UNIT 1: PROBLEM SOLVING

UNIT STRUCTURE

1.0 Objectives
1.1 Introduction
1.2 Problem Solving Aspect
  1.2.1 Decomposition
  1.2.2 Pattern recognition
  1.2.3 Pattern generalisation and abstraction
  1.2.4 Algorithm design
1.3 Top-Down Design
  1.3.1 Modular Design
  1.3.2 Advantages of Top-Down Design
1.4 Implementation Of Algorithm
  1.4.1 Definition
  1.4.2 Features of Algorithm
  1.4.3 Steps involved in algorithm development
1.5 Program Verification
  1.5.1 Basic steps in algorithms correctness verification
1.6 Algorithm Efficiency
  1.6.1 Redundant Computations
  1.6.2 Referencing Array Elements
  1.6.3 Inefficiency due to late termination
  1.6.4 Early detection of desired output condition
  1.6.5 Trading storage for efficient gains
1.7 Algorithm Analysis
  1.7.1 Computational Complexity
  1.7.2 The Order of Notation
  1.7.3 Rules for using the Big-O Notation
  1.7.4 Worst and Average Case Behavior
1.8 Summary
1.9 Questions for Exercises
1.10 Suggested Readings

1.0 OBJECTIVES

Solving problems is the core of computer science. Programmers must first understand how a human solves a problem, then understand how to translate this "algorithm" into something a computer can do, and finally how to "write" the specific syntax (required by a computer) to get the job done. It is sometimes the case that a machine will solve a problem in a completely different way than a human.

After completing this chapter, you will be able to explain the basic concepts of problem solving; list the steps involved in program development; list the advantages of top-down programming; describe the analysis of algorithm efficiency; and discuss the analysis of algorithm complexity.

1.1 INTRODUCTION

A computer is a very powerful and versatile machine capable of performing a multitude of different tasks, yet it has no intelligence or thinking power. The intelligence Quotient (I.Q) of a computer is zero. A computer performs many tasks exactly in the same manner as it is told to do. This places responsibility on the user to instruct the computer in a correct

and precise manner, so that the machine is able to perform the required job in a proper way. A wrong or ambiguous instruction may sometimes prove disastrous.

In order to instruct a computer correctly, the user must have clear understanding of the problem to be solved. Apart from this he should be able to develop a method, in the form of series of sequential steps, to solve it. Once the problem is well-defined and a method of solving it is developed, then instructing the computer to solve the problem becomes relatively easier task.

Thus, before attempt to write a computer program to solve a given problem. It is necessary to formulate or define the problem in a precise manner. Once the problem is defined, the steps required to solve it, must be stated clearly in the required order.

1.2 PROBLEM SOLVING ASPECT

Problem solving is a creative process. It is an act of defining a problem, determining the cause of the problem, identifying, prioritizing, and selecting alternatives for a solution and implementing a solution.

A problem can be solved successfully only after making an effort to understand the problem. To understand the problem, the following questions help:

- What do we know about the problem?
- What is the information that we have to process in order the find the solution?
- What does the solution look like?
- What sort of special cases exist?
- How can we recognize that we have found the solution?

It is important to see if there are any similarities between the current problem and other problems that have already been solved. We have to be sure that the past experience does not hinder us in developing new methodology or technique for solving a problem.

The important aspect to be considered in problem-solving is the ability to view a problem from a variety of angles. There is no universal method for solving a given problem. Different strategies appear to be good for different problems. Computational thinking is made up of four parts:

- Decomposition
- Pattern recognition
- Pattern generalisation and abstraction
- Algorithm design

Let's take a look at what each of these mean:

1.2.1 Decomposition

It is about breaking down a big problem into the smaller problems that make it up. For example if I gave you a cake and asked you to bake me another one you might struggle, but if you watched me making the cake and worked out the ingredients then you'd stand a much better chance of replicating it. If you can look at a problem and work out the main steps of that problem then you'll stand a much better chance of solving it.

Let's look at an example, the equation to work out the roots of a quadratic equation.

