1.1 The Role of Algorithms in Computing

SPPU : April-18, 19, March-20, Marks 5

1.1.1 What are Algorithms ?

In this section we will first understand *"What is algorithm?"* and *"When it is required?"*

Definition of Algorithm : The algorithm is defined as a collection of unambiguous instructions occurring in some specific sequence and such an algorithm should produce output for given set of input in finite amount of time.

This definition of algorithm is represented in Fig. 1.1.1.

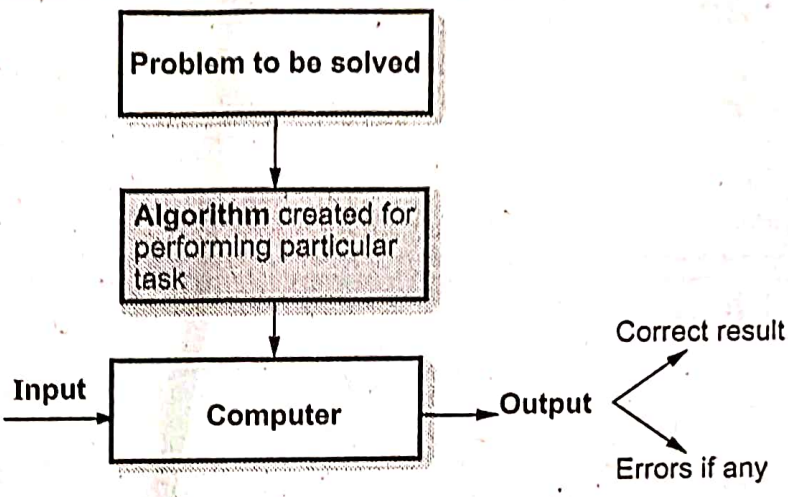


Fig. 1.1.1 Notion of algorithm

After understanding the problem statement we have to create an algorithm carefully for the given problem. The algorithm is then converted into some programming language and then given to some computing device (computer). The computer then executes this algorithm which is actually submitted in the form of source program. During the process of execution it requires certain set of input. With the help of algorithm (in the form of program) and input set, the result is produced as an output. If the given input is invalid then it should raise appropriate error message ; otherwise correct result will be produced as an output.

1.1.2 Properties of Algorithm

Simply writing the sequence of instructions as an algorithm is not sufficient to accomplish certain task. It is necessary to have following properties associated with an algorithm :

1. **Non-ambiguity :** Each step in an algorithm should be non-ambiguous. That means each instruction should be clear and precise. The instruction in an algorithm should not denote any conflicting meaning. This property also indicate the *effectiveness* of algorithm.
2. **Range of input :** The range of input should be specified. This is because normally the algorithm is input driven and if the range of the input is not been specified then algorithm can go in an infinite state.

3. **Multiplicity :** The same algorithm can be represented in several different ways. That means we can write in simple English the sequence of instructions or we can write it in the form of pseudo code. Similarly for solving the same problem we can write several different algorithms. For instance : for searching a number from the given list we can use sequential search or a binary search method. Here *"searching"* is a task and use of either a *"sequential search method"* or *"binary search method"* is an algorithm.
4. **Speed :** The algorithms are written using some specific ideas (which is popularly known as logic of algorithm). But such algorithms should be efficient and should produce the output with fast speed.
5. **Finiteness :** The algorithm should be finite. That means after performing required operations it should terminate.

1.1.3 Algorithms as Technology

- Algorithms are used as a tool to utilize reasonable amount of execution time and memory space.
- Different algorithms can be used to same problem. Each algorithm may take different amount of time.
- For example - For sorting the elements using bubble sort algorithm requires more time than that of merge sort or quick sort algorithm.
- Execution of algorithm depends upon two factors -
  1. Selection of **efficient algorithm** and
  2. **Fast hardware** upon which the algorithm is to be executed.
- For example if there are two algorithms **insertion sort**(less efficient sorting algorithm) and **merge sort**(more efficient sorting algorithm) and there are two computers - computer A and computer B. The computer B is more efficient than computer A. If the merge sort is executed on computer B then this execution is efficient than executing insertion sort on computer B. Thus hardware technology plays an important role in algorithm execution.
- Algorithms are core or fundamental factor for following technologies -
  - Advanced computer architectures
  - Object oriented systems
  - Fast networking applications that can be wired or wireless
  - Applications that use Graphical User Interfaces(GUI)
  - Integrated web technologies

- Ever **increasing storage space of computers** also affect the processing of algorithm. That means the larger problems, or the problems handling large amount of data can be executed efficiently on increased storage space of the computer.

