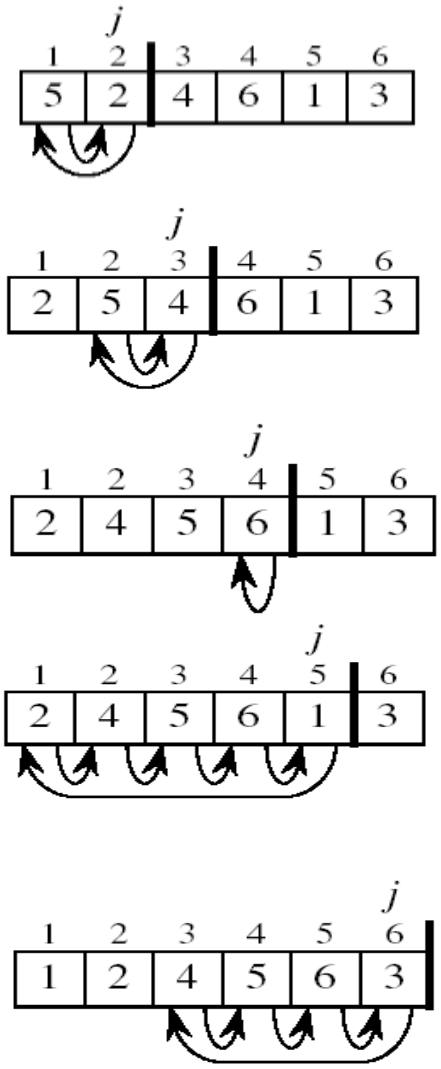


To insert 12, we need to make room for it by moving first 36 and then 24.



Example



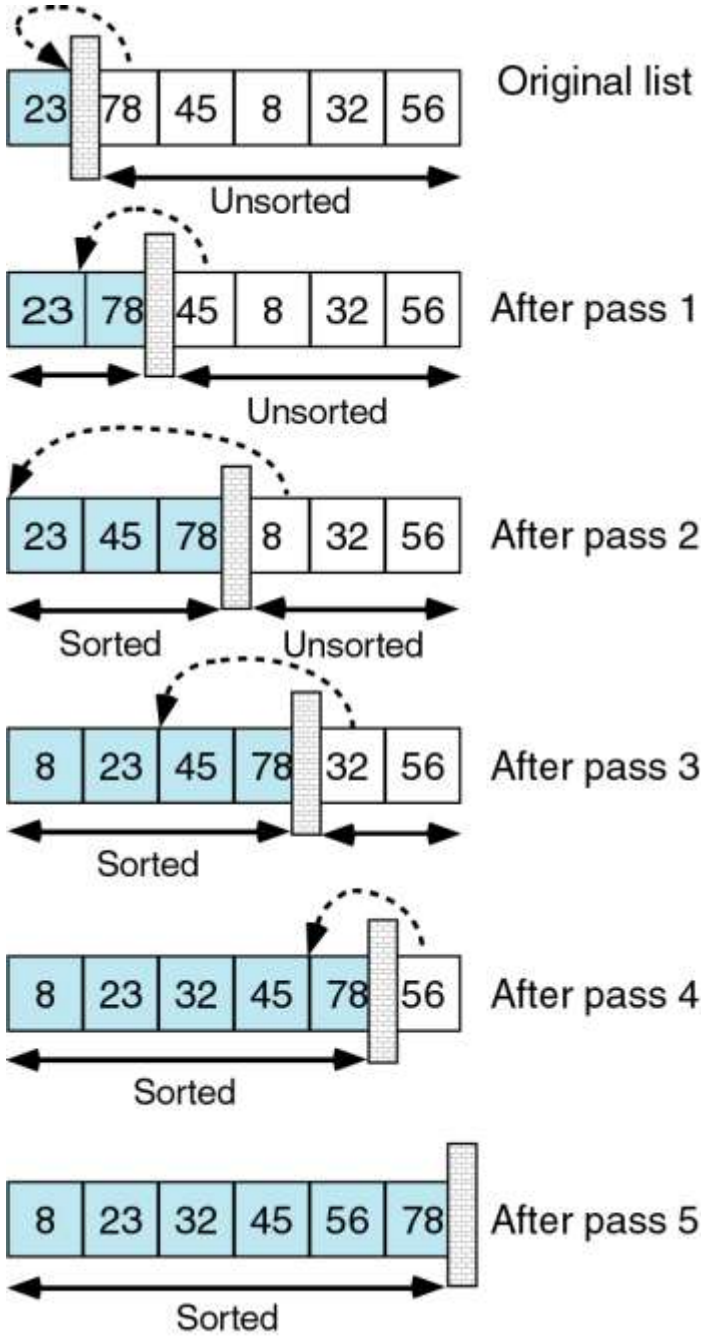


Insertion Sort Algorithm

```
for j=2 to A.length
   key = A[j]
   i = j -1
   while( A[i] > key and i > 0 )
      A[i + 1] = A[i]
      i = i -1
   A[i + 1] = key
```

