LECTURE NOTES ON
DESIGN AND ANALYSIS OF ALGORITHMS

Revision 4

July 2013

L. V. NARASIMHA PRASAD
Professor and Head
Department of Computer Science and Engineering

VARDHAMAN COLLEGE OF ENGINEERING
Shamshabad,Hyderabad– 501 218

Chapter 1

Basic Concepts

1.0. Algorithm

An Algorithm is a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time. No matter what the input values may be, an algorithm terminates after executing a finite number of instructions. In addition every algorithm must satisfy the following criteria:

> Input: there are zero or more quantities, which are externally supplied;
>
> Output: at least one quantity is produced;
>
> Definiteness: each instruction must be clear and unambiguous;
>
> Finiteness: if we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after a finite number of steps;
>
> Effectiveness: every instruction must be sufficiently basic that it can in principle be carried out by a person using only pencil and paper. It is not enough that each operation be definite, but it must also be feasible.

In formal computer science, one distinguishes between an algorithm, and a program. A program does not necessarily satisfy the fourth condition. One important example of such a program for a computer is its operating system, which never terminates (except for system crashes) but continues in a wait loop until more jobs are entered.

We represent algorithm using a pseudo language that is a combination of the constructs of a programming language together with informal English statements.

1.1. Performance of a program:

The performance of a program is the amount of computer memory and time needed to run a program. We use two approaches to determine the performance of a program. One is analytical, and the other experimental. In performance analysis we use analytical methods, while in performance measurement we conduct experiments.

Time Complexity:

The time needed by an algorithm expressed as a function of the size of a problem is called the *time complexity* of the algorithm. The time complexity of a program is the amount of computer time it needs to run to completion.

The limiting behavior of the complexity as size increases is called the asymptotic time complexity. It is the asymptotic complexity of an algorithm, which ultimately determines the size of problems that can be solved by the algorithm.

Space Complexity:

The space complexity of a program is the amount of memory it needs to run to completion. The space need by a program has the following components:

Instruction space: Instruction space is the space needed to store the compiled version of the program instructions.

Data space: Data space is the space needed to store all constant and variable values. Data space has two components:

- Space needed by constants and simple variables in program.
- Space needed by dynamically allocated objects such as arrays and class instances.

Environment stack space: The environment stack is used to save information needed to resume execution of partially completed functions.

Instruction Space: The amount of instructions space that is needed depends on factors such as:

- The compiler used to complete the program into machine code.
- The compiler options in effect at the time of compilation
- The target computer.

1.2. Algorithm Design Goals

The three basic design goals that one should strive for in a program are:

1. Try to save Time
2. Try to save Space
3. Try to save Face

A program that runs faster is a better program, so saving time is an obvious goal. Like wise, a program that saves space over a competing program is considered desirable. We want to "save face" by preventing the program from locking up or generating reams of garbled data.

1.3. Classification of Algorithms

If 'n' is the number of data items to be processed or degree of polynomial or the size of the file to be sorted or searched or the number of nodes in a graph etc.

1 Next instructions of most programs are executed once or at most only a few times. If all the instructions of a program have this property, we say that its running time is a constant.

Log n When the running time of a program is logarithmic, the program gets slightly slower as n grows. This running time commonly occurs in programs that solve a big problem by transforming it into a smaller problem, cutting the size by some constant fraction., When n is a million, log n is a doubled. Whenever n doubles, log n increases by a constant, but log n does not double until n increases to n^2.

n When the running time of a program is linear, it is generally the case that a small amount of processing is done on each input element. This is the optimal situation for an algorithm that must process n inputs.

n. log n This running time arises for algorithms that solve a problem by breaking it up into smaller sub-problems, solving then independently, and then combining the solutions. When n doubles, the running time more than doubles.

n^2 When the running time of an algorithm is quadratic, it is practical for use only on relatively small problems. Quadratic running times typically arise in algorithms that process all pairs of data items (perhaps in a double nested loop) whenever n doubles, the running time increases four fold.

n^3 Similarly, an algorithm that process triples of data items (perhaps in a triple–nested loop) has a cubic running time and is practical for use only on small problems. Whenever n doubles, the running time increases eight fold.

2^n Few algorithms with exponential running time are likely to be appropriate for practical use, such algorithms arise naturally as "brute–force" solutions to problems. Whenever n doubles, the running time squares.

1.4. Complexity of Algorithms

The complexity of an algorithm M is the function f(n) which gives the running time and/or storage space requirement of the algorithm in terms of the size 'n' of the input data. Mostly, the storage space required by an algorithm is simply a multiple of the data size 'n'. Complexity shall refer to the running time of the algorithm.

The function f(n), gives the running time of an algorithm, depends not only on the size 'n' of the input data but also on the particular data. The complexity function f(n) for certain cases are:

1. Best Case : The minimum possible value of f(n) is called the best case.
2. Average Case : The expected value of f(n).
3. Worst Case : The maximum value of f(n) for any key possible input.

The field of computer science, which studies efficiency of algorithms, is known as analysis of algorithms.

Algorithms can be evaluated by a variety of criteria. Most often we shall be interested in the rate of growth of the time or space required to solve larger and larger instances of a problem. We will associate with the problem an integer, called the size of the problem, which is a measure of the quantity of input data.

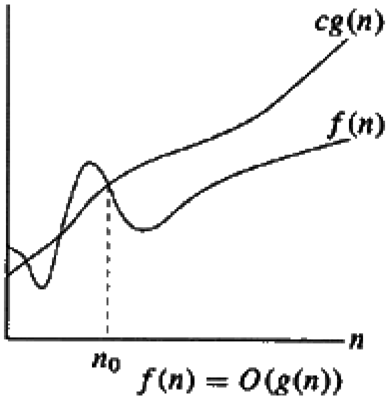
1.5. Rate of Growth:

The following notations are commonly use notations in performance analysis and used to characterize the complexity of an algorithm:

1. Big–OH (O) [1],
2. Big–OMEGA (Ω),
3. Big–THETA (θ) and
4. Little–OH (o)

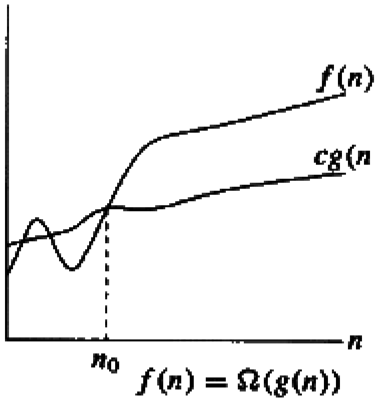
Big–OH O (Upper Bound)

f(n) = O(g(n)), (pronounced order of or big oh), says that the growth rate of f(n) is less than or equal ($\leq$) that of g(n).



Big–OMEGA Ω (Lower Bound)

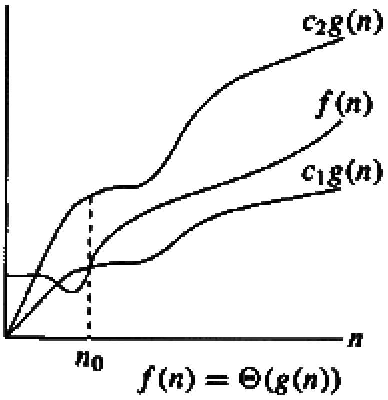
f(n) = Ω(g(n)) (pronounced omega), says that the growth rate of f(n) is greater than or equal to ($\geq$) that of g(n).



Big–THETA θ (Same order)

[1] In 1892, P. Bachmann invented a notation for characterizing the asymptotic behavior of functions. His invention has come to be known as *big oh notation*.

f(n) = Θ(g(n)) (pronounced theta), says that the growth rate of f(n) equals (=) the growth rate of g(n) [if f(n) = O(g(n)) and T(n) = Ω (g(n)].



Little–OH (o)

T(n) = o(p(n)) (pronounced little oh), says that the growth rate of T(n) is less than the growth rate of p(n) [if T(n) = O(p(n)) and T(n) ≠ θ(p(n))].

1.6. Analyzing Algorithms

Suppose 'M' is an algorithm, and suppose 'n' is the size of the input data. Clearly the complexity f(n) of M increases as n increases. It is usually the rate of increase of f(n) we want to examine. This is usually done by comparing f(n) with some standard functions. The most common computing times are:

O(1), O($\log_2$ n), O(n), O(n. $\log_2$ n), O(n^2), O(n^3), O(2^n), n! and n^n

Numerical Comparison of Different Algorithms

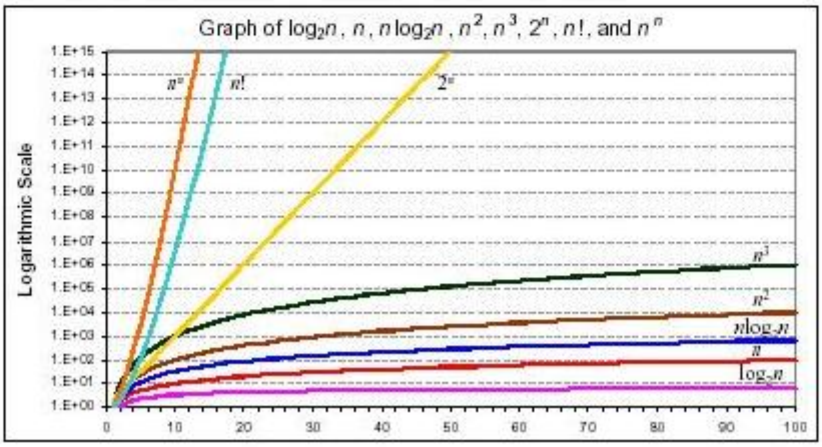
The execution time for six of the typical functions is given below:

n	**$\log_2$ n**	**n*$\log_2$n**	**n^2**	**n^3**	**2^n**
1	0	0	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4096	65,536
32	5	160	1024	32,768	4,294,967,296
64	6	384	4096	2,62,144	Note 1
128	7	896	16,384	2,097,152	Note 2
256	8	2048	65,536	1,677,216	????????

Note1: The value here is approximately the number of machine instructions executed by a 1 gigaflop computer in 5000 years.

Note 2: The value here is about 500 billion times the age of the universe in nanoseconds, assuming a universe age of 20 billion years.

Graph of log n, n, n log n, n^2, n^3, 2^n, n! and n^n



O(log n) does not depend on the base of the logarithm. To simplify the analysis, the convention will not have any particular units of time. Thus we throw away leading constants. We will also throw away low–order terms while computing a Big–Oh running time. Since Big-Oh is an upper bound, the answer provided is a guarantee that the program will terminate within a certain time period. The program may stop earlier than this, but never later.

One way to compare the function f(n) with these standard function is to use the functional 'O' notation, suppose f(n) and g(n) are functions defined on the positive integers with the property that f(n) is bounded by some multiple g(n) for almost all 'n'. Then,

$$f(n) = O(g(n))$$

Which is read as "f(n) is of order g(n)". For example, the order of complexity for:

- Linear search is O (n)
- Binary search is O (log n)
- Bubble sort is O (n^2)
- Merge sort is O (n log n)

1.7. The rule of sums

Suppose that $T_1(n)$ and $T_2(n)$ are the running times of two programs fragments P_1 and P_2, and that $T_1(n)$ is O(f(n)) and $T_2(n)$ is O(g(n)). Then $T_1(n) + T_2(n)$, the running time of P_1 followed by P_2 is O(max f(n), g(n)), this is called as rule of sums.

For example, suppose that we have three steps whose running times are respectively O(n^2), O(n^3) and O(n. log n). Then the running time of the first two steps executed sequentially is O (max(n^2, n^3)) which is O(n^3). The running time of all three together is O(max (n^3, n. log n)) which is O(n^3).

1.8. The rule of products

If $T_1(n)$ and $T_2(n)$ are O(f(n) and O(g(n)) respectively. Then $T_1(n)*T_2(n)$ is O(f(n) g(n)). It follows term the product rule that O(c f(n)) means the same thing as O(f(n)) if 'c' is any positive constant. For example, $O(n^2/2)$ is same as $O(n^2)$.
Suppose that we have five algorithms A_1–A_5 with the following time complexities:

A_1 :	n
A_2 :	n log n
A_3 :	n^2
A_4:	n^3
A_5:	2^n

The time complexity is the number of time units required to process an input of size 'n'. Assuming that one unit of time equals one millisecond. The size of the problems that can be solved by each of these five algorithms is:

Algorithm	Time complexity	Maximum problem size		
		1 second	1 minute	1 hour
A_1	n	1000	6×10^4	3.6×10^6
A_2	n log n	140	4893	2.0×10^5
A_3	^{n2}	31	244	1897
A_4	^{n3}	10	39	153
A_5	2^n	9	15	21

The speed of computations has increased so much over last thirty years and it might seem that efficiency in algorithm is no longer important. But, paradoxically, efficiency matters more today then ever before. The reason why this is so is that our ambition has grown with our computing power. Virtually all applications of computing simulation of physical data are demanding more speed.

The faster the computer run, the more need are efficient algorithms to take advantage of their power. As the computer becomes faster and we can handle larger problems, it is the complexity of an algorithm that determines the increase in problem size that can be achieved with an increase in computer speed.

Suppose the next generation of computers is ten times faster than the current generation, from the table we can see the increase in size of the problem.

Algorithm	Time Complexity	Maximum problem size before speed up	Maximum problem size after speed up
A_1	n	S1	10 S1
A_2	n log n	S2	≈10 S2 for large S2
A_3	^{n2}	S3	3.16 S3
A_4	^{n3}	S4	2.15 S4
A_5	2^n	S5	S5 + 3.3

Instead of an increase in speed consider the effect of using a more efficient algorithm. By looking into the following table it is clear that if minute as a basis for comparison, by replacing algorithm A4 with A3, we can solve a problem six times larger; by replacing A4 with A2 we can solve a problem 125 times larger. These results are for more impressive than the two fold improvement obtained by a ten fold increase in speed. If an hour is used as the basis of comparison, the differences are even more significant.

We therefore conclude that the asymptotic complexity of an algorithm is an important measure of the goodness of an algorithm.

1.9. The Running time of a program

When solving a problem we are faced with a choice among algorithms. The basis for this can be any one of the following:

i. We would like an algorithm that is easy to understand, code and debug.

ii. We would like an algorithm that makes efficient use of the computer's resources, especially, one that runs as fast as possible.

1.10. Measuring the running time of a program

The running time of a program depends on factors such as:

1. The input to the program.

2. The quality of code generated by the compiler used to create the object program.

3. The nature and speed of the instructions on the machine used to execute the program, and

4. The time complexity of the algorithm underlying the program.

The running time depends not on the exact input but only the size of the input. For many programs, the running time is really a function of the particular input, and not just of the input size. In that case we define T(n) to be the worst case running time, i.e. the maximum overall input of size 'n', of the running time on that input. We also consider $T_{avg}(n)$ the average, over all input of size 'n' of the running time on that input. In practice, the average running time is often much harder to determine than the worst case running time. Thus, we will use worst–case running time as the principal measure of time complexity.

Seeing the remarks (2) and (3) we cannot express the running time T(n) in standard time units such as seconds. Rather we can only make remarks like the running time of such and such algorithm is proportional to n^2. The constant of proportionality will remain un-specified, since it depends so heavily on the compiler, the machine and other factors.

1.11. Asymptotic Analysis of Algorithms:

Our approach is based on the *asymptotic complexity* measure. This means that we don't try to count the exact number of steps of a program, but how that number grows with the size of the input to the program. That gives us a measure that will work for different operating systems, compilers and CPUs. The asymptotic complexity is written using big-O notation.

Rules for using big-O:

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. $O(20\ n^3) = O(n^3)$

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).

1.12. Calculating the running time of a program:

Let us now look into how big-O bounds can be computed for some common algorithms.

Example 1:

Let's consider a short piece of source code:

```
x = 3*y + 2;
z = z + 1;
```

If y, z are scalars, this piece of code takes a *constant* amount of time, which we write as O(1). In terms of actual computer instructions or clock ticks, it's difficult to say exactly how long it takes. But whatever it is, it should be the same whenever this piece of code is executed. O(1) means *some* constant, it might be 5, or 1 or 1000.

Example 2:

$2n^2 + 5n - 6 = O(2^n)$ $2n^2 + 5n - 6 = O(n^3)$ $2n^2 + 5n - 6 = O(n^2)$ $2n^2 + 5n - 6 \neq O(n)$	$2n^2 + 5n - 6 \neq \Theta(2^n)$ $2n^2 + 5n - 6 \neq \Theta(n^3)$ $2n^2 + 5n - 6 = \Theta(n^2)$ $2n^2 + 5n - 6 \neq \Theta(n)$
$2n^2 + 5n - 6 \neq \Omega(2^n)$ $2n^2 + 5n - 6 \neq \Omega(n^3)$ $2n^2 + 5n - 6 = \Omega(n^2)$ $2n^2 + 5n - 6 = \Omega(n)$	$2n^2 + 5n - 6 = o(2^n)$ $2n^2 + 5n - 6 = o(n^3)$ $2n^2 + 5n - 6 \neq o(n^2)$ $2n^2 + 5n - 6 \neq o(n)$

Example 3:

If the first program takes $100n^2$ milliseconds and while the second takes $5n^3$ milliseconds, then might not $5n^3$ program better than $100n^2$ program?

As the programs can be evaluated by comparing their running time functions, with constants by proportionality neglected. So, $5n^3$ program be better than the $100n^2$ program.

$$5 n^3/100 n^2 = n/20$$

for inputs $n < 20$, the program with running time $5n^3$ will be faster than those the one with running time $100 n^2$. Therefore, if the program is to be run mainly on inputs of small size, we would indeed prefer the program whose running time was $O(n^3)$

However, as 'n' gets large, the ratio of the running times, which is n/20, gets arbitrarily larger. Thus, as the size of the input increases, the $O(n^3)$ program will take significantly more time than the $O(n^2)$ program. So it is always better to prefer a program whose running time with the lower growth rate. The low growth rate function's such as O(n) or O(n log n) are always better.

Example 4:

Analysis of simple for loop

Now let's consider a simple for loop:

```
for (i = 1; i<=n; i++)
        v[i] = v[i] + 1;
```

This loop will run exactly n times, and because the inside of the loop takes constant time, the total running time is proportional to n. We write it as O(n). The actual number of instructions might be 50n, while the running time might be 17n microseconds. It might even be 17n+3 microseconds because the loop needs some time to start up. The big-O notation allows a multiplication factor (like 17) as well as an additive factor (like 3). As long as it's a linear function which is proportional to n, the correct notation is O(n) and the code is said to have *linear* running time.

Example 5:

Analysis for nested for loop

Now let's look at a more complicated example, a nested for loop:

```
for (i = 1; i<=n; i++)
        for (j = 1; j<=n; j++)
                a[i,j] = b[i,j] * x;
```

The outer for loop executes N times, while the inner loop executes n times for every execution of the outer loop. That is, the inner loop executes $n \times n = n^2$ times. The assignment statement in the inner loop takes constant time, so the running time of the code is $O(n^2)$ steps. This piece of code is said to have *quadratic* running time.

Example 6:

Analysis of matrix multiply

Lets start with an easy case. Multiplying two n × n matrices. The code to compute the matrix product C = A * B is given below.

```
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];
```

There are 3 nested *for* loops, each of which runs n times. The innermost loop therefore executes n*n*n = n^3 times. The innermost statement, which contains a scalar sum and product takes constant O(1) time. So the algorithm overall takes O(n^3) time.

Example 7:

Analysis of bubble sort

The main body of the code for bubble sort looks something like this:

```
for (i = n-1; i<1; i--)
        for (j = 1; j<=i; j++)
                if (a[j] > a[j+1])
                        swap a[j] and a[j+1];
```

This looks like the double. The innermost statement, the if, takes O(1) time. It doesn't necessarily take the same time when the condition is true as it does when it is false, but both times are bounded by a constant. But there is an important difference here. The outer loop executes n times, but the inner loop executes a number of times that depends on i. The first time the inner for executes, it runs i = n-1 times. The second time it runs n-2 times, etc. The total number of times the inner if statement executes is therefore:

(n-1) + (n-2) + ... + 3 + 2 + 1

This is the sum of an arithmetic series.

$$\sum_{i=1}^{N-1} (n-i) = \frac{n(n-i)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

The value of the sum is n(n-1)/2. So the running time of bubble sort is O(n(n-1)/2), which is O((n^2-n)/2). Using the rules for big-O given earlier, this bound simplifies to O((n^2)/2) by ignoring a smaller term, and to O(n^2), by ignoring a constant factor. Thus, bubble sort is an O(n^2) algorithm.

Example 8:

Analysis of binary search

Binary search is a little harder to analyze because it doesn't have a for loop. But it's still pretty easy because the search interval halves each time we iterate the search. The sequence of search intervals looks something like this:

$n, n/2, n/4, ..., 8, 4, 2, 1$

It's not obvious how long this sequence is, but if we take logs, it is:

$\log_2 n, \log_2 n - 1, \log_2 n - 2, ..., 3, 2, 1, 0$

Since the second sequence decrements by 1 each time down to 0, its length must be $\log_2 n + 1$. It takes only constant time to do each test of binary search, so the total running time is just the number of times that we iterate, which is $\log_2 n + 1$. So binary search is an $O(\log_2 n)$ algorithm. Since the base of the log doesn't matter in an asymptotic bound, we can write that binary search is $O(\log n)$.

1.13. General rules for the analysis of programs

In general the running time of a statement or group of statements may be parameterized by the input size and/or by one or more variables. The only permissible parameter for the running time of the whole program is 'n' the input size.

1. The running time of each assignment read and write statement can usually be taken to be O(1). (There are few exemptions, such as in PL/1, where assignments can involve arbitrarily larger arrays and in any language that allows function calls in arraignment statements).

2. The running time of a sequence of statements is determined by the sum rule. I.e. the running time of the sequence is, to with in a constant factor, the largest running time of any statement in the sequence.

3. The running time of an if–statement is the cost of conditionally executed statements, plus the time for evaluating the condition. The time to evaluate the condition is normally O(1) the time for an if–then–else construct is the time to evaluate the condition plus the larger of the time needed for the statements executed when the condition is true and the time for the statements executed when the condition is false.

4. The time to execute a loop is the sum, over all times around the loop, the time to execute the body and the time to evaluate the condition for termination (usually the latter is O(1)). Often this time is, neglected constant factors, the product of the number of times around the loop and the largest possible time for one execution of the body, but we must consider each loop separately to make sure.

Chapter 2

Advanced Data Structures and Recurrence Relations

2.1. Priority Queue, Heap and Heap Sort:

Heap is a data structure, which permits one to insert elements into a set and also to find the largest element efficiently. A data structure, which provides these two operations, is called a priority queue.

2.1.1. Max and Min Heap data structures:

A max heap is an almost complete binary tree such that the value of each node is greater than or equal to those in its children.

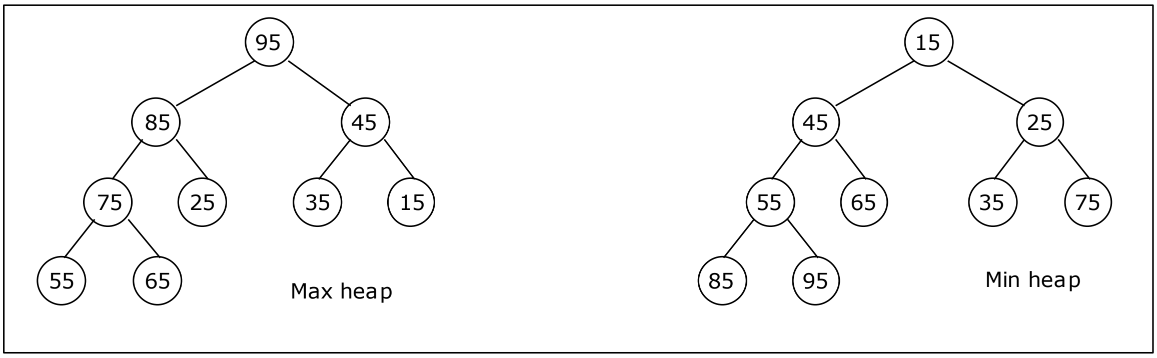


Figure 2.1. Max. and Min heap

A min heap is an almost complete binary tree such that the value of each node is less than or equal to those in its children. Figure 2.1 shows the maximum and minimum heap tree.

2.1.2. Representation of Heap Tree:

Since heap is a complete binary tree, a heap tree can be efficiently represented using one dimensional array. This provides a very convenient way of figuring out where children belong to.

- The root of the tree is in location 1.
- The left child of an element stored at location *i* can be found in location 2**i*.
- The right child of an element stored at location *i* can be found in location 2*i + 1.
- The parent of an element stored at location *i* can be found at location floor(*i*/2).

For example let us consider the following elements arranged in the form of array as follows:

X[1]	X[2]	X[3]	X[4]	X[5]	X[6]	X[7]	X[8]
65	45	60	40	25	50	55	30

The elements of the array can be thought of as lying in a tree structure. A heap tree represented using a single array looks as follows:

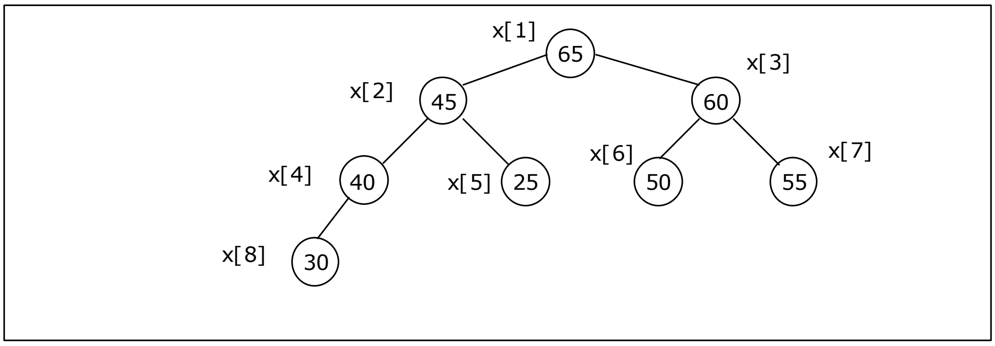


Figure 2.2. Heap Tree

2.1.3. Operations on heap tree:

The major operations required to be performed on a heap tree:

1. Insertion,
2. Deletion and
3. Merging.

Insertion into a heap tree:

This operation is used to insert a node into an existing heap tree satisfying the properties of heap tree. Using repeated insertions of data, starting from an empty heap tree, one can build up a heap tree.

Let us consider the heap (max) tree. The principle of insertion is that, first we have to adjoin the data in the complete binary tree. Next, we have to compare it with the data in its parent; if the value is greater than that at parent then interchange the values. This will continue between two nodes on path from the newly inserted node to the root node till we get a parent whose value is greater than its child or we reached the root.

For illustration, 35 is added as the right child of 80. Its value is compared with its parent's value, and to be a max heap, parent's value greater than child's value is satisfied, hence interchange as well as further comparisons are no more required.

As another illustration, let us consider the case of insertion 90 into the resultant heap tree. First, 90 will be added as left child of 40, when 90 is compared with 40 it requires interchange. Next, 90 is compared with 80, another interchange takes place. Now, our process stops here, as 90 is now in root node. The path on which these comparisons and interchanges have taken places are shown by *dashed line*.

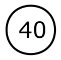
The algorithm Max_heap_insert to insert a data into a max heap tree is as follows:

```
Max_heap_insert (a, n)
{
    //inserts the value in a[n] into the heap which is stored at a[1] to a[n-1]
    integer i, n;
    i = n;
    item = a[n]  ;
    while ( (i > 1) and (a[ $\lfloor i/2 \rfloor$ ] < item ) do
    {
        a[i] = a[ $\lfloor i/2 \rfloor$ ] ;        // move the parent down
        i = $\lfloor i/2 \rfloor$  ;
    }
    a[i] = item ;
    return true ;
}
```

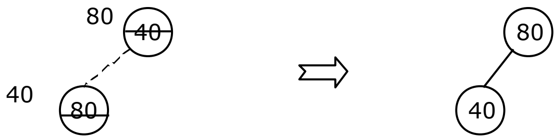
Example:

Form a heap by using the above algorithm for the given data 40, 80, 35, 90, 45, 50, 70.