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

On first look it might appear a little scary, but if we decompose it we should stand a better chance of solving it:

1. b^2
2. $4ac$
3. $b^2 - 4ac$
4. $\sqrt{b^2 - 4ac}$
5. $-b + \sqrt{b^2 - 4ac}$

6. $2a$

7. repeat for $-b - \sqrt{b^2 - 4ac}$

By noting the steps down to solve a problem we can often recognise patterns, and by giving a list of steps we are one step closer to creating an algorithm.

1.2.2 Pattern recognition

Often breaking down a problem into its components is a little harder than taking apart an algorithm, we often get given a set of raw data then are asked to find the pattern behind it:

1, 4, 7, 10, 13, 16, 19, 22, 25, …

This is pretty easy with number sets, the above has the pattern $A_n = A_{n-1} + 3$.
But pattern recognition might also involve recognising shapes, sounds or images. If your camera highlights faces when you point it at some friends, then it is recognising the pattern of a face in a picture.

Figure 1.1: The face was automatically detected by pattern recognition

1.2.3 Pattern generalisation and abstraction

Once we have recognised a pattern we need to put it in its simplest terms so that it can be used whenever we need to use it. For example, if you were studying the patterns of how people speak, we might notice that all proper English sentences have a subject and predicate.

1.2.4 Algorithm design

Once we have our patterns and abstractions we can start to write the steps that a computer can use to solve the problem. We do this by creating **Algorithms**. Algorithms aren't computer code, but are independent instructions that could be turned into computer code. We often write these independent instructions as **pseudo code**. Examples of algorithms could be to describe orbit of the moon, the steps involved in setting up a new online shopping account or the sequences of tasks involved for a robot to build a new car.

Example Find the factorial of a given number
Input: Positive valued integer number
Output: Factorial of that number
Process: Solution technique which transforms input into output. Factorial of a number can be calculated by the formula n! = 1*2*3*4….*n

1.3 TOP DOWN DESIGN

A design is the path from the problem to a solution in code. Program Design is both a product and a process. The process results in a theoretical framework for describing the effects and consequences of a program as they are related to its development and implementation.

A well designed program is more likely to be:

- Easier to read and understand later
- Less of bugs and errors
- Easier to extend to add new features
- Easier to program in the first place

1.3.1 Modular Design

Once the problem is defined clearly, several design methodologies can be applied. An important approach is Top-Down programming design. It is a structured design technique which breaks up the problem into a set of sub-problems called Modules and creates a hierarchical structure of modules.

While applying top-down design to a given problem, consider the following guidelines:

- A problem is divided it into smaller logical sub-problems, called Modules
- Each module should be independent and should have a single task to do
- Each module can have only one entry point and one exit point, so that the logic flow of the program is easy to follow
- When the program is executed, it must be able to move from one module to the next in sequence, until the last module is executed
- Each module should be of manageable size, in order to make the design and testing easier

1.3.2 Advantages of Top-Down Design

Top-down design has the following advantages:

- Breaking up the problem into parts helps us to clarify what is to be done
- At each step of refinement, the new parts become more focussed and, therefore, easier to design
- Modules may be reused
- Breaking the problem into parts allows more than one person to work on the solution simultaneously

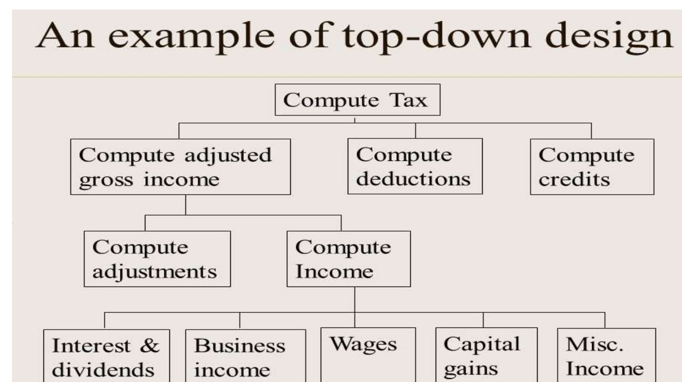

Figure 1.2: Schematic breakdown of a problem into subtasks

1.4 IMPLEMENTATION OF ALGORITHM

The first step in the program development is to devise and describe a precise plan of what you want the computer to do. This plan, expressed as a sequence of operations, is called an algorithm. An algorithm is just an outline or idea behind a program.