Review Questions

1. *Explain characteristics of good algorithm ? List out the problems solved by algorithm.* **SPPU : April-18, Marks 5**
2. *Write a short note on algorithm as a technology with example.* **SPPU : April-19, Marks 5**
3. *What are algorithms ? Explain algorithm as technology with example.* **SPPU : March-20, Marks 5**

1.2 Evolution of Algorithms

SPPU : April-18, Marks 5

- **Introduction to Concept of Algorithm :** The first algorithm is developed by famous Indian sanskrit grammarian Panini in the text Maheshwar Sutra. This text consists of compact rules and algorithms for Phonetics and syntax of Sanskrit language. He proposed formal language theory almost parallel to modern theory and gave concepts parallel to modern mathematical functions.
- **Structured Programming :** In 1936 Alan Turing introduced the concept of **procedures** due to which the approach for systematic coding is developed. This led to concept of **structured programming.**
- **Correctness of Algorithm :** The concept of proving the **correctness** of **algorithm** is developed.
- **Analysis of Algorithm :** More and more efficient algorithms are then getting developed for solving the problems. Prof. Donald Knuth introduced the concept of **analysis of algorithms.**
- **Complexity Theory :** Later on the concept of **complexity theory** is introduced in which the problems were divided into two classes - **tractable and non-tractable.** The tractable problems is a class of problems which are solvable in reasonable amount of time while non-tractable problems cannot be solved in reasonable amount of time. The concept of **lower bound on running time** is emerged while solving the problems using complexity theory.

Parallel Algorithmic Development : Later on it is found that finding the proof of correctness is very difficult because some proofs can be lengthy or there is no guarantee that the proof will be correct. This led to two developments.

i) **Automatic proof assistants** - Programs which try to generate the detailed proof for a given algorithm with some assistance from a human analyst,

ii) **Parallel development :** The idea that an algorithm should be developed in parallel, along with its proof of correctness.

- **Formal Method of Specification :** A new format popularly known as formal methods of specification is introduced to define the proof of correctness.
- **Time Space-trade off :** While searching for effective algorithms researchers found the concept of time-space trade-off. The time space tradeoff is a concept which states that for making algorithm time efficient, the space is utilized more or for making the algorithm space efficient, the space is consumed a lot by the algorithm.
- **Approximation Algorithm :** There are lot of problems which are non-tractable. In such a case, the approximation algorithms are developed to solve such problems. Approximation algorithms are those algorithms which gives the expected result within a time limit which is a small multiple of the theoretical limit.
- **Randomness :** There are some algorithms that make use of random number generations for computing the result. For example genetic algorithms makes use of random number generation.

Review Question

1. *Write short note on evolution of algorithm.* **SPPU : April-18, Marks 5**

1.3 Design of Algorithm

SPPU : Dec.-09, Marks 6

Algorithm is basically a sequence of instructions written in simple English language. The algorithm is broadly divided into two sections as shown in Fig. 1.3.1.

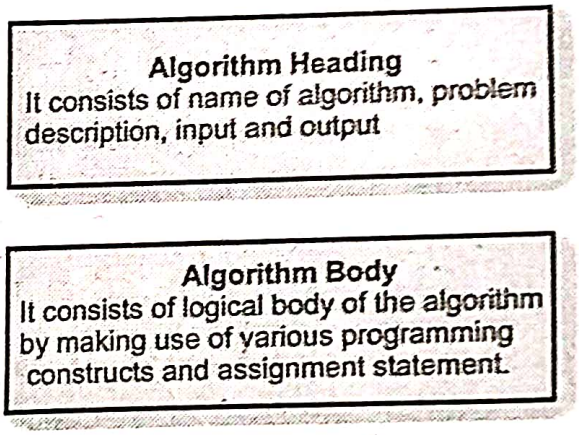
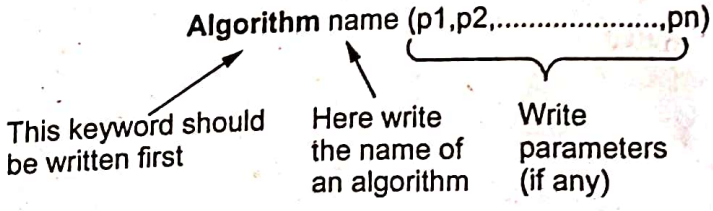


Fig. 1.3.1 Structure of algorithm

Let us understand some rules for writing the algorithm.