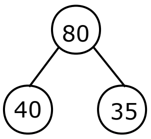
1. Insert 40:



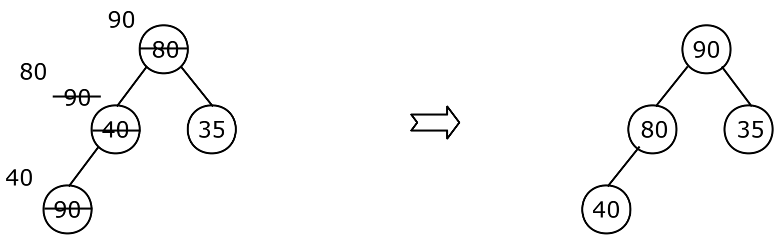
2. Insert 80:



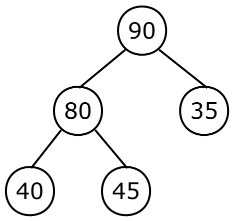
3. Insert 35:



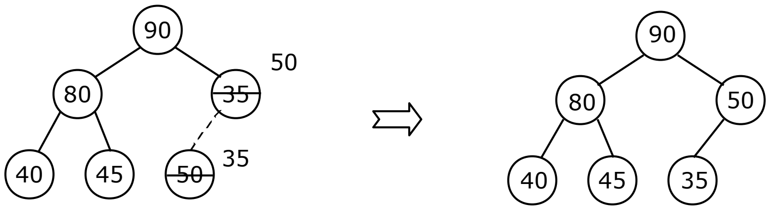
4. Insert 90:



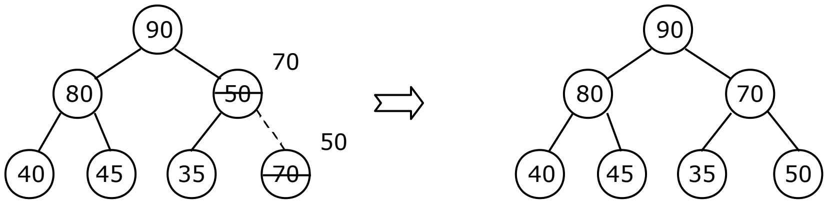
5. Insert 45:



6. Insert 50:



7. Insert 70:



Deletion of a node from heap tree:

Any node can be deleted from a heap tree. But from the application point of view, deleting the root node has some special importance. The principle of deletion is as follows:

- Read the root node into a temporary storage say, ITEM.
- Replace the root node by the last node in the heap tree. Then re-heap the tree as stated below:
  - Let newly modified root node be the current node. Compare its value with the value of its two child. Let X be the child whose value is the largest. Interchange the value of X with the value of the current node.
  - Make X as the current node.
  - Continue re-heap, if the current node is not an empty node.

The algorithm for the above is as follows:

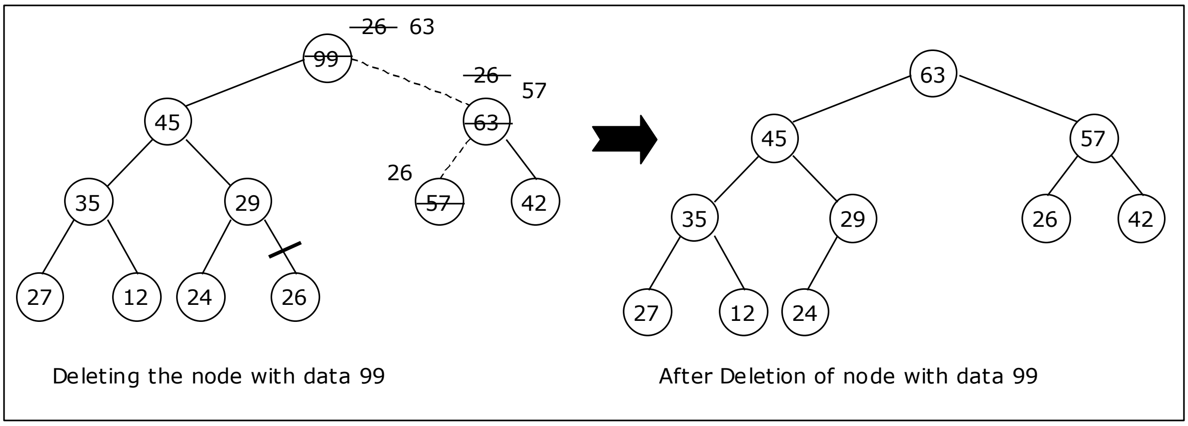
delmax (a, n, x)

```
// delete the maximum from the heap a[n] and store it in x
{
        if (n = 0) then
        {
                write ("heap is empty");
                return false;
        }
        x = a[1]; a[1] = a[n];
        adjust (a, 1, n-1);
        return true;
}
```

adjust (a, i, n)

```
// The complete binary trees with roots a(2*i) and a(2*i + 1) are combined with a(i) to form a
single heap, 1 ≤ i ≤ n. No node has an address greater than n or less than 1. //
{
        j = 2 *i  ;
        item = a[i] ;
        while (j ≤ n) do
        {
                if ((j < n) and (a (j) < a (j + 1)) then j ← j + 1;
                        // compare left and right child and let j be the larger child
                if (item ≥ a (j)) then break;
                                                // a position for item is found
                else a[ ⌊ j / 2 ⌋ ] = a[j]  // move the larger child up a level
                j = 2 * j;
        }
        a [ ⌊ j / 2 ⌋ ] = item;
}
```

Here the root node is 99. The last node is 26, it is in the level 3. So, 99 is replaced by 26 and this node with data 26 is removed from the tree. Next 26 at root node is compared with its two child 45 and 63. As 63 is greater, they are interchanged. Now, 26 is compared with its children, namely, 57 and 42, as 57 is greater, so they are interchanged. Now, 26 appear as the leave node, hence re-heap is completed. This is shown in figure 2.3.



Deleting the node with data 99

After Deletion of node with data 99

Figure 2.3.

2.1.4. Merging two heap trees:

Consider, two heap trees H1 and H2. Merging the tree H2 with H1 means to include all the node from H2 to H1. H2 may be min heap or max heap and the resultant tree will be min heap if H1 is min heap else it will be max heap.

Merging operation consists of two steps: Continue steps 1 and 2 while H2 is not empty:

1. Delete the root node, say x, from H2. Re-heap H2.
2. Insert the node x into H1 satisfying the property of H1.

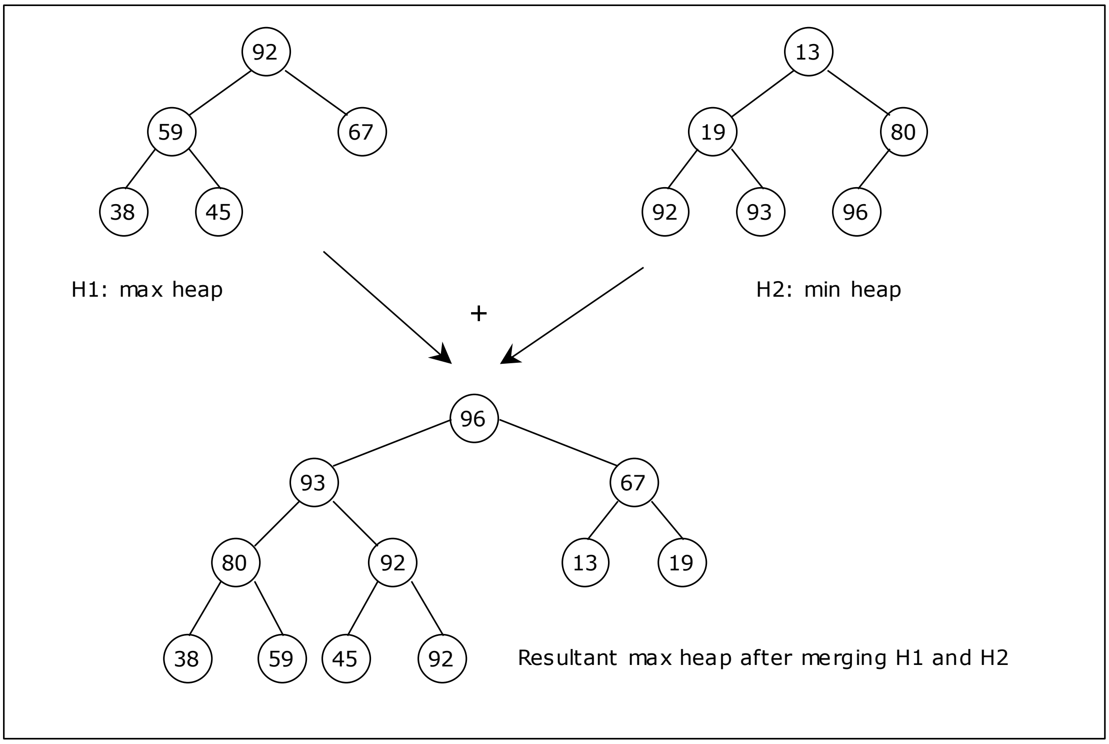


Figure 2.4. Merging of two heaps.

2.1.5. Applications of heap tree:

They are two main applications of heap trees known:

1. Sorting (Heap sort) and
2. Priority queue implementation.

2.2. HEAP SORT:

A heap sort algorithm works by first organizing the data to be sorted into a special type of binary tree called a heap. Any kind of data can be sorted either in ascending order or in descending order using heap tree. It does this with the following steps:

1. Build a heap tree with the given set of data.
2. a. Remove the top most item (the largest) and replace it with the last element in the heap.

   b. Re-heapify the complete binary tree.

    c. Place the deleted node in the output.

3. Continue step 2 until the heap tree is empty.

Algorithm:

This algorithm sorts the elements a[n]. Heap sort rearranges them in-place in non-decreasing order. First transform the elements into a heap.

```
heapsort(a, n)
{
        heapify(a, n);
        for i = n to 2 by – 1 do
        {
                temp = a[I];
                a[i] =  a[1];
                a[1] = t;
                adjust (a, 1, i – 1);
        }
}

heapify (a, n)
//Readjust the elements in a[n] to form a heap.
{
        for i ← ⌊ n/2 ⌋ to 1 by – 1 do adjust (a, i, n);
}

adjust (a, i, n)
// The complete binary trees with roots a(2*i) and a(2*i+1) are combined
// with a(i) to form a single heap, 1≤i≤n. No node has an address greater
// than n or less than 1.
{
        j = 2 *i  ;
        item = a[i] ;
        while (j ≤ n) do
        {
                if ((j < n) and (a (j) < a (j + 1)) then j ← j + 1;
                        // compare left and right child and let j be the larger child
                if (item ≥ a (j)) then break;
                                                // a position for item is found
                else a[ ⌊ j / 2 ⌋ ] = a[j]  // move the larger child up a level
                j = 2 * j;
        }
        a [ ⌊ j / 2 ⌋ ] = item;
}
```

Time Complexity:

Each ‘n’ insertion operations takes O(log k), where ‘k’ is the number of elements in the heap at the time. Likewise, each of the ‘n’ remove operations also runs in time O(log k), where ‘k’ is the number of elements in the heap at the time. Since we always have k ≤ n, each such operation runs in O(log n) time in the worst case.

Thus, for *n* elements it takes O(n log n) time, so the priority queue sorting algorithm runs in O(n log n) time when we use a heap to implement the priority queue.

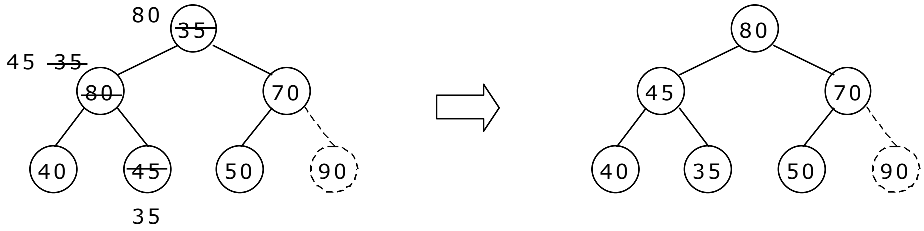
Example 1:

Form a heap from the set of elements (40, 80, 35, 90, 45, 50, 70) and sort the data using heap sort.

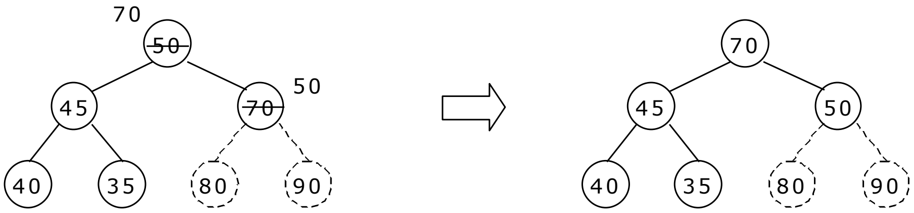
Solution:

First form a heap tree from the given set of data and then sort by repeated deletion operation:

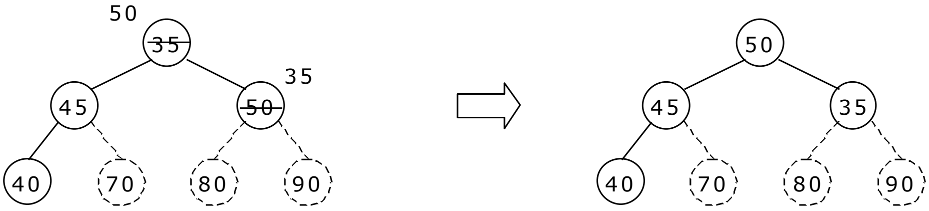
1. Exchange root 90 with the last element 35 of the array and re-heapify



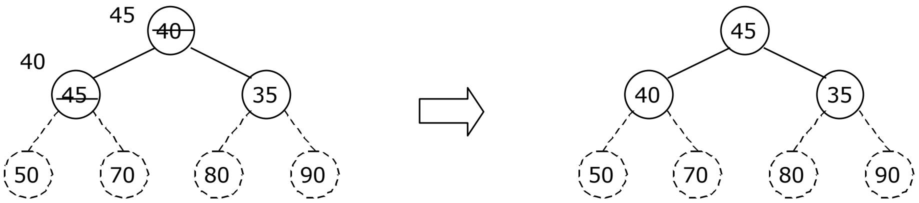
2. Exchange root 80 with the last element 50 of the array and re-heapify



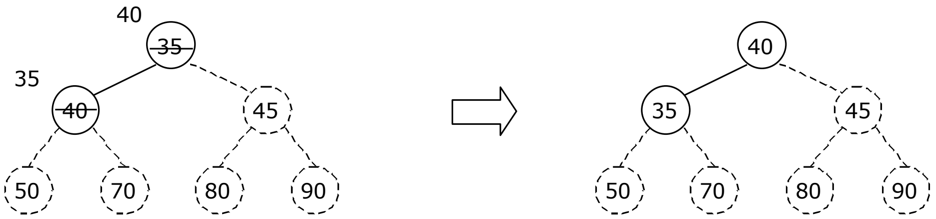
3. Exchange root 70 with the last element 35 of the array and re-heapify



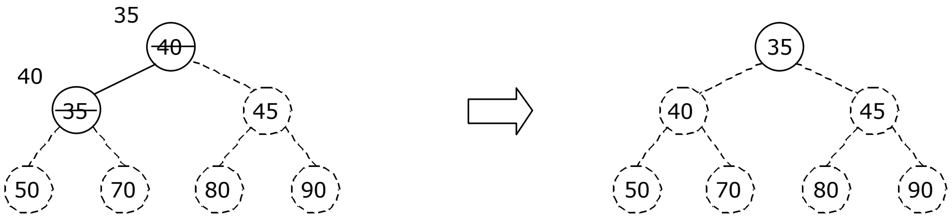
4. Exchange root 50 with the last element 40 of the array and re-heapify



5. Exchange root 45 with the last element 35 of the array and re-heapify



6. Exchange root 40 with the last element 35 of the array and re-heapify



The sorted tree

2.3. Priority queue implementation using heap tree:

Priority queue can be implemented using circular array, linked list etc. Another simplified implementation is possible using heap tree; the heap, however, can be represented using an array. This implementation is therefore free from the complexities of circular array and linked list but getting the advantages of simplicities of array.

As heap trees allow the duplicity of data in it. Elements associated with their priority values are to be stored in from of heap tree, which can be formed based on their priority values. The top priority element that has to be processed first is at the root; so it can be deleted and heap can be rebuilt to get the next element to be processed, and so on.

As an illustration, consider the following processes with their priorities:

Process	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}
Priority	5	4	3	4	5	5	3	2	1	5

These processes enter the system in the order as listed above at time 0, say. Assume that a process having higher priority value will be serviced first. The heap tree can be formed considering the process priority values. The order of servicing the process is successive deletion of roots from the heap.

2.4. Binary Search Trees:

A binary search tree has binary nodes and the following additional property. Given a node t, each node to the left is “smaller” than t, and each node to the right is “larger”. This definition applies recursively down the left and right sub-trees. Figure 2.5 shows a binary search tree where characters are stored in the nodes.

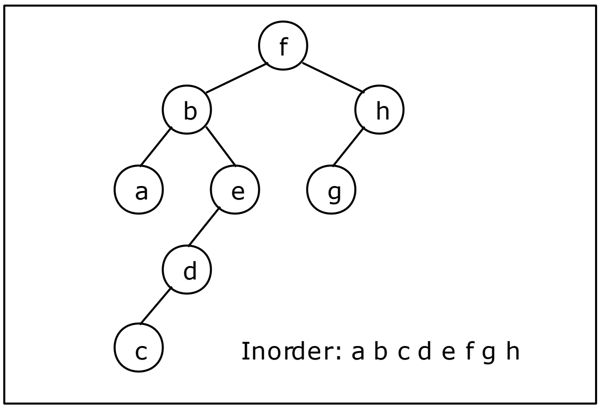


Figure 2.5. Binary Search tree

Figure 2.5 also shows what happens if you do an inorder traversal of a binary search tree: you will get a list of the node contents in sorted order. In fact, that’s probably how the name inorder originated.

Binary Tree Searching:

The search operation starts from root node R, if item is less than the value in the root node R, we proceed to the left child; if item is greater than the value in the node R, we proceed to its right child. The process will be continued till the item is found or we reach to a dead end. The Figure 2.6 shows the path taken when searching for a node “c”.

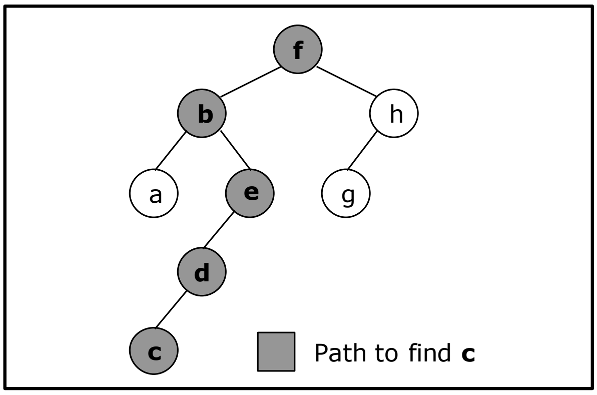


Figure 2.6. Searching a binary tree

Why use binary search trees?

Binary search trees provide an efficient way to search through an ordered collection of items. Consider the alternative of searching an ordered list. The search must proceed sequentially from one end of the list to the other. On average, n/2 nodes must be compared for an ordered list that contains n nodes. In the worst case, all n nodes might need to be compared. For a large collection of items, this can get very expensive.

The inefficiency is due to the one-dimensionality of a linked list. We would like to have a way to jump into the middle of the list, in order to speed up the search process. In essence, that's what a binary search tree does. The longest path we will ever have to search is equal to the height of the tree. The efficiency of a binary search tree thus depends on the height of the tree. For a tree holding n nodes, the smallest possible height is log(n).

To obtain the smallest height, a tree must be balanced, where both the left and right sub trees have approximately the same number of nodes. Also, each node should have as many children as possible, with all levels being full except possibly the last. Figure 2.7 shows an example of a well-constructed tree.

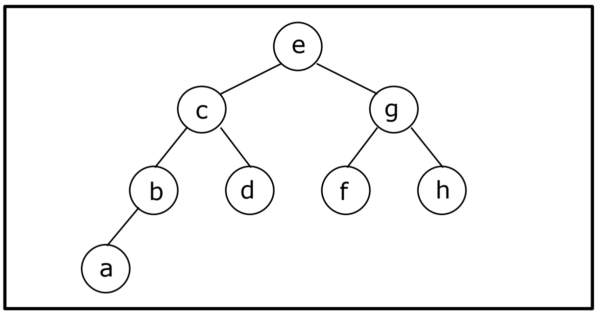


Figure 2.7. Well constructed binary search tree.

Unfortunately, trees can become so unbalanced that they're no better for searching than linked lists. Such trees are called *degenerate trees*. Figure 2.8 shows an example. For a degenerate tree, an average of n/2 comparisons are needed, with a worst case of n comparisons – the same as for a linked list.

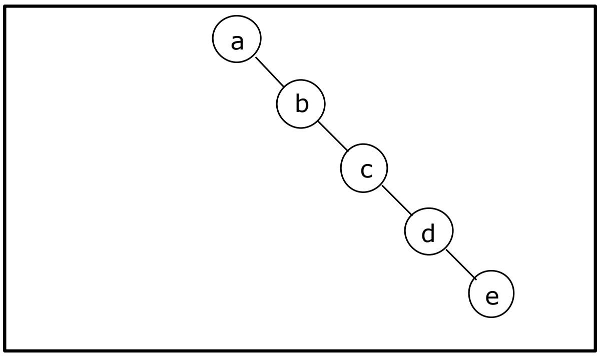


Figure 2.8. A degenerate binary search tree.

When nodes are being added and deleted in a binary search tree, it's difficult to maintain the balance of the tree. We will investigate methods of balancing trees in the next section.

Inserting Nodes into a Binary Search Tree:

When adding nodes to a binary search tree, we must be careful to maintain the binary search tree property. This can be done by first searching the tree to see whether the key we are about to add is already in the tree. If the key cannot be found, a new node is allocated and added at the same location where it would go if the search had been successful. Figure 2.9 shows what happens when we add some nodes to a tree.

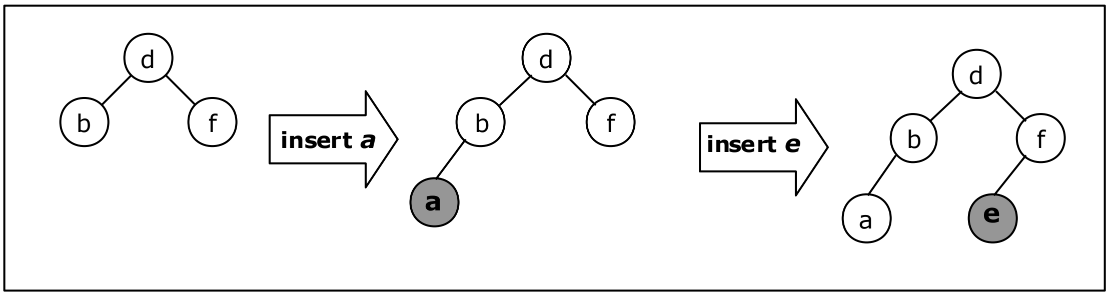


Figure 2.9. Inserting nodes into a binary search tree.

Deleting nodes from a Binary search tree:

Deletions from a binary search tree are more involved than insertions. Given a node to delete, we need to consider these tree cases:

1. The node is a leaf
2. The node has only one child.
3. The node has two children.

Case 1: It is easy to handle because the node can simply be deleted and the corresponding child pointer of its parent set to null. Figure 2.10 shows an example.

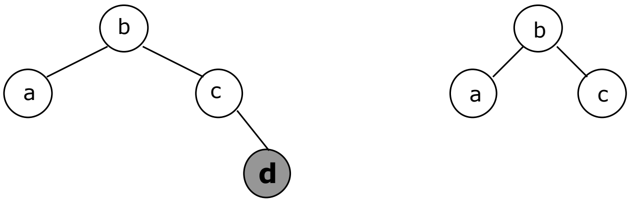


Figure 2.10. Deleting a leaf node.

Case 2: It is almost as easy to manage. Here, the single child can be promoted up the tree to take the place of the deleted node, as shown in figure 2.11.

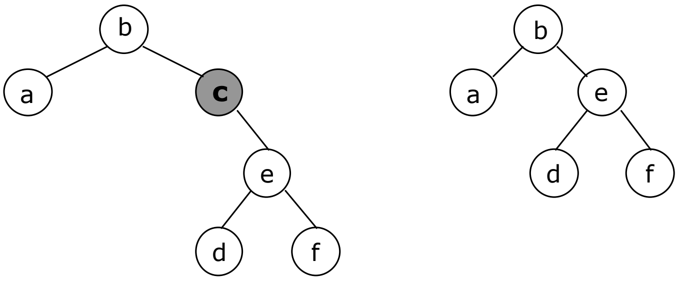


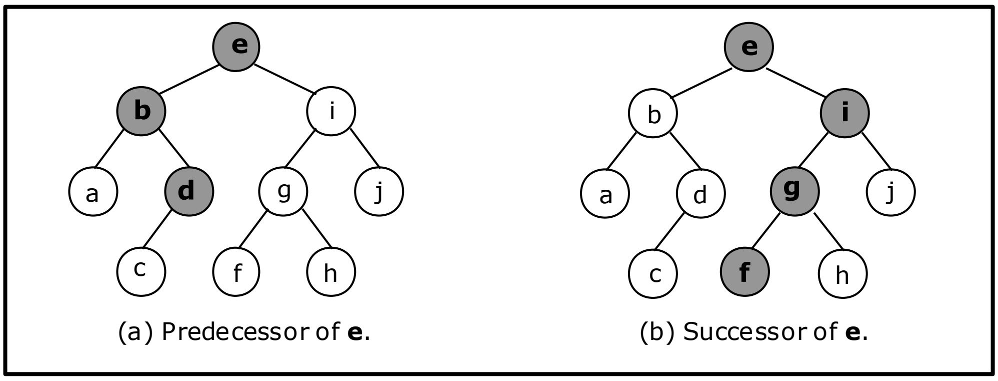
Figure 2.11. Deleting a node that has one child.

Case 3: The node to be deleted has two children, is more difficult. We must find some node to take the place of the one deleted and still maintain the binary search tree property. There are two obvious cases:

- The inorder predecessor of the deleted node.
- The inorder successor of the deleted node.

We can detach one of these nodes from the tree and insert it where the node to be deleted.

The predecessor of a node can be found by going down to the left once, and then all the way to the right as far as possible. To find the successor, an opposite traversal is used: first to the right, and then down to the left as far as possible. Figure 2.12 shows the path taken to find both the predecessor and successor of a node.

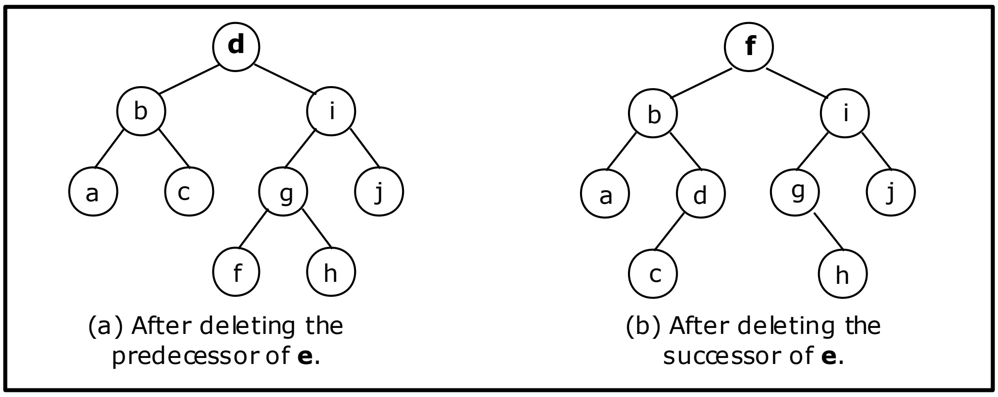


(a) Predecessor of **e**.

(b) Successor of **e**.

Figure 2.12. Finding predecessor and successor of nodes.

Both the predecessor and successor nodes are guaranteed to have no more than one child, and they may have none. Detaching the predecessor or successor reduces to either case 1 or 2, both of which are easy to handle. In figure 2.13 (a), we delete a node from the tree and use its predecessor as the replacement and in figure 2.13 (b), we delete a node from the tree and use its successor as the replacement.



(a) After deleting the predecessor of **e**.

(b) After deleting the successor of **e**.

Figure 2.13. Deleting a node that has two children.

2.5. Balanced Trees:

For maximum efficiency, a binary search tree should be balanced. Every un-balanced trees are referred to as degenerate trees, so called because their searching performance degenerates to that of linked lists. Figure 2.14 shows an example.

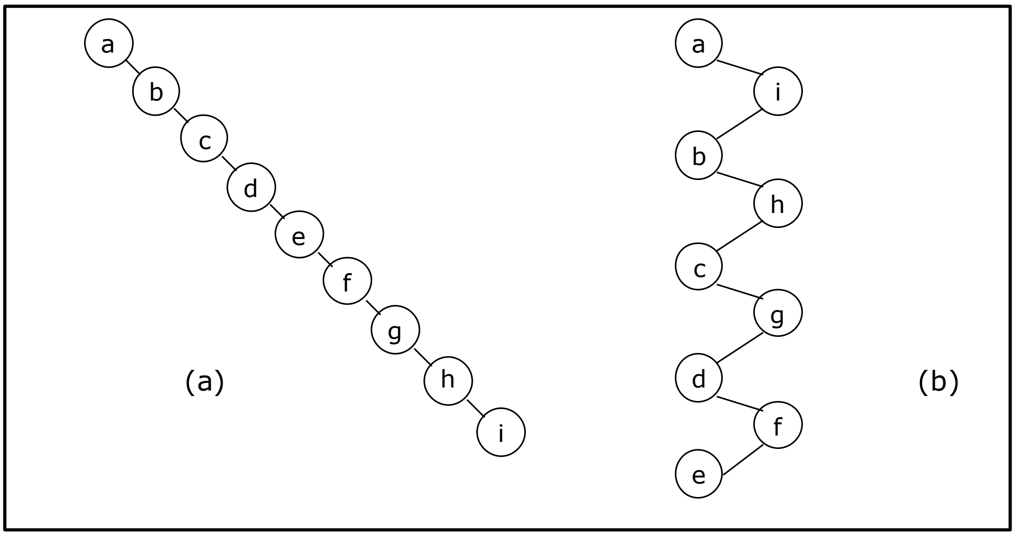


Figure 2.14. A degenerate binary search tree.

