Unit – 2

Introduction to computer problem solving: Introduction, the problem-solving aspect, top down design, implementation of algorithms, the efficiency of algorithms, the analysis of algorithms.

* * * * *

2.1 Introduction

Computer problem solving can be considered as demanding. It is the process of requiring much thought, careful planning, logical precision, persistence and attention to detail

2.1.1 Programs and Algorithms

The computer solution to a problem is a set of explicit and unambiguous instructions expressed in a programming language. A program can also be considered as an algorithm expressed in a programming language. In general, an algorithm is a solution to a problem that is independent of any programming language.

Once we have program ready, we can obtain the computer solution to a problem by supplying the input or data to the program. The program takes the input, manipulates it and produces output, which is the computer solution to the problem. An algorithm consists of a set of explicit and unambiguous finite steps which, when carried out for a given set of initial conditions, produce the corresponding output and terminate in a finite time.

2.1.2 Requirements for solving problems by computer

In daily life, we employ algorithms to solve our problems. For example, to search for some one's telephone number, we need to employ an algorithm. A telephone directory contains hundreds of thousands of names and telephone numbers yet we have little trouble finding the desired telephone number we are seeking. Why do we have so little difficulty with a problem of great size? Why because, here there is an advantage of the order in the directory to eliminate large sections of the list and home in on the desired name and number. For suppose we want to search for the telephone number of M. Jack, we don't start from page1 and continue till we reach to the name M. Jack. We also do not search for a name whose telephone number is 240240. To do that kind of searching we don't have an approach where we can start from page1 and search number by number, why because the telephone directory is not ordered as per telephone numbers. If we have such kind of provision, then searching is straight forward.

From the above examples, what we can observe is if the data is organized, then the performance of the algorithm will be increased. When the data structure is linked symbolically with an algorithm, we can expect high performance.

2.2 The Problem – Solving Aspect

Problem solving is a creative process which largely defies systemization and mechanization. There is no universal method that has to be followed, for problem solving. Different strategies appear to work for different people.

2.2.1 Problem definition phase

A problem can be solved successfully, only after we understand what the problem is? There will be no useful progress in solving problem until we fully understand what we are trying to solve? This preliminary investigation is known as "Problem Definition Phase". During this phase, one should focus on what we should do rather than How to do it. That means, from the given problem statement, we must try to extract a set of precisely defined tasks. A lot of care must be taken in working out precisely what must be done.

2.2.2 Getting started on a problem

There are many ways to solve most problems and also many solutions to most problems. Because of this problem solving doesn't become easy. Why because, when we have multiple solutions to solve a problem, then it is difficult to recognize which solutions are productive and non-productive.

In most of the scenarios the problem solvers do not have any idea where to start on the problem, even after the problem definition phase. These situations occur, because people are mainly concentrating on implementation details, even before understanding the problem. The best solution is to first understand the problem perfectly and then go for implementation.

Note: The sooner you start coding your program the longer it is going to take

2.2.3 The use of specific examples

When we stuck in where to start, better to use some rules of thumb to get a start with the problem. A best approach that allows us to start on a problem is to pick a specific example of the general problem we wish to solve and try to work out mechanism that will allow us to solve this particular problem.

Ex: If you want to find out the maximum number in a set of numbers, chose a particular set of numbers and work out on the mechanism for finding the maximum in this set.

This approach of focusing on a particular problem can often give us the idea for making a start on the solution to our problem. Ideally, the specifications for our particular problem need to be examined very carefully to try to establish whether or not the proposed algorithm can meet those requirements. If the fill specifications are difficult to formulate sometimes a well-chosen set of test cases can give us a degree of confidence in the generality of our solution.

2.2.4 Similarities among problems

We already saw that one way to make a start on problem by considering the specific examples. Another best way is use of past experience to solve our current problem and it is better to see if there are any similarities between current problem and other problems that we have solved or we have seen solved previously. Sometimes it blocks us from discovering a better solution to a problem so, in some cases better to solve a problem without using past experience.

The more experience on tools and techniques that are required to handle a given problem will help us to solve the problem. It is unlikely that our experience will always help us to solve the problems. As a final analysis, every problem must be considered on its merits. In order to solve a problem a skill that we need to improve is the ability to view a problem from a variety of angles. Once one has developed this skill it should be possible to get started on any problem.

2.2.5 Working backwards from the solution

In some cases, we can assume that we already have the solution to the problem and then try to work backwards to the starting conditions. Even a guess at the solution to the problem may be enough to give us a position to start on the problem. Whatever the attempts that we make to get started on a problem we should write down as we go along the various steps and explorations made.

Once we have solved a problem, consciously reflect back on the way we went about discovering the solution.

2.2.6 General problem solving strategies

There are a number of general and powerful computational strategies that are used in various computer science problems. The problems can be phrased in these strategies and achieve very considerable gains in computational efficiency.

The most popular and widely used one is *Divide and Conquer* strategy. The basic idea is to divide the original problem into two or more sub problems which can be hopefully solved more efficiently by the same technique. The sub problems can be split further till we reach to smaller problems which we can solve without splitting them. This approach is widely used in sorting, selection and searching algorithms It is also possible to apply the divide and conquer strategy in reverse to some problems. The resulting *binary doubling* strategy can give the same sort of gains in computation efficiency.

Another strategy is *dynamic programming.* This strategy can be used when we have to build up a solution to a problem via sequence of intermediate steps. The idea is that a good solution to a large problem can be built up from good or optimal solutions to smaller problems. This can be used in Operations research. Greedy search, backtracking and branch and bound evaluation are all variations on the basic dynamic programming idea. They all tend to guide a computation in such a way that the minimum amount of effort is expended on exploring solutions that have been established to be suboptimal.

2.3 Top down design

The primary goal of computer problem solving is to design an algorithm, which can be implemented as a correct and efficient computer program. Once we have a defined a problem to be solved, and an idea of how to solve it, we can use powerful techniques to design algorithms. Problem solvers generally focus on a very limited span of logic or instructions. A technique for algorithm design that tries to accommodate this human limitation is known as *top-down design* or *step-wise refinement*.