1.4.1 Definition

A set of sequential steps usually written in Ordinary Language to solve a given problem is called **Algorithm**. It may be possible to solve to problem in more than one ways, resulting in more than one algorithm. The choice of various algorithms depends on the factors like reliability, accuracy and easy to modify. The most important factor in the choice of algorithm is the time requirement to execute it, after writing code in High-level language with the help of a computer. The algorithm which will need the least time when executed is considered the best.

An algorithm can be defined as **“a complete, unambiguous, finite number of logical steps for solving a specific problem.”**

In computer programming, there are often many different algorithms to accomplish any given task. Each algorithm has advantages and disadvantages in different situations. Sorting is one place where a lot of research has been done, because computers spend a lot of time sorting lists. Three reasons for using algorithms are efficiency, abstraction and reusability.

1.4.2 Features of Algorithm

Efficiency: Certain types of problems, like sorting, occur often in computing. Efficient algorithms must be used to solve such problems considering the time and cost factor involved in each algorithm.

Abstraction: Algorithms provide a level of abstraction in solving problems because many seemingly complicated problems can be distilled into simpler ones for which well-known algorithms exist. Once we see a more complicated problem in a simpler light, we can think of the simpler problem as just an abstraction of the more complicated one. For example, imagine trying to find the shortest way to route a packet between two gateways in an internet. Once we realize that this problem is just a variation of the more general shortest path problem, we can solve it using the generalised approach.

Reusability: Algorithms are often reusable in many different situations. Since many well-known algorithms are the generalizations of more complicated ones, and since many complicated problems can be distilled into simpler ones, an efficient means of solving certain simpler problems potentially lets us solve many complicated problems.

An algorithm must possess the following properties
1. Finiteness: An algorithm must terminate in a finite number of steps
2. Definiteness: Each step of the algorithm must be precisely and unambiguously stated
3. Effectiveness: Each step must be effective, in the sense that it should be primitive easily convert able into program statement) can be performed exactly in a finite amount of time.
4. Generality: The algorithm must be complete in itself so that it can be used to solve problems of a specific type for any input data.
5. Input/output: Each algorithm must take zero, one or more quantities as input data produce one or more output values. An algorithm can be written in English like sentences or in any standard representation sometimes, algorithm written in English like languages are called Pseudo Code

1.4.3 Steps involved in algorithm development

Step1. Identification of input: For an algorithm, there are quantities to be supplied called input and these are fed externally. The input is to be indentified first for any specified problem.

Step2: Identification of output: From an algorithm, at least one quantity is produced, called for any specified problem.

Step3 : Identification the processing operations : All the calculations to be performed in order to lead to output from the input are to be identified in an orderly manner.

Step4 : Processing Definiteness : The instructions composing the algorithm must be clear and there should not be any ambiguity in them.

Step5 : Processing Finiteness : If we go through the algorithm, then for all cases, the algorithm should terminate after a finite number of steps.

Step6 : Possessing Effectiveness : The instructions in the algorithm must be sufficiently basic and in practice they can be carries out easily.

1.5 PROGRAM VERIFICATION

When an algorithm is designed it should be analyzed at least from the following points of view:

• *Correctness:* This means to verify if the algorithm leads to the solution of the problem

(hopefully after a finite number of processing steps).

• *Efficiency:* This means to establish the amount of resources (memory space and processing time) needed to execute the algorithm on a machine (a formal one or a physical one).