The first tree was built by inserting the keys 'a' through 'i' in sorted order into binary search tree. The second tree was built using the insertion sequence. a-g-b-f-c-e-d. This pathological sequence is often used to test the balancing capacity of a tree. Figure 2.15 shows what the tree in 2.14-(b) above would look like if it were *balanced*.

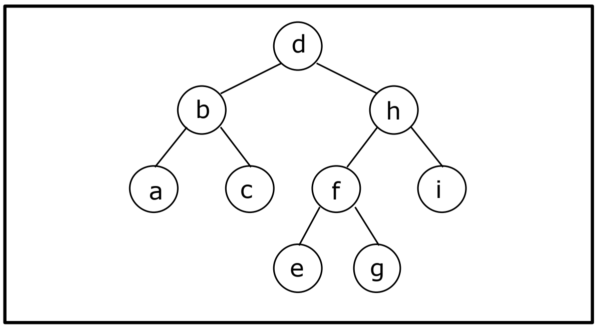


Figure 2.15 Balanced Tree

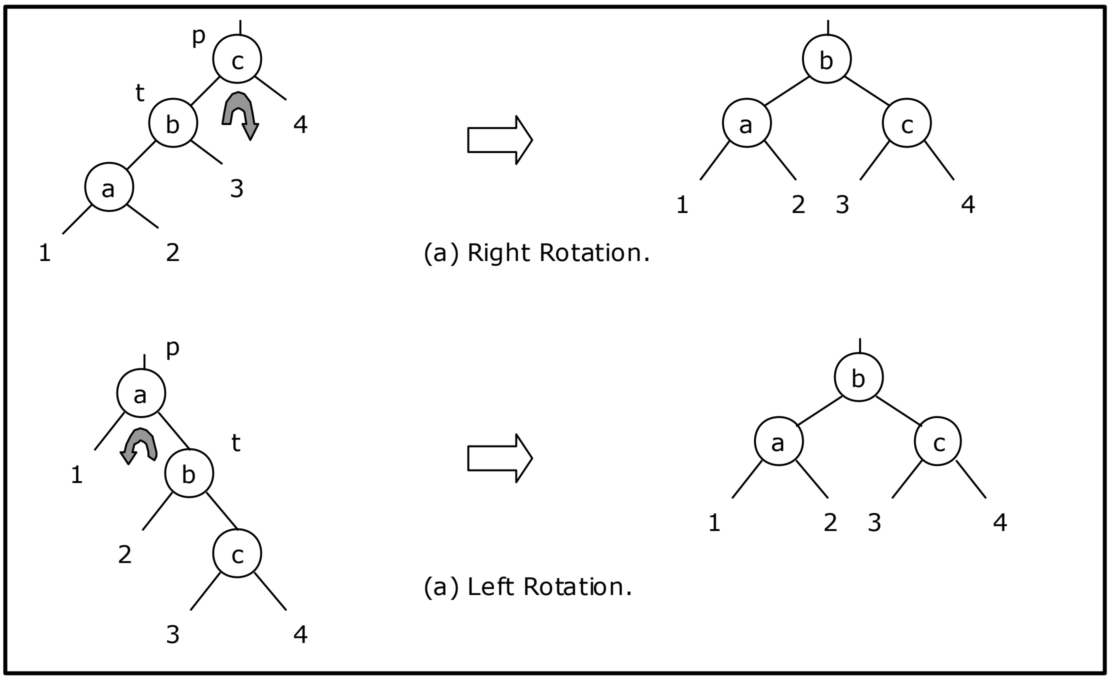
Balancing Acts:

There are two basic ways used to keep trees balanced.

1) Use tree rotations.
2) Allow nodes to have more than two children.

Tree Rotations:

Certain types of tree-restructurings, known as rotations, can aid in balancing trees. Figure 2.16 shows two types of single rotations.

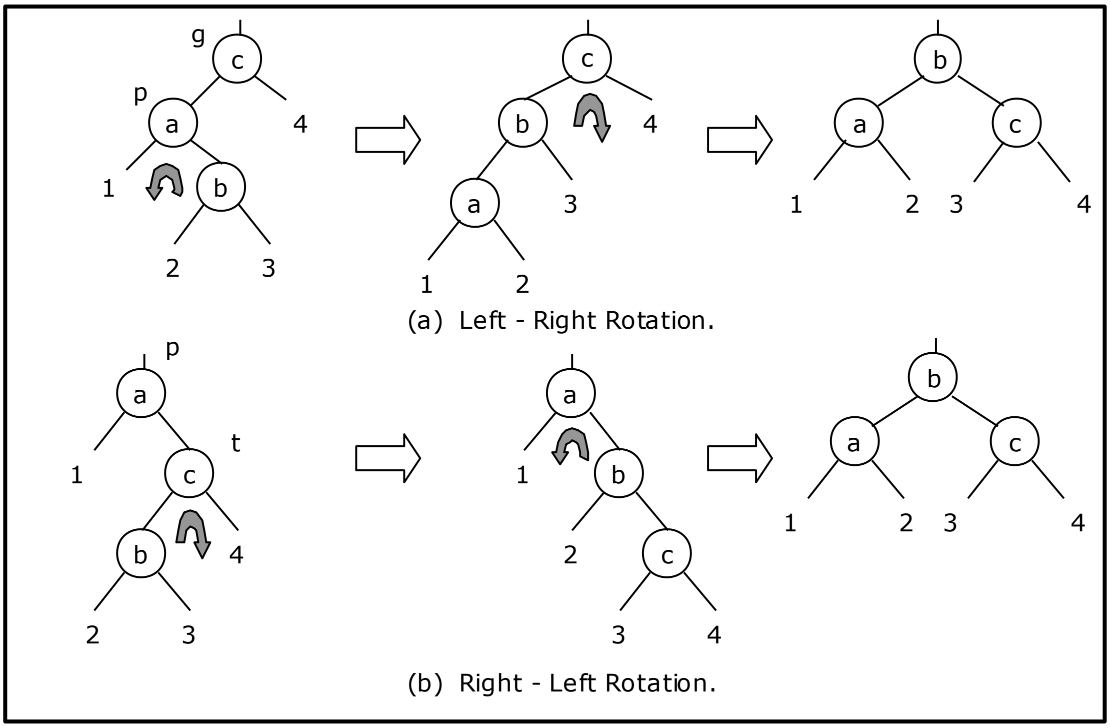


(a) Right Rotation.

(a) Left Rotation.

Figure 2.16. Single Rotations

Another type of rotation is known as a double rotation. Figure 2.17 shows the two symmetrical cases.



(a) Left - Right Rotation.

(b) Right - Left Rotation.

Figure 2.17. Double Rotations

Both single and double rotations have one important feature: in order traversals of the trees before and after the rotations are preserved. Thus, rotations help balance trees, but still maintain the binary search tree property.

Rotations don't guarantee a balanced tree, however for example, figure 2.18 shows a right rotations that makes a tree more un-balanced. The trick lies in determining when to do rotation and what kind of notations to do.

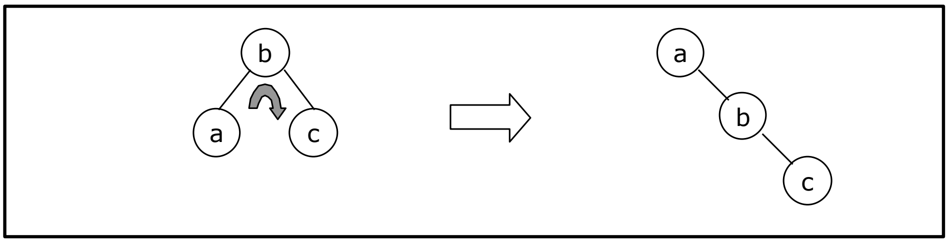


Figure 2.18. Un-balanced rotation.

The Red-black trees have certain rules to be used to determine when a how to rotate.

2.6. *Dictionary:*

A Dictionary is a collection of pairs of form (*k*, *e*), where *k* is a key and *e* is the element associated with the key *k* (equivalently, *e* is the element whose key is *k*). No two pairs in a dictionary should have the same key.

Examples:

1. A word dictionary is a collection of elements; each element comprises a word, which is the key and the meaning of the word, pronunciation and etymologies etc. are associated with the word.

2. A telephone directory with a collection of telephone numbers and names.

3. The list of students enrolled for the data structures course, compiler and Operating System course. The list contains (CourseName, RollID) pairs.

The above last two examples are dictionaries, which permit two or more (key, element) pairs having the same key.

Operations on Dictionaries:

- Get the element associated with a specified key from the dictionary.
  For example, **get(*k*)** returns the element with the key *k.*
- Insert or put an element with a specified key into the dictionary.
  For example, **put(*k*, *e*)** puts the element *e* whose key is *k* into the dictionary.

- Delete or remove an element with a specified key.
  For example, **remove(*k*)** removes the element with key *k.*

Disjoint Set Operations

2.7. Disjoint Set Operations

Set:

A set is a collection of distinct elements. The Set can be represented, for examples, as S1={1,2,5,10}.

Disjoint Sets:

The disjoints sets are those do not have any common element.
For example S1={1,7,8,9} and S2={2,5,10}, then we can say that S1 and S2 are two disjoint sets.

Disjoint Set Operations:

The disjoint set operations are

1. Union
2. Find

Disjoint set Union:

If Si and Sj are tow disjoint sets, then their union Si U Sj consists of all the elements x such that x is in Si or Sj.

Example:

S1={1,7,8,9}	S2={2,5,10}

S1 U S2={1,2,5,7,8,9,10}

Find:

Given the element I, find the set containing i.

Example:

S1={1,7,8,9}	S2={2,5,10}	s3={3,4,6}

Then,

Find(4)= S3	Find(5)=S2	Find97)=S1

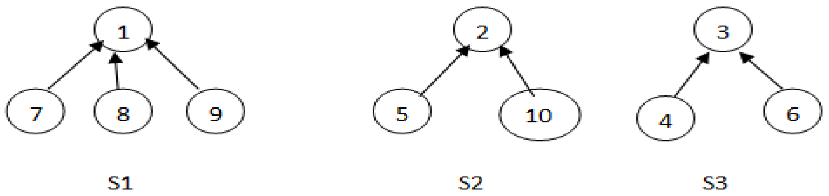
Set Representation:

The set will be represented as the tree structure where all children will store the address of parent / root node. The root node will store null at the place of parent address. In the given set of elements any element can be selected as the root node, generally we select the first node as the root node.

Example:

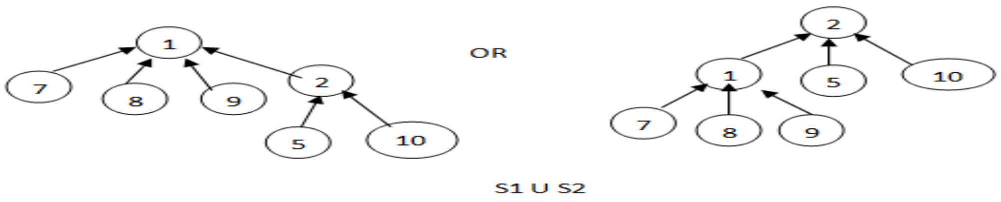
S1={1,7,8,9}	S2={2,5,10}	s3={3,4,6}

Then these sets can be represented as



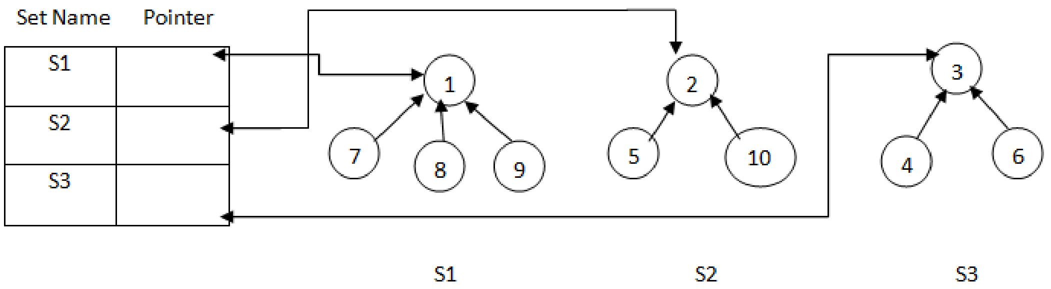
Disjoint Union:

To perform disjoint set union between two sets Si and Sj can take any one root and make it sub-tree of the other. Consider the above example sets S1 and S2 then the union of S1 and S2 can be represented as any one of the following.



Find:

To perform find operation, along with the tree structure we need to maintain the name of each set. So, we require one more data structure to store the set names. The data structure contains two fields. One is the set name and the other one is the pointer to root.

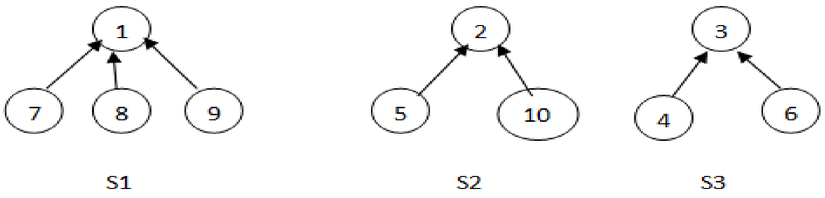


2.7.1 Union and Find Algorithms:

In presenting Union and Find algorithms, we ignore the set names and identify sets just by the roots of trees representing them. To represent the sets, we use an array of 1 to n elements where n is the maximum value among the elements of all sets. The index values represent the nodes (elements of set) and the entries represent the parent node. For the root value the entry will be '-1'.

Example:

For the following sets the array representation is as shown below.



i	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
p	-1	-1	-1	3	2	3	1	1	1	2

Algorithm for Union operation:

To perform union the **SimpleUnion(i,j)** function takes the inputs as the set roots i and j . And make the parent of i as j i.e, make the second root as the parent of first root.

```
Algorithm SimpleUnion(i,j)
{
        P[i]:=j;
}
```

Algorithm for find operation:

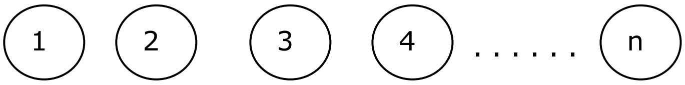
The SimpleFind(i) algorithm takes the element i and finds the root node of i. It starts at I until it reaches a node with parent value -1.

```
Algorithms SimpleFind(i)
{
        while( P[i]≥0) do i:=P[i];
        return i;
```

```
    }
```

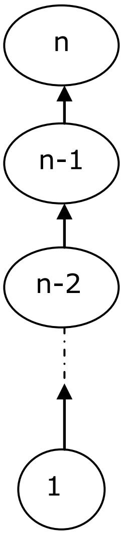
2.7.2 Analysis of SimpleUnion(i,j) and SimpleFind(i):

Although the SimpleUnion(i,j) and SimpleFind(i) algorithms are easy to state, their performance characteristics are not very good. For example, consider the sets



Then if we want to perform following sequence of operations Union(1,2) , Union(2,3)……. Union(n-1,n) and sequence of Find(1), Find(2)……… Find(n).

The sequence of Union operations results the degenerate tree as below.



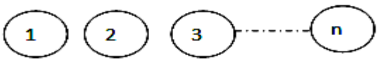
Since, the time taken for a Union is constant, the n-1 sequence of union can be processed in time O(n). And for the sequence of Find operations it will take time complexity of $O(\sum_{i=1}^{n} i) = O(n^2)$.

We can improve the performance of union and find by avoiding the creation of degenerate tree by applying weighting rule for Union.

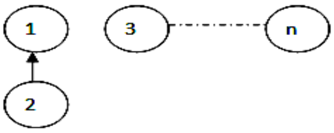
2.7.3 Weighting rule for Union:

If the number of nodes in the tree with root I is less than the number in the tree with the root j, then make 'j' the parent of i; otherwise make 'i' the parent of j.

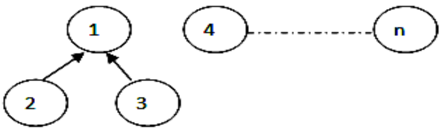
Consider Set



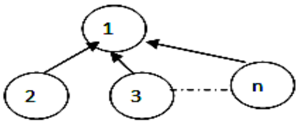
Union(1,2)



Union (1,3)



Union(1,n)



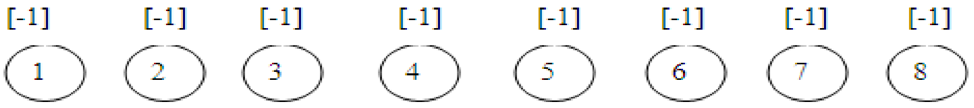
To implement weighting rule we need to know how many nodes are there in every tree. To do this we maintain "count" field in the root of every tree. If 'i' is the root then count[i] equals to number of nodes in tree with root i.
Since all nodes other than roots have positive numbers in parent (P) field, we can maintain count in P field of the root as negative number.

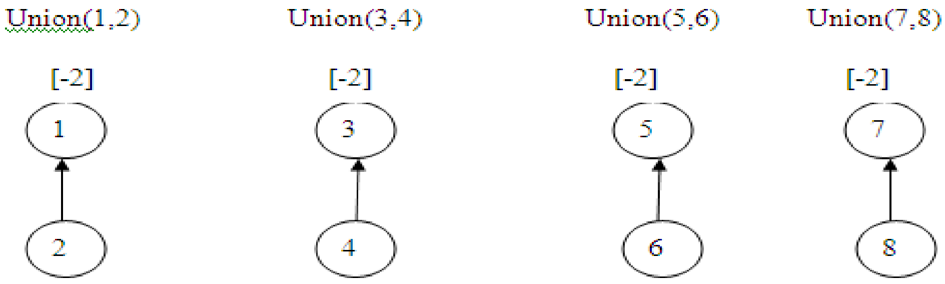
```
Algorithm WeightedUnion(i,j)
//Union sets with roots i  and j , i≠j using the weighted rule
// P[i]=-count[i] and p[j]=-count[j]
{
        temp:= P[i]+P[j];
        if (P[i]>P[j]) then
        {
         // i has fewer nodes
             P[i]:=j;
             P[j]:=temp;
         }
         else
         {
         // j has fewer nodes
             P[j]:=i;
             P[i]:=temp;
         }

}
```

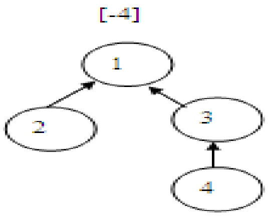
2.7.4 Collapsing rule for find:

If j is a node on the path from i to its root and p[i]≠root[i], then set P[j] to root[i].
Consider the tree created by WeightedUnion() on the sequence of 1≤i≤8.
Union(1,2), Union(3,4), Union(5,6) and Union(7,8)

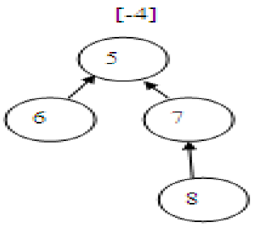




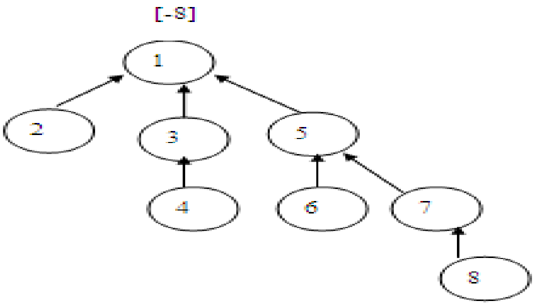
Union(1,3)



Union(5,7)



Union(1,5)



Now process the following eight find operations

Find(8), Find(8)……………………..Find(8)

If SimpleFind() is used each Find(8) requires going up three parent link fields for a total of 24 moves .
When Collapsing find is used the first Find(8) requires going up three links and resetting three links. Each of remaining seven finds require going up only one link field. Then the total cost is now only 13 moves.(3 going up + 3 resets + 7 remaining finds).

```
Algorithm CollapsingFind(i)
// Find the root of the tree containing element i
// use the collapsing rule to collapse all nodes from i to root.
{
        r:=i;
        while(P[r]>0) do r:=P[r];  //Find root
         while(i≠r)
         {
                 //reset the parent node from element i to the root
                 s:=P[i];
                 P[i]:=r;
                 i:=s;
         }
}
```

Recurrence Relations

Recurrence Relation for a sequence of numbers S is a formula that relates all but a finite number of terms of S to previous terms of the sequence, namely, $\{a_0, a_1, a_2, \ldots\ldots, a_{n-1}\}$, for all integers n with $n \geq n_{0,}$ where n_0 is a nonnegative integer. Recurrence relations are also called as difference equations.

Sequences are often most easily defined with a recurrence relation; however the calculation of terms by directly applying a recurrence relation can be time consuming. The process of determining a closed form expression for the terms of a sequence from its recurrence relation is called solving the relation. Some guess and check with respect to solving recurrence relation are as follows:

- Make simplifying assumptions about inputs
- Tabulate the first few values of the recurrence
- Look for patterns, guess a solution
- Generalize the result to remove the assumptions

Examples: Factorial, Fibonnaci, Quick sort, Binary search etc.

Recurrence relation is an equation, which is defined in terms of itself. There is no single technique or algorithm that can be used to solve all recurrence relations. In fact, some recurrence relations cannot be solved. Most of the recurrence relations that we encounter are linear recurrence relations with constant coefficients.

Several techniques like substitution, induction, characteristic roots and generating function are available to solve recurrence relations.

2.8. The Iterative Substitution Method:

One way to solve a divide-and-conquer recurrence equation is to use the iterative substitution method. This is a "plug-and-chug" method. In using this method, we assume that the problem size n is fairly large and we than substitute the general form of the recurrence for each occurrence of the function T on the right-hand side. For example, performing such a substitution with the merge sort recurrence equation yields the equation.

$$\begin{aligned} T(n) &= 2\ (2\ T(n/2^2) + b\ (n/2)) + b\ n \\ &= 2^2\ T(n/2^2) + 2\ b\ n \end{aligned}$$

Plugging the general equation for T again yields the equation.

$$\begin{aligned} T(n) &= 2^2\ (2\ T(n/2^3) + b\ (n/2^2)) + 2\ b\ n \\ &= 2^3\ T(n/2^3) + 3\ b\ n \end{aligned}$$

The hope in applying the iterative substitution method is that, at some point, we will see a pattern that can be converted into a general closed-form equation (with T only appearing on the left-hand side). In the case of merge-sort recurrence equation, the general form is:

$$T(n) = 2^i\ T\ (n/2^i) + i\ b\ n$$

Note that the general form of this equation shifts to the base case, T(n) = b, where $n = 2^i$, that is, when i = log n, which implies:

T(n) = b n + b n log n.

In other words, T(n) is O(n log n). In a general application of the iterative substitution technique, we hope that we can determine a general pattern for T(n) and that we can also figure out when the general form of T(n) shifts to the base case.

2.9. The Recursion Tree:

Another way of characterizing recurrence equations is to use the recursion tree method. Like the iterative substitution method, this technique uses repeated substitution to solve a recurrence equation, but it differs from the iterative substitution method in that, rather than being an algebraic approach, it is a visual approach.

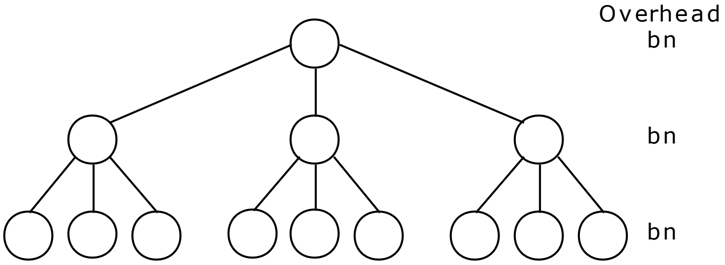
In using the recursion tree method, we draw a tree R where each node represents a different substitution of the recurrence equation. Thus, each node in R has a value of the argument n of the function T (n) associated with it. In addition, we associate an overhead with each node v in R, defined as the value of the non-recursive part of the recurrence equation for v.

For divide-and-conquer recurrences, the overhead corresponds to the running time needed to merge the subproblem solutions coming from the children of v. The recurrence equation is then solved by summing the overheads associated with all the nodes of R. This is commonly done by first summing values across the levels of R and then summing up these partial sums for all the levels of R.

For example, consider the following recurrence equation:

$$T(n) = \begin{cases} b & \text{if } n < 3 \\ 3T(n/3) + bn & \text{if } n \geq 3 \end{cases}$$

This is the recurrence equation that we get, for example, by modifying the merge sort algorithm so that we divide an unsorted sequence into three equal – sized sequences, recursively sort each one, and then do a three-way merge of three sorted sequences to produce a sorted version of the original sequence. In the recursion tree R for this recurrence, each internal node v has three children and has a size and an overhead associated with it, which corresponds to the time needed to merge the sub-problem solutions produced by v's children. We illustrate the tree R as follows:



The overheads of the nodes of each level, sum to bn. Thus, observing that the depth of R is $\log_3$ n, we have that T(n) is O(n log n).

2.10. The Guess-and-Test Method:

Another method for solving recurrence equations is the guess-and-test technique. This technique involves first making a educated guess as to what a closed-form solution of the recurrence equation might look like and then justifying the guesses, usually by induction. For example, we can use the guess-and-test method as a kind of "binary search" for finding good upper bounds on a given recurrence equation. If the justification of our current guess fails, then it is possible that we need to use a faster-growing function, and if our current guess is justified "too easily", then it is possible that we need to use a slower-growing function. However, using this technique requires case careful, in each mathematical step we take, in trying to justify that a certain hypothesis holds with respect to our current "guess".

Example 2.10.1: Consider the following recurrence equation:

$T(n) = 2T(n/2) + b\, n \log n$. (assuming the base case $T(n) = b$ for $n < 2$)

This looks very similar to the recurrence equation for the merge sort routine, so we might make the following as our first guess:

First guess: $T(n) < c\, n \log n$.

for some constant $c>0$. We can certainly choose c large enough to make this true for the base case, so consider the case when $n > 2$. If we assume our first guesses an inductive hypothesis that is true for input sizes smaller than n, then we have:

$$\begin{aligned} T(n) &= 2T(n/2) + b\, n \log n \\ &\le 2(c(n/2)\log(n/2)) + b\, n \log n \\ &\le c\, n(\log n - \log 2) + b\, n \log n \\ &\le c\, n \log n - c\, n + b\, n \log n. \end{aligned}$$

But there is no way that we can make this last line less than or equal to $c\, n \log n$ for $n \ge 2$. Thus, this first guess was not sufficient. Let us therefore try:

Better guess: $T(n) \le c\, n \log^2 n$.

for some constant $c > 0$. We can again choose c large enough to make this true for the base case, so consider the case when $n \ge 2$. If we assume this guess as an inductive hypothesis that is true for input sizes smaller than n, then we have inductive hypothesis that is true for input sizes smaller than n, then we have:

$$\begin{aligned} T(n) &= 2T(n/2) + b\, n \log n \\ &\le 2(c(n/2)\log^2(n/2)) + b\, n \log n \\ &\le c\, n(\log^2 n - 2\log n + 1) + b\, n \log n \\ &\le c\, n \log^2 n - 2c\, n \log n + c\, n + b\, n \log n \\ &\le c\, n \log^2 n \end{aligned}$$

Provided $c \ge b$. Thus, we have shown that $T(n)$ is indeed $O(n \log^2 n)$ in this case.
We must take care in using this method. Just because one inductive hypothesis for $T(n)$ does not work, that does not necessarily imply that another one proportional to this one will not work.

Example 2.10.2: Consider the following recurrence equation (assuming the base case T(n) = b for n < 2): T (n) = 2T (n/2) + log n

This recurrence is the running time for the bottom-up heap construction. Which is O(n). Nevertheless, if we try to prove this fact with the most straightforward inductive hypothesis, we will run into some difficulties. In particular, consider the following:

First guess: $T(n) \leq c\, n$.

For some constant c > 0. We can choose c large enough to make this true for the base case, so consider the case when n ≥ 2. If we assume this guess as an inductive hypothesis that is true of input sizes smaller than n, then we have:

$$\begin{aligned} T(n) &= 2T(n/2) + \log n \\ &\leq 2\,(c\,(n/2)) + \log n \\ &= c\,n + \log n \end{aligned}$$

But there is no way that we can make this last line less than or equal to cn for n > 2. Thus, this first guess was not sufficient, even though T (n) is indeed O (n). Still, we can show this fact is true by using:

Better guess: $T(n) \leq c\,(n - \log n)$

For some constant c > 0. We can again choose c large enough to make this true for the base case; in fact, we can show that it is true any time n < 8. So consider the case when n ≥ 8. If we assume this guess as an inductive hypothesis that is true for input sizes smaller than n, then we have:

$$\begin{aligned} T(n) &= 2T(n/2) + \log n \\ &\leq 2\,c\,((n/2) - \log(n/2)) + \log n \\ &= c\,n - 2\,c \log n + 2\,c + \log n \\ &= c\,(n - \log n) - c \log n + 2\,c + \log n \\ &\leq c\,(n - \log n) \end{aligned}$$

Provided c ≥ 3 and n ≥ 8. Thus, we have shown that T (n) is indeed O (n) in this case.

2.11. The Master Theorem Method:

Each of the methods described above for solving recurrence equations is ad hoc and requires mathematical sophistication in order to be used effectively. There is, nevertheless, one method for solving divide-and-conquer recurrence equations that is quite general and does not require explicit use of induction to apply correctly. It is the master method. The master method is a "cook-book" method for determining the asymptotic characterization of a wide variety of recurrence equations. It is used for recurrence equations of the form:

$$T(n) = \begin{cases} c & \text{if } n < d \\ a\,T(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

Where d > 1 is an integer constant, a > 0, c > 0, and b > 1 are real constants, and f (n) is a function that is positive for n ≥ d.