1. Algorithm is a procedure consisting of heading and body. The heading consists of keyword **Algorithm** and name of the algorithm and parameter list. The syntax is,



2. Then in the heading section we should write following things :

```
//Problem Description :
//Input :
//Output :
```

3. Then body of an algorithm is written, in which various programming constructs like if, for, while or some assignment statements may be written.
4. The compound statements should be enclosed within { and } brackets.
5. Single line comments are written using // as beginning of comment.
6. The **identifier** should begin by letter and not by digit. An identifier can be a combination of alphanumeric string.

   It is not necessary to write data types explicitly for identifiers. It will be represented by the context itself. Basic data types used are integer, float, char, Boolean and so on. The pointer type is also used to point memory location. The compound data type such as structure or record can also be used.
7. Using assignment operator ← an assignment statement can be given.

   For instance : Variable ← expression
8. There are other types of operators such as Boolean operators such as true or false. Logical operators such as **and, or, not**. And relational operators such as <, <=, >, >=, =, ≠
9. The array indices are stored with in square brackets '[' ']'. The index of array usually start at zero. The multidimensional arrays can also be used in algorithm.
10. The inputting and outputting can be done using **read** and **write**.

    For example :

```
write("This message will be displayed on console");
read(val);
```

11. The conditional statements such as if-then or if-then-else are written in following form :

```
if (condition) then statement
if (condition) then statement else statement
```

If the **if-then** statement is of compound type then { and } should be used for enclosing block.

12. **while** statement can be written as :

```
while (condition) do
{
        statement 1
        statement 2
            :
            :
        statement n
}
```

While the condition is true the block enclosed with { } gets executed otherwise statement after } will be executed.

13. The general form for writing for loop is :

```
for variable ← value1 to valuen do
{
        statement 1
        statement 2
            :
            :
        statement n
}
```

Here $value_1$ is initialization condition and $value_n$ is a terminating condition. The **step** indicates the increments or decrements in $value_1$ for executing the **for** loop.

Sometime a keyword **step** is used to denote increament or decreament the value of variable for example

```
for i←1 to n step 1
{
    Write (i)
}
```

Here variable i is incremented by 1 at each iteration

14. The **repeat until** statement can be written as :

```
repeat
        statement 1
        statement 2
            :
            :
        statement n
until (condition)
```

15. The **break** statement is used to exit from inner loop. The **return** statement is used to return control from one point to another. Generally used while exiting from function.

Note that statements in an algorithm executes in sequential order i.e. in the same order as they appear-one after the other.

Some Examples

Example 1.3.1 *Write an algorithm to count the sum of n numbers.*

Solution :

```
Algorithm sum (1, n)
//Problem Description: This algorithm is for finding the
//sum of given n numbers
//Input: 1 to n numbers
```

```
//Output: The sum of n numbers
 result ← 0
 for i ← 1 to n do i ← i+1
     result ← result+i
 return result
```

Example 1.3.2 *Write an algorithm to check whether given number is even or odd.*

Solution :

```
Algorithm eventest (val)
//Problem Description: This algorithm test whether given
//number is even or odd
//Input: the number to be tested i.e. val
//Output: Appropriate messages indicating even or oddness
if (val%2=0) then
            write ("Given number is even")
else
            write("Given number is odd")
```

Example 1.3.3 *Write an algorithm for sorting the elements.*

Solution :

```
Algorithm sort (a,n)
//Problem Description: sorting the elements in ascending order
//Input:An array a in which the elements are stored and n
//is total number of elements in the array
//Output: The sorted array
for i ← 1 to n do
for j ← i+1 to n-1 do
{
if(a[i]>a[j]) then
{
            temp ← a[i]
            a[i] ← a[j]
            a[j] ← temp
 }
}
write ("List is sorted")
```

Example 1.3.4 *Write an algorithm to find factorial of n number.*

Solution :

```
Algorithm fact (n)
//Problem Description: This algorithm finds the factorial
//of given number n
//Input: The number n of which the factorial is to be
//calculated.
```

```
//Output:factorial value of given n number.
if(n ← 1) then
        return 1
else
        return n*fact(n-1)
```

Example 1.3.5 *Write an algorithm to perform multiplication of two matrices.*

Solution :

```
Algorithm Mul(A,B,n)
//Problem Description:This algorithm is for computing
//multiplication of two matrices
//Input:The two matrices A,B and order of them as n
//Output:The multiplication result will be in matrix C
for i ← 1 to n do
 for j ← 1 to n do
  C[i,j] ← 0
  for k ← 1 to n do
      C[i,j] ← C[i,j]+A[i,k]B[k,j]
```

Example 1.3.6 *Write an algorithm to compute GCD of two given integers.*

Solution :

```
Algorithm GCD_Euclid(a,b)
// Problem Description: This algorithm computes the GCD of
//two numbers a and b using Euclid's method.
//Input: two integers a and b.
//Output: GCD value of a and b.
while (b ≠ 0)do
{
c ← a mod b
a ← b
b ← c
}
return a
```