Top-down design approach allows us to build our won solutions to a problem in a step-wise fashion. In this way, specific and complex details of the implementation are encountered only at the stage when we have done sufficient ground work on the overall structure and relationships among the various parts of the problem.

2.3.1 Breaking a problem into subproblems

In top-down design, we take the general statements one at a time, and break them down into a set of more preciously subtasks. These subtasks should more accurately describe how the goal is to be reached. With each splitting of a task into subtasks it is essential that the way in which the subtasks need to interact with each other be preciously defined. The process of repeatedly breaking a task down into subtasks and then each subtask into still smaller subtasks must continue until we end up with the subtasks that can be directly written as program statements. For most of the algorithms we only need to go down to two or three levels, which is not true in large software projects. These kind of solutions works in block-structured languages like Pascal and Algol. A schematic break-down of a problem is shown in the below figure.

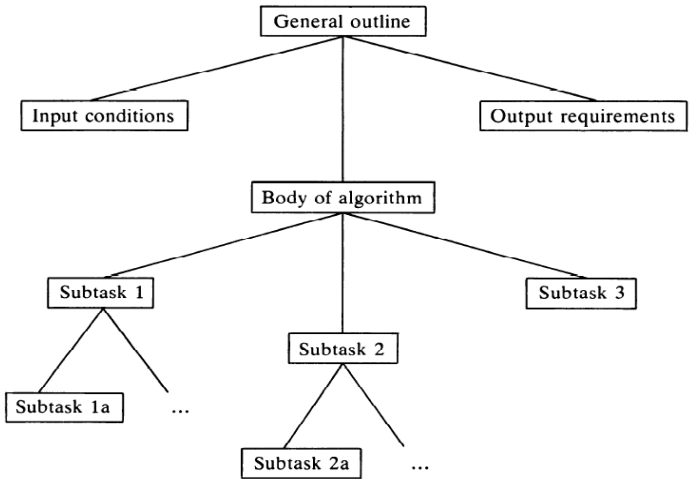


Fig 2.1 Schematic breakdown of a problem into subtasks using top-down design.

2.3.2 Choice of a suitable data structure

One of the most important decision that we have to take, while generating computer solutions to problems is the choice of appropriate data structures. Data structure is a particular way of organizing data in a computer. The developer must choose the appropriate data structure for better performance. If the developer chooses bad data structure, the system does not perform well and it leads to inefficient implementation. On the other hand, an appropriate choice leads to simple, transparent and efficient implementation.

In some problems, the data structures need to be considered at the very beginning stage i.e. even before the top-down design and in some other problems we will consider after knowing the implementation details. Some of the data structures are

- Array
- Stack
- Queue
- Linked List
- Graph
- B-Tree, etc.,

Data structures and algorithms are linked to each other. A small change in data organization may have significant influence on the algorithm required to solve the problem.

2.3.3 Construction of loops

Implementation of sub tasks will have a series of iterative statements or loops along with input/output statements, expressions and assignments make up the heart of program implementations. Loops will provide the facility to execute a portion of the program repetitively, based on a condition

To construct any loop, we must take in to account three things:

(1) Initial conditions: Applied before the loop begins to execute.
(2) The invariant relation: Applied after each iteration of the loop.
(3) Termination condition: Under which the iterative process must terminate.

In constructing loops, the problems that problem solvers face are: Setting the initial conditions correct, setting the loop to execute the right number of times.

2.3.4 Establishing initial conditions for loops

To set initial conditions for loops, the effective strategy is to set the loop variables to the values that they would have to assume in order to solve the smallest problem associated with the loop. Typically, the numbers of iterations n that must be made by a loop are in the range of $0 \le i \le n$. The smallest problem usually corresponds to the case where i either equal 0 or 1.

Ex: Suppose we want to sum a set of numbers in an array using an iterative construct. The loop variables are i and loop index, and s the variable for accumulating the sum of the array elements.

```
sum← 0
i← 0
while i<n
begin
     s  ←  s+a[i]
     i++
end
```

In array, the array index always start with 0 and ends with n-1. So the initial value for i is always start with 0 where it starts form first element in the array.

2.3.5 Finding the iterative construct

Once we have the conditions for solving the smallest problem the next step is to try to extend it to the next smallest problem (in this case when i ← 1).

The solution for n ← 1 is:

$i \leftarrow 1$ Solution for n ← 1

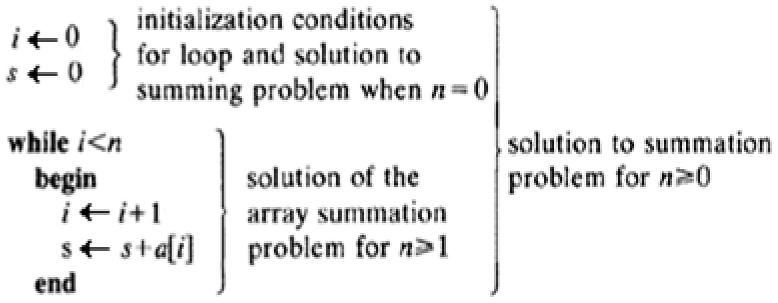
$s \leftarrow a[1]$

This solution for n=1can be built from the solution for n=0 using the values for i and s when n=0 and the two expressions.