1.5.1 Basic steps in algorithms correctness verification

To verify if an algorithms really solves the problem for which it is designed we can use one of the following strategies:

• ***Experimental analysis (testing).*** We test the algorithm for different instances of the problem (for different input data). The main advantage of this approach is its simplicity while the main disadvantage is the fact that testing cannot cover always all possible instances of input data (it is difficult to know how much testing is enough). However the experimental analysis allows sometimes to identify situations when the algorithm doesn't work.

• ***Formal analysis (proving).*** The aim of the formal analysis is to prove that the algorithm works for any instance of data input. The main advantage of this approach is that if it is rigorously applied it guarantee the correctness of the algorithm. The main disadvantage is the difficulty of finding a proof, mainly for complex algorithms. In this case the algorithm is decomposed in sub algorithms and the analysis is focused on these (simpler) sub- algorithms.
On the other hand the formal approach could lead to a better understanding of the algorithms. This approach is called formal due to the use of formal rules of logic to show that an algorithm meets its specification.

The main steps in the formal analysis of the correctness of an algorithm are:

• Identification of the properties of input data (the so-called *problem's preconditions*).

• Identification of the properties which must be satisfied by the output data (the so-called *problem's post conditions*).

• Proving that starting from the preconditions and executing the actions specified in the algorithms one obtains the post-conditions.

When we analyze the correctness of an algorithm a useful concept is that of *state*.
The algorithm's state is the set of the values corresponding to all variables used in the algorithm. The state of the algorithm changes (usually by variables assignments) from one processing step to another processing step. The basic idea of correctness verification is to establish which should be the state corresponding to each processing step such that at the end of the algorithm the post-conditions are satisfied. Once we established these intermediate states it is sufficient to verify that each processing step ensures the transformation of the current state into the next state.
When the processing structure is a sequential one (for example a sequence of assignments) then the verification process is a simple one (we must only analyze the effect of each assignment on the algorithm's state). Difficulties may arise in analyzing loops because there are many sources of errors: the initializations may be wrong, the processing steps inside the loop may be wrong or the stopping condition may be wrong.
A formal method to prove that a loop statement works correctly is the mathematical induction method.

Example 1.5.1.1. Let us consider the following algorithm whose aim is to find the minimal value in a finite sequence of real numbers:

minimum(a[1..n])
$min \leftarrow a[1]$
for $i \leftarrow 2, n$ **do**
if $min > a[i]$ **then** $min \leftarrow a[i]$ **endif**
endfor
return *min*

The input array can be arbitrary, thus the preconditions set consists only of $\{n \geq 1\}$ (meaning that the array is not empty).
The post-condition is represented by the property of the minimal value: $min \leq a[i]$ for all
$i = 1, n$.
We only have to prove that this post-condition is satisfied after the algorithm's execution. Thus, we will prove by using the mathematical induction that $min \leq a[i]$, $i = 1, n$.
Since $min = a[1]$ and the value of min is replaced only with a smaller one it follows that
$min \leq a[1]$.
Let us suppose that $min \leq a[k]$ for all $k = 1, i - 1$. Now, we only have to prove that
$min \leq a[i]$.
In the **for** loop, the value of min is modified as follows:
• If $min \leq a[i]$ then min keeps its old value.
• If $min > a[i]$ then the value of min is replaced with $a[i]$.
Thus in both situations min will satisfy $min \leq a[i]$. According to the mathematical induction method it follows that the post-condition is satisfied, thus the algorithm is correct.

Let us consider now the following algorithm:

minim($a[1..n]$)
for $i \leftarrow 1, n - 1$ **do**
if $a[i] < a[i + 1]$ **then** $min \leftarrow a[i]$
else $min \leftarrow a[i + 1]$
endfor
return *min*

In this case the processing step of the loop body will lead to $min = \min\{a[i], a[i + 1]\}$, thus
one cannot anymore prove that $min = \min\{a[1..i]\}$ implies $min = \min\{a[1..i + 1]\}$. On the other hand it is easy to find a counter example (for instance (2, 1, 3, 5, 4)) for which the above algorithm doesn't work. However it can be proved that it computes $\min\{a[n - 1], a[n]\}$.