Example 1.3.7 *Easter sunday is in principle the first sunday after the full moon after the spring equinox. Is this rule sufficiently precise to be called an algorithm ? Justify your answer.* **SPPU : Dec.-09, Marks 6**

Solution : There are two types of day calculations. By lunar age and by solar age. The lunar age starts at 1 and increases to 29 or 30 and then again start at 1. Each period of 29 or 30 makes a lunar month. Thus every alternate month is of 29 days. Hence in a year, if there are 12 months then.

$$\underset{\downarrow}{12} \times \underset{\downarrow}{29} + \underset{\downarrow}{6} = 354 \text{ days}$$

(months) (days) (6 months have 30 days)

The solar year is 11 days longer than the lunar age. Hence after 2 years there will be 22 more days in solar year than lunar age. To match both solar and lunar years we must add those many number of days to lunar calender. After 19 years there will be 209 days extra (As 19 X 11 = 209) in solar year than lunar year. If we perform 209 mod 30 then still we get 29 days exact. A cycle of lunar age is repeated after 30 days, hence by considering this issue, a sequence number called **Golden Number** is calculated as -

GN = Number of years mod 19+1.

Each year is associated with **Epact.**

Epact is nothing but the number of excess days in solar year. This epact is associated with the golden number as follows.

Epact = (11 x (GN – 1)) mod 30.

Thus there are some more calculations required to compute the epact of each year. The date of full moon is obtained for each epact and the first sunday following the full moon day of corresponding epact. This sunday is actually an **ester sunday.** Thus **computations** are involved in this ester sunday calculations. More over these calculations must be performed in some specific sequence. Hence this computation can be represented by **algorithm.**

1.4 Need of Correctness of Algorithm

SPPU : Dec.-18, Marks 7

1. To ensure that the algorithm is developed to correctly **satisfy the functional requirements,** there is a need for correctness of algorithm.
2. To ensure that the **solution** for the given problem is **valid** we need to check the correctness of algorithm.
3. To check the given problem has **finite or infinite solution** it is essential to check the correctness of algorithm.
4. The need for correctness of algorithm is for **efficient execution** of given task on computers.
5. To ensure that **all the cases** in the problem statements are **covered,** it is necessary to check the correctness of algorithm.
6. For some application correctness of algorithm means safety of human life or very costly equipment. Hence it is necessary to ensure that the algorithm is correct.

Basic steps in algorithmic correctness

Following are the basic steps that need to be followed for checking the correctness of algorithm -

1. Identification of the properties of input data. These properties of data are called **preconditions.**
2. Identification of the properties which must be satisfied by the output data. These properties are called **postconditions.**

3. Proving that starting from the preconditions and executing each step specified in the algorithms one obtains the postconditions.

Example 1.4.1 *Compare between definiteness and effectiveness of an algorithm with example.*

SPPU : Dec.-18, Marks 7

Solution : Definiteness : 1) The algorithm must always terminate after some finite number of states. 2) Each step of algorithm must be precisely defined. Also, the actions must be precisely defined. These actions has to be carried out rigorously and unambiguously.

Effectiveness : 1) All operations specified in the algorithm must be sufficiently basic that they can be done exactly as per specification. 2) Each operation in the algorithmic steps must have finite length.