$i \leftarrow i+1$ Generalized Solution for n > 0

$s \leftarrow s+a[i]$

The same two steps can be used to extend the solution for n=1, 2, … These two steps extend the solution from the (i-1)th case to ith case(i>=1). They can be therefore used as the basis of our iterative construct for solving the problem for n≥1.



```
i ← 0      } initialization conditions
s ← 0      } for loop and solution to
             summing problem when n = 0

while i<n
  begin            solution of the
    i ← i+1        array summation
    s ← s+a[i]     problem for n≥1
  end
```

solution to summation problem for n≥0

2.3.6 Termination of loops

There are various ways to terminate the loops. The simplest condition for terminating a loop occurs when we know in advance that how much iteration needs to be made. In this case, we can directly use the termination facilities available in programming languages.

```
Ex: while x>0
       begin
           -----
           -----
      end
```

This loop terminates unconditionally after n iterations. A second way in which a loop can be terminated is, when some conditional expression becomes false.

```
Ex: while ((x>0) and (x<10))
      begin
            -----
            -----
       end
```

In these types of loops, we can’t directly determine in advance how many iterations there will be before the loop will terminate. In fact, there is no guarantee that these kinds of loops will terminate at all. The responsibility of loop termination depends on algorithm designer.

The other way in which loop can be terminated is forcing the condition under which the loop will continue to iterate to become false.

2.4 Implementation of algorithms

If an algorithm has been designed properly, the path of execution should flow in a straight line from top to bottom. The programs developed using this approach is much easier to understand debug.

2.4.1 Use of procedures to emphasize modularity

Modularizing the program along with top-down approach allows both the development of implementation and the readability of the main program. Using this approach we can develop independent procedures to perform some specific and well-defined tasks.

One thing to remember is the implementation process should not take too far, to a part at which the implementation becomes difficult to read because of the division. When we are implementing large software projects, a good strategy is to first complete the overall design in top-down approach. Later, the main program is designed in such a way that, it has calls to various procedures that will be needed in the final implementation where all the independent procedures are implemented separately

2.4.2 Choice of variable names

Another implementation detail which makes programs more meaningful and easier to understand is to choose appropriate variables and constant names.

Ex: Suppose, we are dealing with days of week, then instead of using a variable name like a, c, x, better to choose a variable called *day*.

This kind of selection makes programs much more self-documenting. A variable should have only one role in a program. It is also good to define all variables and constants at the start of each procedure.

2.4.3 Documentation of programs

A good programming practice is to always write programs, which can be executed and used by even beginner users with the requirements of the program. This means, the program must provide, the details regarding what responses it requires from the user, during execution. The program should catch incorrect responses to its requests and inform the user in an appropriate manner.

2.4.4 Debugging Programs

While implementing algorithm, it is always necessary to perform some tests to ensure that the program is working correctly according to its specifications. There may be chance of having syntax, logical errors in even small programs, which are undiscovered during compilation phase. When a program is compiled, the compiler discovers syntax errors, which occur when statements are written incorrectly. These compile time errors are easy to fix. The more difficult errors to find and correct are program logic errors, i.e. a program does not perform the task correctly. Some logic errors are obvious immediately upon running the program; the results produced by the program are wrong so the statement that generates those results is suspect.

To make the logical errors identification process easy, it is better to keep some statements in the program which print some additional information to the desired output, which can be made conditionally executable. If we do this, then the need to remove these statements at the time when we are satisfied with the program becomes unnecessary. Another useful technique for debugging programs begins after the program is coded, but before it is compiled and run, and is called a **program trace.** The programmer is executing the program manually using pencil and paper to keep track changes to key variables. The simplest way to do this is, draw a table consisting of steps executed against all the variables used in the program. Then we must execute statements in the section one by one and update our variables as each variable is changed.

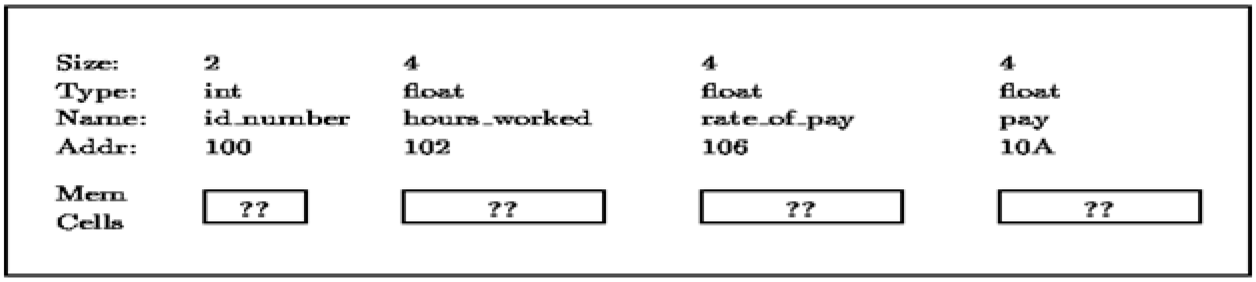

Size: 2 4 4 4
Type: int float float float
Name: id_number hours_worked rate_of_pay pay
Addr: 100 102 106 10A
Mem Cells
??
??
??
??

Fig 2.2 Allocation of Memory cells or objects

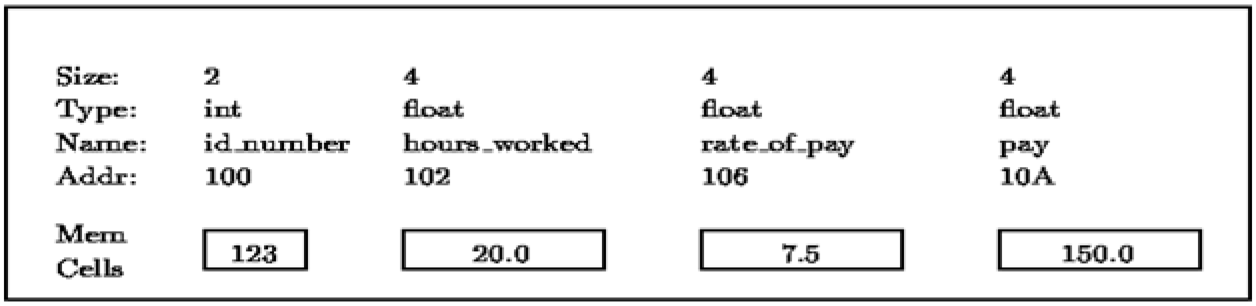

Size: 2 4 4 4
Type: int float float float
Name: id_number hours_worked rate_of_pay pay
Addr: 100 102 106 10A
Mem Cells
123
20.0
7.5
150.0

Fig 2.3 Computation of pay

2.4.5 Program testing

We need to check whether the program solves the smallest possible problem, whether it handles the case when all data values are the same, and so on. These kinds of unusual cases can cause a program to falter.