The master method for solving such recurrence equations involves simply writing down the answer based on whether one of the three cases applies. Each case is distinguished by comparing f (n) to the special function $n^{\log_b a}$ (we will show later why this special function is so important.

The master theorem: Let f (n) and T (n) be defined as above.

1. If there is a small constant ε > 0, such that f (n) is $O(n^{\log_b a - \varepsilon})$, then T (n) is $\Theta\left(n^{\log_b a}\right)$.
2. If there is a constant K ≥ 0, such that f (n) is $\Theta\left(n^{\log_b a} \log^k n\right)$, then T (n) is $\Theta\left(n^{\log_b a} \log^{k+1} n\right)$.
3. If there are small constant ε > 0 and δ < 1, such that f (n) is $\Omega\left(n^{\log_b a + \varepsilon}\right)$ and af(n/b) < δf(n), for n ≥ d, then T (n) is Θ (f(n)).

Case 1 characterizes the situation where f (n) is polynomial smaller than the special function, $n^{\log_b a}$.

Case 2 characterizes the situation when f (n) is asymptotically close to the special function, and

Case 3 characterizes the situation when f (n) is polynomially larger than the special function.

We illustrate the usage of the master method with a few examples (with each taking the assumption that T (n) = c for n < d, for constants c > 1 and d > 1).

Example 2.11.1: Consider the recurrence T (n) = 4 T (n/2) + n

In this case, $n^{\log_b a} = n^{\log_2 4} = n^2$. Thus, we are in case 1, for f (n) is O ($n^{2-\varepsilon}$) for ε = 1. This means that T (n) is Θ (n^2) by the master.

Example 2.11.2: Consider the recurrenceT (n) = 2 T (n/2) + n log n

In this case, $n^{\log_b a} = n^{\log_2 2} = n$. Thus, we are in case 2, with k=1, for f (n) is Θ (n log n). This means that T (n) is Θ (n $\log^2 n$) by the master method.

Example 2.11.3: consider the recurrence T (n) = T (n/3) + n

In this case $n^{\log_b a} = n^{\log_3 1} = n^0 = 1$. Thus, we are in Case 3, for f (n) is $\Omega(n^{0+\varepsilon})$, for ε = 1, and af(n/b) = n/3 = (1/3) f(n). This means that T (n) is Θ (n) by the master method.

Example 2.11.4: Consider the recurrence T (n) = 9 T (n/3) + $n^{2.5}$

In this case, $n^{\log_b a} = n^{\log_3 9} = n^2$. Thus, we are in Case 3, since f (n) is $\Omega(n^{2+\varepsilon})$ (for ε=1/2) and af(n/b) = 9 $(n/3)^{2.5}$ = $(1/3)^{1/2}$ f(n). This means that T (n) is Θ ($n^{2.5}$) by the master method.

Example 2.11.5: Finally, consider the recurrence T (n) = 2T ($n^{1/2}$) + log n
Unfortunately, this equation is not in a form that allows us to use the master method. We can put it into such a form, however, by introducing the variable k = log n, which lets us write:

$$T(n) = T(2^k) = 2\,T(2^{k/2}) + k$$

Substituting into this the equation S (k) = T (2^k), we get that

S (k) = 2 S (k/2) + k

Now, this recurrence equation allows us to use master method, which specifies that S (k) is O (k log k). Substituting back for T (n) implies T (n) is O (log n log log n).

2.12. CLOSED FROM EXPRESSION

There exists a closed form expression for many summations

Sum of first n natural numbers (Arithmetic progression)

$$\sum_{i=1}^{n} i = 1 + 2 + \ldots\ldots\ldots + n = \frac{n(n+1)}{2}$$

Sum of squares of first n natural numbers

$$\sum_{i=1}^{n} i^2 = 1 + 4 + 9 + \ldots\ldots\ldots + n^2 = \frac{n(n+1)\,(2n+1)}{6}$$

Sum of cubes

$$\sum_{i=1}^{n} i^3 = 1^3 + 2^3 + 3^3 + \ldots\ldots\ldots + n^3 = \frac{n^2\,(n+1)^2}{4}$$

Geometric progression

$$\sum_{i=0}^{n} 2^i = 2^0 + 2^1 + 2^2 + \ldots\ldots\ldots + 2^n = 2^{n+1} - 1$$

$$\sum_{i=0}^{n} r^i = \frac{r^{n+1} - 1}{r - 1}$$

$$\sum_{i=1}^{n} r^i = \frac{r^n - 1}{r - 1}$$

2.13. SOLVING RECURRENCE RELATIONS

Example 2.13.1. Solve the following recurrence relation:

$$T(n) = \begin{cases} 2 & , n = 1 \\ 2.T\left(\frac{n}{2}\right) + 7 & , n > 1 \end{cases}$$

Solution: We first start by labeling the main part of the relation as Step 1:

Step 1: Assume n > 1 then,

$$T(n) = 2.T\left(\frac{n}{2}\right) + 7$$

Step 2: Figure out what $T\left(\frac{n}{2}\right)$ is; everywhere you see n, replace it with $\frac{n}{2}$.

$$T\left(\frac{n}{2}\right) = 2.T\left(\frac{n}{2^2}\right) + 7$$

Now substitute this back into the last T (n) definition (last line from step 1):

$$T(n) = 2.\left[2.\ T\left(\frac{n}{2^2}\right) + 7\right] + 7$$

$$= 2^2.T\left(\frac{n}{2^2}\right) + 3.7$$

Step 3: let's expand the recursive call, this time, it's $T\left(\frac{n}{2^2}\right)$:

$$T\left(\frac{n}{2^2}\right) = 2.T\left(\frac{n}{2^3}\right) + 7$$

Now substitute this back into the last T(n) definition (last line from step 2):

$$T(n) = 2^2.\left[2.\ T\left(\frac{n}{2^3}\right) + 7\right] + 7$$

$$2^3.T\left(\frac{n}{2^3}\right) + 7.7$$

From this, first, we notice that the power of 2 will always match i, current. Second, we notice that the multiples of 7 match the relation $2^i - 1$: 1, 3, 7, 15. So, we can write a general solution describing the state of T (n).

Step 4: Do the i^{th} substitution.

$$T(n) = 2^i.\ T\left(\frac{n}{2^i}\right) + \left(2^i - 1\right).\ 7$$

However, how many times could we take these "steps"? Indefinitely? No… it would stop when n has been cut in half so many times that it is effectively 1. Then, the original definition of the recurrence relation would give us the terminating condition T (1) = 1 and us restrict size n = 2^i

When, $1 = \frac{n}{2^i}$

$$\Rightarrow 2^i = n$$

$$\Rightarrow \log_2 2^i = \log_2 n$$

$$\Rightarrow i.\log_2 2 = \log_2 n$$

$$\Rightarrow i = \log_2 n$$

Now we can substitute this "last value for i" back into a our general Step 4 equation:

$$\begin{aligned} T(n) &= 2^i . T\left(\frac{n}{2^i}\right) + \left(2^i - 1\right) . 7 \\ &= 2^{\log_2 n} . T\left(\frac{n}{2^{\log_2 n}}\right) + \left(2^{\log_2 n} - 1\right) . 7 \\ &= n . T(1) + (n - 1) . 7 \\ &= n . 2 + (n - 1) . 7 \\ &= 9 . n - 7 \end{aligned}$$

This implies that T (n) is **O (n).**

Example 2.13.2. Imagine that you have a recursive program whose run time is described by the following recurrence relation:

$$T(n) = \begin{cases} 1 & , n = 1 \\ 2.T\left(\frac{n}{2}\right) + 4.n & , n > 1 \end{cases}$$

Solve the relation with iterated substitution and use your solution to determine a tight big-oh bound.

Solution:

Step 1: Assume n > 1 then,

$$T(n) = 2 . T\left(\frac{n}{2}\right) + 4 . n$$

Step 2: figure out what $T\left(\frac{n}{2}\right)$ is:

$$T\left(\frac{n}{2}\right) = 2 . T\left(\frac{n}{2^2}\right) + 4 . \frac{n}{2}$$

Now substitute this back into the last T (n) definition (last line from step 1):

$$T(n) = 2 \;.\; \left[2 \;.\; T\left(\frac{n}{2^2}\right) + 4 \;.\; \frac{n}{2}\right] \; + \; 4 \;.\; n$$

$$= 2^2 \;.\; T\left(\frac{n}{2^2}\right) + 2 \;.\; 4 \;.\; n$$

Step 3: figure out what $T\left(\frac{n}{2^2}\right)$ is:

$$T\left(\frac{n}{2^2}\right) = 2 \;.\; T\left(\frac{n}{2^3}\right) + 4 \;.\; \frac{n}{2^2}$$

Now substitute this back into the last T (n) definition (last time from step 2):

$$T(n) = 2^2 \;.\; \left[2 \;.\; T\left(\frac{n}{2^3}\right) + 4 \;.\; \frac{n}{2^2}\right] \; + \; 2 \;.\; 4 \;.\; n$$

$$= 2^3 \;.\; T\left(\frac{n}{2^3}\right) + 3 \;.\; 4 \;.\; n$$

Step 4: Do the ith substitution.

$$\mathrm{T}(\mathrm{n}) = 2^i \;.\; T\left(\frac{n}{2^i}\right) + i \;.\; 4 \;.\; n$$

The parameter to the recursive call in the last line will equal 1 when i = $\log_2$ n (use the same analysis as the previous example). In which case T (1) = 1 by the original definition of the recurrence relation.

Now we can substitute this back into a our general Step 4 equation:

$$\mathrm{T}(\mathrm{n}) = 2^i \;.\; T\left(\frac{n}{2^i}\right) + i \;.\; 4 \;.\; n$$

$$= 2^{\log_2 n} \;.\; T\left(\frac{n}{2^{\log_2 n}}\right) + \log_2 n \;.\; 4 \;.\; n$$

$$= n \;.\; T(1) + 4 \;.\; n \;.\; \log_2 n$$

$$= \mathrm{n} + 4 \;.\; \mathrm{n} \;.\; \log_2 \mathrm{n}$$

This implies that T (n) is **O(n log n).**

Example 2.13.3. Write a recurrence relation and solve the recurrence relation for the following fragment of program:

Write T(n) in terms of T for fractions of n, for example, T(n/k) or T(n - k).

int findmax (int a[ ], int start, int end)

```
{
        int mid, max1, max2;

        if (start == end)                              // easy case (2 steps)
                return (a [start]);

        mid = (start + end) / 2;                       // pre-processing (3)
        max1 = findmax (a, start, mid);                // recursive call: T(n/2) + 1
        max2 = findmax (a, mid+1, end);                // recursive call: T(n/2) + 1
        if (max1 >= max2)                              //post-processing (2)
                return (max1);
        else
                return (max2);
}
```

Solution:

The Recurrence Relation is:

$$T(n) = \begin{cases} 2, & \text{if } n = 1 \\ 2\,T(n/2) + 8, & \text{if } n > 1 \end{cases}$$

Solving the Recurrence Relation by **iteration method:**

Assume $n > 1$ then $T(n) = 2\,T(n/2) + 8$

Step 1: Substituting $n = n/2$, $n/4$ . . . . for n in the relation:

Since, $T(n/2) = 2\,T((n/2)/2) + 8$, and $T(n/4) = 2T((n/4)/2) + 8$,

$$T(n) = 2\,[\quad T(n/2) \quad] + 8 \qquad (1)$$

$$T(n) = 2\,[\,2 \quad T(n/4) \quad + \; 8\,] \quad + 8 \qquad (2)$$

$$T(n) = 2\,[\,2\,[\,2\,T(n/8) \quad + 8] \; + \; 8\,] \quad + 8 \qquad (3)$$

$$= 2x2x2\,T(n/2x2x2) + 2x2x8 + 2x8 + 8,$$

Step 2: Do the i^{th} substitution:

$$T(n) = 2^i\,T(n/2^i) + 8\,(2^{i-1} + \ldots + 2^2 + 2 + 1)$$

$$= 2^i\,T(n/2^i) + 8\,\left(\sum_{k=0}^{i-1} 2^k\right)$$

$$= 2^i\,T(n/2^i) + 8\,(2^i - 1) \quad \text{(the formula for geometric series)}$$

Step 3: Define i in terms of n:

If $n = 2^i$, then $i = \log n$ so, $T(n/2^i) = T(1) = 2$

$$\begin{aligned} T(n) &= 2^i\,T(n/2^i) + 8\,(2^i - 1) \\ &= n\,T(1) + 8\,(n-1) \\ &= 2n + 8n - 8 = 10n - 8 \end{aligned}$$

Step 4: Representing in big-O function:

T (n) = 10n - 8 is **O(n)**

Example 2.13.4. If K is a non negative constant, then prove that the recurrence

$$T(n) = \begin{cases} k & , n = 1 \\ 3 . T\left(\frac{n}{2}\right) + k . n & , n > 1 \end{cases}$$

has the following solution (for n a power of 2)

$$T(n) = 3k \; . \; n^{\log_2 3} \; - 2k \; .n$$

Solution:

Assuming n > 1, we have T (n) = $3T\left(\frac{n}{2}\right) + K \; . \; n$ (1)

Substituting $n = \frac{n}{2}$ for n in equation (1), we get

$$T\left(\frac{n}{2}\right) = 3T\left(\frac{n}{4}\right) + k\,\frac{n}{2} \qquad (2)$$

Substituting equation (2) for $T\left(\frac{n}{2}\right)$ in equation (1)

$$\text{T (n)} = 3\left(3T\left(\frac{n}{4}\right) + k\,\frac{n}{2}\right) + k \; . \; n$$

$$T(n) = 3^2\, T\left(\frac{n}{4}\right) + \frac{3}{2}\, k . \; n + k \; . \; n \qquad (3)$$

Substituting $n = \frac{n}{4}$ for n equation (1)

$$T\left(\frac{n}{4}\right) = 3T\left(\frac{n}{8}\right) + k\,\frac{n}{4} \qquad (4)$$

Substituting equation (4) for $T = \frac{n}{4}$ in equation (3)

$$T(n) = 3^2\left(3\,T\left(\frac{n}{8}\right) + k\,\frac{n}{4}\right) + \frac{3}{2}\, k \; . \; n + k \; . \; n$$

$$3^3\, T\left(\frac{n}{8}\right) + \frac{9}{4}\, k . n + \frac{3}{2}\, k . n + k . n \qquad (5)$$

Continuing in this manner and substituting n = 2^i, we obtain

$$T(n) = 3^i\ T\left(\frac{n}{2^i}\right) + \frac{3^{i-1}}{2^{i-1}}\ k.n + \ldots\ldots\ldots + \frac{9kn}{4} + \frac{3}{2}\ k.\ n + k.n$$

$$T(n) = 3^i\, T\left(\frac{n}{2^i}\right) + \left(\frac{\left(\frac{3}{2}\right)^i - 1}{\frac{3}{2} - 1}\right)\ k\ .\ n \qquad \text{as} \quad \sum_{i=0}^{n-1} r^i \ = \frac{r^n - 1}{r - 1}$$

$$= 3^i\, T\left(\frac{n}{2^i}\right) + 2\ .\left(\frac{3}{2}\right)^i\ k\ .\ n - 2k\ n \qquad (6)$$

As n = 2^i , then i = $\log_2 n$ and by definition as T (1) = k

$$T(n) = 3^i\ k + 2.\frac{3^i}{n}\ .\ kn - 2kn$$

$$= 3^i\ (3k) - 2kn$$

$$= 3k\ .\ 3^{\log_2 n} - 2kn$$

$$= 3kn^{\log_2 3} - 2kn$$

Example 2.13.5. Towers of Hanoi

The Towers of Hanoi is a game played with a set of donut shaped disks stacked on one of three posts. The disks are graduated in size with the largest on the bottom. The objective of the game is to transfer all the disks from post B to post A moving one disk at a time without placing a larger disk on top of a smaller one. What is the minimum number of moves required when there are n disks?

In order to visualize the most efficient procedure for winning the game consider the following:

1. Move the first n-1 disks, as per the rules in the most efficient manner possible, from post B to post C.

2. Move the remaining, largest disk from post B to post A.

3. Move the n - 1 disks, as per the rules in the most efficient manner possible, from post C to post A.

Let m_n be the number of moves required to transfer n disks as per the rules in the most efficient manner possible from one post to another. Step 1 requires moving n - 1 disks or m_{n-1} moves. Step 2 requires 1 move. Step 3 requires m_{n-1} moves again. We have then,

$M_n = m_{n-1} + 1 + m_{n-1}$, for $n > 2 = 2m_{n-1} + 1$

Because only one move is required to win the game with only one disk, the initial condition for this sequence is $m_1 = 1$. Now we can determine the number of moves required to win the game, in the most efficient manner possible, for any number of disks.

$m_1 = 1$

$m_2 = 2(1) + 1 = 3$

$m_3 = 2(3) + 1 = 7$

$m_4 = 2(7) + 1 = 15$

$m_5 = 2(15) + 1 = 31$

$m_6 = 2(31) + 1 = 63$

$m_7 = 2(63) + 1 = 127$

Unfortunately, the recursive method of determining the number of moves required is exhausting if we wanted to solve say a game with 69 disks. We have to calculate each previous term before we can determine the next.

Lets look for a pattern that may be useful in determining an explicit formula for m_n. In looking for patterns it is often useful to forego simplification of terms. $m_n = 2m_{n-1} + 1$, $m_1 = 1$

$m_1 = 1$

$m_2 = 2(1) + 1 = 2+1$

$m_3 = 2(2+1) + 1 = 2^2+2+1$

$m_4 = 2(2^2+2+1) + 1 = 2^3+2^2+2+1$

$m_5 = 2(2^3+2^2+2+1) + 1 = 2^4+2^3+2^2+2+1$

$m_6 = 2(2^4+2^3+2^2+2+1) + 1 = 2^5+2^4+2^3+2^2+2+1$

$m_7 = 2(2^5+2^4+2^3+2^2+2+1) + 1 = 2^6+2^5+2^4+2^3+2^2+2+1$

So we guess an explicit formula:

$$M_k = 2^{k-1} + 2^{k-2} + \ldots\ldots + 2^2 + 2 + 1$$

By sum of a Geometric Sequence, for any real number r except 1, and any integer $n \geq 0$.

$$\sum_{i=0}^{n} r^i = \frac{r^{n+1}-1}{r-1}$$

our formula is then

$$m_k = \sum_{k=0}^{n-1} 2^k = \frac{2^n-1}{2-1} = 2^n - 1$$

This is nothing but **O (2^n)**

Thus providing an explicit formula for the number of steps required to win the Towers of Hanoi Game.

Example 2.13.6 Solve the recurrence relation $T (n) = 5 T (n/5) + n/\log n$

Using iteration method,

$T (n) = 25 T (n/25) + n/\log (n/5) + n/\log n$

$=125 T (n/125) + n/ \log (n/25) + n/\log (n/5) + n/\log n$

$= 5^k T (n/5^k) + n/\log (n/5^{k-1}) + n/\log (n/5^{k-2}) \ldots . + n/\log n$

When $n = 5^k$

$= n * T(1) + n/\log 5 + n/\log 25 + n/\log 125 \ldots .. + n/\log n$

$= c n + n (1/\log 5 + 1/\log 25 + 1/\log 125 + \ldots .. 1/\log n)$

$= c n + n \log_5 2 (1/1 + 1/2 + 1/3 + \ldots . 1/k)$

$= c n + \log_5 2 n \log (k)$

$= c n + \log_5 2 n \log (\log n)$

$= \theta (n \log (\log n))$

Example 2.13.7. Solve the recurrence relation $T (n) = 3 T (n/3 + 5) + n/2$

Use the substitution method, guess that the solution is $T (n) = O (n \log n)$

We need to prove $T (n) < = c n \log n$ for some c and all $n > n_o$.

Substitute $c (n/3 + 5) \log (n/3 + 5)$ for $T (n/3 + 5)$ in the recurrence

$T (n) <= 3 * c (n/3 + 5) * (\log (n/3 + 5)) + n/2$

If we take n_0 as 15, then for all $n > n_0$, $n/3+5 <= 2n/3$, so

$T (n) <= (c n + 15 c) * (\log n/3 + \log 2) + n/2$

$<= c n \log n - c n \log 3 + 15 c \log n - 15 c \log 3 + c n + 15 c + n/2$

$<= c n \log n - (c n (\log 3-1) + n/2 - 15 c \log n - 15 c (\log 3 - 1)$

$<= c n \log n$, by choosing an appropriately large constant c

Which implies $T (n) = O (n \log n)$

Similarly, by guessing $T (n) >= c_1 n \log n$ for some c_1 and all $n > n_0$ and substituting in the recurrence, we get.

$T (n) >= c_1 n \log n - c_1 n \log 3 + 15 c_1 \log n - 15 c_1 \log 3 + c_1 n + 15 c_1 + n/2$

$>= c_1 n \log n + n/2 + c_1 n (1 - \log 3) + 15 c_1 + 15 c_1 (\log n - \log 3)$

$>= c_1$ n log n + n (0.5 + c_1 (1 - log 3) + 15 c_1 + 15 c_1 (log n - log 3)

by choosing an appropriately small constant c_1 (say c_1 = 0.01) and for all n > 3

T (n) >= c1 n log n

Which implies T (n) = Ω (n log n)

Thus, T (n) = θ (n log n)

Example 2.13.8. Solve the recurrence relation T (n) = 2 T (n/2) + n/log n.

Using iteration method,

T (n) = 4 T (n/4) + n/log (n/2) + n/log n

= 8 T (n/8) + n/log (n/4) + n/log (n/2) + n/log n

= 2^k T (n/2^k) + n/log (n/2^{k-1}) + n/log (n/2^{k-2}) . . . . . + n/log n

When n = 2^k

= n * T(1) + n/log 2 + n/log 4 + n/log 8 . . . . . + n/log n

= c n + n (1/log 2 + 1/log 4 + 1/log 8 + ... 1/log n)

= c n + n (1/1 + 1/2 + 1/3 + ... 1/k)

= c n + n log (k)

= c n + n log (log n)

= θ (n log (log n))

Example 2.13.9: Solve the recurrence relation:

T (n) = T (n/2) + T (n/4) + T (n/8) + n

Use the substitution method, guess that the solution is T (n) = O (n log n).

We need to prove T (n) <= c n log n for some c and all n > no.

Substituting the guess in the given recurrence, we get

T (n) <= c (n/2) log (n/2) + c (n/4) log (n/4) + c (n/8) log (n/8) + n

<= c (n/2) (log n - 1) + c (n/4) (log n - 2) + c (n/8) (log n - 3) + n

<= c n log n (1/2 + 1/4 + 1/8) – c n/2 – c n/2 – 3 c n/8 + n

<= c n log n (7/8) - (11 c n – 8 n) / 8

<= c n log n (7/8)   (if c = 1)

<= c n log n

From the recurrence equation, it can be inferred that T (n) > = n. So, T (n) = Ω (n)

Example 2.13.10: Solve the recurrence relation: T (n) = 28 T (n/3) + cn^3

Let us consider : $n^{\log_b^a} \Rightarrow \frac{f(n)}{n^{\log_b^a}} = \frac{cn^3}{n^{\log_3^2}}$

According to the law of indices: $\frac{a^x}{a^y}$, we can write a^{x-y}

$$\frac{c\,n^3}{n^{\log_3^{28}}} = c\,n^{3-\log_3^{28}} = c\ n^r \qquad \text{where } r = 3 - \log_3{}^{28} < 0$$

It can be written as: h (n) = O (n^r) where r < 0

f (n) for these from the table can be taken O (1)

$$T(n) = n\log_3^{28}\ [T(1) + h(n)]$$

$$= n\log_3^{28}\ [T(1) + 0(1)] = \theta\left(n^{\log_3^{28}}\right)$$

Example 2.13.12: Solve the recurrence relation T (n) = T (√n) + c. n > 4

$$T(n) = T(n^{1/2}) + C$$

$$T(n)^{1/2} = T(n^{1/2})^{1/2} + C$$

$$T(n^{1/2}) = T(n^{1/4}) + C + C$$

$$T(n) = T(n^{1/4}) + 2c$$

$$T(n) = T(n^{1/2}) + 3c$$

$$= T(n^{1/2i}) + i\ c$$

$$= T(n^{1/n}) + C\log_2 n$$

$$T\left(\sqrt[n]{n}\right) + C\ \log_2 n = \theta\ \textbf{(log n)}$$

Example 2.13.13: Solve the recurrence relation

$$T(n) = \begin{cases} 1 & n \le 4 \\ 2T(\sqrt{n}) + \log n & n > 4 \end{cases}$$

$$T(n) = 2\left\lfloor 2T\ (n^{1/2})^{1/2} + \log\sqrt{n}\right\rfloor + \log n$$

$$= 4\ T\ (n^{1/4}) + 2 \log n^{1/2} + \log n$$

$$T\ (n^{1/4}) = 2\ T\ (n^{1/2})^{1/4} + \log n^{1/4}$$

$$T(n) = 4\left[\ 2T\left(n^{1/2}\right)^{1/4} + \log n^{1/4}\right] + 2\log n^{1/4} + \log n$$

$$= 8\ T\ (n^{1/8}) + 4 \log n^{1/4} + 2 \log n^{1/2} + \log n$$

$$T\ (n^{1/8}) = 2\ T\ (n^{1/2}\ )^{1/8} + \log n^{1/8}$$

$$T(n) = 8\left\lfloor 2T\,(n^{1/16}) + \log n^{1/8}\right\rfloor + 4\log n^{1/4} + 2\log n^{1/2} + \log n$$

$$= 16\ T\ (n^{1/6}) + 8 \log n^{1/8} + 4 \log n^{1/4} + 2 \log n^{1/2} + \log n$$

$$= 2^i\ T(n^{1/2^i}) + 2^{i-1} \log n^{1/2^{i-1}} + 2^{i-2} \log n^{1/2^{i-2}} + 2^{i-3} \log n^{1/2^{i-3}} + 2^{i-4} \log n^{1/2^{i-4}}$$

$$= 2^i\ T\left(n^{1/2^i}\right) + \sum_{k=1}^{n} 2^{i-k} \log n^{1/2^{i-k}}$$

$$= nT\left(n^{1/n}\right) + \sum \frac{n}{2^k} \log n^{k/n}$$

$$= \theta\ \mathbf{(log\ n)}$$

Chapter 3

Divide and Conquer

3.1. General Method

Divide and conquer is a design strategy which is well known to breaking down efficiency barriers. When the method applies, it often leads to a large improvement in time complexity. For example, from O (n^2) to O (n log n) to sort the elements.

Divide and conquer strategy is as follows: divide the problem instance into two or more smaller instances of the same problem, solve the smaller instances recursively, and assemble the solutions to form a solution of the original instance. The recursion stops when an instance is reached which is too small to divide. When dividing the instance, one can either use whatever division comes most easily to hand or invest time in making the division carefully so that the assembly is simplified.

Divide and conquer algorithm consists of two parts:

Divide : Divide the problem into a number of sub problems. The sub problems are solved recursively.

Conquer : The solution to the original problem is then formed from the solutions to the sub problems (patching together the answers).

Traditionally, routines in which the text contains at least two recursive calls are called divide and conquer algorithms, while routines whose text contains only one recursive call are not. Divide–and–conquer is a very powerful use of recursion.

3.2. Control Abstraction of Divide and Conquer

A control abstraction is a procedure whose flow of control is clear but whose primary operations are specified by other procedures whose precise meanings are left undefined. The control abstraction for divide and conquer technique is DANDC(P), where P is the problem to be solved.

```
DANDC (P)
{
        if SMALL (P) then return S (p);
        else
        {
                divide p into smaller instances $p_1$, $p_2$, …. $P_k$, $k \geq 1$;
                apply DANDC to each of these sub problems;
                return (COMBINE (DANDC ($p_1$) , DANDC ($p_2$),…., DANDC ($p_k$));
        }
}
```

SMALL (P) is a Boolean valued function which determines whether the input size is small enough so that the answer can be computed without splitting. If this is so function 'S' is invoked otherwise, the problem 'p' into smaller sub problems. These sub problems p_1, p_2, . . . , p_k are solved by recursive application of DANDC.

If the sizes of the two sub problems are approximately equal then the computing time of DANDC is:

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ 2\,T(n/2) + f(n) & \text{otherwise} \end{cases}$$

Where, T (n) is the time for DANDC on 'n' inputs
g (n) is the time to complete the answer directly for small inputs and
f (n) is the time for Divide and Combine

3.3. Binary Search

If we have 'n' records which have been ordered by keys so that $x_1 < x_2 < \ldots < x_n$. When we are given a element 'x', binary search is used to find the corresponding element from the list. In case 'x' is present, we have to determine a value 'j' such that a[j] = x (successful search). If 'x' is not in the list then j is to set to zero (un successful search).

In Binary search we jump into the middle of the file, where we find key a[mid], and compare 'x' with a[mid]. If x = a[mid] then the desired record has been found. If x < a[mid] then 'x' must be in that portion of the file that precedes a[mid], if there at all. Similarly, if a[mid] > x, then further search is only necessary in that past of the file which follows a[mid]. If we use recursive procedure of finding the middle key a[mid] of the un-searched portion of a file, then every un-successful comparison of 'x' with a[mid] will eliminate roughly half the un-searched portion from consideration.