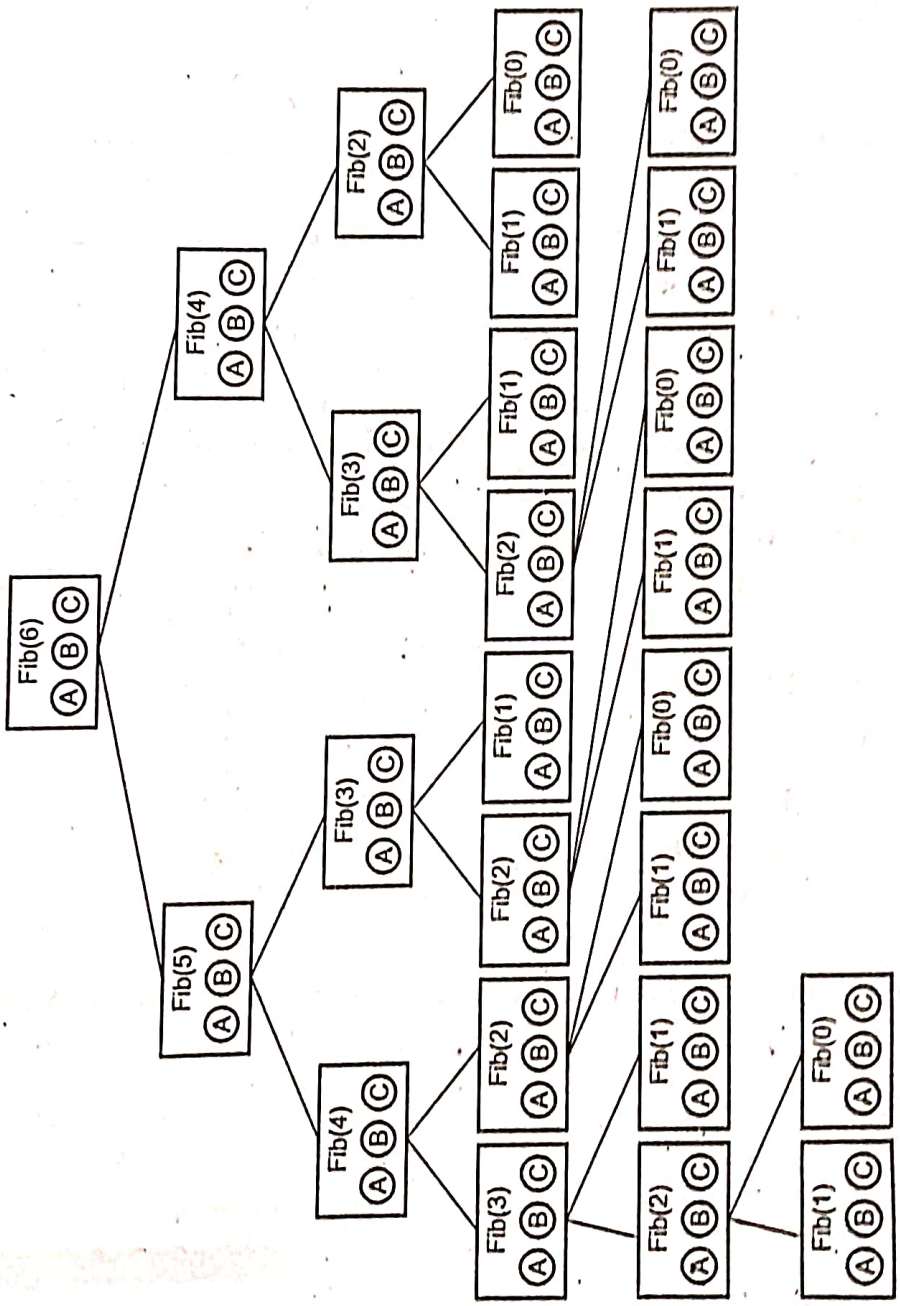


Fig. 1.4.1

1.5 Confirming Correctness of Algorithm - Sample Examples

SPPU : April-18, May-19, March-20, Marks 5

One simple method of confirming the correctness of algorithm is making use of assertions. An assertion is a logical statement or expression which is true. For example -

Example : Consider an algorithm for finding the smallest value among the three numbers. The assertions are denoted in curly brackets.

```
Algorithm Find_Min(int a,int b,int c)
{
    if a<b then                    //{a<b}
        if a<c then//{a<b, a<c}
            min<-a
        else          //{a<b, c<a}
            min<c
        endif
    else  //{b<a}
        if b<c then          //{b<a,b<c}
            min<-b
        else          //{b<a, c<b}
            min<-c
        endif
    endif
}
```

Finally we will get the minimum value from integers a,b and c in the variable min.

Example : Consider a problem of Traveling Salesman. This problem states that - " there is a traveling salesman who has to travel through different cities. He can visit each city one and only once. The total tour must be of minimum cost. That means total distance traveled by him must be least. "

For solving such problem using algorithm,

- Every tour is examined and cost of the tour is computed.
- The tour with minimum cost is examined and retained.

The algorithm will terminate because there are finite number of tours. But for large number of tours such algorithm need not be efficient as all the paths need to be examined.

This tells us that **algorithms correctness does not necessarily imply anything about its efficiency.**

Review Questions

1. *How to confirm the correctness of algorithm ? Explain with example.*

   SPPU : April-18, May-19, Marks 5

2. *What is need of correctness of algorithm ? What is loop invariant property ?*

   SPPU : March-20, Marks 5

1.6 Iterative Algorithm Design Issues

SPPU : May-19, March-20, Marks 6

- Iterative algorithm is an algorithm which has at least one iterative component or loop. Hence the part of algorithmic statements will be executed for n number of times.
- Even small amount of time spent on execution of loop will directly affect the efficiency of overall algorithm. This situation can be worst if nested loops (a loop within another loop) is present in the algorithm.
- There are various algorithmic design issues for iterative algorithms -
  i) Correct use of loops in the program.
  ii) Factors that affect the efficiency of algorithm.
  iii) Estimation and specification of execution time.
  iv) Order notations.