We should design algorithms to be very general so that they will handle a whole class of problems rather than just one specific case. This causes, in the long run, a lot of wasted effort and time. We should not use fixed constant values, rather we should use variables.

Ex : we should not use statements of the form:

while i < 100 do (while(i<100))

The 100 should always be replaced by a variable i.e.,

while i < n do (while(i<n))

2.5 The efficiency of algorithms

An algorithm is a complete, detailed step by step method for solving a problem independently. Efficiency of an algorithm depends on its design and implementation. Since every algorithm uses computer resources to run, execution time and internal memory usage are important considerations to analyze an algorithm. We need to design algorithms which are economical in terms of usage of CPU time and memory, because of the high cost of computing resources.

2.5.1 Redundant Computations

Many of the inefficiencies arise because of redundant computations or unnecessary storage is used, while implementing the algorithms. They cause serious problems when they are part of loops. The most common mistake using loops is to repeatedly recalculate part of an expression that remains constant throughout the entire execution phase of the loop.

Ex:

```
$x \leftarrow 0$
$i \leftarrow 0$
while $i < n$
begin
    $x \leftarrow x + 0.01$
    $y \leftarrow (a*a*a+c)*x*x+b*b*x$
    print 'x ='x,'y ='y
end
```

```
$a3c \leftarrow (a*a*a+c)$
$b2 \leftarrow b*b$
$x \leftarrow 0$
$i \leftarrow 0$
while $i < n$
begin
    $x \leftarrow x + 0.01$
    $y \leftarrow a3c*x*x+b2*x$
    print 'x ='x,'y ='y
end
```

This loop does twice the number of multiplications necessary to complete the computation. The unnecessary multiplications and additions can be removed by pre-computing two other constants a3c and b2 before executing the loop.

2.5.2 Referencing array elements

If proper care is not taken, redundant computations also cause severe problems in array processing. For example, consider the two versions of an algorithm for finding maximum element and its position in an array.

Version 1:

```
$p \leftarrow 1$
$i \leftarrow 2$
while $i < n$
begin
    if $a[i] > a[p]$ then
        $p \leftarrow i$
end
$max \leftarrow a[p]$
```

Version 2:

```
$p \leftarrow 1$
$max \leftarrow a[1]$
$i \leftarrow 2$
while $i < n$
begin
    if $a[i] > max$ then
        $max \leftarrow a[i]$
        $p \leftarrow i$
end
```

Version 2 is preferred over version1, because the conditional test (array[i]>max) is more efficient to perform than the array[i] > array[pos] in version 1. It is more efficient because to use the variable max only one memory reference instruction is required, where as to use the variable array[pos] requires two memory references first to locate start of 'array' and then to get the value of 'pos' .

2.5.3 Inefficiency due to late termination

Another place where inefficiency can occur into an implementation is due to performing more tests than actually required to solve a problem.

For example, if in the linear search process, all the list elements are checked for a particular element even if the point is reached where it was known that the element cannot occur later. In second example where the inner loop should not proceed beyond n-i.

```
$i \leftarrow 1$
while $i < n-1$
begin
    $j \leftarrow 1$
    while $j < n-1$
    begin
        if $a[j] > a[j+1]$ then
            "exchange $a[j]$ with $a[j+1]$"
    end
end
```

```
$i \leftarrow 1$
while $i < n-1$
begin
    $j \leftarrow n-i$
    while $j < n-1$
    begin
        if $a[j] > a[j+1]$ then
            "exchange $a[j]$ with $a[j+1]$"
    end
end
```

Note: We should always try to take advantage of any order or structure in a problem.

2.5.4 Early detection of desired output conditions

We must include some additional steps and test to detect the conditions for early termination. If they are kept inexpensive then it is worth including them. That is, when early termination is possible, we always have to trade tests and may be even storage to bring about the early termination.

2.5.5 Trading storage for efficiency gains

The trade between storage and efficiency id often used to improve the performance of an algorithm. Sometimes we pre-compute or save some intermediate results and avoid having to do a lot of unnecessary testing and computation later on.

In order to speed up the algorithm, we need to use least number of loops. This makes programs much harder to read and debug. So better to stick to the rule of having one loop do one job

2.6 The analysis of algorithms

The good solution to a problem is economical in the use of computing and human resources. The below are the qualities and capabilities of a good algorithm:

- Simple but powerful and general solutions.
- Can be easily understood by others, i.e., implementation is easy clear and concise.
- Can be easily modified, if needed.
- Economical in terms of usage of computer time, storage and peripherals.
- Documented
- Can be used as a sub procedure for other problems.
- Can run on any computer

We also require some quantitative measures to complete the evaluation of goodness of an algorithm. Using these, we can predict the performance of an algorithm and compare it with other algorithms which can solve the same problem.

2.6.1 Computational Complexity

In order to evaluate the quantitative measure of an algorithm's performance, we need to set up a computational model, which reflects its behavior under specified input conditions. We can measure an algorithm's performance in terms of the size of the problem being solved (n). More computing resources are needed to solve the large problems. As 'n' increases, the cost increases.

Ex: Problem for n=200 takes twice as long as a problem for n =100.

The below table illustrates the comparative cost for a range of n values.

$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n
1	2	2	4	8	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	132,877	10^8	10^{12}	$>>10^{2500}$

Table 2.1 Computational cost as a function of problem size for a range of computational complexities.

2.6.2 The order notation

The Order notation is a standard notation developed to represent functions which bound the computing time for algorithms. It is usually referred as *"O-notation"*.