Since the array size is roughly halved often each comparison between 'x' and a[mid], and since an array of length 'n' can be halved only about $\log_2 n$ times before reaching a trivial length, the worst case complexity of Binary search is about $\log_2 n$

3.3.1. Algorithm

```
Algorithm BINSRCH (a, n, x)
//      array a(1 : n) of elements in increasing order, n ≥ 0,
//      determine whether 'x' is present, and if so, set j such that x = a(j)
//      else return j

{
        low :=1 ; high :=n ;
        while (low ≤ high) do
        {
                mid :=|(low + high)/2|
                if (x < a [mid]) then high:=mid – 1;
                else if (x > a [mid]) then low:= mid + 1
                        else return mid;
        }
        return 0;
 }
```

low and *high* are integer variables such that each time through the loop either 'x' is found or *low* is increased by at least one or *high* is decreased by at least one. Thus we have two sequences of integers approaching each other and eventually *low* will become greater than *high* causing termination in a finite number of steps if 'x' is not present.

3.3.2. Example for Binary Search

Let us illustrate binary search on the following 9 elements:

Index	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
Elements	-15	-6	0	7	9	23	54	82	101

The number of comparisons required for searching different elements is as follows:

1. Searching for x = 101

low	high	mid
1	9	5
6	9	7
8	9	8
9	9	9
		found

Number of comparisons = 4

2. Searching for x = 82

low	high	mid
1	9	5
6	9	7
8	9	8
		found

Number of comparisons = 3

3. Searching for x = 42

low	high	mid
1	9	5
6	9	7
6	6	6
7	6	not found

Number of comparisons = 4

4. Searching for x = -14

low	high	mid
1	9	5
1	4	2
1	1	1
2	1	not found

Number of comparisons = 3

Continuing in this manner the number of element comparisons needed to find each of nine elements is:

Index	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
Elements	-15	-6	0	7	9	23	54	82	101
Comparisons	*3*	*2*	*3*	*4*	*1*	*3*	*2*	*3*	*4*

No element requires more than 4 comparisons to be found. Summing the comparisons needed to find all nine items and dividing by 9, yielding 25/9 or approximately 2.77 comparisons per successful search on the average.

There are ten possible ways that an un-successful search may terminate depending upon the value of x.

If $x < a[1]$, $a[1] < x < a[2]$, $a[2] < x < a[3]$, $a[5] < x < a[6]$, $a[6] < x < a[7]$ or $a[7] < x < a[8]$ the algorithm requires 3 element comparisons to determine that 'x' is not present. For all of the remaining possibilities BINSRCH requires 4 element comparisons. Thus the average number of element comparisons for an unsuccessful search is:

$(3 + 3 + 3 + 4 + 4 + 3 + 3 + 3 + 4 + 4) / 10 = 34/10 = 3.4$

The time complexity for a successful search is $O(\log n)$ and for an unsuccessful search is $\Theta(\log n)$.

Successful searches			**un-successful searches**
$\Theta(1)$,	$\Theta(\log n)$,	$\Theta(\log n)$	$\Theta(\log n)$
Best	average	worst	best, average and worst

3.3.3. Analysis for worst case

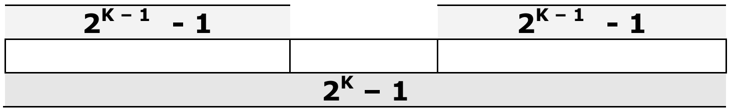
Let T (n) be the time complexity of Binary search

The algorithm sets mid to [n+1 / 2]

Therefore,

$$
\begin{aligned}
T(0) &= 0 \\
T(n) &= 1 && \text{if } x = a\,[mid] \\
&= 1 + T([(n + 1) / 2] - 1) && \text{if } x < a\,[mid] \\
&= 1 + T(n - [(n + 1)/2]) && \text{if } x > a\,[mid]
\end{aligned}
$$

Let us restrict 'n' to values of the form $n = 2^K - 1$, where 'k' is a non-negative integer. The array always breaks symmetrically into two equal pieces plus middle element.



Algebraically this is $\left[\frac{n + 1}{2}\right] = \left[\frac{2^K - 1 + 1}{2}\right] = 2^{K - 1}$ for $K \geq 1$

Giving,

$$
\begin{aligned}
T(0) &= 0 \\
T(2^k - 1) &= 1 && \text{if } x = a\,[mid] \\
&= 1 + T(2^{K-1} - 1) && \text{if } x < a\,[mid] \\
&= 1 + T(2^{k-1} - 1) && \text{if } x > a\,[mid]
\end{aligned}
$$

In the worst case the test x = a[mid] always fails, so

$$
\begin{aligned}
&w(0) = 0 \\
&w(2^k - 1) = 1 + w(2^{k-1} - 1)
\end{aligned}
$$

This is now solved by repeated substitution:

$$\begin{aligned} w(2^k - 1) &= 1 + w(2^{k-1} - 1) \\ &= 1 + [1 + w(2^{k-2} - 1)] \\ &= 1 + [1 + [1 + w(2^{k-3} - 1)]] \\ &= \ldots\ldots \\ &= \ldots\ldots \\ &= i + w(2^{k-i} - 1) \end{aligned}$$

For $i \leq k$, letting $i = k$ gives $w(2^k - 1) = K + w(0) = k$

But as $2^K - 1 = n$, so $K = \log_2(n + 1)$, so

$$w(n) = \log_2(n + 1) = O(\log n)$$

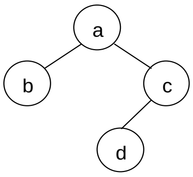
for $n = 2^K - 1$, concludes this analysis of binary search.

Although it might seem that the restriction of values of 'n' of the form 2^K-1 weakens the result. In practice this does not matter very much, w(n) is a monotonic increasing function of 'n', and hence the formula given is a good approximation even when 'n' is not of the form 2^K-1.

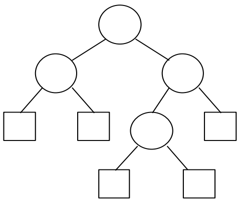
3.4. External and Internal path length:

The lines connecting nodes to their non-empty sub trees are called edges. A non- empty binary tree with n nodes has n–1 edges. The size of the tree is the number of nodes it contains.

When drawing binary trees, it is often convenient to represent the empty sub trees explicitly, so that they can be seen. For example:

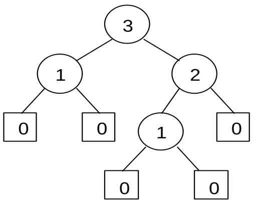


The tree given above in which the empty sub trees appear as square nodes is as follows:

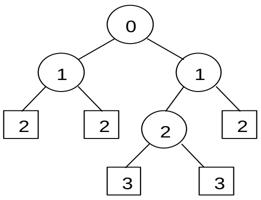


The square nodes are called as external nodes E(T). The square node version is sometimes called an extended binary tree. The round nodes are called internal nodes I(T). A binary tree with n internal nodes has n+1 external nodes.

The height h(x) of node 'x' is the number of edges on the longest path leading down from 'x' in the extended tree. For example, the following tree has heights written inside its nodes:



The depth d(x) of node 'x' is the number of edges on path from the root to 'x'. It is the number of internal nodes on this path, excluding 'x' itself. For example, the following tree has depths written inside its nodes:



The internal path length I(T) is the sum of the depths of the internal nodes of 'T':

$$I(T) \quad = \sum_{x \,\in\, I(T)} d(x)$$

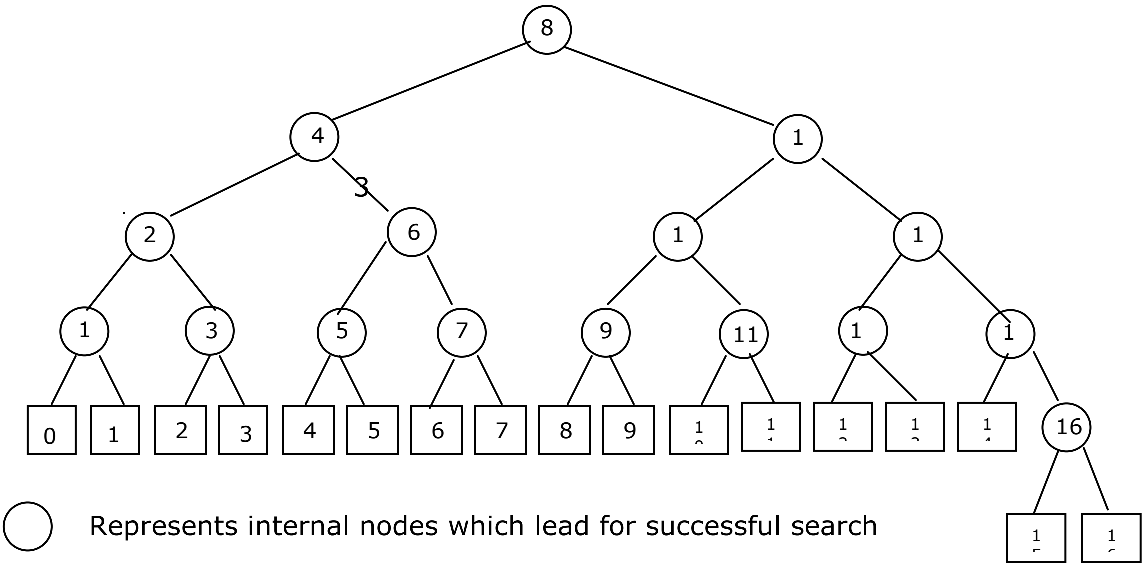
The external path length E(T) is the sum of the depths of the external nodes:

$$E(T) \quad = \sum_{x \,\in\, E(T)} d(x)$$

For example, the tree above has I(T) = 4 and E(T) = 12.

A binary tree T with 'n' internal nodes, will have I(T) + 2n = E(T) external nodes.

A binary tree corresponding to binary search when n = 16 is



○ Represents internal nodes which lead for successful search

□ External square nodes, which lead for unsuccessful search.

Let C_N be the average number of comparisons in a successful search.

C'_N be the average number of comparison in an un successful search.

Then we have,

$$C_N = 1 + \frac{\text{internal path length of tree}}{N}$$

$$C'_N = \frac{\text{External path length of tree}}{N + 1}$$

$$C_N = \left(1 + \frac{1}{N}\right) C'_N - 1$$

External path length is always 2N more than the internal path length.

3.5. Merge Sort

Merge sort algorithm is a classic example of divide and conquer. To sort an array, recursively, sort its left and right halves separately and then merge them. The time complexity of merge mort in the *best case, worst case* and *average case* is O(n log n) and the number of comparisons used is nearly optimal.

This strategy is so simple, and so efficient but the problem here is that there seems to be no easy way to merge two adjacent sorted arrays together in place (The result must be build up in a separate array).

The fundamental operation in this algorithm is merging two sorted lists. Because the lists are sorted, this can be done in one pass through the input, if the output is put in a third list.

The basic merging algorithm takes two input arrays ‘a’ and ’b’, an output array ‘c’, and three counters, *a ptr, b ptr* and *c ptr,* which are initially set to the beginning of their respective arrays. The smaller of *a[a ptr]* and *b[b ptr]* is copied to the next entry in ‘c’, and the appropriate counters are advanced. When either input list is exhausted, the remainder of the other list is copied to ‘c’.

To illustrate how merge process works. For example, let us consider the array ‘a’ containing 1, 13, 24, 26 and ‘b’ containing 2, 15, 27, 38. First a comparison is done between 1 and 2. 1 is copied to ‘c’. Increment *a ptr* and *c ptr.*

1	*2*	*3*	*4*
1	13	24	26
h ptr			

5	*6*	*7*	*8*
2	15	27	28
j ptr			

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1							
i ptr							

and then 2 and 13 are compared. 2 is added to ‘c’. Increment *b ptr* and *c ptr.*

1	*2*	*3*	*4*
1	13	24	26
	h ptr		

5	*6*	*7*	*8*
2	15	27	28
j ptr			

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1	2						
	i ptr						

then 13 and 15 are compared. 13 is added to ‘c’. Increment *a ptr* and *c ptr*.

1	*2*	*3*	*4*
1	13	24	26
	h *ptr*		

5	*6*	*7*	*8*
2	15	27	28
	j *ptr*		

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1	2	13					
		i *ptr*					

24 and 15 are compared. 15 is added to ‘c’. Increment *b ptr* and *c ptr*.

1	*2*	*3*	*4*
1	13	24	26
		h *ptr*	

5	*6*	*7*	*8*
2	15	27	28
	j *ptr*		

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1	2	13	15				
			i *ptr*				

24 and 27 are compared. 24 is added to ‘c’. Increment *a ptr* and *c ptr*.

1	*2*	*3*	*4*
1	13	24	26
		h *ptr*	

5	*6*	*7*	*8*
2	15	27	28
		j *ptr*	

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1	2	13	15	24			
				i *ptr*			

26 and 27 are compared. 26 is added to ‘c’. Increment *a ptr* and *c ptr.*

1	*2*	*3*	*4*
1	13	24	26
			h *ptr*

5	*6*	*7*	*8*
2	15	27	28
		j *ptr*	

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1	2	13	15	24	26		
					i *ptr*		

As one of the lists is exhausted. The remainder of the b array is then copied to ‘c’.

1	*2*	*3*	*4*
1	13	24	26

h
ptr

5	*6*	*7*	*8*
2	15	27	28
		j *ptr*	

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1	2	13	15	24	26	27	28

i
ptr

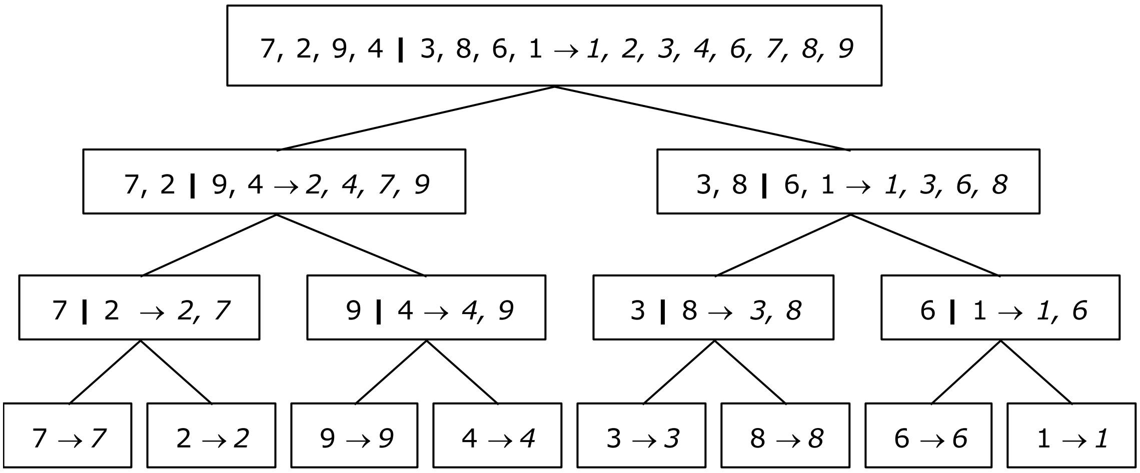
3.5.1. Algorithm

```
Algorithm MERGESORT (low, high)
// a (low : high) is a global array to be sorted.
{
        if (low < high)
        {
                mid := $\lfloor(low + high)/2\rfloor$            //finds where to split the set
                MERGESORT(low, mid)               //sort one subset
                MERGESORT(mid+1, high)   //sort the other subset
                MERGE(low, mid, high)              // combine the results
        }
}
```

```
Algorithm MERGE (low, mid, high)
// a (low : high) is a global array containing two sorted subsets
// in a (low : mid) and in a (mid + 1 : high).
// The objective is to merge these sorted sets into single sorted
// set residing in a (low : high). An auxiliary array B is used.
{
        h :=low; i := low; j:= mid + 1;
        while ((h ≤ mid) and (J ≤ high)) do
        {
                if (a[h] ≤ a[j]) then
                {
                        b[i] := a[h]; h := h + 1;
                }
                else
                {
                        b[i] :=a[j]; j := j + 1;
                }
                i := i + 1;
        }
        if (h > mid) then
                for k := j to high do
                {
                        b[i] := a[k]; i := i + 1;
                }
        else
                for k := h to mid do
                {
                        b[i]  := a[K];  i := i + l;
                }
        for k := low to high do
                a[k] := b[k];
}
```

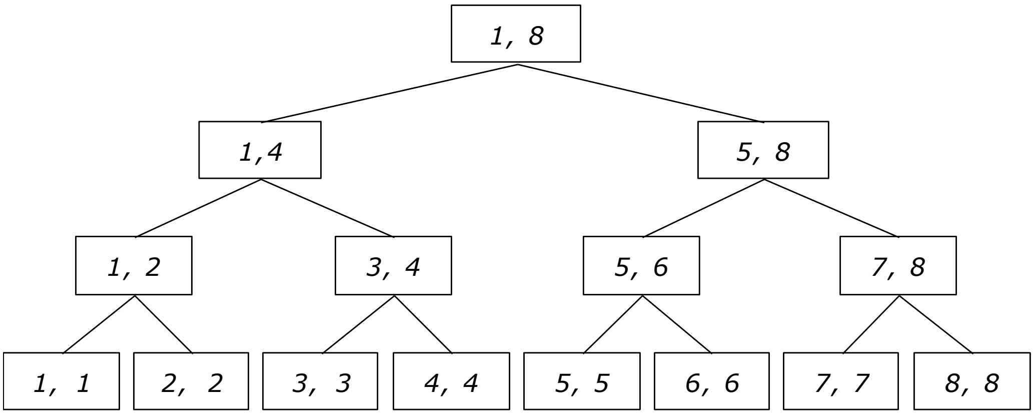
3.5.2. Example

For example let us select the following 8 entries 7, 2, 9, 4, 3, 8, 6, 1 to illustrate merge sort algorithm:



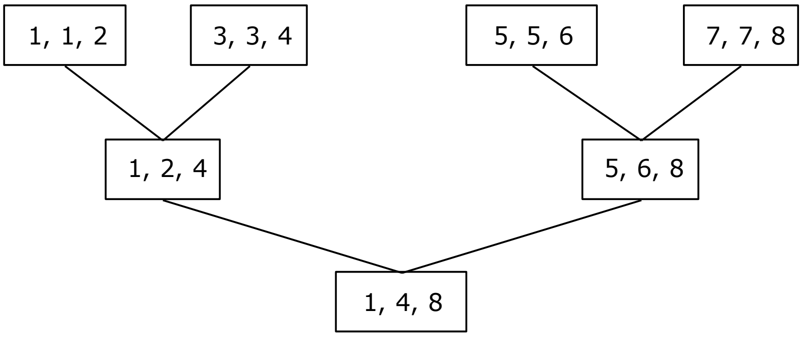
3.5.3. Tree Calls of MERGESORT(1, 8)

The following figure represents the sequence of recursive calls that are produced by MERGESORT when it is applied to 8 elements. The values in each node are the values of the parameters low and high.



3.5.3. Tree Calls of MERGE()

The tree representation of the calls to procedure MERGE by MERGESORT is as follows:



3.5.4. Analysis of Merge Sort

We will assume that ‘n’ is a power of 2, so that we always split into even halves, so we solve for the case $n = 2^k$.

For n = 1, the time to merge sort is constant, which we will be denote by 1. Otherwise, the time to merge sort ‘n’ numbers is equal to the time to do two recursive merge sorts of size n/2, plus the time to merge, which is linear. The equation says this exactly:

$$T(1) = 1$$
$$T(n) = 2\,T(n/2) + n$$

This is a standard recurrence relation, which can be solved several ways. We will solve by substituting recurrence relation continually on the right–hand side.

We have, $T(n) = 2T(n/2) + n$

Since we can substitute n/2 into this main equation

$$\begin{aligned} 2\,T(n/2) &= 2\,(2\,(T(n/4)) + n/2) \\ &= 4\,T(n/4) + n \end{aligned}$$

We have,

$$\begin{aligned} T(n/2) &= 2\,T(n/4) + n \\ T(n) &= 4\,T(n/4) + 2n \end{aligned}$$

Again, by substituting n/4 into the main equation, we see that

$$\begin{aligned} 4T\,(n/4) &= 4\,(2T(n/8)) + n/4 \\ &= 8\,T(n/8) + n \end{aligned}$$

So we have,

$$\begin{aligned} T(n/4) &= 2\,T(n/8) + n \\ T(n) &= 8\,T(n/8) + 3n \end{aligned}$$

Continuing in this manner, we obtain:

$$T(n) = 2^k\,T(n/2^k) + K.\,n$$

As $n = 2^k$, $K = \log_2 n$, substituting this in the above equation

$$\begin{aligned} T(n) &= 2^{\log_2 n}\;T\left(\frac{2^k}{2^k}\right) + \log_2 n\;.\;n \\ &= n\,T(1) + n \log n \\ &= n \log n + n \end{aligned}$$

Representing this in O notation:

T(n) = **O(n log n)**

We have assumed that $n = 2^k$. The analysis can be refined to handle cases when 'n' is not a power of 2. The answer turns out to be almost identical.

Although merge sort's running time is O(n log n), it is hardly ever used for main memory sorts. The main problem is that merging two sorted lists requires linear extra memory and the additional work spent copying to the temporary array and back, throughout the algorithm, has the effect of slowing down the sort considerably.
The Best and worst case time complexity of Merge sort is O(n log n).

3.6. Strassen's Matrix Multiplication:

The matrix multiplication of algorithm due to Strassens is the most dramatic example of divide and conquer technique (1969).

The usual way to multiply two n x n matrices A and B, yielding result matrix 'C' as follows :

```
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];
```

This algorithm requires n^3 scalar multiplication's (i.e. multiplication of single numbers) and n^3 scalar additions. So we naturally cannot improve upon.

We apply divide and conquer to this problem. For example let us considers three multiplication like this:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Then c_{ij} can be found by the usual matrix multiplication algorithm,

$$C_{11} = A_{11} \,.\, B_{11} + A_{12} \,.\, B_{21}$$

$$C_{12} = A_{11} \,.\, B_{12} + A_{12} \,.\, B_{22}$$

$$C_{21} = A_{21} \,.\, B_{11} + A_{22} \,.\, B_{21}$$

$$C_{22} = A_{21} \,.\, B_{12} + A_{22} \,.\, B_{22}$$

This leads to a divide–and–conquer algorithm, which performs nxn matrix multiplication by partitioning the matrices into quarters and performing eight (n/2)x(n/2) matrix multiplications and four (n/2)x(n/2) matrix additions.

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 8\,T(n/2) \end{aligned}$$

Which leads to T (n) = O (n^3), where n is the power of 2.

Strassens insight was to find an alternative method for calculating the C_{ij}, requiring seven (n/2) x (n/2) matrix multiplications and eighteen (n/2) x (n/2) matrix additions and subtractions:

$$P = (A_{11} + A_{22})\,(B_{11} + B_{22})$$

$$Q = (A_{21} + A_{22})\,B_{11}$$

$$R = A_{11}\,(B_{12} - B_{22})$$

$$S = A_{22}\,(B_{21} - B_{11})$$

$$T = (A_{11} + A_{12})\,B_{22}$$

$$U = (A_{21} - A_{11})\,(B_{11} + B_{12})$$

$$V = (A_{12} - A_{22})\,(B_{21} + B_{22})$$

$$C_{11} = P + S - T + V$$

$$C_{12} = R + T$$

$$C_{21} = Q + S$$

$$C_{22} = P + R - Q + U.$$

This method is used recursively to perform the seven (n/2) x (n/2) matrix multiplications, then the recurrence equation for the number of scalar multiplications performed is:

$$T(1) = 1$$
$$T(n) = 7\,T(n/2)$$

Solving this for the case of $n = 2^k$ is easy:

$$T(2^k) = 7\,T(2^{k-1})$$
$$= 7^2\,T(2^{k-2})$$
$$= \text{- - - - - -}$$
$$= \text{- - - - - -}$$
$$= 7^i\,T(2^{k-i})$$

Put i = k

$$= 7^k\,T(1)$$
$$= 7^k$$

That is, $T(n) = 7^{\log_2 n}$

$$= n^{\log_2 7}$$
$$= O(n^{\log_2 7}) = O(n^{2.81})$$

So, concluding that Strassen's algorithm is asymptotically more efficient than the standard algorithm. In practice, the overhead of managing the many small matrices does not pay off until 'n' revolves the hundreds.

The main reason for the slowness of Algorithms like SIS is that all comparisons and exchanges between keys in a sequence $w_1, w_2, \ldots, w_n$ take place between adjacent pairs. In this way it takes a relatively long time for a key that is badly out of place to work its way into its proper position in the sorted sequence.

C.A.R. Hoare his devised a very efficient way of implementing this idea in the early 1960's that improves the $O(n^2)$ behavior of SIS algorithm with an expected performance that is O(n log n).

In essence, the quick sort algorithm partitions the original array by rearranging it into two groups. The first group contains those elements less than some arbitrary chosen value taken from the set, and the second group contains those elements greater than or equal to the chosen value.

The chosen value is known as the *pivot element*. Once the array has been rearranged in this way with respect to the pivot, the very same partitioning is recursively applied to each of the two subsets. When all the subsets have been partitioned and rearranged, the original array is sorted.

The function partition() makes use of two pointers 'i' and 'j' which are moved toward each other in the following fashion:

- Repeatedly increase the pointer 'i' until a[i] >= pivot.

- Repeatedly decrease the pointer ‘j’ until a[j] <= pivot.
- If j > i, interchange a[j] with a[i]

- Repeat the steps 1, 2 and 3 till the ‘i’ pointer crosses the ‘j’ pointer. If ‘i’ pointer crosses ‘j’ pointer, the position for pivot is found and place pivot element in ‘j’ pointer position.

The program uses a recursive function quicksort(). The algorithm of quick sort function sorts all elements in an array ‘a’ between positions ‘low’ and ‘high’.

- It terminates when the condition low >= high is satisfied. This condition will be satisfied only when the array is completely sorted.

- Here we choose the first element as the ‘pivot’. So, pivot = x[low]. Now it calls the partition function to find the proper position j of the element x[low] i.e. pivot. Then we will have two sub-arrays x[low], x[low+1], ... x[j-1] and x[j+1], x[j+2], . . .x[high].

- It calls itself recursively to sort the left sub-array x[low], x[low+1], ... x[j-1] between positions low and j-1 (where j is returned by the partition function).

- It calls itself recursively to sort the right sub-array x[j+1], x[j+2], ... x[high] between positions j+1 and high.

3.7.1. Algorithm

```
Algorithm QUICKSORT(low, high)
/* sorts the elements a(low), . . . . . , a(high) which reside in the global array A(1 : n)
into ascending order a (n + 1) is considered to be defined  and must be greater than all
elements in a(1 : n); A(n + 1) = + ∝ */
{
    if low < high then
    {
        j := PARTITION(a, low, high+1);
                                  // J is the position of the partitioning element
        QUICKSORT(low, j – 1);
        QUICKSORT(j + 1 , high);
    }
}
```

```
Algorithm PARTITION(a, m, p)
{
    V ← a(m); i ← m; j ← p;                   // A (m) is the partition element
    do
    {
        loop  i  := i + 1    until a(i) ≥ v          // i moves left to right
        loop  j  := j – 1    until a(j) ≤ v          // p moves right to left
        if (i < j) then INTERCHANGE(a, i, j)
    }  while (i ≥ j);
    a[m] :=a[j]; a[j] :=V;    // the partition element belongs at position P
    return j;
}
```

```
Algorithm INTERCHANGE(a, i, j)
{
        P:=a[i];
        a[i] := a[j];
        a[j] := p;
}
```

3.7.2. Example

Select first element as the pivot element. Move 'i' pointer from left to right in search of an element larger than pivot. Move the 'j' pointer from right to left in search of an element smaller than pivot. If such elements are found, the elements are swapped. This process continues till the 'i' pointer crosses the 'j' pointer. If 'i' pointer crosses 'j' pointer, the position for pivot is found and interchange pivot and element at 'j' position.

Let us consider the following example with 13 elements to analyze quick sort:

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*Remarks*
38	08	16	06	79	57	24	56	02	58	04	70	45	
pivot				i						j			swap i & j
				04						79			
					i			j					swap i & j
					02			57					
						j	i						
(24	08	16	06	04	02)	**38**	(56	57	58	79	70	45)	swap pivot & j
pivot					j, i								swap pivot & j
(02	08	16	06	04)	**24**								
pivot, j	i												swap pivot & j
02	(08	16	06	04)									
	pivot	i		j									swap i & j
		04		16									
			j	i									
	(06	04)	**08**	(16)									swap pivot & j
	pivot, j	i											
	(04)	**06**											swap pivot & j
	04 pivot, j, i												
				16 pivot, j, i									
(02	**04**	**06**	**08**	**16**	**24)**	38							

							(56	57	58	79	70	45)	
							pivot	i				j	swap i & j
								45				57	
								j	i				
							(45)	**56**	(58	79	70	57)	swap pivot & j
							45 pivot, j, i						swap pivot & j
									(58 pivot	79 i	70	57) j	swap i & j
										57		79	
										j	i		
									(57)	**58**	(70	79)	swap pivot & j
									57 pivot, j, i				
											(70	79)	
											pivot, j	i	swap pivot & j
											70		
												79 pivot, j, i	
							(45	**56**	**57**	**58**	**70**	**79)**	
02	**04**	**06**	**08**	**16**	**24**	**38**	**45**	**56**	**57**	**58**	**70**	**79**	

3.7.3. *Analysis of Quick Sort:*

Like merge sort, quick sort is recursive, and hence its analysis requires solving a recurrence formula. We will do the analysis for a quick sort, assuming a random pivot (and no cut off for small files).

