Chapter One

Basic Concepts in Algorithmic Analysis

1.1 Introduction:

An *algorithm* is a *computational procedure* that consists of a finite set of instructions which, given an input from some set of possible inputs, enables us to obtain an output if such an output exists or else obtain nothing at all if there is no output for that particular input through a systematic execution of the instructions. A well-defined **sequence of computational steps that accepts a set of values as input and produces a set of values as output.**

Example : The problem of sorting:

- Input : a sequence of n numbers: $a_1, a_2, \ldots a_n$

- Output : a reordering of the input: $a'_1, a'_2, \ldots a'_n$

where $a'_1 <= a'_2 <= \ldots <= a'_n$

In particular, we may be interested in finding the most efficient algorithm for solving a particular problem (Efficiency). An algorithm can be represented by:

1. Flowcharts.
2. Pseudo Codes.
3. Prose Codes (Pseudo + Documentation).

- *Algorithm Analysis*: is an abstract or mathematical comparisons between algorithms - without implementing them - by specifying the time and the space needed for solving that algorithm.

- *Algorithm Design*: is a set of methods, ideas, and tricks for developing a fast algorithm. It is observed that the number of basic design techniques is reasonably small.

Computational problems

• A computational problem specifies an input-output relationship

What does the input look like?

What should the output be for each input?

• Example:

Input: an integer number n

Output: Is the number prime?

• Example:
Input: A list of names of people
Output: The same list sorted alphabetically

• Example:
Input: A picture in digital format
Output: An English description of what the picture shows.

Algorithms specification

• A tool for solving a well-specified computational problem
• Algorithms must be:
Correct: For each input produce an appropriate output
Efficient: run as quickly as possible, and use as little memory as possible – more about this later.

Components of an Algorithm

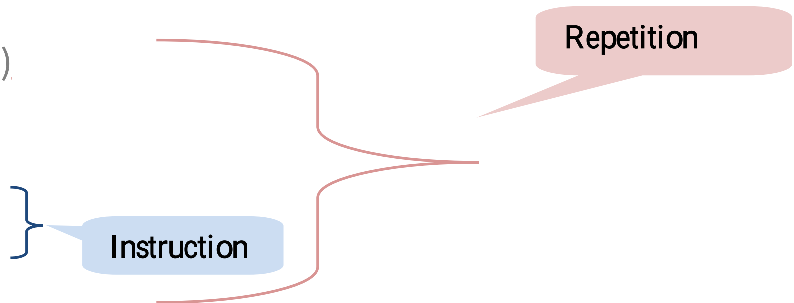
- •Algorithm name
- Input
- Output
- Variables and values
- Instructions
- Procedures name (involving instructions)
- Selections (between instructions)
- Repetitions
- Documentation

Values & Variables
• Represent quantities, amounts or measurements
• May be numerical or alphabetical (or other things)
• Often have a unit related to their purpose

• Example:

```
procedure Sum1_to_n(num)
{
// do initialization
count = 1
sum = 0
while (count <= num)

{
add count to sum
add 1 to count
}
}
```



1.7 Why Analyze Algorithms?

There may be several different ways to solve a particular problem. For example, there are several methods for sorting numbers. How can you decide which method is the best in a certain situation? How would you define "best" – is it the fastest method or the one that takes up the least amount of memory space?

Understanding the relative efficiencies of algorithms designed to do the same task is very important in every area of computing. This is how computer scientists decide which algorithm to use for a particular application. As mentioned earlier, an algorithm can be analyzed in terms of ***time efficiency or space utilization***. We will consider only the former right now. The running time of an algorithm is influenced by several factors:

1) Speed of the machine running the program
2) Language in which the program was written. For example, programs written in assembly language generally run faster than those written in C or C++, which in turn tend to run faster than those written in Java.
3) Efficiency of the compiler that created the program
4) The size of the input: processing 1000 records will take more time than processing 10 records.
5) Organization of the input: if the item we are searching for is at the top of the list, it will take less time to find it than if it is at the bottom.

The first three items in the list are problematic. We don't want to use an exact measurement of running time: To say that a particular algorithm written in C++ and running on a Pentium IV takes some number of milliseconds to run tells us nothing about the general time efficiency of the algorithm, because the measurement is specific to a given environment. The measurement will be of no

use to someone in a different environment. We need a general metric for the time efficiency of an algorithm; one that is independent of processor or language speeds, or compiler efficiency. The fourth item in the list is not environment-specific, but it is an important consideration. An algorithm will run slower if it must process more data but this decrease in speed is not because of the construction of the algorithm. It's simply because there is more work to do. As a result of this consideration, **we usually express the running time of an algorithm as a function of the size of the input. Thus, if the input size is n, we express the running time as T(n).** This way we take into account the input size but not as a defining element of the algorithm. Finally, the last item in the list requires us to consider another aspect of the input, which again is not part of the actual algorithm.

Algorithm Analysis: The Big-O Notation

Just as a problem is analyzed before writing the algorithm and the computer program, after an algorithm is designed it should also be analyzed. Usually, there are various ways to design a particular algorithm. Certain algorithms take very little computer time to execute, whereas others take a considerable amount of time.
Let us consider the following problem. The holiday season is approaching and a gift shop is expecting sales to be double or even triple the regular amount. They have hired extra delivery people to deliver the packages on time. The company calculates the shortest distance from the shop to a particular destination and hands the route to the driver. Suppose that 50 packages are to be delivered to 50 different houses. The shop, while making the route, finds that the 50 houses are one mile apart and are in the same area. (See Figure 1-1, in which each dot represents a house and the distance between houses is 1 mile.)