An algorithm in which the dominant mechanism is executed cn^2 times for c a constant, and n the problem size, is said to have an order n^2complexity which is written as $O(n^2)$.

f(n) = O(g(n)) if there exist constants c > 0 and N such that

f(n) ≤ c*g(n) for all n ≥ N

holds for all the values of n that are finite and positive.

Ex: If an algorithm requires ($3n^2+6n+3$) comparisons to complete its task. Then,

$f(n) = 3n^2+6n+3$

We can say the above algorithm has an asymptotic complexity of $O(n^2)$.

2.6.3 Worst and average case behavior

To analyze any given algorithm three measures of performance are actually considered: The best, *worst* and *average* case behavior of the algorithm.

- The worst-case complexity of the algorithm is the function defined by the maximum number of steps taken on any instance of size n and worst case time is W(n) = O(n)
- The best-case complexity of the algorithm is the function defined by the minimum number of steps taken on any instance of size n and best case time is B(n) = O(1).
- Finally, the average-case complexity of the algorithm is the function defined by the average number of steps taken on any instance of size n and average case time is A(n) = $\frac{n+1}{2}$.

2.6.4 Probabilistic average case analysis

- Suppose we wish to characterize the behavior of an algorithm that linearly searches an ordered list of elements for some value x.
- In best case, the algorithm will examine only starting value before it terminates where as in the worst case, the algorithm should examine all n values in the list before terminating.

0	1	2	3	4	5
8	5	4	7	9	10

  - Best case = searching key element present at first index i.e., key value is 8 and best case time is B(n) = O(1).
  - Worst case=searching key element present at last index i.e., key value is 10 and worst case time is W(n) = O(n).
- For probabilistic average case analysis, we assume that all possible cases of termination are equally likely, i.e., the probability that x will be found at position 1 is 1/n, and at position 2 is 1/n and so on. So, the average search cost is the sum of all possible search costs.

1	2	3	- -	n

$$Average\ case = \frac{all\ possible\ case\ time}{no\ of\ \ cases}$$

$$Average\ \text{time} = \frac{1+2+3+--+n}{n} = \frac{n+1}{2}$$

Ex: if n=5, the average search cost = $\frac{1+2+3+4+5}{5} = 3$

Fundamental algorithms: Exchanging the values of two variables, counting, summation of a set of numbers, factorial computation, sine function computation, generation of the Fibonacci sequence, reversing the digits of an integer.

* * * * *

2.7 Exchanging the values of two variables

Problem Definition: Given two variables, *x* and *y*, exchange the values assigned to them.

Algorithm Development:

To define the problem more clearly, consider the below example. Let us say, two variables x and y contains two different values as shown below.

Starting Configuration: x y

$$\boxed{245} \qquad \boxed{356}$$

This means that memory cell or variable *x* contains value 245 and memory cell or variable *y* contains value 356. Our intention is to replace *x* value with 356 and *y* value with 245. Our requirement is as shown below.

Target Configuration: x y

$$\boxed{356} \qquad \boxed{245}$$

In order to change the value of a variable we can use the assignment operator ← . We want to assign the value held by *y* to *x* and value held by *x* to *y*. We could perhaps make the exchange with the below assignment statements.

$x \leftarrow y$ …………….. (1)

and $y \leftarrow x$ …………….. (2)

In (1) value stored by variable *y* will be copied into variable *x*.

When work through these two steps, to make sure they have the desired effect.

Initially, x y

$$\boxed{245} \qquad \boxed{356}$$

After executing, $x \leftarrow y$ statement,

x y

$$\boxed{356} \qquad \boxed{356}$$

The assignment (1) has changed the value of *x* but has left the value of *y*, unchanged. As per our target configuration, we can see that now *x* contains the value of *y*.

After executing, $y \leftarrow x$ statement,

x y

$$\boxed{356} \qquad \boxed{356}$$

When we executed step (2), x is not changed while y takes on the value that currently belongs to x. whatever the result we obtained, doesn't meet our requirement. The problem in this approach is, when we executed the statement,

$$x \leftarrow y.$$

We have lost the value that originally belonged to x. The value which has lost is supposed to be the final value of y.

So, we need to redefine the problem as follows:

New value of $x \leftarrow$ Old value of y

New value of $y \leftarrow$ Old value of x

What we have done in our present approach is that,

New value of $y \leftarrow$ New value of x

So, we need to come up with a solution, in which we should not destroy the old value of x, when we make the assignment,

$$x \leftarrow y$$

The solution is to use a temporary variable t, and copy the original value of x in to this variable before executing step (1).

Solution:

$$t \leftarrow x$$

$$x \leftarrow y$$

After these two steps, we have

x	*t*	*y*
356	245	356

Now we can observe that, we have still old value of x, stored in t, which is to be assigned to y.

$$y \leftarrow t$$

After execution of this step, we have:

x	*t*	*y*
356	245	245

Now X and Y values are interchanged as per our target configuration.

Algorithm:

```
Step 1: Start.
Step 2: Get the values of X and Y
Step 3: Save the original value of X in T
Step 4: Assign the original value of Y to X.
Step 5: Assign the original value of X, which is stored in T, to Y
Step 6: Stop
```

Applications: Sorting Algorithms

2.8 Counting

Problem Definition: Given a set of n student's examination marks (in the range of 0 to 100) make a count of number of students that passed in the examination. A pass is awarded for all marks of 50 and above.

Algorithm Development:

Suppose that we are given a set of marks,

71, 42, 88, 65, 31, 56, 91

In order to find out how many no. of students passed in this set, we need to start from left, examine the first mark (i.e. 71) , see if it is >= 50, and remember that one student has passed so far. The second mark is then examined and count of no of students passed will not be changed. When we arrive at third mark (i.e. 88), we see that it is >= 50, so add one to the previous count. We should repeat this process until all marks have been tested.