We will take T (0) = T (1) = 1, as in merge sort.

The running time of quick sort is equal to the running time of the two recursive calls plus the linear time spent in the partition (The pivot selection takes only constant time). This gives the basic quick sort relation:

$$T(n) = T(i) + T(n - i - 1) + C\,n \qquad - \qquad (1)$$

Where, i = $|S_1|$ is the number of elements in S_1.

3.7.3.1. Worst Case Analysis

The pivot is the smallest element, all the time. Then i=0 and if we ignore T(0)=1, which is insignificant, the recurrence is:

$$T(n) = T(n - 1) + C\,n \qquad n > 1 \qquad - \qquad (2)$$

Using equation – (1) repeatedly, thus

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

$$T(n-2) = T(n-3) + C(n-2)$$

- - - - - - - -

$$T(2) = T(1) + C(2)$$

Adding up all these equations yields

$$T(n) = T(1) + \sum_{i=2}^{n} i$$

$$= \mathbf{O(n^2)} \qquad - \qquad (3)$$

3.7.3.2. Best Case Analysis

In the best case, the pivot is in the middle. To simply the math, we assume that the two sub-files are each exactly half the size of the original and although this gives a slight over estimate, this is acceptable because we are only interested in a Big – oh answer.

$$T(n) = 2T(n/2) + Cn \qquad - \qquad (4)$$

Divide both sides by n

$$\frac{T(n)}{n} = \frac{T(n/2)}{n/2} + C \qquad - \qquad (5)$$

Substitute n/2 for 'n' in equation (5)

$$\frac{T(n/2)}{n/2} = \frac{T(n/4)}{n/4} + C \qquad - \qquad (6)$$

Substitute n/4 for 'n' in equation (6)

$$\frac{T(n/4)}{n/4} = \frac{T(n/8)}{n/8} + C \qquad - \qquad (7)$$

- - - - - - - -

- - - - - - - -

Continuing in this manner, we obtain:

$$\frac{T(2)}{2} = \frac{T(1)}{1} + C \qquad - \qquad (8)$$

We add all the equations from 4 to 8 and note that there are log n of them:

$$\frac{T(n)}{n} = \frac{T(1)}{1} + C \log n \qquad - \qquad (9)$$

Which yields, T (n) = C n log n + n = **O(n log n)** - (10)

This is exactly the same analysis as merge sort, hence we get the same answer.

3.7.3.3. Average Case Analysis

The number of comparisons for first call on partition: Assume left_to_right moves over k smaller element and thus k comparisons. So when right_to_left crosses left_to_right it has made n-k+1 comparisons. So, first call on partition makes n+1 comparisons. The average case complexity of quicksort is

$T(n)$ = *comparisons for first call on quicksort*

$+$

$\{\Sigma\ 1 <= nleft, nright <= n\ [T(nleft) + T(nright)]\}n = (n+1) + 2\ [T(0) + T(1) + T(2) + \text{-----} + T(n-1)]/n$

$nT(n) = n(n+1) + 2\ [T(0) + T(1) + T(2) + \text{-----} + T(n-2) + T(n-1)]$

$(n-1)T(n-1) = (n-1)n + 2\ [T(0) + T(1) + T(2) + \text{-----} + T(n-2)]$ \

Subtracting both sides:

$nT(n) - (n-1)T(n-1) = [\ n(n+1) - (n-1)n] + 2T(n-1) = 2n + 2T(n-1)$

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

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

The recurrence relation obtained is:

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

Using the method of subsititution:

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

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

$T(n-2)/(n-1) \quad = \quad 2/(n-1) + T(n-3)/(n-2)$

$T(n-3)/(n-2) \quad = \quad 2/(n-2) + T(n-4)/(n-3)$

. .

. .

T(3)/4 = 2/4 + T(2)/3

T(2)/3 = 2/3 + T(1)/2 T(1)/2 = 2/2 + T(0)

Adding both sides:

T(n)/(n+1) + [T(n-1)/n + T(n-2)/(n-1) + ------------- + T(2)/3 + T(1)/2]

= [T(n-1)/n + T(n-2)/(n-1) + ------------- + T(2)/3 + T(1)/2] + T(0) +

[2/(n+1) + 2/n + 2/(n-1) + ---------- +2/4 + 2/3]

Cancelling the common terms:

T(n)/(n+1) = 2[1/2 +1/3 +1/4+--------------+1/n+1/(n+1)]

$$T(n) = (n+1)2[\sum_{2 \le k \le n+1} 1/k$$

$$=2(n+1) [\int_{2}^{n+1} \frac{1}{x} dx]$$

$$=2(n+1)[\log (n+1) - \log 2]$$

$$=2n \log (n+1) + \log (n+1)-2n \log 2 - \log 2$$

T(n)= O(n log n)

3.8. *Straight insertion sort:*

Straight insertion sort is used to create a sorted list (initially list is empty) and at each iteration the top number on the sorted list is removed and put into its proper place in the sorted list. This is done by moving along the sorted list, from the smallest to the largest number, until the correct place for the new number is located i.e. until all sorted numbers with smaller values comes before it and all those with larger values comes after it. For example, let us consider the following 8 elements for sorting:

Index	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
Elements	27	412	71	81	59	14	273	87

Solution:

Iteration 0:	unsorted	412	71	81	59	14	273	87
	Sorted	27						
Iteration 1:	unsorted	412	71	81	59	14	273	87
	Sorted	27	412					
Iteration 2:	unsorted	71	81	59	14	273	87	
	Sorted	27	71	412				
Iteration 3:	unsorted	81	39	14	273	87		
	Sorted	27	71	81	412			
Iteration 4:	unsorted	59	14	273	87			

	Sorted	274	59	71	81	412			
Iteration 5:	unsorted	14	273	87					
	Sorted	14	27	59	71	81	412		
Iteration 6:	unsorted	273	87						
	Sorted	14	27	59	71	81	273	412	
Iteration 7:	unsorted	87							
	Sorted	14	27	59	71	81	87	273	412

Chapter 4

Greedy Method

4.1. GENERAL METHOD

Greedy is the most straight forward design technique. Most of the problems have n inputs and require us to obtain a subset that satisfies some constraints. Any subset that satisfies these constraints is called a feasible solution. We need to find a feasible solution that either maximizes or minimizes the objective function. A feasible solution that does this is called an optimal solution.

The greedy method is a simple strategy of progressively building up a solution, one element at a time, by choosing the best possible element at each stage. At each stage, a decision is made regarding whether or not a particular input is in an optimal solution. This is done by considering the inputs in an order determined by some selection procedure. If the inclusion of the next input, into the partially constructed optimal solution will result in an infeasible solution then this input is not added to the partial solution. The selection procedure itself is based on some optimization measure. Several optimization measures are plausible for a given problem. Most of them, however, will result in algorithms that generate sub-optimal solutions. This version of greedy technique is called *subset paradigm*. Some problems like Knapsack, Job sequencing with deadlines and minimum cost spanning trees are based on *subset paradigm.*

For the problems that make decisions by considering the inputs in some order, each decision is made using an optimization criterion that can be computed using decisions already made. This version of greedy method is *ordering paradigm*. Some problems like optimal storage on tapes, optimal merge patterns and single source shortest path are based on *ordering paradigm.*

4.2. CONTROL ABSTRACTION

```
Algorithm Greedy (a, n)
// a(1 : n) contains the 'n' inputs
{
solution := φ;                // initialize the solution to empty
for i:=1 to n do
{
x := select (a);
if  feasible (solution, x) then
solution := Union (Solution, x);
}
return solution;
}
```

Procedure Greedy describes the essential way that a greedy based algorithm will look, once a particular problem is chosen and the functions select, feasible and union are properly implemented.

The function select selects an input from ‘a’, removes it and assigns its value to ‘x’. Feasible is a Boolean valued function, which determines if ‘x’ can be included into the solution vector. The function Union combines ‘x’ with solution and updates the objective function.

4.3. KNAPSACK PROBLEM

Let us apply the greedy method to solve the knapsack problem. We are given ‘n’ objects and a knapsack. The object ‘i’ has a weight w_i and the knapsack has a capacity ‘m’. If a fraction x_i, $0 < x_i < 1$ of object i is placed into the knapsack then a profit of $p_i\ x_i$ is earned. The objective is to fill the knapsack that maximizes the total profit earned.

Since the knapsack capacity is ‘m’, we require the total weight of all chosen objects to be at most ‘m’. The problem is stated as:

$$\text{maximize} \sum_{i=1}^{n} p_i\ x_i$$

$$\text{subject to} \sum_{i=1}^{n} a_i\ x_i \leq M \qquad \text{where, } 0 \leq x_i \leq 1 \text{ and } 1 \leq i \leq n$$

The profits and weights are positive numbers.

4.3.1. Algorithm

If the objects are already been sorted into non-increasing order of p[i] / w[i] then the algorithm given below obtains solutions corresponding to this strategy.

Algorithm GreedyKnapsack (m, n)

```
// P[1 : n] and w[1 : n] contain the profits and weights respectively of
// Objects ordered so that p[i] / w[i] > p[i + 1] / w[i + 1].
// m is the knapsack size and x[1: n] is the solution vector.
{
for i := 1 to n do x[i] := 0.0                    // initialize x
        U := m;
for i := 1 to n do
{
if  (w(i) > U) then break;
        x [i] := 1.0; U := U – w[i];
        }
if (i ≤ n) then x[i] := U / w[i];
}
```

Running time:

The objects are to be sorted into non-decreasing order of p_i / w_i ratio. But if we disregard the time to initially sort the objects, the algorithm requires only O(n) time.

4.3.2. Example:

Consider the following instance of the knapsack problem: n = 3, m = 20, (p_1, p_2, p_3) = (25, 24, 15) and (w_1, w_2, w_3) = (18, 15, 10).

1. First, we try to fill the knapsack by selecting the objects in some order:

x_1	x_2	x_3	$\sum w_i x_i$	$\sum p_i x_i$
1/2	1/3	1/4	18 x 1/2 + 15 x 1/3 + 10 x 1/4 = 16.5	25 x 1/2 + 24 x 1/3 + 15 x 1/4 = 24.25

2. Select the object with the maximum profit first (p = 25). So, x_1 = 1 and profit earned is 25. Now, only 2 units of space is left, select the object with next largest profit (p = 24). So, x_2 = 2/15

x_1	x_2	x_3	$\sum w_i x_i$	$\sum p_i x_i$
1	2/15	0	18 x 1 + 15 x 2/15 = 20	25 x 1 + 24 x 2/15 = 28.2

3. Considering the objects in the order of non-decreasing weights w_i.

x_1	x_2	x_3	$\sum w_i x_i$	$\sum p_i x_i$
0	2/3	1	15 x 2/3 + 10 x 1 = 20	24 x 2/3 + 15 x 1 = 31

4. Considered the objects in the order of the ratio p_i / w_i .

p_1/w_1	**p_2/w_2**	**p_3/w_3**
25/18	24/15	15/10
1.4	1.6	1.5

Sort the objects in order of the non-increasing order of the ratio p_i / x_i. Select the object with the maximum p_i / x_i ratio, so, x_2 = 1 and profit earned is 24. Now, only 5 units of space is left, select the object with next largest p_i / x_i ratio, so x_3 = ½ and the profit earned is 7.5.

x_1	x_2	x_3	$\sum w_i x_i$	$\sum p_i x_i$
0	1	1/2	15 x 1 + 10 x 1/2 = 20	24 x 1 + 15 x 1/2 = 31.5

This solution is the optimal solution.

4.4. OPTIMAL STORAGE ON TAPES

There are 'n' programs that are to be stored on a computer tape of length 'L'. Each program 'i' is of length l_i, $1 \leq i \leq n$. All the programs can be stored on the tape if and only if the sum of the lengths of the programs is at most 'L'.

We shall assume that whenever a program is to be retrieved from this tape, the tape is initially positioned at the front. If the programs are stored in the order $i = i_1, i_2, \ldots\ldots, i_n$, the time t_J needed to retrieve program i_J is proportional to

$$\sum_{1 \leq k \leq j} l_{i_k}$$

If all the programs are retrieved equally often then the expected or mean retrieval time (MRT) is:

$$\frac{1}{n} \cdot \sum_{1 \leq J \leq n} t_j$$

For the optimal storage on tape problem, we are required to find the permutation for the 'n' programs so that when they are stored on the tape in this order the MRT is minimized.

$$d(I) = \sum_{J=1}^{n} \sum_{K=1}^{J} l_{i_k}$$

4.4.1. Example

Let n = 3, $(l_1, l_2, l_3) = (5, 10, 3)$. Then find the optimal ordering?

Solution:

There are n! = 6 possible orderings. They are:

Ordering I	d(I)		
1, 2, 3	5 + (5 +10) +(5 + 10 + 3)	=	38
1, 3, 2	5 + (5 + 3) + (5 + 3 + 10)	=	31
2, 1, 3	10 + (10 + 5) + (10 + 5 + 3)	=	43
2, 3, 1	10 + (10 + 3) + (10 + 3 + 5)	=	41
3, 1, 2	3 + (3 + 5) + (3 + 5 + 10)	=	29
3, 2, 1	3 + (3 + 10) + (3 + 10 + 5)	=	34

From the above, it simply requires to store the programs in non-decreasing order (increasing order) of their lengths. This can be carried out by using a efficient sorting algorithm (Heap sort). This ordering can be carried out in O (n log n) time using heap sort algorithm.

The tape storage problem can be extended to several tapes. If there are m > 1 tapes, T_o, ... ,T_{m-1}, then the programs are to be distributed over these tapes.

The total retrieval time (RT) is $\sum_{J=0}^{m-1} d(I_J)$

The objective is to store the programs in such a way as to minimize RT.

The programs are to be sorted in non decreasing order of their lengths l_i's, $l_1 \leq l_2 \leq$ .. . . $l_{n.}$

The first 'm' programs will be assigned to tapes T_o, ... ,T_{m-1} respectively. The next 'm' programs will be assigned to T_0, ... ,T_{m-1} respectively. The general rule is that program i is stored on tape T_i mod m.

4.4.2. Algorithm:

The algorithm for assigning programs to tapes is as follows:

Algorithm Store (n, m)

```
// n is the number of programs and m the number of tapes
{
j := 0;                         // next tape to store on
for i :=1 to n do
     {
          Print ('append program', i, 'to permutation for tape', j);
j := (j + 1) mod m;
     }
}
```

On any given tape, the programs are stored in non-decreasing order of their lengths.

4.5. JOB SEQUENCING WITH DEADLINES

When we are given a set of 'n' jobs. Associated with each Job i, deadline $d_i \geq 0$ and profit $P_i \geq 0$. For any job 'i' the profit pi is earned iff the job is completed by its deadline. Only one machine is available for processing jobs. An optimal solution is the feasible solution with maximum profit.

Sort the jobs in 'j' ordered by their deadlines. The array d [1 : n] is used to store the deadlines of the order of their p-values. The set of jobs j [1 : k] such that j [r], 1 ≤ r ≤ k are the jobs in 'j' and d (j [1]) ≤ d (j[2]) ≤ . . . ≤ d (j[k]). To test whether J U {i} is feasible, we have just to insert i into J preserving the deadline ordering and then verify that d [J[r]] ≤ r, 1 ≤ r ≤ k+1.

4.5.1. Example:

Let n = 4, (P_1, P_2, P_3, $P_{4,}$) = (100, 10, 15, 27) and (d_1 d_2 d_3 d_4) = (2, 1, 2, 1). The feasible solutions and their values are:

S. No	Feasible Solution	Procuring sequence	Value	Remarks
1	1,2	2,1	110	
2	1,3	1,3 or 3,1	115	

3	1,4	4,1	127	**OPTIMAL**
4	2,3	2,3	25	
5	3,4	4,3	42	
6	1	1	100	
7	2	2	10	
8	3	3	15	
9	4	4	27	

4.5.2. Algorithm:

The algorithm constructs an optimal set J of jobs that can be processed by their deadlines**.**

Algorithm GreedyJob (d, J, n)

```
// J is a set of jobs that can be completed by their deadlines.
{
        J := {1};
        for i := 2 to n do
        {
                if (all jobs in J U {i} can be completed by their dead lines)
                then J := J U {i};
        }
}
```

4.6. OPTIMAL MERGE PATERNS

Given ‘n’ sorted files, there are many ways to pair wise merge them into a single sorted file. As, different pairings require different amounts of computing time, we want to determine an optimal (i.e., one requiring the fewest comparisons) way to pair wise merge ‘n’ sorted files together. This type of merging is called as 2-way merge patterns. To merge an n-record file and an m-record file requires possibly n + m record moves, the obvious choice choice is, at each step merge the two smallest files together. The two-way merge patterns can be represented by binary merge trees.

4.6.1. Algorithm to Generate Two-way Merge Tree:

```
struct treenode
{
treenode * lchild;
treenode * rchild;
};
```

Algorithm TREE (n)

```
// list is a global of n single node binary trees
{
for i := 1  to n – 1 do
{
pt  ← new treenode
(pt → lchild) ← least (list);            // merge two trees with smallest lengths
(pt → rchild) ← least (list);
(pt → weight) ← ((pt → lchild) → weight) + ((pt → rchild) → weight);
```

```
 insert (list, pt);
          }
return least (list);                          // The tree left in list is the merge tree
}
```

4.6.2. Example 1:

Suppose we are having three sorted files X_1, X_2 and X_3 of length 30, 20, and 10 records each. Merging of the files can be carried out as follows:

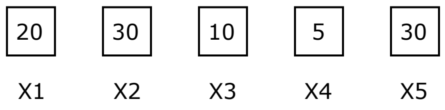
S.No	First Merging	Record moves in first merging	Second merging	Record moves in second merging	Total no. of records moves
1.	X_1 & X_2 = T1	50	T_1 & X_3	60	50 + 60 = 110
2.	X_2 & X_3 = T1	30	T_1 & X_1	60	30 + 60 = 90

The Second case is optimal.

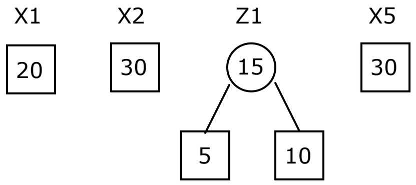
4.6.3. Example 2:

Given five files (X1, X2, X3, X4, X5) with sizes (20, 30, 10, 5, 30). Apply greedy rule to find optimal way of pair wise merging to give an optimal solution using binary merge tree representation.

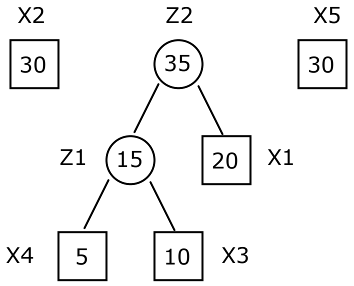
Solution:



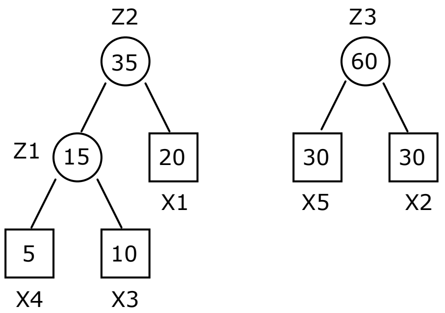
Merge X_4 and X_3 to get 15 record moves. Call this Z_1.



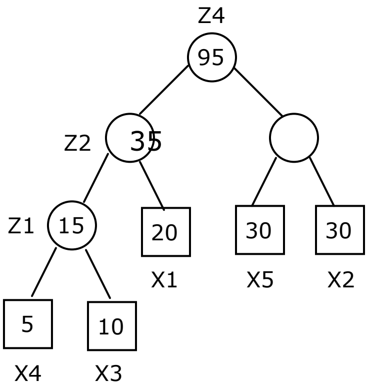
Merge Z_1 and X_1 to get 35 record moves. Call this Z_2.



Merge X_2 and X_5 to get 60 record moves. Call this Z_3.



Merge Z_2 and Z_3 to get 90 record moves. This is the answer. Call this Z_4.



Therefore the total number of record moves is 15 + 35 + 60 + 95 = 205. This is an optimal merge pattern for the given problem.

4.7. Huffman Codes

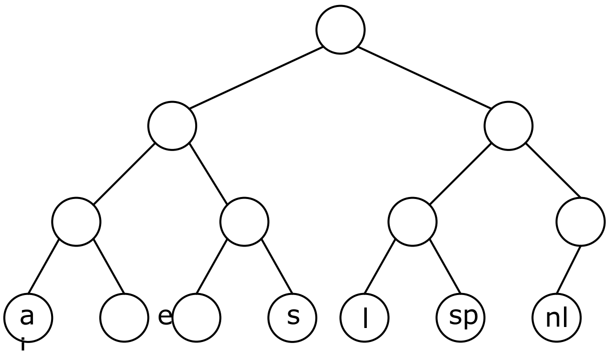
Another application of Greedy Algorithm is file compression.

Suppose that we have a file only with characters a, e, i, s, t, spaces and new lines, the frequency of appearance of a's is 10, e's fifteen, twelve i's, three s's, four t's, thirteen banks and one newline.

Using a standard coding scheme, for 58 characters using 3 bits for each character, the file requires 174 bits to represent. This is shown in table below.

Character	Code	Frequency	Total bits
A	000	10	30
E	001	15	45
I	010	12	36
S	011	3	9
T	100	4	12
Space	101	13	39
New line	110	1	3

Representing by a binary tree, the binary code for the alphabets are as follows:

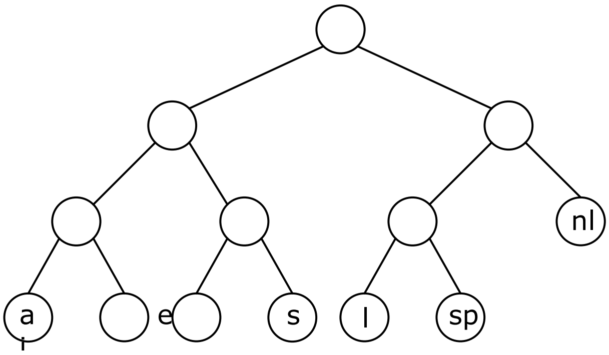


The representation of each character can be found by starting at the root and recording the path. Use a 0 to indicate the left branch and a 1 to indicate the right branch.

If the character c_i is at depth d_i and occurs f_i times, the cost of the code is equal to $\sum d_i \; f_i$

With this representation the total number of bits is 3x10 + 3x15 + 3x12 + 3x3 + 3x4 + 3x13 + 3x1 = 174

A better code can be obtained by with the following representation.



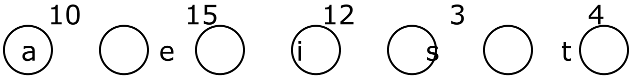
The basic problem is to find the full binary tree of minimal total cost. This can be done by using Huffman coding (1952).

4.7.1. Huffman's Algorithm:

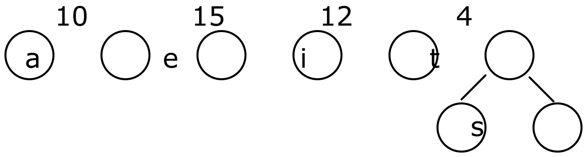
Huffman's algorithm can be described as follows: We maintain a forest of trees. The weights of a tree is equal to the sum of the frequencies of its leaves. If the number of characters is 'c'. c - 1 times, select the two trees T1 and T2, of smallest weight, and form a new tree with sub-trees T1 and T2. Repeating the process we will get an optimal Huffman coding tree.

4.7.2. Example:

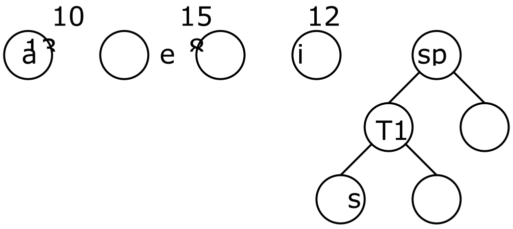
The initial forest with the weight of each tree is as follows:



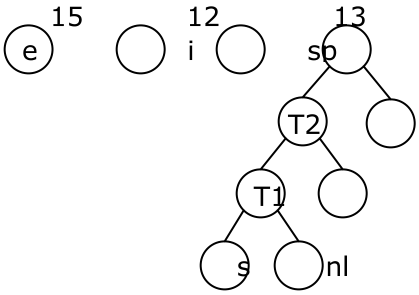
The two trees with the lowest weight are merged together, creating the forest, the Huffman algorithm after the first merge with new root T_1 is as follows: The total weight of the new tree is the sum of the weights of the old trees.



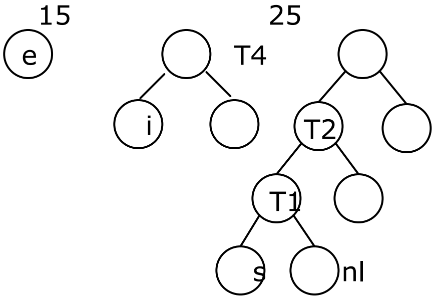
We again select the two trees of smallest weight. This happens to be T_1 and t, which are merged into a new tree with root T_2 and weight 8.



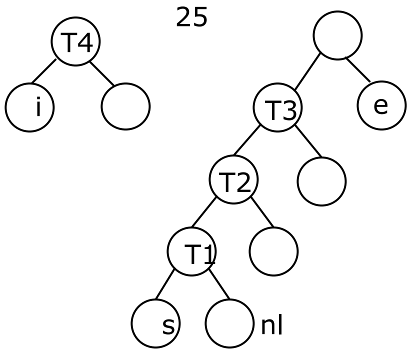
In next step we merge T_2 and a creating T_3, with weight 10+8=18. The result of this operation in



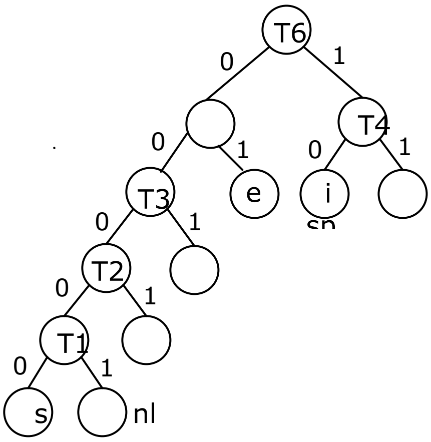
After third merge, the two trees of lowest weight are the single node trees representing i and the blank space. These trees merged into the new tree with root T_4.



The fifth step is to merge the trees with roots e and T_3. The results of this step is



Finally, the optimal tree is obtained by merging the two remaining trees. The optimal trees with root T_6 is:



The full binary tree of minimal total cost, where all characters are obtained in the leaves, uses only 146 bits.

Character	Code	Frequency	Total bits (Code bits X frequency)
A	001	10	30
E	01	15	30
I	10	12	24
S	00000	3	15
T	0001	4	16
Space	11	13	26
New line	00001	1	5
		Total :	146

4.8. GRAPH ALGORITHMS

4.8.1. Basic Definitions:

- **Graph G** is a pair (V, E), where V is a finite set (set of vertices) and E is a finite set of pairs from V (set of edges). We will often denote n := |V|, m := |E|.

- Graph G can be **directed**, if E consists of ordered pairs, or undirected, if E consists of unordered pairs. If (u, v) $\in$ E, then vertices u, and v are adjacent.

- We can assign weight function to the edges: $w_G(e)$ is a weight of edge e $\in$ E. The graph which has such function assigned is called **weighted**.

- **Degree** of a vertex v is the number of vertices u for which (u, v) $\in$ E (denote deg(v)). The number of **incoming edges** to a vertex v is called **in–degree** of the vertex (denote indeg(v)). The number of **outgoing edges** from a vertex is called **out-degree** (denote outdeg(v)).

4.8.2. Representation of Graphs:

Consider graph G = (V, E), where V= {v_1, v_2,….,v_n}.

Adjacency matrix represents the graph as an n x n matrix A = ($a_{i,j}$), where

$$a_{i,\,j} = \begin{cases} 1, & \textit{if } (v_i,\, v_j) \in E, \\ 0, & \textit{otherwise} \end{cases}$$

The matrix is symmetric in case of undirected graph, while it may be asymmetric if the graph is directed.

We may consider various modifications. For example for weighted graphs, we may have

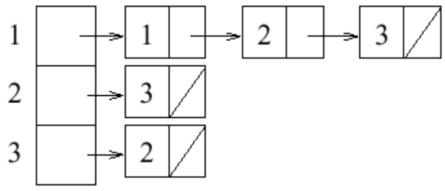
$$a_{i,\,j} = \begin{cases} w(v_i,\, v_j), & \textit{if } (v_i,\, v_j) \in E, \\ \textit{default}, & \textit{otherwise}, \end{cases}$$

Where default is some sensible value based on the meaning of the weight function (for example, if weight function represents length, then default can be $\propto$, meaning value larger than any other value).

Adjacency List: An array Adj [1 . . . . n] of pointers where for 1 $\leq$ v $\leq$ n, Adj [v] points to a linked list containing the vertices which are adjacent to v (i.e. the vertices that can be reached from v by a single edge). If the edges have weights then these weights may also be stored in the linked list elements.

	1	2	3
1	1	1	1
2	0	0	1
3	0	1	0

Adjacency matrix



Adjacency list

4.8.3. Paths and Cycles:

A path is a sequence of vertices (v_1, v_2, ... , v_k), where for all i, (v_i, v_{i+1}) ε E. **A path is simple** if all vertices in the path are distinct.