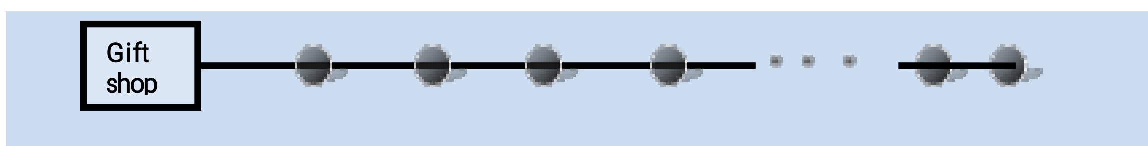


FIGURE 1-1 Gift shop and each dot representing a house

To deliver 50 packages to their destinations, one of the drivers picks up all 50 packages, Drives one mile to the first house and delivers the first package. Then he drives another mile and delivers the second package, drives another mile and delivers the third package, and so on.

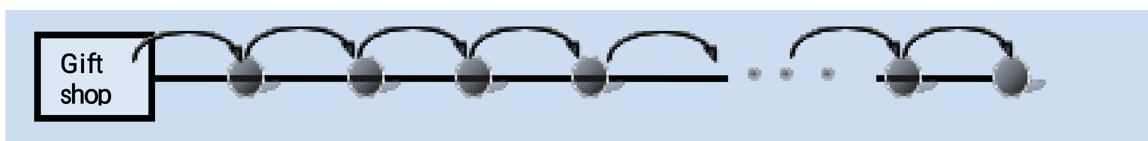


Figure 1-2 illustrates this delivery scheme.

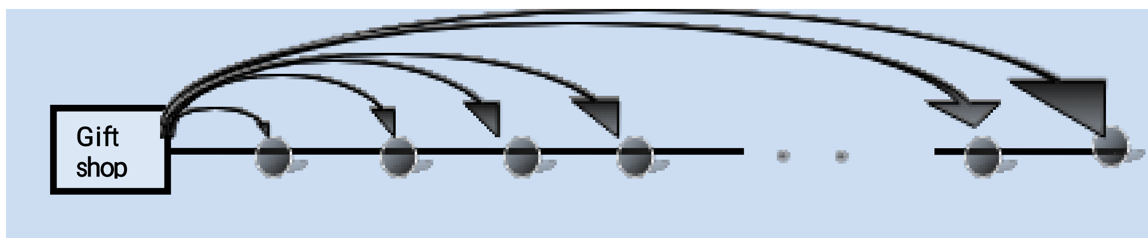
It now follows that using this scheme, the distance driven by the driver to deliver the packages is: 1 + 1+ 1+ … + 1 =50 miles

Therefore, the total distance traveled by the driver to deliver the packages and then getting back to the shop is:

50 + 50 = 100 miles

Another driver has a similar route to deliver another set of 50 packages. The driver looks at the route and delivers the packages as follows: The driver picks up the first package, drives one mile to the first house, delivers the package, and then comes back to the shop. Next, the driver picks up the second package, drives 2 miles, delivers the second package, and

then returns to the shop. The driver then picks up the third package, drives 3 miles, delivers the package, and comes back to the shop. Figure 1-3 illustrates this delivery scheme.



The driver delivers only one package at a time. After delivering a package, the driver comes back to the shop to pick up and deliver the second package. Using this scheme, the total distance traveled by this driver to deliver the packages and then getting back to the store is: $2 * (1 + 2 + 3 + \ldots + 50) = 2550$ miles

Now suppose that there are n packages to be delivered to n houses, and each house is one mile apart from each other, as shown in Figure 1-1. If the packages are delivered using the first scheme, the following equation gives the total distance traveled: $1 + 1 + \ldots + 1 + n = 2n$ … (1-1)

If the packages are delivered using the second method, the distance traveled is:

$2 * (1 + 2 + 3 + \ldots + n) = n^2 + n$ ... (1-2)

In Equation (1-1), we say that the distance traveled is a function of n. Let us consider

Equation (1-2). In this equation, for large values of n, we will find that the term consisting of n^2 will become the dominant term and the term containing n will be negligible. In this case, we say that the distance traveled is a function of n^2. Table 1-1 evaluates Equations (1-1) and (1-2) for certain values of n. (The table also shows the value of n^2.)

EXAMPELE: 1

Consider the following algorithm. (Assume that all variables are properly declared.)

```
cout << "Enter two numbers";                              //Line 1
cin >> num1 >> num2;                                      //Line 2
if (num1 >= num2)                                         //Line 3
max = num1;                                               //Line 4
else                                                      //Line 5
max = num2;                                               //Line 6
cout << "The maximum number is: " << max << endl;         //Line 7
```

T(n) = 8 = O(1)

Line 1 has one operation, <<; Line 2 has two operations; Line 3 has one operation, >=; Line 4 has one operation, =; Line 6 has one operation; and Line 7 has three operations. Either Line 4 or Line 6 executes. Therefore, the total number of operations executed in the preceding code is 1 + 2 + 1 + 1 + 3 = 8. In this algorithm, the number of operations executed is fixed.

EXAMPELE: 1

Problem of search
Input: array of integer numbers [1...n], integer number x
Output: return the index of x if it found otherwise return -1

```
1. read x
2. for (int i=0; i<n; i++)
3.       if(x==arr[i])
4.       k=i;
5.       if (k==0)
6.       cout<<-1;
7.    else
8.      cout<<k;
```

بطريقة أخرى:

```
While (arr [i] != x)
 i++;
 if (i>n)
 cout<< -1;
else
 cout<< i;
```

EXAMPELE: 1

```
For( int i=0; i<n; i++)          n+1
 For( j=0; j<n ; j++)            n(n+1)
  Cout<< array[i][j]             n^2
```

$T(n) = 2n^2 + 2n + 1 = O(n^2)$

EXAMPELE: 1

Array Re ordering:

```
For( i=0; i<n-1 ; i++)           n
 For( j=I+1; j<n; j++)           n (n+1) /2  -1
  Cout<< array[i]                n (n-1) /2  -2
```

$T(n) = n+n(n+1)/2 -1 + n(n-1)/2 \quad -2$

$= n+n^2/2+n/2 +n^2/2+n/2 -3$

$= n+n^2 +n-3$

$= n^2+2n-3 = O(n^2)$

```
For (i=0; i<n; i++ )             n+1
   For (j=i+1; j<n; j++)         n(n+1)/2
   Statement;                    n(n+1)/2 -1
```