1.6 ALGORITHM EFFICIENCY

The importance of efficiency with respect to time was emphasised by Ada Lovelace in 1843 as applying to Charles Babbage's mechanical analytical engine:

"In almost every computation a great variety of arrangements for the succession of the processes is possible, and various considerations must influence the selections amongst them for the purposes of a calculating engine. One essential object is to choose that arrangement which shall tend to reduce to a minimum the time necessary for completing the calculation".

Early electronic computers were severely limited both by the speed of operations and the amount of memory available. In some cases it was realized that there was a space–time trade-off, whereby a task could be handled either by using a fast algorithm which used quite a lot of working memory, or by using a slower algorithm which used very little working memory. The engineering trade-off was then to use the fastest algorithm which would fit in the available memory.

There are many ways in which the resources used by an algorithm can be measured: the two most common measures are speed and memory usage; other measures could include transmission speed, temporary disk usage, long-term disk usage, power consumption, total cost of ownership, response time to external stimuli, etc. Many of these measures depend on the size of the input to the algorithm (i.e. the amount of data to be processed); they might also depend on the way in which the data is arranged (e.g. some sorting algorithms perform poorly on data which is already sorted, or which is sorted in reverse order).

In practice, there are other factors which can affect the efficiency of an algorithm, such as requirements for accuracy and/or reliability.

Every algorithm uses some of the computer's resources like central processing time and internal memory to complete its task. Because of high cost of computing resources, it is desirable to design algorithms that are

economical in the use of CPU time and memory. Efficiency considerations for algorithm are tied in with design, implementation and analysis of algorithm.

There is no simpler way of designing efficient algorithm, but a few suggestions as shown below can be useful in designing an efficient algorithm.

1.6.1 Redundant Computations

Redundant computations are unnecessary computations result in inefficiency in the implementation of the algorithms. When redundant calculations are embedded inside the loop for the variable which remains unchanged throughout the entire execution phase of the loop, the results are more serious.

Consider the following code in which the value x*x*x*z is redundantly calculated in the loop:

```
a=0;
for i=0 to n
        a=a+1;
        b=(x*x*x*z)*a*a+y*y*a;
        print a, b;
next i
```

This redundant calculation can be removed by small modification in the program:

```
a=0;
j=x*x*x*z;
k=y*y
for i=0 to n
        a=a+1;
        b=j*a*a+k*a;
        print a, b;
next i
```

1.6.2 Referencing Array Elements

For using array element, we require two memory references and an additional operation to locate the correct value for use.so, efficient program must not refer to the same array element again and again if the value of the array element does not change.

For example:

Case 1

```
x=1;
for i=0 to n
        if (a[i]>a[x])
                x=i;
next i;
max=a[x];
```

Case 2

```
x=1;
max=a[x];
for i=0 to n
        if (a[i]>max)
                max=a[i];
next i;
```

In the above examples, case 2 is more efficient algorithm than case 1.

1.6.3 Inefficiency due to late termination

Another place where inefficiency can come into implementation is where considerably more tests are done than are required to solve the problem at hand. For example, in the case of the below bubble sort algorithm, the inner loop should not proceed beyond n-i, because last i elements are already sorted.

```
for i=0 to n
        for j=0 to n-1
                if(a[j]>a[j+1])
```

//swap values a[j], a[j+1]

1.6.4 Early detection of desired output condition

Sometimes the loops can be terminated early, if the desired output conditions are met. This saves a lot of unfruitful execution. For example, in the bubble sort algorithm, if during the current pass of the inner loop there is no exchanges in the data, then the list can be assumed to be sorted and the search can be terminated before running the outer loop for *n* times.

1.6.5 Trading storage for efficient gains

A trade between storage and efficiency is often used to improve the performance of an algorithm. This can be done if we save some intermediary results and avoid having to do a lot of unnecessary testing and computation later on. One strategy for speeding up the execution of an algorithm is to implement it using the least number of loops.