$\Theta(n^2)$ running time in worst and average case

Insertion Sort Example



Selection Sort

Algorithm *SelectionSort(A[0..n-1])*

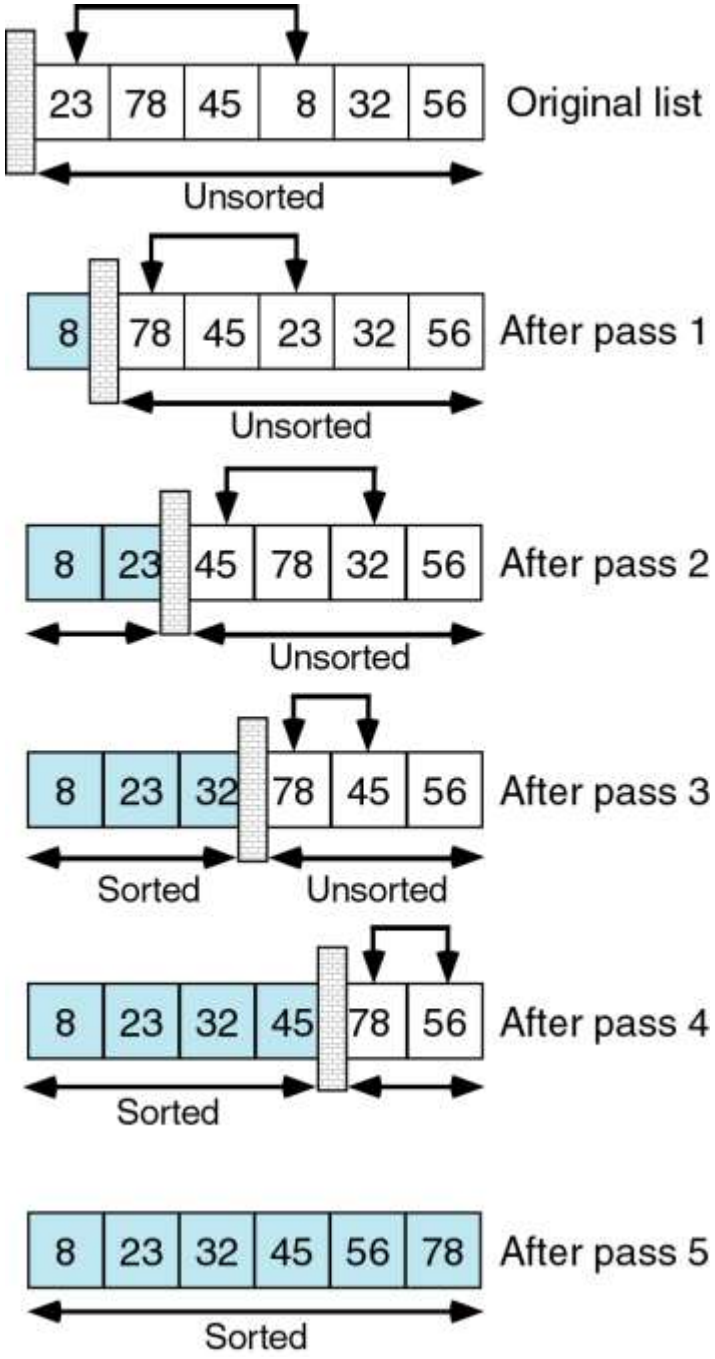
//The algorithm sorts a given array by selection sort

//Input: An array A[0..n-1] of orderable elements

//Output: Array A[0..n-1] sorted in ascending order

```
for i ← 0 to n – 2 do
    min ← i
    for j ← i + 1 to n – 1 do
            if A[j] < A[min]
                    min ← j
    swap A[i] and A[min]
```

Selection Sort Example



Complexity Analysis of Selection Sort

Input: Given **n** input elements.

Output: Number of steps incurred to sort a list.

Logic: If we are given n elements, then in the first pass, it will do **n-1** comparisons; in the second pass, it will do **n-2**; in the third pass, it will do **n-3** and so on. Thus, the total number of comparisons can be found by;

Output;

$(n-1) + (n-2) + (n-3) + (n-4) + \ldots\ldots + 1$

$$Sum = \frac{n(n-1)}{2}$$

i.e., $O(n^2)$