$T(n)=n+1+n(n+1)/2 + n(n+1)/2 -1$

$=n+n^2/2 + n/2 + n^2/2 + n/2$

$=n+n^2 +n$

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

When n=4

i	j	statement
0	1 2 3 4	3
1	2 3 4	2
2	3 4	1
3	4	
4		

```
For (I=1; I<n; )          log n+1
 {
   Cout<<I;               log n
   I = I *2;              log n
}
```

$T(n) = \log_2 n+1$

```
For (i=n;  i>=1;  i=i/2)        log n+1
S=s+I                           log n
```

$T(n) = \log n+1 = O(\log n)$

```
For(i=1;  i< n;  i++)             n
   For( j=1;  j<=n;  j*=٢)        n (log n+1)
      Sum =sum+ x                 n log n
```

$T(n)=n + n (\log n+1) + n \log n$

$= n+ n\log n +n + n\log n =2n + 2n\log n$

When n=10

```
For(i=1; i<n; i=i+3)          (n-1)/3 +1
   Statement ;                (n-1)/3
```

```
For(i=1; i<17; i=i+4)         (17-1)/4 +1
  Statement;                  (17-1)/4
```

TABLE 1-1 Various values of n, 2n, n^2, and $n^2 + n$

N	2n	n^2	$n^2 + n$
1	2	1	2
10	20	100	110
100	200	10,000	10,100
1000	2000	1,000,000	1,001,000
10,000	20,000	100,0000,0000	100,010,000

While analyzing a particular algorithm, **we usually count the number of operations Performed by the algorithm.** We focus on the number of operations, not on the actual Computer time to execute the algorithm. This is because a particular algorithm can be implemented on a variety of computers and the speed of the computer can affect the execution time. However, **the number of operations performed by the algorithm would be the same on each computer**. Let us consider the following examples.

EXAMPLE: 1-2

Consider the following algorithm:

```
cout << "Enter positive integers ending with -1" << endl;          //Line1
count = 0;                                                          //Line2
sum = 0;                                                            //Lin3
cin >> num;                                                         //Lin4
while (num != -1)                                                   //Lin5
{
sum = sum + num;                                                    //Line6
count++;                                                            //Line7
cin >> num;                                                         //Line8
}
cout << "The sum of the numbers is: " << sum << endl;               //Line9
if (count != 0)                                                     //Line10
average = sum / count;                                              //Line11
else                                                                //Line 12
average = 0;                                                        //Line 13
cout << "The average is: " << average << endl;                      //Line 14
```

This algorithm has 5` operations (Lines 1 through 4) before the while loop. Similarly, there are nine or eight operations after the while loop, depending on whether Line 11 or Line 13 executes. Line 5 has one operation, and four operations within the while loop (Lines 6 through 8).
Thus, Lines 5 through 8 have five operations. If the while loop executes 10 times, these five operations execute 10 times. One extra operation is also executed at Line 5 to terminate the loop. Therefore, the number of operations executed is 51 from Lines 5 through 8.
If the while loop executes 10 times, the total number of operations executed is:

10 * 5 + 1 + 5 + 8
that is: 10 * 5 +14

We can generalize it to the case when the while loop executes n times. If the while loop
executes n times, the number of operations executed is:

5n + 15 or 5n + 14

In these expressions, for very large values of n, the term 5n becomes the dominating term and the terms 15 and 14 become negligible.
Usually, in an algorithm, certain operations are dominant. For example, in the preceding algorithm, to add numbers, the dominant operation is in Line 6. Similarly, in a search algorithm, because the search item is compared with the

items in the list, the dominant operations would be comparison, that is, the relational operation. Therefore, in the case of a search algorithm, we count the number of comparisons. For another example, suppose that we write a program to multiply matrices. The multiplication of matrices involves addition and multiplication. Because multiplication takes more computer time to execute, to analyze a matrix multiplication algorithm, we count the number of multiplications.
In addition to developing algorithms, we also provide a reasonable analysis of each algorithm. If there are various algorithms to accomplish a particular task, the algorithm analysis allows the programmer to choose between various options.
Suppose that an algorithm performs f(n) basic operations to accomplish a task, where n is the size of the problem. Suppose that you want to determine whether an item is in a list. Moreover, suppose that the size of the list is n. To determine whether the item is in the list, there are various algorithms, as you will see in Chapter 9. However, the basic method is to compare the item with the items in the list. Therefore, the performance of the algorithm depends on the number of comparisons.
Thus, in the case of a search, n is the size of the list and f(n) becomes the count function, that is, f(n) gives the number of comparisons done by the search algorithm. Suppose that, on a particular computer, it takes c units of computer time to execute one operation. **Thus, the computer time it would take to execute f(n) operations is *c f(n)*. Clearly, the constant c depends on the speed of the computer** and, therefore, varies from computer to computer. However, f(n), the number of basic operations, is the same on each computer. If we know how the function f(n) grows as the size of the problem grows, we can determine the efficiency of the algorithm. Consider Table 1-2.

TABLE 1-2 Growth rates of various functions

N	Log_2n	$n \log_2 n$	n^2	2^n
1	0	0	1	2
2	1	2	4	4
4	2	8	16	16
8	3	24	64	256
16	4	64	256	65,536
32	5	160	1024	4,294,967,296

Table 1-2 shows how certain functions grow as the parameter n, that is, the problem size, grows. Suppose that the problem size is doubled. From Table 1-2, it follows that if the number of basic operations is a function of $f(n) = n^2$, the number of basic operations is **quadrupled**. If the number of basic operations is a function

of $f(n) = 2^n$, the number of basic operations is **squared**. However, if the number of operations is a function of $f(n) = \log_2 n$, the change in the number of basic operations is **insignificant**

The remainder of this section develops a notation that shows how a function f(n) grows as n increases without bound. **That is, we develop a notation that is useful in describing the behavior of the algorithm, which gives us the most useful information about the algorithm.**
First, we define the term asymptotic. Let **f** be a function of n. By the term asymptotic, we mean the study of the function f as n becomes larger and larger without bound.
Consider the functions $g(n) = n^2$ and $f(n) = n^2 + 4n + 20$. Clearly, the function g does not contain any linear term, that is, the coefficient of n in g is zero. Consider Table 1-4.