Order in which marks are examined	**Marks**	**Test Condition** (If Marks > = 50 then add 1 to the count else do not change the count value)	**Counting details for passes.**
↓	71	71 > = 50 (True)	Previous Count = 0, Current Count = 1
	42	42 > = 50 (False)	Previous Count = 1, Current Count = 1
	88	88 > = 50 (True)	Previous Count = 1, Current Count = 2
	65	65 > = 50 (True)	Previous Count = 2, Current Count = 3
	31	31 > = 50 (False)	Previous Count = 3, Current Count = 3
	56	56 > = 50 (True)	Previous Count = 3, Current Count = 4
	91	91 > = 50 (True)	Previous Count = 4, Current Count = 5

Therefore, Number of students passed = 5

After each mark has been processed the current count reflects the number of students that have passed in the marks list, so far we have examined.

From the example, we can see that every time, we need to increase the count based on the previous value. That is,

current_count $\leftarrow$ previous_count + 1

For example,

When we arrive at mark 56, the Previous_count = 3

Current count therefore becomes 4. Simillarly, when we get to the next mark (i.e. 91), the current_count of 4 must assume the role of previous_count.This means that whenever a new current_count is generated it must then assume the role of previous_count, before the next mark is considered.

current_count ← previous_count + 1 …………. (1)

previous_count ← current_count …………. (2)

These two steps can be repeatedly applied to obtain the count required. In conjunction with the conditional test and input of the next mark we execute step (1), followed by step (2). Followed by step (1), followed by step (2) and so on..

When we substitute step (2) in step (1), we get

current_count ← current_count + 1

(new value) (old value)

Note : Initially, count ← 0

While less than n marks have been examined do

(a) Read the next mark
(b) If current mark satisfies pass requirement then add one to count.

Algorithm:

```
Step 1: Start
Step 2: Prompt then read the number of marks to be processed.
Step 3: While there are still marks to be processed repeatedly, do
        (a). read the next mark.
        (b). if it is pass (.i.e >=50) then add one to the count
Step 4: Write out the total number of passes.
Step 5: Stop
```

Applications: All forms of counting

2.9 Summation of a set of numbers

Problem Definition: Given a set of n numbers design an algorithm that adds these numbers and returns the resultant sum. Assume n is greater than or equal to zero.

Algorithm Development:

In a manual approach, we write the numbers down one under the other and start adding up the right-hand column.

For example, consider the addition of 992, 891, 766.

```
992
891
766
___
... 9
___
```

In designing an algorithm, to do this task, we need to follow a different approach. The computer has a built-in device which accepts two numbers to be added, performs the addition, and returns the sum of the two numbers.

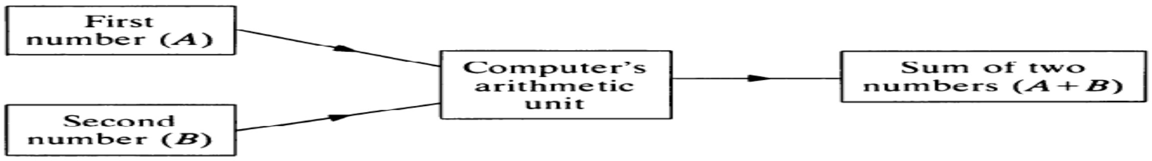


Fig 2.4 Schematic mechanism for computer's addition process.

The simplest way to instruct the computer's arithmetic unit to add a set of numbers is to write down an expression that specifies the addition that we wish to be performed. For the above three numbers, we can write the expression as follows:

$s \leftarrow 992 + 891 + 766$ ………………….(1)

The value resulted from the expression (1) at the RHS after the evaluation will be stored in the memory cell or variable called S.

Expression (1) adds three specific numbers as required. Suppose we want to perform the addition of three other numbers, we need to write a different program statement.

The better solution is to replace all the constants with variables. The expression (1) will become

$s \leftarrow a + b + c$ ----------------------(2)

Expression (2) adds any three numbers provided they have been previously assigned as values or contents of a, b and c respectively. The second approach also has drawback that, it adds only three numbers.

But our intention is to design a more general algorithm which can take n number of numbers and add them, where n can take a wide range of values. To solve this conventionally, we can write as:

$$s \leftarrow (a_1 + a_2 + a_3 + a_4 + ... + a_n) \qquad (3)$$

(or)

$$s \leftarrow \sum_{i \leftarrow 1}^{n} a_i \qquad (4)$$

Note: Computers can do repetitive things and computer's adding device can only add two numbers at a time.

The above problem can be solved using the following approach:

Start by adding first two numbers $a_1$and a_2 . That is,

$$s \leftarrow a_1 + a_2 \qquad (1)$$

We could then proceed by adding a_3 to the s computed in step (1).

$$s \leftarrow s + a_3 \qquad (2)$$

In a similar manner,

$$\begin{array}{ll} s \leftarrow s + a_4 & \\ s \leftarrow s + a_5 & \\ . \quad . \quad . & (3,....,n-1) \\ . \quad . \quad . & \\ . \quad . \quad . & \\ s \leftarrow s + a_n & \end{array}$$

From step (2) onwards we are actually repeating the same process over and over – the only difference is that values of a ad s change with each step. For general ith step, we have

$$s \leftarrow s + a_i \qquad (i)$$

This general step can be placed in a loop to iteratively generate the sum of n numbers. The algorithm should work for all values of n >=0

Algorithm:

```
Step 1: Start
Step 2: Prompt and read the 'n' numbers that need to be summed up
Step 3: Initialize sum for zero numbers
Step 4: While less than n numbers have been summed repeatedly do
          (a). read in next number
          (b). add the number to the previous sum to get the current sum
Step 5: Write the value of sum of n numbers
```

Applications: Average calculations, Variance and least squares calculations

2.10 Factorial computation

Problem Definition: Given a number n, compute n factorial (n!) where n>=0.