1.6.1 Use of Loops

To construct a loop three aspects need to be considered -

i) **Initial condition :** This condition must be true before execution of the loop.

ii) **Invariant relation :** This is a kind of relation that must hold before, during and after each iteration of the loop.

iii) **Terminate condition :** This is a condition under which the loop must terminate.

For example - Consider following loop

```
for(int i=1;i<=10; i++)
{
        printf("Hello");
}
```

Clearly the above loop will execute the printf statement for 10 times and we will get Hello to be printed 10 times on the console. Here

i=0 is **initial condition**

i++ means **incrementing** i on each iteration.

i<=10 is a **terminate condition.** That means when i value goes beyond 10 the loop will not be executed.

1.6.2 Efficiency of Algorithms

- The design and implementation of algorithm have deep influence on their efficiency. Every algorithm on its implementation makes use of CPU time and internal memory (RAM). The cost of utilization of these resources need to be considered while designing the algorithm.
- Today, designing the efficient algorithm is a major requirement.

- There is no standard method for designing efficient algorithm.
- There is a need to generate generalized algorithms as well as specific problem solving algorithm. It always depends upon the problem to be solved.
- There are some **optimization techniques** that improve the efficiency of execution of loop. These are

i. Removing redundant computations inside the loop

- Most of the times the algorithm becomes inefficient due to **redundant computations** or unnecessary storage.
- The effect of redundant instructions is serious when they are used within the loop. These loops repeatedly calculate the part of expression that remains constant throughout the execution phase of the loop. For example -

Consider following code

```
a=5; b=10;x=1;
for(int i=0;i<10;i++)
{
                    x=x+1;
                    y=(a*a*a)+(b*b)+x;
                    printf("x = %d, y= %d"x,y);
}
```

- In above code fragment computation of (a*a*a) and (b*b) remains unchanged through the execution of loop. The loop unnecessarily performs multiplication within the loop. This makes the algorithm inefficient.
- To make the above code efficient we need to remove the redundant computations inside the loop. The code can be made efficient as follows -

```
a=5; b=10;x=1;
int temp1=a*a*a;
int temp2=b*b;
for(int i=0;i<10;i++)
{
    x=x+1;
    y=temp1+temp2+x;
    printf("x = %d, y= %d"x,y);
}
```

ii. Referencing array element

- Accessing array element inside the loop, makes the algorithm inefficient. For example consider following code -

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

```
        if (a [i] < a[position])
        {
                position = i;
        }
}
min_element = a[position];
```

In above code the array a[position] is accessed each time during the loop execution. If we denote it using some variable then the execution of above loop can be made efficient.

- The above code can be modified as follows to make it efficient -

```
position = 0;
min_element=a[position];
for (i = 1; i < n; i + +)
{
        if (a [i] < min_element)
        {
                min_element=a[i];
                position = i;
        }
}
```

iii. Late detection of terminating condition

Sometime more tests are conducted than required. This also makes the algorithm inefficient. For example consider the example of linear search. Suppose we want to search the names from alphabetically stored names and we are searching for the name starting with letter 'J'. Now when we get the name starting from letter 'J' we should not scan the letter starting from 'K','L'... and so on.

iv. Early detection of desired output condition

Due to nature of input data, it is possible to detect the desired output condition early. For example - if we are sorting the already sorted data using bubble sort method, then instead of applying bubble sort loop we must examine the input. How can one determine that input is already sorted ? Well, for that matter, we need to check whether there have been any exchanges in the current pass of the inner loop. If there have been no exchanges in the current pass, the data must be already sorted and the loop can be terminated early.

1.6.3 Estimating and Specifying Execution Time

The time complexity of an algorithm is the **amount of computer time required** by an algorithm to run to completion.

It is **difficult to compute the time complexity** in terms of **physically clocked time**. For instance in multiuser system, executing time depends on many **factors** such as -

- System load
- Number of other programs running
- Instruction set used
- Speed of underlying hardware

The time complexity is therefore given in terms of **frequency count.**

Frequency count is a count denoting number of times of execution of statement.

Definition : The frequency count is a count that denotes how many times particular statement is executed.