TABLE 1-4 Growth rate of n2 and n2 + 4n + 20

n	$g(n) = n^2$	$f(n)=n^2+4n+20$
10	100	160
50	2500	2720
100	10,000	10,420
1000	1,000,000	1,004,020
10,000	100,000,000	100,040,020

Clearly, as n becomes larger and larger the term 4n + 20 in f(n) becomes insignificant, and the term n2 becomes the dominant term. For large values of n, we can predict the
behavior of f(n) by looking at the behavior of g(n). In algorithm analysis, if the complexity of a function can be described by the complexity of a quadratic function without the linear term, we say that the function is of $O(n^2)$, called Big-O of n^2.
Let f and g be real-valued functions. Assume that f and g are nonnegative, that is, for all real numbers n, f(n) >= 0 and g(n) >= 0.
Definition: We say that **f(n) is Big-O of g(n)**, written **f(n) = O(g(n))**, if there exists positive constants c and n0 such that f(n) <=cg(n) for all n >= n0.

Example: 1-5

In the following, f(n) is a nonnegative real-valued function.

Function	Big-O
$f(n) = an + b$, where a and b are real numbers and a is nonzero.	$f(n) = O(n)$
$f(n) = n^2 + 5n + 1$	$f(n) = O(n^2)$
$f(n) = 4n^6 + 3n3 + 1$	$f(n) = O(n^6)$
$f(n) = 10n^7 + 23$	$f(n) = O(n^7)$
$f(n) = 6n^{15}$	$f(n) = O(n^{15})$

Example: 1-6

Suppose that $f(n) = 2 \log2n + a$, where a is a real number. It can be shown that $f(n)=O(\log2n)$.

Example: 1-7

Consider the following code, where m and n are int variables and their values are nonnegative:

```
for (int i = 0; i < m; i++)                //Line 1
for (int j = 0; j < n; j++)                //Line 2
cout << i * j << endl;                     //Line 3
```

This code contains nested for loops. The outer for loop, at Line 1, executes m times. For each iteration of the outer loop, the inner loop, at Line 2, executes n times. For each iteration of the inner loop, the output statement in Line 3 executes. It follows that the total number of iterations of the nested for loop is mn. So the number of times the statement in Line 3 executes is mn. Therefore, this algorithm is O(mn). **Note that if m = n, then this algorithm is $O(n^2)$.**

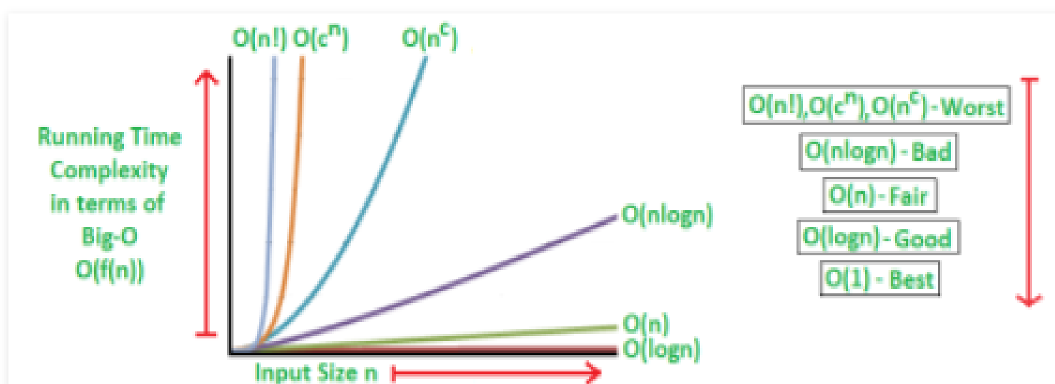
Table 1-5 shows some common Big-O functions that appear in the algorithm analysis.
Let $f(n) = O(g(n))$ where n is the problem size.

TABLE 1-5 Some Big-O functions that appear in algorithm analysis

Function g (n)	Growth rate of f (n)
g (n) = 1	The growth rate is constant and so does not depend on n, the size of the problem

g (n) = $\log_2 n$	The growth rate is a function of log2n. Because a logarithm function grows slowly, the growth rate of the function f is also slow.
g (n) = n	The growth rate is **linear.** The growth rate of f is directly proportional to the size of the problem.
g (n) = $n\log_2 n$	The growth rate is faster than the linear algorithm.
g (n) = n^2	The growth rate of such functions increases quickly with the size of the problem. The growth rate is **quadrupled** when the problem size is doubled.
g (n) = 2^n	The growth rate is exponential. The growth rate is **squared** when the problem size is doubled.

Where, n is the input size and c is a positive constant.



Analyzing Some Simple Programs

General Rules:

1) All basic statements (assignments, reads, writes, conditional testing, library calls) run in constant time: O(1).

2) The time to execute a loop is the sum, over all times around the loop, of the time to execute all the statements in the loop, plus the time to evaluate the condition for termination. Evaluation of basic termination conditions is O(1) in each iteration of the loop.

3) The complexity of an algorithm is determined by the complexity of the most frequently executed statements. If one set of statements have a running time

of $O(n^3)$ and the rest are O(n), then the complexity of the algorithm is $O(n^3)$. This is a result of the Summation Rule.