Algorithm Development:

We know that n! can be written as,

$$n! = 1 \times 2 \times 3 \times ...... \times (n-1) \times n \qquad \text{for } n>=1$$

and by definition

$$0! = 1$$

One important thing to remember is that the computer's arithmetic unit can only multiply two numbers at a time.

Applying the factorial definition we get

$0! = 1$

$1! = 1\times1$

$2! = 1\times2$

$3! = 1\times2\times3$

$4! = 1\times2\times3\times4$

$5! = 1\times2\times3\times4\times5$

.

.

From the above we can generalize the following thing, for n>=1,

$$n! = n \times (n-1)! \qquad \text{for } n >=1$$

Using this definition,

$1! = 1 \times 0!$

$2! = 2 \times 1!$

$3! = 3 \times 2!$

$4! = 4 \times 3!$

$5! = 5 \times 4!$

.

.

If we assume fact = 0!=1 , then

fact $\leftarrow$ 1	=	0!
fact $\leftarrow$ fact*1	=	1!
fact $\leftarrow$ fact*2	=	2!
fact $\leftarrow$ fact*3	=	3!
fact $\leftarrow$ fact*4	=	4!

.

.

From step (2) onwards, the same process is repeated over and over. For general (i+1)th step we have

fact $\leftarrow$ fact * i (i+1)

This step should be placed in a loop to iteratively generate n!. The instance where n = 0 is a special case which must be accounted for directly by the assignment.

fact $\leftarrow$ 1 (by definition of 0!)

Algorithm:

```
Step 1: Start
Step 2: Prompt and read the ‘n’, whose factorial is to be computed
Step 3: Set product ‘fact’ for 0!(special case). Also set product count to 0.
Step 4: While less than n products have been calculated repeatedly do
(a)     increment product count
(b)     compute the ith product fact by multiplying i by the most recent product
Step 5: Print the result n!
Step 6: Stop
```

Applications: Probability, statistical and mathematical computations

2.11 Sine function computation

Problem Definition: Design an algorithm to evaluate the function sin(x) as defined by the infinite series expansion.

$$\sin(x) = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + ...$$

Algorithm Development:

From the sin(x) expression, we see that powers and factorials are following the sequence.

1, 3, 5, 7, ………..

are required. We can generate this odd sequence by starting with 1and successively adding 2. Our other problem is to compute the general term $\frac{x^i}{i!}$, which can be expressed as:

$$\frac{x^i}{i!} = \frac{x}{1} \times \frac{x}{2} \times \frac{x}{3} \times \frac{x}{4} \times ..... \times \frac{x}{i} \qquad \text{for } i \geq 1$$

The following steps can be used to compute the expression:

```
$fp \leftarrow 1$
$j \leftarrow 0$
while $j < i$
begin
    $j \leftarrow j + 1$
    $fp \leftarrow fp * \frac{x}{j}$
end
```

The algorithm can be completed by implementing the additions and subtractions and making the appropriate termination. With this approach each term is computed by starting at the smallest j value and working upwards.

$$\frac{x^3}{3!} = \frac{x*x*x}{3*2*1} \qquad \frac{x^2}{3*2} \quad \frac{x^1}{1!} \qquad i=3$$

$$\frac{x^5}{5!} = \frac{x*x*x*x*x}{5*4*3*2*1} \qquad \frac{x^2}{5*4} \quad \frac{x^3}{3!} \qquad i=5$$

$$\frac{x^7}{7!} = \frac{x*x*x*x*x*x*x}{7*6*5*4*3*2*1} \qquad \frac{x^2}{7*6} \quad \frac{x^5}{5!} \qquad i=7$$

Each of the terms $\frac{x^2}{3*2}, \frac{x^2}{5*4}, \frac{x^2}{7*6}$ …. can be described by the general term:

$$\frac{x^2}{i(i-1)} \qquad \text{for } i = 3, 5, 7, \ldots\ldots$$

Therefore, to generate consecutive terms of the sine series we can use:

$$\text{current } i^{th} \text{ term} = \frac{x^2}{i(i-1)} * \text{(previous term)}$$

$$\text{sign} \leftarrow -\text{sign}$$

will generate alternative positive and negative terms. The initial conditions are therefore

$$\begin{aligned} &tsin \leftarrow x \\ &term \leftarrow x \\ &i \leftarrow 1 \end{aligned}$$

The i^{th} term and summation which can be generated iteratively from their predecessors are:

$$\begin{aligned} &i \leftarrow i+2 \\ &term \leftarrow -\frac{term * x * x}{i*(i-1)} \\ &tsin = tsin + term \end{aligned}$$

We now have an effective iterative mechanism for generating successive terms of the sine function. We need to evaluate sin(x) to a finite number of times. x is in range of $-1 \le x \le 1$. Fix an acceptable error level (1×10^{-6}) and generate successive terms until the contribution of the current term is less than the acceptable error.

Algorithm:

```
Step 1: Start
Step 2: Setup initial conditions for the first term that can't be computed iteratively.
Step 3: While the absolute value of current term is greater than the acceptable error do
    a. Identify the current i^th term.
    b. Generate the current term from its predecessor.
    c. Add current term with the appropriate sign to the accumulated sum for the sine
       function
Step 4: Stop
```

Applications: Mathematical and statistical computations.

2.12 Generation of the Fibonacci sequence

Problem Definition: Generate and print the first 'n' terms of the Fibonacci sequence, where n>=1. The first few terms are: 0, 1, 1, 2, 3, 5, 8, 13, ………………

Each term beyond the first two is derived from the sum of its two nearest predecessors.

Algorithm Development:

From the definition, given that:

New term ← preceding term + term before preceding term

Let us define:

a as the term before the preceding term

b as the preceding term

c new term

Then to start with we have:

	a←0	first Fibonacci number
	b←1	second Fibonacci number
and	c←a + b	third Fibonacci number (from definition)

When the new term c has been generated we have the third Fibonacci number. To generate the fourth or next Fibonacci number, we need to apply the same definition again. But, before that we need to make some adjustments. The fourth Fibonacci number is derived from the sum of second and third Fibonacci numbers. That means, the second Fibonacci number becomes the term before the preceding term and third Fibonacci number becomes preceding term.