For Example : Consider following code for counting the frequency count

```
void fun()
{
 int a;
 a=10; .....1
 printf("%d",a); .....1
}
```

The frequency count of above program is **2.**

Example 1.6.1 *Obtain the frequency count for the following code.*

```
 void fun()
{
 int a;
 a=0; .....1
 for(i=0;i<n;i++) .....n+1
 {
    a  = a+i; .....n
 }
 printf("%d",a); .....1
 }
```

Solution : The frequency count of above code is **2n + 3.**

The for loop in above given fragment of code is executed n times when the condition is true and one more time when the condition becomes false. Hence for the for loop the frequency count is n + 1. The statement inside the for loop will be executed only when the condition inside the for loop is true. Therefore this statement will be executed for n times. The last printf statement will be executed for once.

Example 1.6.2 *Obtain the frequency count for the following code.*

```
void fun(int a[][],int b[][])
{
int c[3][3];
for(i=0;i<m;i++) .....m+1
{
 for(j=0;j<n;j++) .....m(n+1)
 {
  c[i][j]=a[i][j]+b[i][j]; .....m.n
 }
  }
```

Solution : The frequency count = (m + 1) + m(n + 1) + mn = 2m + 2mn + 1 = **2m(1 + n) + 1**

Example 1.6.3 *Obtain the frequency count for the following code.*

```
for(i=1;i<=n;i++)
{
for(j=1;j<=n;j++)
{
 c[i][j]=0;
 for(k=1;k<=n;k++)
   c[i][j]=c[i][j]+a[i][k]*b[k][j];
}
}
```

Solution :

Statement	Frequency Count
for(i=1;i<=n;i++)	n + 1
for(j=1;j<=n;j++)	n.(n + 1)
c[i][j]=0;	n.(n)
for(k=1;k<=n;k++)	n.n(n + 1)
c[i][j]=c[i][j]+a[i][k]*b[k][j];	n.n.n
Total	$2n^3 + 3n^2 + 2n + 1$

After counting the frequency count, the constant terms can be neglected and only the order of magnitude is considered. The time complexity is denoted in terms of algorithmic notations. The Big oh notation is a most commonly used algorithmic notation. For the above frequency count all the constant terms are neglected and only the order of magnitude of the polynomial is considered. Hence the time complexity for the above code can be $O(n^3)$. The higher order is the polynomial is always considered.

1.6.4 Algorithmic Strategies

Problem solving strategy is a general approach by which many problems can be solved algorithmically. These problems may belong to different areas of computing. These strategies are also called as **algorithmic techniques** or **algorithmic paradigms.**

1. **Brute Force :** This is straightforward technique with naïve approach.
2. **Divide and Conquer :** The problem is divided into smaller sub-instances and solution of these sub-instances help to solve the main problem.
3. **Dynamic Programming :** The results of smaller, reoccurring instances are obtained to solve the problem.
4. **Greedy Technique :** To solve the problem locally optimal decisions are made.
5. **Backtracking :** This method is based on trial and error. Each time the solution from the set of solutions is picked and checked if it is the accurate solution or not.

Review Question

1. *Explain issues related to iterative algorithm design.*

SPPU : May-19, Marks 6, March-20, Marks 5

1.7 Problem Solving Principles

The problem solving steps and principles are given below -

1. **Understand the Problem :** For understanding the problem statement, read the problem description carefully and ask the questions such as - What is unknown in the problem ? What is data ? What is the condition ? Then draw the figure and introduce the notations. Separate out various parts of conditions.
2. **Make Decisions :** After finding the required set of input for the given problem we have to analyze the input and need to decide certain issues such as **capabilities of computational devices, selection of problem solving approach, suitable data structures** and **algorithmic techniques**. These decisions serve as the base for the **design of algorithm.**
3. **Design or Specify the Problem :** The design or specification of the problem can be using natural language, flowchart or algorithms.
4. **Verify the Problem :** Problem verification involves checking the correctness of an algorithm. The proof of correctness of the problem can be complex sometimes.
5. **Analyze the Problem :** For analyzing the problem the time and space requirements are considered. The **amount of time** required by algorithm and **amount of space** required by the algorithm is computed and then the analysis is made.

6. **Implement :** The implementation of the problem can be done using some suitable **programming language.**

1.8 Classification of Problem

Generally any computing problem can be solved by an algorithm.There is large number of computing problems and some of them can be classified as,

1. Sorting
2. Searching
3. Numerical Problems
4. Geometric Problems
5. Combinatorial Problems
6. Graph Problems
7. String Processing Problems.