Example 1

Compute the big-Oh running time of the following C++ code segment:

```
for (i = 2; i < n; i++)
        sum += i
```

The number of iterations of a for loop is equal to the top index of the loop minus the bottom index, plus one more instruction to account for the final conditional test. Note: if the for loop terminating condition is i <= n, rather than i < n, then the number of times the conditional test is performed is:

((top_index) – bottom_index) + 1)

In this case, we have n - 2 + 1 = n - 1. The assignment in the loop is executed n - 2 times. So, we have (n - 1) + (n - 2) = (2n - 3) instructions executed = O(n).

Complexity problems may ask for "number of instructions executed" which means you need to provide an equation in terms of n of the precise number of instructions executed. Or, we may just ask for the complexity in which case you need only provide a big-Oh (or big-Theta) expression.

Example 2

Consider the sorting algorithm shown below. Find the number of instructions executed and the complexity of this algorithm.

.

```
1)    for (i = 1; i < n; i++)
2)    {      SmallPos = i;
3)           Smallest = Array[SmallPos];
4)           for (j = i+1; j <= n; j++)          n(n+1)/2 -1
5)                 if (Array[j] < Smallest)
                    {
6)                        SmallPos = j;
7)                        Smallest = Array[SmallPos]
                    }
8)           Array[SmallPos] = Array[i];
9)           Array[i] = Smallest;   }
```

assume n=4

I	j	N0.of times
1	2	
	3	
	4	
	5	3
2	3	
	4	
	5	2
3	4	
	5	1
4		

Statement 1 is executed n times (n - 1 + 1); statements 2, 3, 8, and 9 are executed (n – 1) times each, once on each pass through the outer loop. On the first pass through this loop with i = 1, statement 4 is executed **n** times; statement 5 is executed n - 1 times, and assuming a worst case where the elements of the array are in descending order, statements 6 and 7 are executed n - 1 times.

On the second pass through the outer loop with i = 2, statement 4 is executed n - 1 times and statements 5, 6, and 7 are executed n - 2 times, etc. Thus, statement 4 is executed (n) + (n-1) +... + 2 times and statements 5, 6, and 7 are executed (n-1) + (n-2) + ... + 2 + 1 times. The first sum is equal to n(n+1)/2 –1, and the second is equal to n(n-1)/2.

Thus, the total computing time is:

$$
\begin{aligned}
T(n) &= (n) + 4(n-1) + n(n+1)/2 - 1 + 3[n(n-1) / 2] \\
&= n + 4n - 4 + (n^2 + n)/2 - 1 + (3n^2 - 3n) / 2 \\
&= 5n - 5 + (4n^2 - 2n) / 2 \\
&= 5n - 5 + 2n^2 - n \\
&= 2n^2 + 4n - 5 \\
&= O(n^2)
\end{aligned}
$$

Example 3

The following program segment initializes a two-dimensional array A (which has n rows and n columns) to be an n x n identity matrix – that is, a matrix with 1's on the diagonal and 0's everywhere else. More formally, if A is an n x n ident ty matrix, then:

A x M= M x A = M, for any n x n matrix M.

What is the complexity of this C++ code?

```
1)  cin >> n;                       // Same as: n = GetInteger();
2)  for (i = 1; i <= n; i ++)
3)          for (j = 1; j <= n; j ++)
4)                  A[i][j] = 0;
5)  for (i = 1; i <= n; i ++)
6)          A[i][i] = 1;
```

A program such as this can be analyzed in parts, and then we can use the summation rule to find a total running time for the entire program. Line 1 takes O(1) time. The instructions in lines 5 and 6 are executed O(n) times. The instructions in lines 3 and 4 are executed n times every time we execute this loop. The outer loop of line 2 is executed n times, yielding a time complexity of $O(n^2)$, due to the inner loop begin executed O(n) times. Thus the running time of the

segment is O(1) + $O(n^2)$ + O(n). We apply the summation rule to conclude that the running time of the segment is $O(n^2)$.

Example 4

Consider the following two examples of nested loops intended to sum each of the rows of an N × N *matrix*, storing the row sums in the one-dimensional vector *rows* and the overall total in Grand Total:

- *Program - 1:*

```
GrandTotal = 0;
for (int k = 0 ; k < n-1 ; ++k )
{
    rows[ k ] = 0;
    for ( int j = 0 ; j < n-1 ; ++j )
    {
            rows[ k ] = rows[ k ] + matrix[ k ][ j ];
            GrandTotal = GrandTotal + matrix[ k ][ j ];
    }
}
```

- *Program - 2:* GrandTotal = 0;

```
for (int k = 0 ; k < n-1 ; ++k )
{
    rows[ k ] = 0;

    for ( int j = 0 ; j < n-1 ; ++j )
            rows[ k ] = rows[ k ] + matrix[ k ][ j ];

    GrandTotal = GrandTotal + rows[ k ];
}
```

1.2 Algorithm Design Techniques:

1. Incremental Technique:
   Examples: Bubble sort, Insertion sort, Selection sort.

2. Divide and Conquer Technique:
   Examples: Merge sort, Quick sort, Binary Search.

3. Dynamic Technique:

Examples: Fiboncis numbers, Matrix chain multiplication , Knapsack problem.

4. Greedy Technique:
   Examples: Shortest path problem, Kruskal's algorithm, Prim's algorithm.

5. Graph Technique:
   Examples: Depth-first search, Breadth-first search.

We will start with some simple algorithms related to searching and sorting, then we will present the basic concepts used in the design and analysis of algorithms.