That is,

```
a←0      [1]   term before preceding term
b←1      [2]   preceding term
c←a+b    [3]   new term
a←b      [4]   term before preceding term becomes preceding term
b←c      [5]   preceding term becomes new term
```

To generate next Fibonacci number iterate the steps from 3 to 5 (for n>=2)

Algorithm:

```
Step 1: Start
Step 2: Prompt and read n , the number of Fibonacci numbers to be generated.
Step 3: Assign the first two Fibonacci numbers 0 and 1 to a and b respectively
Step 4: Initialize the count of number generated
Step 5: While less than 'n' Fibonacci numbers have been generated do
        (a) write out next two Fibonacci numbers
        (b) generate next Fibonacci number 'c' from a & b
        (c) Store b value to a
        (d) Store c value to b
        (e) Update count of number of Fibonacci numbers generated, i.
Step 6: Stop
```

Applications: Botany, Electrical Network Theory, sorting and searching

2.13 Reversing the digits of an integer

Problem Definition: Design an algorithm that accepts a positive integer and reverses the order of its digits.

Algorithm Development:

Digit reversal technique is used to remove bias from a set of numbers. The problem definition can be clearly explained as below:

Input : 2345

Output : 5432

In order to solve the problem we need to access individual digits of the given number. The number can be rewritten as follows:

$$2\times10^3+3\times10^2+4\times10^1+5\times10^0$$

To access the digits individually, we need to start from one end and goes to other end. But the question is at which end we should start? Why because visually we can say how many digits we have in given number, but programmatically difficult. So better to establish the identity of the Least Significant Digit (rightmost). To do this we need to chop off the Least Significant Digit in the number. Which means, in the given number 5 should be removed and we should end up with 234.

We can get the number 234 by integer division of the original number by 10.

i.e. 2345 div 10 $\longrightarrow$ 234

This chops off 5 but does not allow us to save it. Here 5 is the remainder we get when we divide 2345 by 10. To get this remainder we can use the mod function.

i.e. 2345 mod 10 $\longrightarrow$ 5

Therefore if we apply the below steps, we get the digit 5 and the number 234.

$r \leftarrow n \bmod 10$ (1) (r=5)

$n \leftarrow n \text{ div } 10$ (2) (n=234)

Applying the same steps to the new value of n, we can obtain digit 4. Using this approach we can iteratively access the individual digits of the given number.

The next thing we need to find out is how to carry out the digit reversal. The first two digits till now we got are 5 and 4, they should appear in the final output as below:

5 followed by 4 (or 54)

If the original number was 45 then we could obtain its reverse by first extracting the 5, multiplying 10, and then adding 4 to give 54.

i.e. $5\times10 + 4 \longrightarrow 54$

Let us assume that the variable *dreverse* is to be used to build the reversed integer. At each stage in building the reversed integer its previous value is used in conjunction with the most recently extracted digit.

The above approach can be explained in terms of variable *dreverse*,

Iteration	**Value of dreverse**
[1] dreverse ← dreverse*10 + 5	5
[2] dreverse ← dreverse*10 + 4	54
[3] dreverse ← dreverse*10 + 3	543
[4] dreverse ← dreverse*10 + 2	5432

Therefore to build the reversed integer we can use the construct:

dreverse ← (previous value of *dreverse*)*10 + (most recently extracted rightmost digit)

dreverse variable can be used on both sides of this expression. Initially *dreverse* must be zero.

The process should be terminated as soon as all digits have been extracted and processed. With each iteration the number of digits in the number being reversed is reduced by one, yielding the sequence shown in the below table.

Number being reversed	**Reversed number being constructed**	**Step**
2345	5	[1]
234	54	[2]
23	543	[3]
2	5432	[4]

Algorithm:

Step 1: Start

Step 2: Read 'n', the positive integer to be reversed

Step 3: Set the initial condition for the reversed integer *dreverse*

Step 4: While the integer being reversed is > 0 do

(a) use the remainder function to extract the rightmost digit of the number being reversed.
(b) increase the previous reversed integer representation *dreverse*by a factor of 10 and add to it the most recently extracted digit to give the current *dreverse* value.
(c) use integer division by 10 to remove the rightmost digit from the number being reversed.

Step 5: Stop

Applications: Hashing and information retrieval, data base applications

Exercise

1. Given two glasses marked A and B. Glass A is full of milkshake and glass B is full of mango. Suggest a way of exchanging the contents of glasses A and B.
2. Design an algorithm that makes the following exchanges: $$\begin{tikzcd} a \arrow[r] & b \arrow[r] & c \arrow[ll, bend left] \end{tikzcd}$$
3. Given two variables of integer type *a* and *b*, exchange their values without using a third temporary variable.
4. Design an algorithm that reads a list of numbers and makes a count of the numbers of negatives and the number of non – negatives members in the set.
5. Redesigns an algorithm so that it only needs to perform n-1 addition to sum n numbers.
6. Develop an algorithm to compute the sum of the first n terms of the series: s=1-3+5-7+…..
7. The binomial theorem of basic algebra indicates that the coefficient ${}^{n}C_{r}$ of the r^{th} power of x in the expansion of $(x+1)^{n}$ is given by ${}^{n}C_{r} = \frac{n!}{r!(n-r)!}$. Design an algorithm that evaluates all coefficients of x for a given value of n.
8. Design an algorithm to find the sum of the first n terms of the series:
$$f_s = 0! + 1! + 2! + 3! + ... + n! \qquad (n \geq 0)$$
9. The first few numbers of the Lucas sequence which is a variation on the Fibonacci sequence are: 1, 3, 4, 7, 11, 18, 29, …….
10. Design an algorithm that reads in a set of n single digits and converts them into a single decimal integer. (set of 5 digits {2, 3, 9, 5, 7} to the integer is 23957).