1.7 ALGORITHM ANALYSIS

Algorithms usually possess the following qualities and capabilities:

- Easily modifiable if necessary.
- They are easy, general and powerful.
- They are correct for clearly defined solution.
- Require less computer time, storage and peripherals i.e. they are more economical.
- They are documented well enough to be used by others who do not have a detailed knowledge of the inner working.
- They are not dependable on being run on a particular computer.
- The solution is pleasing and satisfying to its designer and user.
- They are able to be used as a sub-procedure for other problems.

Two or more algorithms can solve the same problem in different ways. So, quantitative measures are valuable in that they provide a way of comparing the performance of two or more algorithms that are intended to solve the same problem. This is an important step because the use of an algorithm that is more efficient in terms of time, resources required, can save time and money.

"**Analysis of algorithm**" is a field in computer science whose overall goal is an understanding of the complexity of algorithms (in terms of time Complexity), also known as ***execution time*** & storage (or space) requirement taken by that algorithm.

Space Complexity

Space complexity of and algorithm is the number of elementary objects that this algorithm needs to store during its execution. The space occupied by an algorithm is determined by the number and sizes of the variables and data structures used by the algorithm.

Time Complexity

The time , taken by a program P, is the sum of the Compile time & the Run (execution) time. The Compile time does not depends on the instance characteristics (i.e. no. of inputs, no. of outputs, magnitude of inputs, magnitude of outputs etc.).

Thus we are concerned with the running time of a program only.

1.7.1 Computational Complexity

We can characterize an algorithm's performance in terms of the size (usually n) of the problem being solved. More computing resources are needed to solve larger problems in the same class. The table below illustrates the comparative cost of solving the problem for a range of n values.

$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n
1	2	2	4	5	4
3.322	10	33.22	10^2	10^3	$>10^3$
6.644	10^2	664.4	10^4	10^6	$>>10^{25}$

9.966	10^3	9966.0	10^6	10^9	$>>10^{250}$
13.287	10^4	132877	10^8	10^{12}	$>>10^{2500}$

The above table shows that only very small problems can be solved with an algorithm that exhibit exponential behaviour. An exponential problem with n=100 would take immeasurably longer time. At the other extreme, for an algorithm with logarithmic dependency would merely take much less time (13 steps in case of log2n in the above table). These examples emphasize the importance of the way in which algorithms behave as a function of the problem size. Analysis of an algorithm also provides the theoretical model of the inherent computational complexity of a particular problem.

To decide how to characterize the behaviour of an algorithm as a function of size of the problem n, we must study the mechanism very' carefully to decide just what constitutes the dominant mechanism. It may be the number of times a particular expression is evaluated, or the number of comparisons or exchanges that must be made as n grows. For example, comparisons, exchanges, and moves count most in sorting algorithm. The number of comparisons usually dominates so we use comparisons in computational model for sorting algorithms.

1.7.2 The Order of Notation

The 0-notation gives an upper bound to a function within a constant factor. For a given function g(n), we denote by O(g(n)) the set of functions.

0(g(n)) = { f(n) : there exist positive constants c and n0, such that 0 <= f(n) <= cg(n) for all n >— n0 }

Using O-notation, we can often describe the running time of an algorithm merely by inspecting the algorithm's overall structure. For example a double nested loop structure of the following algorithm immediately yields O(n2) upper bound on the worst case running time.

```
for i=0 to n
        for j=0 to n
                print i,j
        next j
next i
```

What we mean by saying "the running time is $O(n^2)$" is that the worst case running time (which is a function of n) is $O(n^2)$. Or equivalently, no matter what particular input of size n is chosen for each value of-n, the running time on that set of inputs is $O(n^2)$.

1.7.3 Rules for using the Big-O Notation

Big-O bounds, because they ignore constants, usually allow for very simple expression for the running time bounds. Below are some properties of big-0 that al1ow bounds to be simplified. The most important property is that big-O gives an upper bound only. If an algorithm is $O(N^2)$, it doesn't have to take N^2 steps (or a constant multiple of N^2). But it can't take more than N^2. So any algorithm that is O(N), is also an $O(N^2)$ algorithm. If this seems confusing, think of big-O as being like "<". Any number that is <N is also $<N^2$.