1.3 Linear Search Problem:

Consider the problem of determining whether a given element x is in A. This problem can be rephrased as follows: Find an index j; $0 \leq j \leq n-1$, such that $x = A[j]$ if x is in A, and $j = -1$ otherwise. A straightforward approach is to scan the entries in *A* and compare each entry with *x*. If after *j* comparisons, $0 \leq j \leq n-1$, the search is *successful*, i.e., $x = A[j]$, *j* is returned; otherwise a value of -1 is returned indicating an *unsuccessful* search. This method is referred to as *sequential search*. It is also called *linear search*, as the maximum number of element comparisons grows linearly with the size of the sequence. The algorithm is shown as Algorithm LINEARSEARCH:

```
Algorithm: LINEARSEARCH
Input: An array A[1..n] of n elements and an element x.
Output: j if x = A[j], 0 ≤ j ≤ n-1, and -1 otherwise.

    1. j ← 0
    2. while (j < n-1) and (x ≠ A[j])
    3.       j ← j + 1
    4. end while
    5. if x = A[j] then return j else return -1
```

➢ # of Comparisons: between 1 and *n*

Remarks

- *Scanning all entries of A is inevitable if no more information about the ordering of the elements in A is given.*
- *If we are given that the elements in A are sorted, say in non-decreasing order, then one can find much more efficient algorithms.*

1.4 Binary Search Problem:

Consider the problem of determining whether a given element x is in a sorted A. This problem can be rephrased as follows: Find an index j; 0 ≤ j ≤ n-1, such that x = A[j] if x is in A, and j = -1 otherwise, where A[j] <= A [j+1] <=..... <= A [n-1].

Example 1.1 Consider searching the array

A[0..13] =

1	4	5	7	8	9	10	12	15	22	23	27	32	35
0	1	2	3	4	5	6	7	8	9	10	11	12	13

In this instance, we want to search for element x = 22. First, we compare x with the middle element A [(0 + 13)/2] = A[6] = 10. Since 22 > A[6], and since it is known that A[i] <= A[i + 1], 0 <= i < 13, x cannot be in A[0..6], and therefore this portion of the array can be discarded. So, we are left with the subarray

A[7..13] =

12	15	22	23	27	32	35
7	8	9	10	11	12	13

Next, we compare x with the middle of the remaining elements A [(7 + 13)/2] = A[10] = 23. Since 22 < A [10], and since A[i] <= A [i+1]. 11 <= i < 13, x cannot be in A [10...13], and therefore this portion of the array can also be discarded. Thus, the remaining portion of the array to be searched is now reduced to

A[7..9] =

12	15	22
7	8	9

Repeating this procedure, we discard A[7..8], which leaves only one entry in the array to be searched, that is A[9] = 22. Finally, we find that x = A[9], and the search is successfully completed.

In general, let A[low..high] be a nonempty array of elements sorted in nondecreasing order. Let A[mid] be the middle element, and suppose that x > A[mid]. We observe that if x is in A, then it must be one of the elements A[mid + 1], A[mid + 2], . . . , A[high]. It follows that we only need to search for x in A[mid + 1..high].

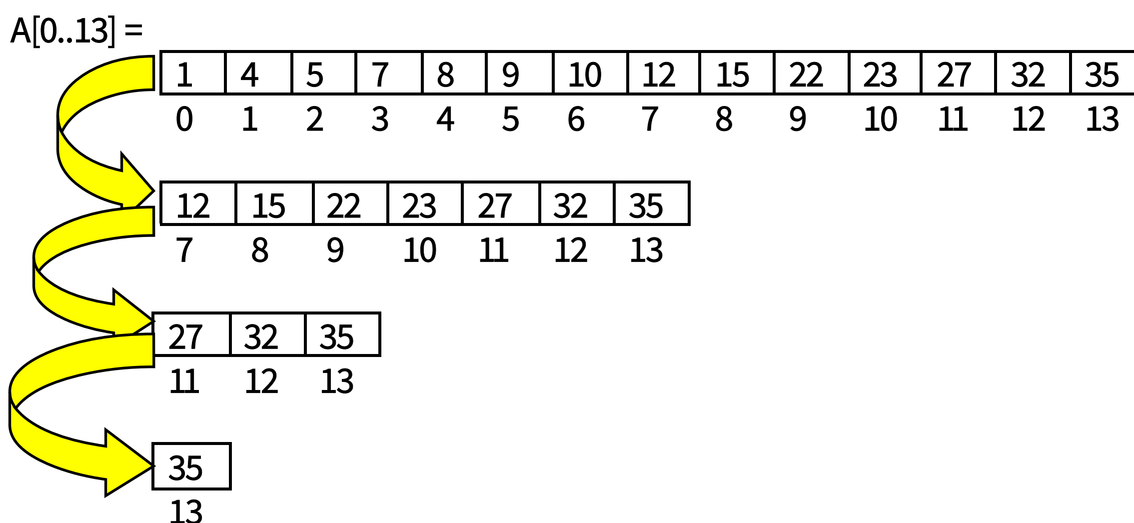
In other words, the entries in A[low..mid] are discarded in subsequent comparisons since, by assumption, A is sorted in non-decreasing order, which implies that x cannot be in this half of the array. Similarly, if x < A[mid], then we only need to search for x in A[low..mid - 1].

This results in an efficient strategy which, because of its repetitive halving, is referred to as binary search. Algorithm BINARYSEARCH gives a more formal description of this method:

```
Algorithm: BINARYSEARCH
Input: An array A[0..n-1] of n elements sorted
      in Non-decreasing order and an element x.
Output: j if x = A[j], 0 ≤ j ≤ n-1,
    and -1 otherwise.
    1. low ← 0; high ← n-1; j ← -1
    2. while (low ≤ high) and (j = -1)
    3.       mid ← [(low + high)/2]
    4.       if x = A[mid] then j ← mid
    5.       else if x < A[mid] then high ← mid - 1
    6.       else low ← mid + 1
    7. end while
    8. Return j
```

Example: Searching for x = 35 or any value greater than 35. The array is sorted in non-decreasing order.