A (simple) cycle is a sequence of vertices (v_1, v_2, ... , v_k, v_{k+1} = v_1), where for all i, (v_i, v_{i+1}) ε E and all vertices in the cycle are distinct except pair v_1, v_{k+1}.

4.8.4. Subgraphs and Spanning Trees:

Subgraphs: A graph G′ = (V′, E′) is a subgraph of graph G = (V, E) iff V′ $\subseteq$ V and E′ $\subseteq$ E.

The undirected graph G is connected, if for every pair of vertices u, v there exists a path from u to v. If a graph is not connected, the vertices of the graph can be divided into **connected components**. Two vertices are in the same connected component iff they are connected by a path.

Tree is a connected acyclic graph. A **spanning tree** of a graph G = (V, E) is a tree that contains all vertices of V and is a subgraph of G. A single graph can have multiple spanning trees.

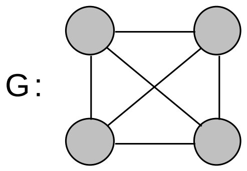
Lemma 1: *Let T be a spanning tree of a graph G. Then*

1. *Any two vertices in T are connected by a unique simple path.*
2. *If any edge is removed from T, then T becomes disconnected.*
3. *If we add any edge into T, then the new graph will contain a cycle.*
4. *Number of edges in T is n-1.*

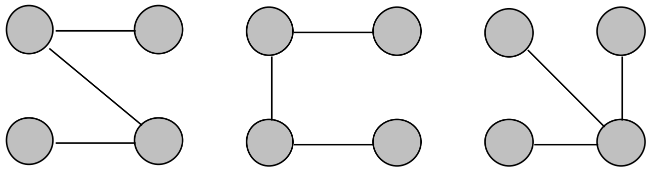
4.8.5. Minimum Spanning Trees (MST):

A spanning tree for a connected graph is a tree whose vertex set is the same as the vertex set of the given graph, and whose edge set is a subset of the edge set of the given graph. i.e., any connected graph will have a spanning tree.

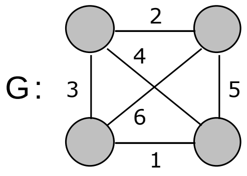
Weight of a spanning tree w (T) is the sum of weights of all edges in T. The Minimum spanning tree (MST) is a spanning tree with the smallest possible weight.



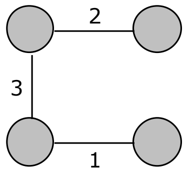
A graph G:



Three (of many possible) spanning trees from graph G:



A weighted graph G:



The minimal spanning tree from weighted graph G:

Here are some examples:

To explain further upon the Minimum Spanning Tree, and what it applies to, let's consider a couple of real-world examples:

1. One practical application of a MST would be in the design of a network. For instance, a group of individuals, who are separated by varying distances, wish to be connected together in a telephone network. Although MST cannot do anything about the distance from one connection to another, it can be used to determine the least cost paths with no cycles in this network, thereby connecting everyone at a minimum cost.
2. Another useful application of MST would be finding airline routes. The vertices of the graph would represent cities, and the edges would represent routes between the cities. Obviously, the further one has to travel, the more it will cost, so MST can be applied to optimize airline routes by finding the least costly paths with no cycles.

To explain how to find a Minimum Spanning Tree, we will look at two algorithms: the Kruskal algorithm and the Prim algorithm. Both algorithms differ in their methodology, but both eventually end up with the MST. Kruskal's algorithm uses edges, and Prim’s algorithm uses vertex connections in determining the MST.

4.8.5. Kruskal’s Algorithm

This is a greedy algorithm. A greedy algorithm chooses some local optimum (i.e. picking an edge with the least weight in a MST).

Kruskal's algorithm works as follows: Take a graph with 'n' vertices, keep on adding the shortest (least cost) edge, while avoiding the creation of cycles, until (n - 1) edges have been added. Sometimes two or more edges may have the same cost. The order in which the edges are chosen, in this case, does not matter. Different MSTs may result, but they will all have the same total cost, which will always be the minimum cost.

4.8.5.1. Algorithm:

The algorithm for finding the MST, using the Kruskal's method is as follows:

```
Algorithm Kruskal (E, cost, n, t)
// E is the set of edges in G. G has n vertices. cost [u, v] is the
// cost of edge (u, v). 't' is the set of edges in the minimum-cost spanning tree.
// The final cost is returned.
{
        Construct a heap out of the edge costs using heapify;
        for  i := 1 to n do parent [i] := -1;
// Each vertex is in a different set.
        i := 0; mincost := 0.0;
        while ((i < n -1) and (heap not empty)) do
        {
                Delete a minimum cost edge (u, v) from the heap and
re-heapify using Adjust;
                j := Find (u); k := Find (v);
                if  (j ≠ k) then
{
                i := i + 1;
                t [i, 1] := u; t [i, 2] := v;
                mincost :=mincost + cost [u, v];
                Union (j, k);
}
}
if  (i ≠ n-1) then write ("no spanning tree");
else return mincost;
}
```

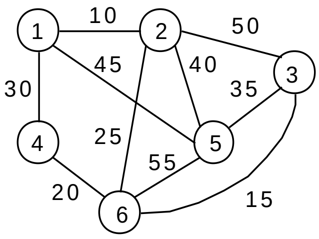
Running time:

- The number of finds is at most 2e, and the number of unions at most n-1. Including the initialization time for the trees, this part of the algorithm has a complexity that is just slightly more than O (n + e).

- We can add at most n-1 edges to tree T. So, the total time for operations on T is O(n).

Summing up the various components of the computing times, we get O (n + e log e) as asymptotic complexity

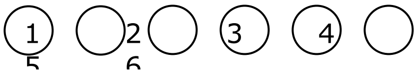
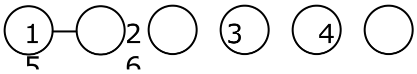
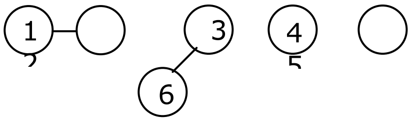
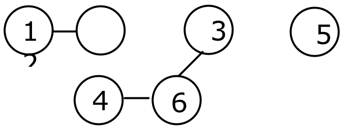
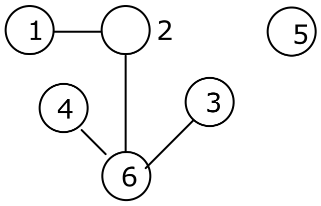
4.8.5.2. Example 1:

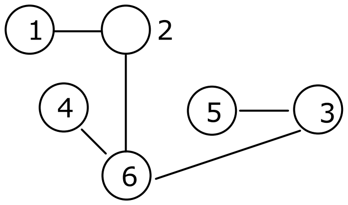


Arrange all the edges in the increasing order of their costs:

Cost	10	15	20	25	30	35	40	45	50	55
Edge	(1, 2)	(3, 6)	(4, 6)	(2, 6)	(1, 4)	(3, 5)	(2, 5)	(1, 5)	(2, 3)	(5, 6)

The edge set T together with the vertices of G define a graph that has up to n connected components. Let us represent each component by a set of vertices in it. These vertex sets are disjoint. To determine whether the edge (u, v) creates a cycle, we need to check whether u and v are in the same vertex set. If so, then a cycle is created. If not then no cycle is created. Hence two **Find**s on the vertex sets suffice. When an edge is included in T, two components are combined into one and a **union** is to be performed on the two sets.

Edge	Cost	Spanning Forest	Edge Sets	Remarks
			{1}, {2}, {3}, {4}, {5}, {6}	
(1, 2)	10		{1, 2}, {3}, {4}, {5}, {6}	The vertices 1 and 2 are in different sets, so the edge is combined
(3, 6)	15		{1, 2}, {3, 6}, {4}, {5}	The vertices 3 and 6 are in different sets, so the edge is combined
(4, 6)	20		{1, 2}, {3, 4, 6}, {5}	The vertices 4 and 6 are in different sets, so the edge is combined
(2, 6)	25		{1, 2, 3, 4, 6}, {5}	The vertices 2 and 6 are in different sets, so the edge is combined
(1, 4)	30	Reject		The vertices 1 and 4 are in the same

				set, so the edge is rejected
(3, 5)	35		{1, 2, 3, 4, 5, 6}	The vertices 3 and 5 are in the same set, so the edge is combined

4.8.6. MINIMUM-COST SPANNING TREES: PRIM'S ALGORITHM

A given graph can have many spanning trees. From these many spanning trees, we have to select a cheapest one. This tree is called as minimal cost spanning tree.

Minimal cost spanning tree is a connected undirected graph G in which each edge is labeled with a number (edge labels may signify lengths, weights other than costs). Minimal cost spanning tree is a spanning tree for which the sum of the edge labels is as small as possible

The slight modification of the spanning tree algorithm yields a very simple algorithm for finding an MST. In the spanning tree algorithm, any vertex not in the tree but connected to it by an edge can be added. To find a Minimal cost spanning tree, we must be selective - we must always add a new vertex for which the cost of the new edge is as small as possible.

This simple modified algorithm of spanning tree is called prim's algorithm for finding an Minimal cost spanning tree.

Prim's algorithm is an example of a greedy algorithm.

4.8.6.1. Algorithm

Algorithm Prim (E, cost, n, t)

```
// E is the set of edges in G. cost [1:n, 1:n] is the cost
// adjacency matrix of an n vertex graph such that cost [i, j] is
// either a positive real number or ∝ if no edge (i, j) exists.
// A minimum spanning tree is computed and stored as a set of
// edges in the array t [1:n-1, 1:2]. (t [i, 1], t [i, 2]) is an edge in
// the minimum-cost spanning tree. The final cost is returned.
{
Let (k, l) be an edge of minimum cost in E;
mincost := cost [k, l];
t [1, 1] := k; t [1, 2] := l;
for  i :=1 to n do                          // Initialize near
if  (cost [i, l] < cost [i, k]) then near [i] := l;
else near [i] := k;
near [k] :=near [l] := 0;
for  i:=2 to n - 1 do                       // Find n - 2 additional edges for t.
{
Let j be an index such that near [j] ≠ 0 and
cost [j, near [j]] is minimum;
t [i, 1] := j; t [i, 2] := near [j];
mincost := mincost + cost [j, near [j]];
near [j] := 0
for  k:= 1 to n do                          // Update near[].
if ((near [k] $\neq$ 0) and (cost [k, near [k]] > cost [k, j]))
then near [k] := j;
}
return mincost;
}
```

Running time:

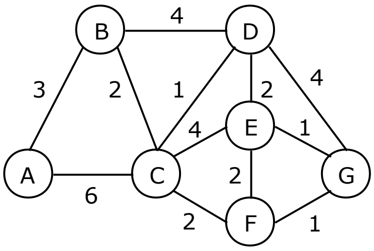
We do the same set of operations with dist as in Dijkstra's algorithm (initialize structure, m times decrease value, n - 1 times select minimum). Therefore, we get O (n^2) time when we implement dist with array, O (n + |E| log n) when we implement it with a heap.

For each vertex u in the graph we dequeue it and check all its neighbors in θ (1 + deg (u)) time. Therefore the running time is:

$$\Theta\left(\sum_{v \in V} 1 + \deg(v)\right) = \Theta\left(n + \sum_{v \in V} \deg(v)\right) = \Theta(n + m)$$

4.8.6.2. EXAMPLE 1:

Use Prim's Algorithm to find a minimal spanning tree for the graph shown below starting with the vertex A.

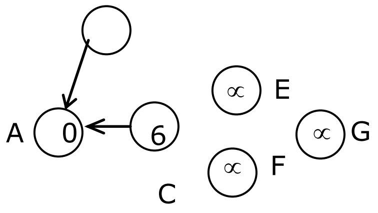


SOLUTION:

The cost adjacency matrix is
$$\begin{pmatrix} 0 & 3 & 6 & \infty & \infty & \infty & \infty \\ 3 & 0 & 2 & 4 & \infty & \infty & \infty \\ 6 & 2 & 0 & 1 & 4 & 2 & \infty \\ \infty & 4 & 1 & 0 & 2 & \infty & 4 \\ \infty & \infty & 4 & 2 & 0 & 2 & 1 \\ \infty & \infty & 2 & \infty & 2 & 0 & 1 \\ \infty & \infty & \infty & 4 & 1 & 1 & 0 \end{pmatrix}$$

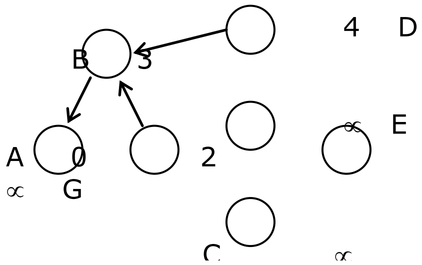
The stepwise progress of the prim's algorithm is as follows:

Step 1:



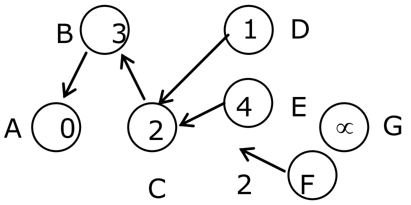
Vertex	A	B	C	D	E	F	G
Status	0	1	1	1	1	1	1
Dist.	0	3	6	∝	∝	∝	∝
Next	*	A	A	A	A	A	A

Step 2:



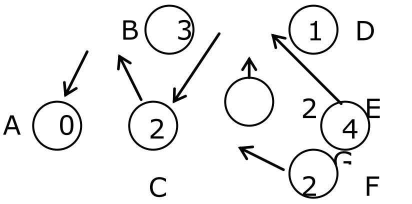
Vertex	A	B	C	D	E	F	G
Status	0	0	1	1	1	1	1
Dist.	0	3	2	4	∝	∝	∝
Next	*	A	B	B	A	A	A

Step 3:



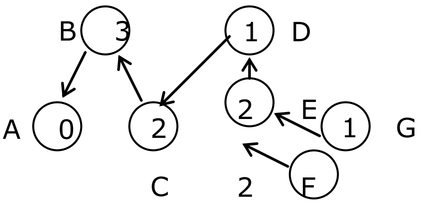
Vertex	A	B	C	D	E	F	G
Status	0	0	0	1	1	1	1
Dist.	0	3	2	1	4	2	∝
Next	*	A	B	C	C	C	A

Step 4:



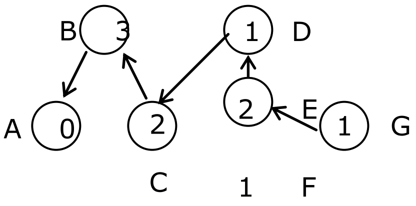
Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	1	1
Dist.	0	3	2	1	2	2	4
Next	*	A	B	C	D	C	D

Step 5:

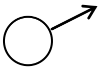


Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	0	1
Dist.	0	3	2	1	2	2	1
Next	*	A	B	C	D	C	E

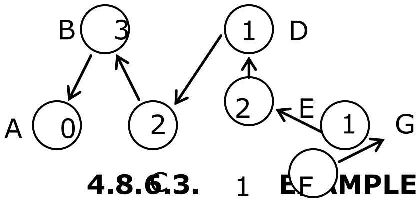
Step 6:



Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	0	1	0
Dist.	0	3	2	1	2	1	1
Next	*	A	B	C	D	G	E



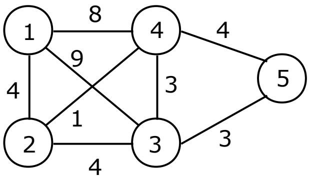
Step 7:



Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	0	0	0
Dist.	0	3	2	1	2	1	1
Next	*	A	B	C	D	G	E

4.8.6.3. EXAMPLE 2:

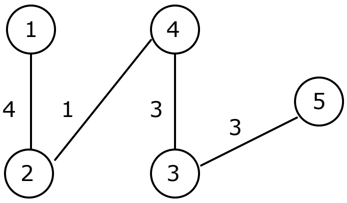
Considering the following graph, find the minimal spanning tree using prim’s algorithm.



The cost adjacent matrix is $\begin{pmatrix} \infty & 4 & 9 & 8 & \infty \\ 4 & \infty & 4 & 1 & \infty \\ 9 & 4 & \infty & 3 & 3 \\ 8 & 1 & 3 & \infty & 4 \\ \infty & \infty & 3 & 4 & \infty \end{pmatrix}$

The minimal spanning tree obtained as:

Vertex 1	Vertex 2
2	4
3	4
5	3
1	2



The cost of Minimal spanning tree = 11.

The steps as per the algorithm are as follows:

Algorithm near (J) = k means, the nearest vertex to J is k.

The algorithm starts by selecting the minimum cost from the graph. The minimum cost edge is (2, 4).

K = 2, l = 4
Min cost = cost (2, 4) = 1

T [1, 1] = 2

T [1, 2] = 4

for i = 1 to 5 Begin i = 1 is cost (1, 4) < cost (1, 2) 8 < 4, No Than near (1) = 2 i = 2 is cost (2, 4) < cost (2, 2) 1 < ∝, Yes So near [2] = 4 i = 3 is cost (3, 4) < cost (3, 2) 1 < 4, Yes So near [3] = 4 i = 4 is cost (4, 4) < cost (4, 2) ∝ < 1, no So near [4] = 2 i = 5 is cost (5, 4) < cost (5, 2) 4 < ∞, yes So near [5] = 4 end near [k] = near [l] = 0 near [2] = near[4] = 0	Near matrix $\begin{array}{\|c\|c\|c\|c\|c\|}\hline 2 & & & & \\ \hline \end{array}$ 1 2 3 4 5 $\begin{array}{\|c\|c\|c\|c\|c\|}\hline 2 & 4 & & & \\ \hline \end{array}$ 1 2 3 4 5 $\begin{array}{\|c\|c\|c\|c\|c\|}\hline 2 & 4 & 4 & & \\ \hline \end{array}$ 1 2 3 4 5 $\begin{array}{\|c\|c\|c\|c\|c\|}\hline 2 & 4 & 4 & 2 & \\ \hline \end{array}$ 1 2 3 4 5 $\begin{array}{\|c\|c\|c\|c\|c\|}\hline 2 & 4 & 4 & 2 & 4 \\ \hline \end{array}$ 1 2 3 4 5 $\begin{array}{\|c\|c\|c\|c\|c\|}\hline 2 & 0 & 4 & 0 & 4 \\ \hline \end{array}$ 1 2 3 4 5	Edges added to min spanning tree: T [1, 1] = 2 T [1, 2] = 4

for i = 2 to n-1 (4) do
 	
<u>i = 2</u>
 	
for j = 1 to 5
j = 1
near(1)≠0 and cost(1, near(1))
2 ≠ 0 and cost (1, 2) = 4
 	
j = 2
near (2) = 0
 	
j = 3		

is near (3) ≠ 0
4 ≠ 0 and cost (3, 4) = 3
 	
j = 4
near (4) = 0
 	
J = 5
Is near (5) ≠ 0
4 ≠ 0 and cost (4, 5) = 4
 	
select the min cost from the above obtained costs, which is 3 and corresponding J = 3
 	
min cost = 1 + cost(3, 4)
= 1 + 3 = 4
 	
T (2, 1) = 3
T (2, 2) = 4
 	
Near [j] = 0
i.e. near (3) =0
 	
for (k = 1 to n)
 	
K = 1
is near (1) ≠ 0, yes
2 ≠ 0
and cost (1,2) > cost(1, 3)
4 > 9, No
 	
K = 2
Is near (2) ≠0, No
 	
K = 3
Is near (3) ≠ 0, No
 	
K = 4
Is near (4) ≠ 0, No
 	
K = 5
Is near (5) ≠ 0
4 ≠ 0, yes
and is cost (5, 4) > cost (5, 3)
4 > 3, yes
than near (5) = 3
 	
i = 3
 	
for (j = 1 to 5)
J = 1
is near (1) ≠0
2 ≠ 0
	2 0 0 0 4
	1 2 3 4 5
	
	2 0 0 0 3
	1 2 3 4 5
		T (2, 1) = 3
		T (2, 2) = 4

cost (1, 2) = 4 J = 2 Is near (2) ≠0, No		
J = 3 Is near (3) ≠ 0, no Near (3) = 0 J = 4 Is near (4) ≠ 0, no Near (4) = 0 J = 5 Is near (5) ≠ 0 Near (5) = 3 ➔ 3 ≠ 0, yes And cost (5, 3) = 3 Choosing the min cost from the above obtaining costs which is 3 and corresponding J = 5 Min cost = 4 + cost (5, 3) = 4 + 3 = 7 T (3, 1) = 5 T (3, 2) = 3 Near (J) = 0 ➔ near (5) = 0 for (k=1 to 5) k = 1 is near (1) ≠ 0, yes and cost(1,2) > cost(1,5) 4 > ∝, No K = 2 Is near (2) ≠ 0 no K = 3 Is near (3) ≠ 0 no K = 4 Is near (4) ≠ 0 no K = 5 Is near (5) ≠ 0 no **i = 4** for J = 1 to 5 J = 1 Is near (1) ≠ 0 2 ≠ 0, yes cost (1, 2) = 4	2 0 0 0 0 1 2 3 4 5	T (3, 1) = 5 T (3, 2) = 3

j = 2 is near (2) ≠ 0, No J = 3 Is near (3) ≠ 0, No Near (3) = 0 J = 4 Is near (4) ≠ 0, No Near (4) = 0 J = 5 Is near (5) ≠ 0, No Near (5) = 0 Choosing min cost from the above it is only '4' and corresponding J = 1 Min cost = 7 + cost (1,2) = 7+4 = 11 T (4, 1) = 1 T (4, 2) = 2 Near (J) = 0 ➔ Near (1) = 0 for (k = 1 to 5) K = 1 Is near (1) ≠ 0, No K = 2 Is near (2) ≠ 0, No K = 3 Is near (3) ≠ 0, No K = 4 Is near (4) ≠ 0, No K = 5 Is near (5) ≠ 0, No End.	0 0 0 0 0 1 2 3 4 5	T (4, 1) = 1 T (4, 2) = 2

4.8.7. The Single Source Shortest-Path Problem: DIJKSTRA'S ALGORITHMS

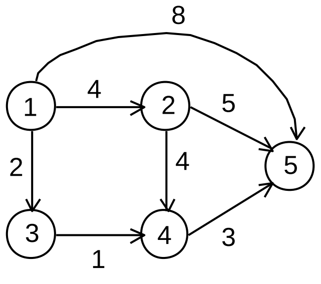
In the previously studied graphs, the edge labels are called as costs, but here we think them as lengths. In a labeled graph, the length of the path is defined to be the sum of the lengths of its edges.

In the single source, all destinations, shortest path problem, we must find a shortest path from a given source vertex to each of the vertices (called destinations) in the graph to which there is a path.

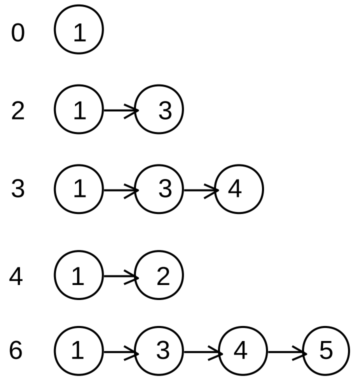
Dijkstra's algorithm is similar to prim's algorithm for finding minimal spanning trees. Dijkstra's algorithm takes a labeled graph and a pair of vertices P and Q, and finds the shortest path between then (or one of the shortest paths) if there is more than one. The principle of optimality is the basis for Dijkstra's algorithms.

Dijkstra's algorithm does not work for negative edges at all.

The figure lists the shortest paths from vertex 1 for a five vertex weighted digraph.



Graph



Shortest Paths

4.8.7.1. Algorithm:

```
Algorithm Shortest-Paths (v, cost, dist, n)
// dist [j], 1 ≤ j ≤ n, is set to the length of the shortest path
// from vertex v to vertex j in the digraph G with n vertices.
// dist [v] is set to zero. G is represented by its
// cost adjacency matrix cost [1:n, 1:n].
{
for  i :=1 to n do
{
            S [i] := false;                    // Initialize S.
dist [i] :=cost [v, i];
      }
S[v] := true; dist[v] := 0.0;                  // Put v in S.
for num := 2 to n – 1 do
      {
            Determine n - 1 paths from v.
            Choose u from among those vertices not in S such that dist[u] is minimum;
S[u] := true;                                  // Put u is S.
for (each w adjacent to u with S [w] = false) do
                  if (dist [w] > (dist [u] + cost [u, w]) then        // Update distances
                        dist [w] := dist [u] + cost [u, w];
      }
}
```

Running time:

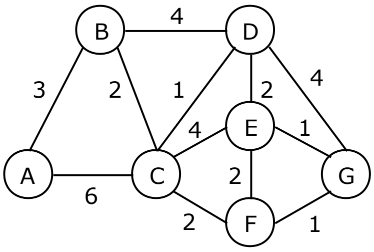
Depends on implementation of data structures for dist.

- Build a structure with n elements A
- at most m = |E| times decrease the value of an item mB

- 'n' times select the smallest value nC
- For <u>array</u> A = O (n); B = O (1); C = O (n) which gives O (n^2) total.
- For <u>heap</u> A = O (n); B = O (log n); C = O (log n) which gives O (n + m log n) total.

4.8.7.2. Example 1:

Use Dijkstras algorithm to find the shortest path from A to each of the other six vertices in the graph:



Solution:

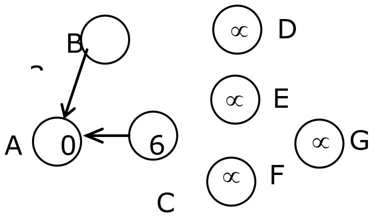
The cost adjacency matrix is $\begin{pmatrix} 0 & 3 & 6 & \infty & \infty & \infty & \infty \\ 3 & 0 & 2 & 4 & \infty & \infty & \infty \\ 6 & 2 & 0 & 1 & 4 & 2 & \infty \\ \infty & 4 & 1 & 0 & 2 & \infty & 4 \\ \infty & \infty & 4 & 2 & 0 & 2 & 1 \\ \infty & \infty & 2 & \infty & 2 & 0 & 1 \\ \infty & \infty & \infty & 4 & 1 & 1 & 0 \end{pmatrix}$

The problem is solved by considering the following information:

- Status[v] will be either '0', meaning that the shortest path from v to v_0 has definitely been found; or '1', meaning that it hasn't.
- Dist[v] will be a number, representing the length of the shortest path from v to v_0 found so far.
- Next[v] will be the first vertex on the way to v_0 along the shortest path found so far from v to v_0

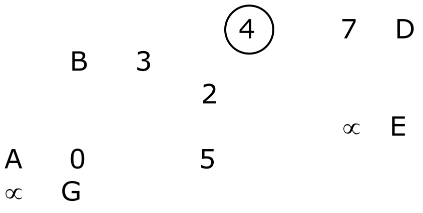
The progress of Dijkstra's algorithm on the graph shown above is as follows:

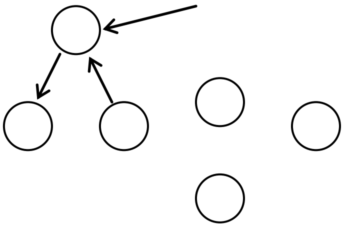
Step 1:



Vertex	A	B	C	D	E	F	G
Status	0	1	1	1	1	1	1
Dist.	0	3	6	∝	∝	∝	∝
Next	*	A	A	A	A	A	A

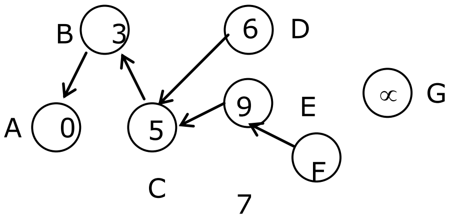
Step 2:





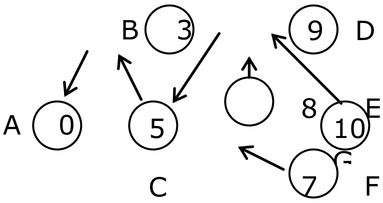
Vertex	A	B	C	D	E	F	G
Status	0	0	1	1	1	1	1
Dist.	0	3	5	7	∝	∝	∝
Next	*	A	B	B	A	A	A

Step 3:



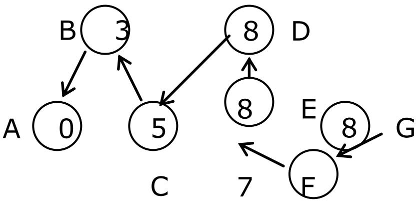
Vertex	A	B	C	D	E	F	G
Status	0	0	0	1	1	1	1
Dist.	0	3	5	6	9	7	∝
Next	*	A	B	C	C	C	A

Step 4:



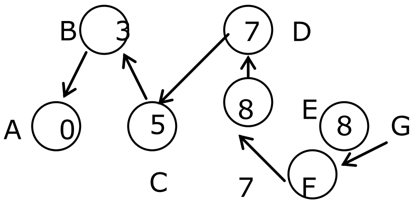
Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	1	1
Dist.	0	3	5	6	8	7	10
Next	*	A	B	C	D	C	D

Step 5:



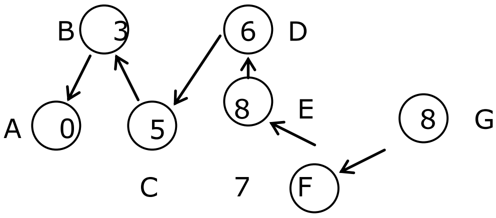
Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	0	1
Dist.	0	3	5	6	8	7	8
Next	*	A	B	C	D	C	F

Step 6:



Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	0	0	1
Dist.	0	3	5	6	8	7	8
Next	*	A	B	C	D	C	F

Step 7:



Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	0	0	0
Dist.	0	3	5	6	8	7	8
Next	*	A	B	C	D	C	F

Chapter 5

Dynamic Programming

Dynamic programming is a name, coined by Richard Bellman in 1955. Dynamic programming, as greedy method, is a powerful algorithm design technique that can be used when the solution to the problem may be viewed as the result of a sequence of decisions. In the greedy method we make irrevocable decisions one at a time, using a greedy criterion. However, in dynamic programming we examine the decision sequence to see whether an optimal decision sequence contains optimal decision subsequence.