1. Ignoring constant factors: O(c f(N)) = O(f(N)), where c is a constant; e.g. $O(2 \text{ O } N^3) = O(N^3)$
2. Ignoring smaller terms: If a<b then O(a+b) = O(b), for example, $O(N^2+N) = O(N^2)$
3. Upper bound only: If a<b then an O(a) algorithm is also an O(b) algorithm. For example, an O(N) algorithm is also an $O(N^2)$ algorithm (but not vice versa).
4. N and log N are bigger than any constant, from an asymptotic view (that means for large enough N). So if k is a constant, an O(N + k) algorithm is also O(N), by ignoring smaller terms. Similarly, an O(log N + k) algorithm is also O(log N).
5. Another consequence of the last item is that an O(N log N + N) algorithm, which is O(N(log N + 1)), can be simplified to O(N log N).

1.7.4 Worst and Average Case Behavior

Worst and average case behaviors of the algorithm are the two measures of performance that are usually considered. These two measures can be applied to both space and time complexity of an algorithm. The worst case complexity for a given problem of size n corresponds to the maximum complexity encountered among all problems of size n. For determination of the worst case complexity of an algorithm, we choose a set of input conditions that force the algorithm to make the least possible progress at each step towards its final goal.

In many practical applications it is very important to have a measure of the expected complexity of an algorithm rather than the worst case behavior. The expected complexity gives a measure of the behavior of the algorithm averaged over all possible problems of size n.

As a simple example: Suppose we wish to characterize the behavior of an algorithm that linearly searches an ordered list of elements for some value x.

1 2 3 4 5……………….. N

In the worst case, the algorithm examines all n values in the list before terminating.

In the average case, the probability that x will be found at position 1 is 1/n, at position 2 is 2/n and so on. Therefore,

Average search cost = 1/n(1+2+3+ = +n)

1n(n12(n-r-:)) (n-i 1)2

1.8 SUMMARY

- In order to instruct a computer correctly, the user must have clear understanding of the problem and should develop a method to solve it.
- Computational thinking is made up of four parts –decomposition, pattern recognition, pattern generalisation & abstraction and algorithm design.
- Decomposition deals about breaking down a big problem into the smaller problems.
- Pattern recognition involves recognising shapes, sounds or images.
- Once we have recognised a pattern we need to put it in its simplest terms-which comes under pattern generalization and abstraction.
- Once the problem is defined clearly, several design methodologies can be applied. An important approach is Top-Down programming design.
- An algorithm is a complete, unambiguous, finite number of logical steps for solving a specific problem.
- An algorithm must possess the following properties - Finiteness, Definiteness, Effectiveness, and Generality.
- To verify if an algorithms really solves the problem for which it is designed one of the following strategies can be used - Experimental analysis (testing) or Formal analysis (proving).
- The importance of efficiency with respect to time was emphasised by Ada Lovelace in 1843 as applying to Charles Babbage's mechanical analytical engine.
- Analysis of algorithm is a field in computer science whose overall goal is an understanding of the complexity of algorithms (in terms of time Complexity & storage (or space) requirement taken by that algorithm.

1.9 Questions for Exercise

1 What are the various steps for Problem-Solving?

2 What are the advantages of top-down design?

3 Give some suggestions to be followed for designing an efficient algorithm.

4 What are the various ways of algorithm analysis?

1.10 Suggested Readings

- ***Algorithms: Design and Analysis****, Harsh Bhasin, 2015*
- ***The Design and Analysis of Computer Algorithms, 6th Edition,*** *Alfred V Aho, John E Hopcroft, 2014*
- **How to Solve it by Computer, *6th Edition,*** *R. G, Dromey, 2001*
- ***Programming Languages*** **(*Second Edition*) ─ *Concepts and Constructs,*** *Ravi Sethi: (Pearson Education, Asia, 1996).*