A[0..13] =



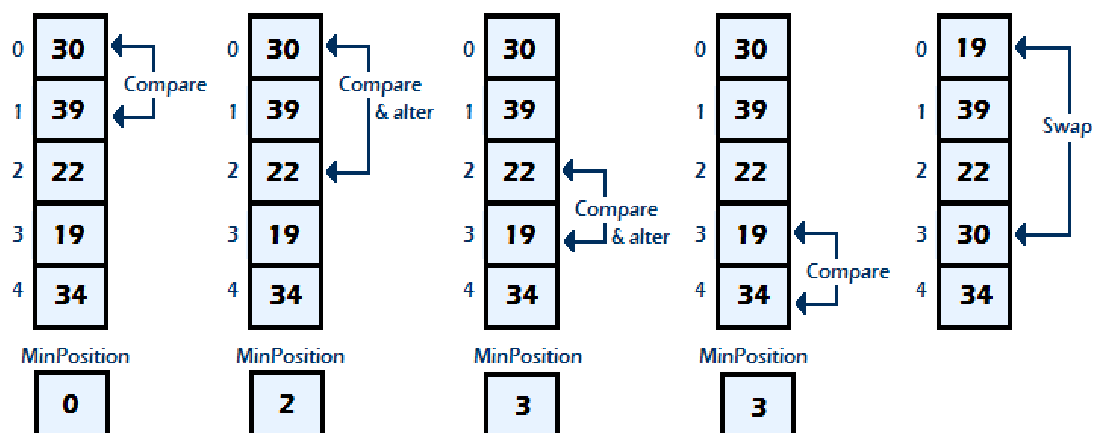
1.5 Selection Sort

Algorithm: SELECTIONSORT
Input: An array A[1..n] of n elements.
Output: A[1..n] sorted in non-decreasing order.

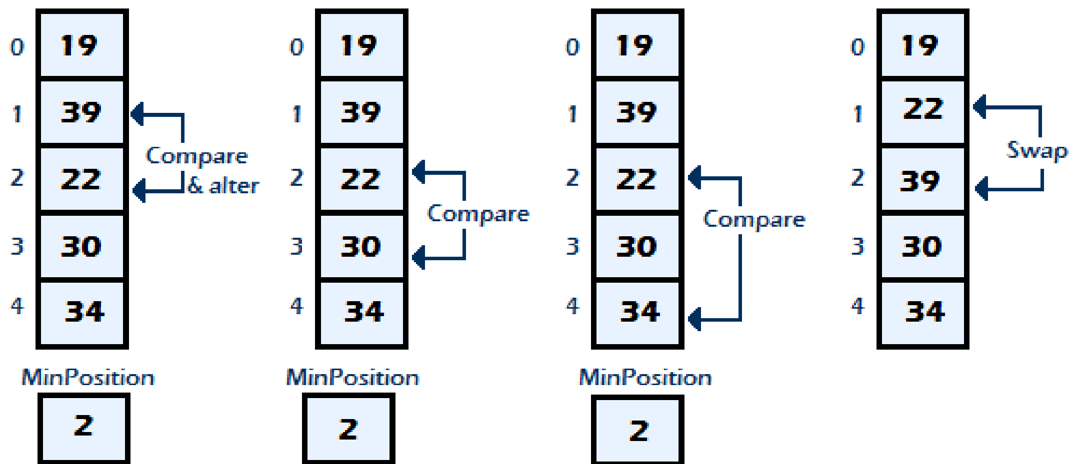
```
    1. for  i ← 0 to n - 2
    2.        k ← i
  //Find the ith smallest element.}
    3.        for j ← i + 1 to n-1
    4.         if A[j] < A[k] then k ← j
    5.        end for
    6.        if k ≠ i then interchange A[i] and A[k]
    7. end for
```

During pass i the smallest value between index i and the last entry in the array is interchanged with the entry at index i.