These important problem types can be discussed in detail as follows -

1. Sorting : **Sorting** means arranging the elements in increasing order or in decreasing order (also called as **ascending order** or **descending order** respectively). The sorting can be done on numbers, characters (alphabets), strings or employees record. For sorting any record we need to choose certain piece of information based on which sorting can be done. For instance: for keeping the employees record in sorting order we will arrange the employees record as per employee ID. Similarly one can arrange the library books according to title of the books. This piece of information which is required to sort the records is called **key**. The important property of this key is that **it should be unique.**

2. Searching : Searching is an activity by which we can find out the desired element from the list. The element which is to be searched is called **search key**. There are many searching algorithms such as sequential search, binary search, Fibonacci search and many more.

How to choose best searching algorithm ?

One cannot declare that particular algorithm is best searching algorithm because some algorithms work faster, but then they may require more memory. Some algorithms work better if the list is almost sorted. Thus efficiency of algorithm is varying at varying situations.

Searching in dynamic set of elements

There may be a set of elements in which repeated addition or deletion of elements occur. In such a situation searching an element is difficult. To handle such lists supporting data structures and algorithms are needed make the list balanced (organized).

3. Numerical Problems : The Numerical problems are based on mathematical equations, systems of equations, computing definite integrals, evaluating functions and so on. These mathematical problems are generally solved by approximate algorithms. These algorithms require manipulating of the real numbers hence we may get wrong output many times.

4. Geometric Problems : The geometric problems is one type of problem solving area in which various operations can be performed on geometric objects such as points, line, polygon. The geometric problems are solved mainly in applications to computer graphics, tomography or in robotics.

5. Combinatorial Problems : The combinatorial problems are related to the problems like computing permutations and combinations. The combinatorial problems are most difficult problems in computing area because of following causes -

- As problem size grows the combinatorial objects grow rapidly and reach to a huge value.
- There is no algorithm available which can solve these problems in finite amount of time.
- Many of these problems fall in the category of unsolvable problems. However there are certain problems that are solvable.

6. Graph Problems : Graph is a collection of vertices and edges. The graph problems involve graph traversal algorithms, shortest path algorithms and topological sorting and so on. Some graph problems are very hard to solve. For example Traveling salesman problem, graph coloring problems.

7. String Processing Problems : String is a collection of characters. In computer science the typical string processing algorithm is pattern matching algorithm. In these algorithms particular word is searched from the text. The algorithms lying in this category are simple to implement.

1.9 Problem Solving Strategies

Problem solving strategy is a general approach by which many problems can be solved algorithmically. These problems may belong to different areas of computing. These strategies are also called as **algorithmic techniques** or **algorithmic paradigms.**

1. **Brute Force :** This is straightforward technique with naïve approach.
2. **Divide and Conquer :** The problem is divided into smaller sub-instances and solution of these sub-instances help to solve the main problem.
3. **Dynamic Programming :** The results of smaller, reoccurring instances are obtained to solve the problem.

4. **Greedy Technique :** To solve the problem locally optimal decisions are made.
5. **Backtracking :** This method is based on trial and error. Each time the solution from the set of solutions is picked and checked if it is the accurate solution or not.

1.10 Classification of Time Complexities

We can classify the time complexity of the algorithm in various classes. These classes are also called as **basic efficiency classes.** Each class possessing certain characteristic. These classes are enlisted in the following table,

Name of efficiency class	Order of growth	Description	Example
Constant	1	As input size grows we get larger running time.	Scanning array elements.
Logarithmic	logn	When we get logarithmic running time then it is sure that the algorithm does not consider all its input rather the problem is divided into smaller parts on each iteration.	Performing binary search operation.
Linear	n	The running time of algorithm depends on the input size n.	Performing sequential search operation.
nlogn	nlogn	Some instance of input is considered for the list of size n.	Sorting the elements using merge sort or quick sort.
Quadratic	n^2	When the algorithm has two nested loops then this type of efficiency occurs.	Scanning matrix elements.
Cubic	n^3	When the algorithm has three nested loops then this type of efficiency occurs.	Performing matrix multiplication.
Exponential	2^n	When the algorithm has very faster rate of growth then this type of efficiency occurs.	Generating all subsets of n elements.
Factorial	n!	When an algorithm is computing all the permutations then this type of efficiency occurs.	Generating all permutations.

Example 1.10.1 *Compare orders of growth of* $\log_2(n)$ *and* $\sqrt{n}$.

Solution : For comparison, we will consider various values of n.

If n = 2

$$\log_2(n) = \log_2 2 = 1$$

$$\sqrt{n} = \sqrt{2} = 1.414$$

If n = 64

$$\log_2(64) = 6$$

$$\sqrt{64} = 8$$

If n = 256

$$\log_2(256) = 8$$

$$\sqrt{256} = 16$$

All these computations show that

$$\log_2(n) < \sqrt{n}$$

❑❑❑