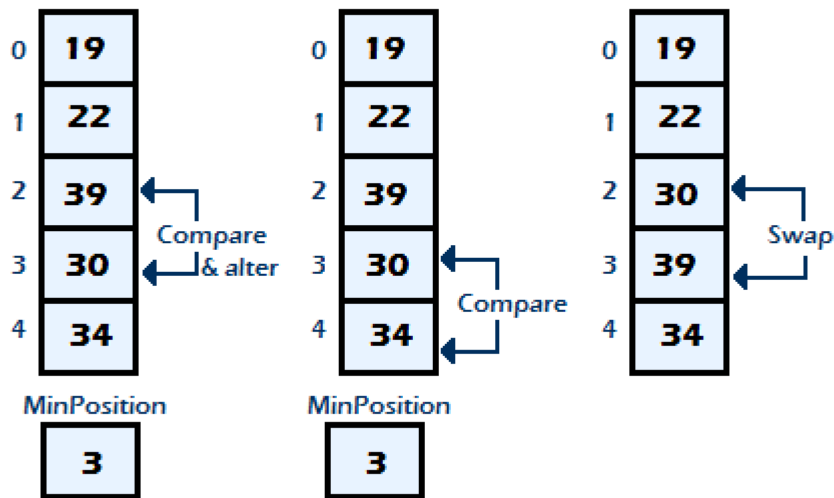
- *Example:*



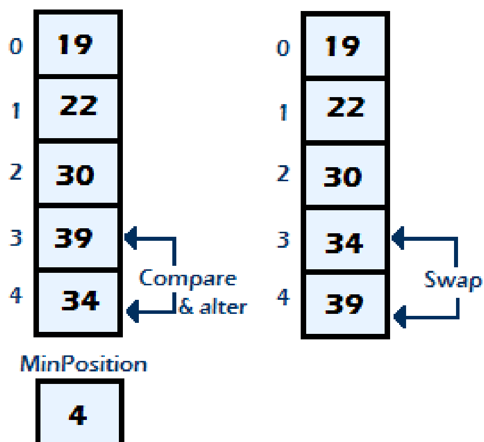
First Pass



Second Pass



Third Pass

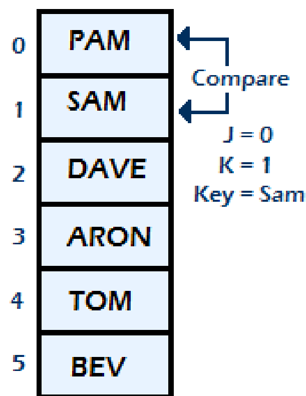


Fourth Pass

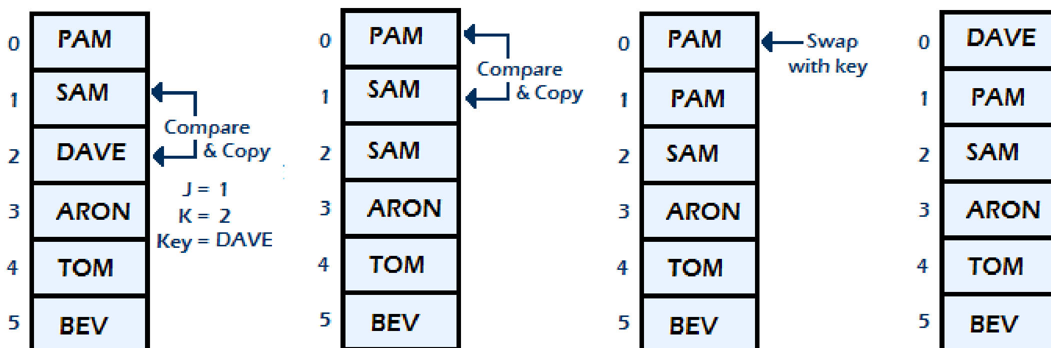
1.6 Insertion Sort

```
Algorithm: INSERTIONSORT
Input: An array A[0..n-1] of n elements.
Output: A[0..n-1] sorted in non-decreasing order.
    1. for  i ← 1 to n-2
    2.      x ← A[i]
    3.      j ← i - 1
    4.      while (j >-1) and (A[ j] > x)
    5.         A[ j + 1] ← A[ j]
    6.         j ← j - 1
    7.      end while
    8.      A[j + 1] ← x
    9. end for
```

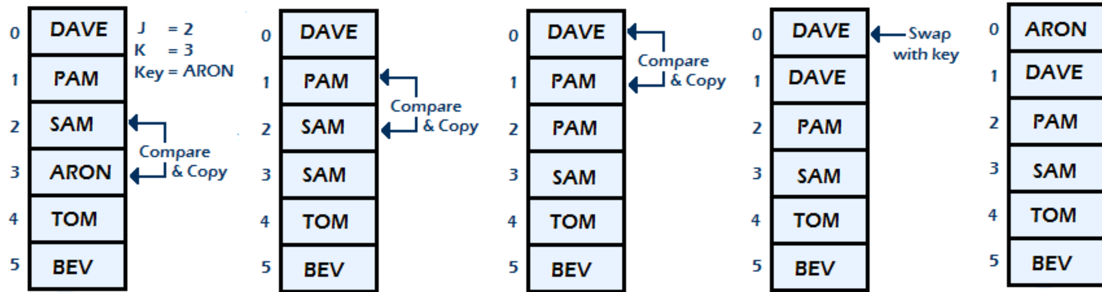
➢ ***Example:***



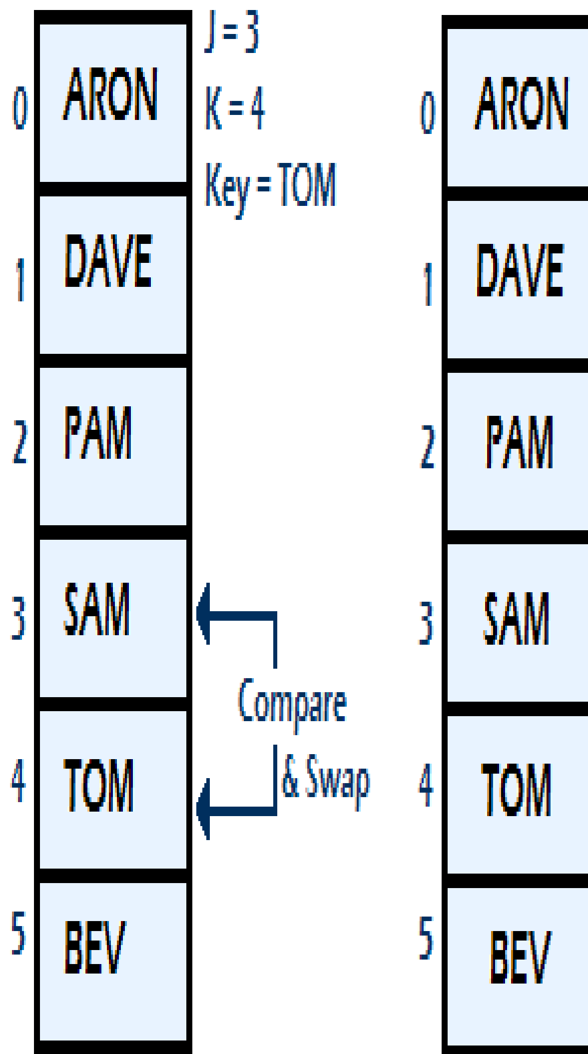
First Pass



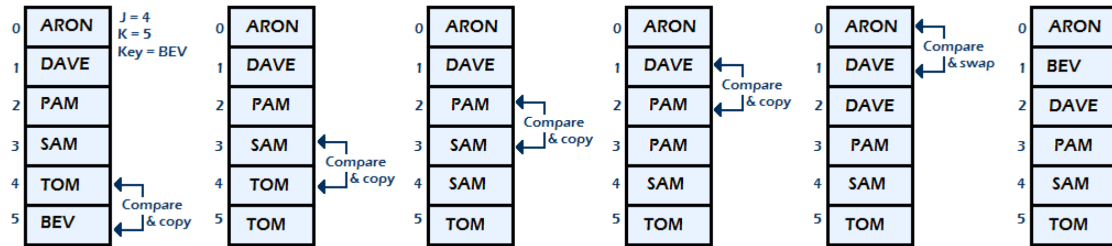
Second Pass



Third Pass



Fourth Pass



Fifth Pass

- Then A[i] is inserted in its proper position.