When optimal decision sequences contain optimal decision subsequences, we can establish recurrence equations, called *dynamic-programming recurrence equations,* that enable us to solve the problem in an efficient way.

Dynamic programming is based on the principle of optimality (also coined by Bellman). The principle of optimality states that no matter whatever the initial state and initial decision are, the remaining decision sequence must constitute an optimal decision sequence with regard to the state resulting from the first decision. The principle implies that an optimal decision sequence is comprised of optimal decision subsequences. Since the principle of optimality may not hold for some formulations of some problems, it is necessary to verify that it does hold for the problem being solved. Dynamic programming cannot be applied when this principle does not hold.

The steps in a dynamic programming solution are:

- Verify that the principle of optimality holds
- Set up the dynamic-programming recurrence equations
- Solve the dynamic-programming recurrence equations for the value of the optimal solution.
- Perform a trace back step in which the solution itself is constructed.

Dynamic programming differs from the greedy method since the greedy method produces only one feasible solution, which may or may not be optimal, while dynamic programming produces all possible sub-problems at most once, one of which guaranteed to be optimal. Optimal solutions to sub-problems are retained in a table, thereby avoiding the work of recomputing the answer every time a sub-problem is encountered

The divide and conquer principle solve a large problem, by breaking it up into smaller problems which can be solved independently. In dynamic programming this principle is carried to an extreme: when we don't know exactly which smaller problems to solve, we simply solve them all, then store the answers away in a table to be used later in solving larger problems. Care is to be taken to avoid recomputing previously computed values, otherwise the recursive program will have prohibitive complexity. In some cases, the solution can be improved and in other cases, the dynamic programming technique is the best approach.

Two difficulties may arise in any application of dynamic programming:

1. It may not always be possible to combine the solutions of smaller problems to form the solution of a larger one.

2. The number of small problems to solve may be un-acceptably large.

There is no characterized precisely which problems can be effectively solved with dynamic programming; there are many hard problems for which it does not seen to be applicable, as well as many easy problems for which it is less efficient than standard algorithms.

5.1 MULTI STAGE GRAPHS

A multistage graph G = (V, E) is a directed graph in which the vertices are partitioned into $k \geq 2$ disjoint sets V_i, $1 \leq i \leq k$. In addition, if <u, v> is an edge in E, then $u \in V_i$ and $v \in V_{i+1}$ for some i, $1 \leq i < k$.

Let the vertex 's' is the source, and 't' the sink. Let c (i, j) be the cost of edge <i, j>. The cost of a path from 's' to 't' is the sum of the costs of the edges on the path. The multistage graph problem is to find a minimum cost path from 's' to 't'. Each set V_i defines a stage in the graph. Because of the constraints on E, every path from 's' to 't' starts in stage 1, goes to stage 2, then to stage 3, then to stage 4, and so on, and eventually terminates in stage k.

A dynamic programming formulation for a k-stage graph problem is obtained by first noticing that every s to t path is the result of a sequence of k – 2 decisions. The i^{th} decision involves determining which vertex in v_{i+1}, $1 \leq i \leq k - 2$, is to be on the path. Let c (i, j) be the cost of the path from source to destination. Then using the forward approach, we obtain:

$$\text{cost}(i, j) = \min_{\substack{l \,\varepsilon\, V_{i+1} \\ <j, l> \,\varepsilon\, E}} \{c(j, l) + \text{cost}(i + 1, l)\}$$

ALGORITHM:

```
Algorithm Fgraph (G, k, n, p)
// The input is a k-stage graph G = (V, E) with n vertices
// indexed in order or stages. E is a set of edges and c [i, j]
// is the cost of (i, j). p [1 : k] is a minimum cost path.
{
    cost [n] := 0.0;
    for j:= n - 1 to 1 step – 1 do
    {                                           // compute cost [j]
        let r be a vertex such that (j, r) is an edge
        of G and c [j, r] + cost [r] is minimum;
        cost [j] := c [j, r] + cost [r];
        d [j] := r:
    }
    p [1] := 1; p [k] := n;                     // Find a minimum cost path.
    for j := 2 to k - 1 do p [j] := d [p [j - 1]];
```

}

The multistage graph problem can also be solved using the backward approach. Let bp(i, j) be a minimum cost path from vertex s to j vertex in V_i. Let Bcost(i, j) be the cost of bp(i, j). From the backward approach we obtain:

$$\text{Bcost}(i, j) = \min_{\substack{l \,\varepsilon\, V_{i-1} \\ <l, j> \,\varepsilon\, E}} \{ \text{Bcost}(i - 1, l) + c(l, j)\}$$

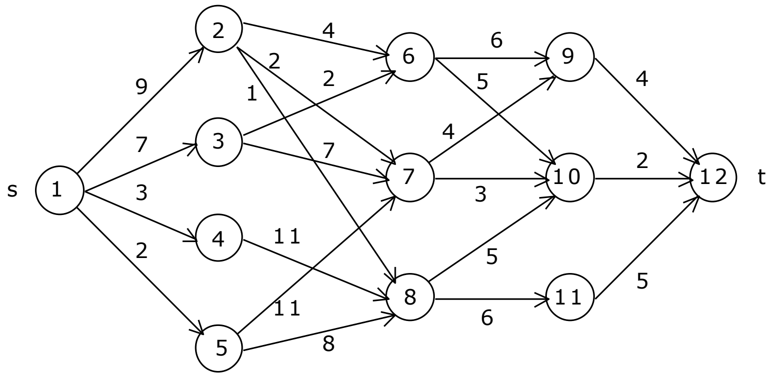
```
Algorithm Bgraph (G, k, n, p)
// Same function as Fgraph
{
    Bcost [1] := 0.0;
    for j := 2 to n do
    {                                       // Compute Bcost [j].
        Let r be such that (r, j) is an edge of
        G and Bcost [r] + c [r, j] is minimum;
        Bcost [j] := Bcost [r] + c [r, j];
        D [j] := r;
    }                                       //find a minimum cost path
    p [1] := 1; p [k] := n;
    for j:= k - 1 to 2 do p [j] := d [p [j + 1]];
}
```

Complexity Analysis:

The complexity analysis of the algorithm is fairly straightforward. Here, if G has $|E|$ edges, then the time for the first for loop is $\Phi(|V| + |E|)$.

EXAMPLE 1:

Find the minimum cost path from s to t in the multistage graph of five stages shown below. Do this first using forward approach and then using backward approach.



FORWARD APPROACH:

We use the following equation to find the minimum cost path from s to t:

$$\text{cost}(i, j) = \min_{\substack{l \; \varepsilon \; V_{i+1} \\ <j,\, l> \; \varepsilon \; E}} \{c(j, l) + \text{cost}(i + 1, l)\}$$

cost (1, 1) = min {c (1, 2) + cost (2, 2), c (1, 3) + cost (2, 3), c (1, 4) + cost (2, 4), c (1, 5) + cost (2, 5)}
= min {9 + cost (2, 2), 7 + cost (2, 3), 3 + cost (2, 4), 2 + cost (2, 5)}

Now first starting with,

cost (2, 2) = min{c (2, 6) + cost (3, 6), c (2, 7) + cost (3, 7), c (2, 8) + cost (3, 8)}
= min {4 + cost (3, 6), 2 + cost (3, 7), 1 + cost (3, 8)}

cost (3, 6) = min {c (6, 9) + cost (4, 9), c (6, 10) + cost (4, 10)}
= min {6 + cost (4, 9), 5 + cost (4, 10)}

cost (4, 9) = min {c (9, 12) + cost (5, 12)} = min {4 + 0) = 4

cost (4, 10) = min {c (10, 12) + cost (5, 12)} = 2

Therefore, cost (3, 6) = min {6 + 4, 5 + 2} = 7

cost (3, 7) = min {c (7, 9) + cost (4, 9) , c (7, 10) + cost (4, 10)}
= min {4 + cost (4, 9), 3 + cost (4, 10)}

cost (4, 9) = min {c (9, 12) + cost (5, 12)} = min {4 + 0} = 4

Cost (4, 10) = min {c (10, 2) + cost (5, 12)} = min {2 + 0} = 2

Therefore, cost (3, 7) = min {4 + 4, 3 + 2} = min {8, 5} = 5

cost (3, 8) = min {c (8, 10) + cost (4, 10), c (8, 11) + cost (4, 11)}
= min {5 + cost (4, 10), 6 + cost (4 + 11)}

cost (4, 11) = min {c (11, 12) + cost (5, 12)} = 5

Therefore, cost (3, 8) = min {5 + 2, 6 + 5} = min {7, 11} = 7

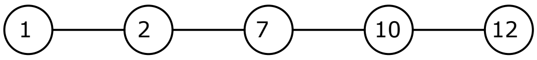
Therefore, cost (2, 2) = min {4 + 7, 2 + 5, 1 + 7} = min {11, 7, 8} = 7

Therefore, cost (2, 3) = min {c (3, 6) + cost (3, 6), c (3, 7) + cost (3, 7)}
= min {2 + cost (3, 6), 7 + cost (3, 7)}
= min {2 + 7, 7 + 5} = min {9, 12} = 9

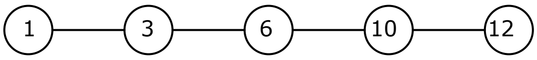
cost (2, 4) = min {c (4, 8) + cost (3, 8)} = min {11 + 7} = 18
cost (2, 5) = min {c (5, 7) + cost (3, 7), c (5, 8) + cost (3, 8)}
= min {11 + 5, 8 + 7} = min {16, 15} = 15

Therefore, cost (1, 1) = min {9 + 7, 7 + 9, 3 + 18, 2 + 15}
= min {16, 16, 21, 17} = 16

The minimum cost path is 16.

The path is 

or



BACKWARD APPROACH:

We use the following equation to find the minimum cost path from t to s:

$$\text{Bcost}(i, J) = \min_{\substack{l \,\varepsilon\, v_{i-1} \\ <l, j> \,\varepsilon\, E}} \{\text{Bcost}(i - 1, l) + c(l, J)\}$$

Bcost (5, 12) = min {Bcost (4, 9) + c (9, 12), Bcost (4, 10) + c (10, 12),
Bcost (4, 11) + c (11, 12)}
= min {Bcost (4, 9) + 4, Bcost (4, 10) + 2, Bcost (4, 11) + 5}

Bcost (4, 9) = min {Bcost (3, 6) + c (6, 9), Bcost (3, 7) + c (7, 9)}
= min {Bcost (3, 6) + 6, Bcost (3, 7) + 4}

Bcost (3, 6) = min {Bcost (2, 2) + c (2, 6), Bcost (2, 3) + c (3, 6)}
= min {Bcost (2, 2) + 4, Bcost (2, 3) + 2}

Bcost (2, 2) = min {Bcost (1, 1) + c (1, 2)} = min {0 + 9} = 9

Bcost (2, 3) = min {Bcost (1, 1) + c (1, 3)} = min {0 + 7} = 7

Bcost (3, 6) = min {9 + 4, 7 + 2} = min {13, 9} = 9

Bcost (3, 7) = min {Bcost (2, 2) + c (2, 7), Bcost (2, 3) + c (3, 7),
Bcost (2, 5) + c (5, 7)}

Bcost (2, 5) = min {Bcost (1, 1) + c (1, 5)} = 2

Bcost (3, 7) = min {9 + 2, 7 + 7, 2 + 11} = min {11, 14, 13} = 11

Bcost (4, 9) = min {9 + 6, 11 + 4} = min {15, 15} = 15

Bcost (4, 10) = min {Bcost (3, 6) + c (6, 10), Bcost (3, 7) + c (7, 10),
Bcost (3, 8) + c (8, 10)}

Bcost (3, 8) = min {Bcost (2, 2) + c (2, 8), Bcost (2, 4) + c (4, 8),
Bcost (2, 5) + c (5, 8)}
Bcost (2, 4) = min {Bcost (1, 1) + c (1, 4)} = 3

Bcost (3, 8) = min {9 + 1, 3 + 11, 2 + 8} = min {10, 14, 10} = 10

Bcost (4, 10) = min {9 + 5, 11 + 3, 10 + 5} = min {14, 14, 15) = 14

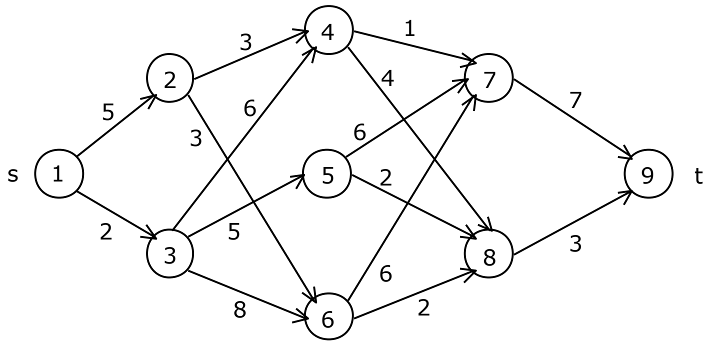
Bcost (4, 11) = min {Bcost (3, 8) + c (8, 11)} = min {Bcost (3, 8) + 6}

= min {10 + 6} = 16

Bcost (5, 12) = min {15 + 4, 14 + 2, 16 + 5} = min {19, 16, 21} = 16.

EXAMPLE 2:

Find the minimum cost path from s to t in the multistage graph of five stages shown below. Do this first using forward approach and then using backward approach.



SOLUTION:

FORWARD APPROACH:

$$\text{cost}(i, J) = \min_{\substack{l \, \varepsilon \, V_{i+1} \\ <J, l> \, \in E}} \{c(j, l) + \text{cost}(i + 1, l)\}$$

cost (1, 1) = min {c (1, 2) + cost (2, 2), c (1, 3) + cost (2, 3)}
= min {5 + cost (2, 2), 2 + cost (2, 3)}

cost (2, 2) = min {c (2, 4) + cost (3, 4), c (2, 6) + cost (3, 6)}
= min {3+ cost (3, 4), 3 + cost (3, 6)}

cost (3, 4) = min {c (4, 7) + cost (4, 7), c (4, 8) + cost (4, 8)}
= min {(1 + cost (4, 7), 4 + cost (4, 8)}

cost (4, 7) = min {c (7, 9) + cost (5, 9)} = min {7 + 0) = 7

cost (4, 8) = min {c (8, 9) + cost (5, 9)} = 3

Therefore, cost (3, 4) = min {8, 7} = 7

cost (3, 6) = min {c (6, 7) + cost (4, 7), c (6, 8) + cost (4, 8)}
= min {6 + cost (4, 7), 2 + cost (4, 8)} = min {6 + 7, 2 + 3} = 5

Therefore, cost (2, 2) = min {10, 8} = 8

cost (2, 3) = min {c (3, 4) + cost (3, 4), c (3, 5) + cost (3, 5), c (3, 6) + cost (3,6)}

cost (3, 5) = min {c (5, 7) + cost (4, 7), c (5, 8) + cost (4, 8)}= min {6 + 7, 2 + 3} = 5

Therefore, cost (2, 3) = min {13, 10, 13} = 10

cost (1, 1) = min {5 + 8, 2 + 10} = min {13, 12} = 12

BACKWARD APPROACH:

$$\text{Bcost}(i, J) = \min_{\substack{l \in v_{i-1} \\ <l, j> \in E}} \{\text{Bcost}(i - 1, l) = c(l, J)\}$$

$$\begin{aligned}
\text{Bcost}(5, 9) &= \min \{\text{Bcost}(4, 7) + c(7, 9), \text{Bcost}(4, 8) + c(8, 9)\} \\
&= \min \{\text{Bcost}(4, 7) + 7, \text{Bcost}(4, 8) + 3\}
\end{aligned}$$

$$\begin{aligned}
\text{Bcost}(4, 7) &= \min \{\text{Bcost}(3, 4) + c(4, 7), \text{Bcost}(3, 5) + c(5, 7), \\
&\qquad \text{Bcost}(3, 6) + c(6, 7)\} \\
&= \min \{\text{Bcost}(3, 4) + 1, \text{Bcost}(3, 5) + 6, \text{Bcost}(3, 6) + 6\}
\end{aligned}$$

$$\begin{aligned}
\text{Bcost}(3, 4) &= \min \{\text{Bcost}(2, 2) + c(2, 4), \text{Bcost}(2, 3) + c(3, 4)\} \\
&= \min \{\text{Bcost}(2, 2) + 3, \text{Bcost}(2, 3) + 6\}
\end{aligned}$$

$$\text{Bcost}(2, 2) = \min \{\text{Bcost}(1, 1) + c(1, 2)\} = \min \{0 + 5\} = 5$$

$$\text{Bcost}(2, 3) = \min (\text{Bcost}(1, 1) + c(1, 3)\} = \min \{0 + 2\} = 2$$

Therefore, $\text{Bcost}(3, 4) = \min \{5 + 3, 2 + 6\} = \min \{8, 8\} = 8$

$$\text{Bcost}(3, 5) = \min \{\text{Bcost}(2, 3) + c(3, 5)\} = \min \{2 + 5\} = 7$$

$$\begin{aligned}
\text{Bcost}(3, 6) &= \min \{\text{Bcost}(2, 2) + c(2, 6), \text{Bcost}(2, 3) + c(3, 6)\} \\
&= \min \{5 + 5, 2 + 8\} = 10
\end{aligned}$$

Therefore, $\text{Bcost}(4, 7) = \min \{8 + 1, 7 + 6, 10 + 6\} = 9$

$$\begin{aligned}
\text{Bcost}(4, 8) &= \min \{\text{Bcost}(3, 4) + c(4, 8), \text{Bcost}(3, 5) + c(5, 8), \\
&\qquad \text{Bcost}(3, 6) + c(6, 8)\} \\
&= \min \{8 + 4, 7 + 2, 10 + 2\} = 9
\end{aligned}$$

Therefore, $\text{Bcost}(5, 9) = \min \{9 + 7, 9 + 3\} = 12$

5.2. All pairs shortest paths

In the all pairs shortest path problem, we are to find a shortest path between every pair of vertices in a directed graph G. That is, for every pair of vertices (i, j), we are to find a shortest path from i to j as well as one from j to i. These two paths are the same when G is undirected.

When no edge has a negative length, the all-pairs shortest path problem may be solved by using Dijkstra's greedy single source algorithm n times, once with each of the n vertices as the source vertex.

The all pairs shortest path problem is to determine a matrix A such that A (i, j) is the length of a shortest path from i to j. The matrix A can be obtained by solving n single-source problems using the algorithm shortest Paths. Since each application of this procedure requires O (n^2) time, the matrix A can be obtained in O (n^3) time.

The dynamic programming solution, called Floyd's algorithm, runs in O (n^3) time. Floyd's algorithm works even when the graph has negative length edges (provided there are no negative length cycles).

The shortest i to j path in G, i ≠ j originates at vertex i and goes through some intermediate vertices (possibly none) and terminates at vertex j. If k is an intermediate vertex on this shortest path, then the subpaths from i to k and from k to j must be shortest paths from i to k and k to j, respectively. Otherwise, the i to j path is not of minimum length. So, the principle of optimality holds. Let A^k (i, j) represent the length of a shortest path from i to j going through no vertex of index greater than k, we obtain:

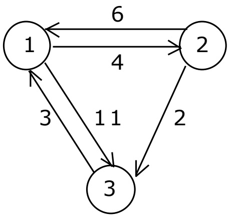
$$A^k (i, j) = \{\min \{\min_{1 \le k \le n} \{A^{k-1} (i, k) + A^{k-1} (k, j)\}, c (i, j)\}$$

```
Algorithm All Paths (Cost, A, n)
// cost [1:n, 1:n] is the cost adjacency matrix of a graph which
// n vertices; A [I, j] is the cost of a shortest path from vertex
// i to vertex j. cost [i, i] = 0.0, for 1 ≤ i ≤ n.
{
        for i := 1 to n do
                for j:= 1 to n do
                        A [i, j] := cost [i, j];                        // copy cost into A.
        for k := 1 to n do
                for i := 1 to n do
                        for j := 1 to n do
                                A [i, j] := min (A [i, j], A [i, k] + A [k, j]);
}
```

Complexity Analysis: A Dynamic programming algorithm based on this recurrence involves in calculating n+1 matrices, each of size n x n. Therefore, the algorithm has a complexity of O (n^3).

Example 1:

Given a weighted digraph G = (V, E) with weight. Determine the length of the shortest path between all pairs of vertices in G. Here we assume that there are no cycles with zero or negative cost.



$$\text{Cost adjacency matrix } (A^0) = \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix}$$

General formula: $\min_{1 \le k \le n} \{A^{k-1} (i, k) + A^{k-1} (k, j)\}, c (i, j)\}$

Solve the problem for different values of k = 1, 2 and 3

Step 1: Solving the equation for, k = 1;

A^1 (1, 1) = min $\{(A^o (1, 1) + A^o (1, 1)), c (1, 1)\}$ = min {0 + 0, 0} = 0

A^1 (1, 2) = min {(A^o (1, 1) + A^o (1, 2)), c (1, 2)} = min {(0 + 4), 4} = 4

A^1 (1, 3) = min {(A^o (1, 1) + A^o (1, 3)), c (1, 3)} = min {(0 + 11), 11} = 11

A^1 (2, 1) = min {(A^o (2, 1) + A^o (1, 1)), c (2, 1)} = min {(6 + 0), 6} = 6

A^1 (2, 2) = min {(A^o (2, 1) + A^o (1, 2)), c (2, 2)} = min {(6 + 4), 0)} = 0

A^1 (2, 3) = min {(A^o (2, 1) + A^o (1, 3)), c (2, 3)} = min {(6 + 11), 2} = 2

A^1 (3, 1) = min {(A^o (3, 1) + A^o (1, 1)), c (3, 1)} = min {(3 + 0), 3} = 3

A^1 (3, 2) = min {(A^o (3, 1) + A^o (1, 2)), c (3, 2)} = min {(3 + 4), $\propto$} = 7

A^1 (3, 3) = min {(A^o (3, 1) + A^o (1, 3)), c (3, 3)} = min {(3 + 11), 0} = 0

$$A^{(1)} = \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

Step 2: Solving the equation for, K = 2;

A^2 (1, 1) = min {(A^1 (1, 2) + A^1 (2, 1), c (1, 1)} = min {(4 + 6), 0} = 0

A^2 (1, 2) = min {(A^1 (1, 2) + A^1 (2, 2), c (1, 2)} = min {(4 + 0), 4} = 4

A^2 (1, 3) = min {(A^1 (1, 2) + A^1 (2, 3), c (1, 3)} = min {(4 + 2), 11} = 6

A^2 (2, 1) = min {(A (2, 2) + A (2, 1), c (2, 1)} = min {(0 + 6), 6} = 6

A^2 (2, 2) = min {(A (2, 2) + A (2, 2), c (2, 2)} = min {(0 + 0), 0} = 0

A^2 (2, 3) = min {(A (2, 2) + A (2, 3), c (2, 3)} = min {(0 + 2), 2} = 2

A^2 (3, 1) = min {(A (3, 2) + A (2, 1), c (3, 1)} = min {(7 + 6), 3} = 3

A^2 (3, 2) = min {(A (3, 2) + A (2, 2), c (3, 2)} = min {(7 + 0), 7} = 7

A^2 (3, 3) = min {(A (3, 2) + A (2, 3), c (3, 3)} = min {(7 + 2), 0} = 0

$$A^{(2)} = \begin{bmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

Step 3: Solving the equation for, k = 3;

A^3 (1, 1) = min {A^2 (1, 3) + A^2 (3, 1), c (1, 1)} = min {(6 + 3), 0} = 0

A^3 (1, 2) = min {A^2 (1, 3) + A^2 (3, 2), c (1, 2)} = min {(6 + 7), 4} = 4

A^3 (1, 3) = min {A^2 (1, 3) + A^2 (3, 3), c (1, 3)} = min {(6 + 0), 6} = 6

A^3 (2, 1) = min {A^2 (2, 3) + A^2 (3, 1), c (2, 1)} = min {(2 + 3), 6} = 5

A^3 (2, 2) = min {A^2 (2, 3) + A^2 (3, 2), c (2, 2)} = min {(2 + 7), 0} = 0

A^3 (2, 3) = min {A^2 (2, 3) + A^2 (3, 3), c (2, 3)} = min {(2 + 0), 2} = 2

A^3 (3, 1) = min {A^2 (3, 3) + A^2 (3, 1), c (3, 1)} = min {(0 + 3), 3} = 3

A^3 (3, 2) = min {A^2 (3, 3) + A^2 (3, 2), c (3, 2)} = min {(0 + 7), 7} = 7

A^3 (3, 3) = min {A^2 (3, 3) + A^2 (3, 3), c (3, 3)} = min {(0 + 0), 0} = 0

$$A^{(3)} \quad = \begin{bmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

5.3. TRAVELLING SALESPERSON PROBLEM

Let G = (V, E) be a directed graph with edge costs C_{ij}. The variable c_{ij} is defined such that $c_{ij} > 0$ for all I and j and $c_{ij} = \alpha$ if < i, j> $\notin$ E. Let |V| = n and assume n > 1. A tour of G is a directed simple cycle that includes every vertex in V. The cost of a tour is the sum of the cost of the edges on the tour. The traveling sales person problem is to find a tour of minimum cost. The tour is to be a simple path that starts and ends at vertex 1.

Let g (i, S) be the length of shortest path starting at vertex i, going through all vertices in S, and terminating at vertex 1. The function g (1, V – {1}) is the length of an optimal salesperson tour. From the principal of optimality it follows that:

$$g(1,\ V - \{1\}) = \min_{2 \le k \le n} \{c_{1k} + g\ (\ k,\ V - \{\ 1,\ k\ \})\} \qquad -- \qquad 1$$

Generalizing equation 1, we obtain (for i $\notin$ S)

$$g\ (i,\ S) = \min_{j \in S} \{c_{i\,j} + g\ (i,\ S - \{\ j\ \})\} \qquad -- \qquad 2$$

The Equation can be solved for g (1, V – 1}) if we know g (k, V – {1, k}) for all choices of k.

Complexity Analysis:

For each value of |S| there are n – 1 choices for i. The number of distinct sets S of size k not including 1 and i is $\binom{n-2}{k}$.

Hence, the total number of g (i, S)'s to be computed before computing g (1, V – {1}) is:

$$\sum_{k=0}^{n-1} (n-1) \binom{n-2}{k}$$

To calculate this sum, we use the binominal theorem:

$$(n - 1) \left[\binom{(n-2}{0} + \binom{(n - 2}{1} + \binom{(n - 2}{2} + ----+ \binom{(n - 2)}{(n - 2)} \right]$$

According to the binominal theorem:

$$\left[\binom{(n-2}{0} + \binom{(n - 2}{1} + \binom{(n - 2}{2} + ----+ \binom{(n - 2)}{(n - 2)} \right] = 2^{n - 2}$$

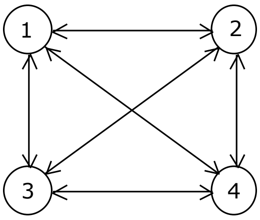
Therefore,

$$\sum_{k=0}^{n-1} (n-1) \binom{n-2}{k} = (n - 1)\ 2^{n-2}$$

This is Φ (n 2^{n-2}), so there are exponential number of calculate. Calculating one g (i, S) require finding the minimum of at most n quantities. Therefore, the entire algorithm is Φ (n^2 2^{n-2}). This is better than enumerating all n! different tours to find the best one. So, we have traded on exponential growth for a much smaller exponential growth. The most serious drawback of this dynamic programming solution is the space needed, which is O (n 2^n). This is too large even for modest values of n.

Example 1:

For the following graph find minimum cost tour for the traveling salesperson problem:



The cost adjacency matrix = $\begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$

Let us start the tour from vertex 1:

$$g(1, V - \{1\}) = \min_{2 \leq k \leq n} \{c_{1k} + g(k, V - \{1, K\})\} \qquad - \qquad (1)$$

More generally writing:

$$g(i, s) = \min \{c_{ij} + g(J, s - \{J\})\} \qquad - \qquad (2)$$

Clearly, $g(i, \Phi) = c_{i1}$, $1 \leq i \leq n$. So,

$g(2, \Phi) = C_{21} = 5$

$g(3, \Phi) = C_{31} = 6$

$g(4, \Phi) = C_{41} = 8$

Using equation – (2) we obtain:

$g(1, \{2, 3, 4\}) = \min \{c_{12} + g(2, \{3, 4\}, c_{13} + g(3, \{2, 4\}), c_{14} + g(4, \{2, 3\})\}$

$g(2, \{3, 4\}) = \min \{c_{23} + g(3, \{4\}), \; c_{24} + g(4, \{3\})\}$
$= \min \{9 + g(3, \{4\}), 10 + g(4, \{3\})\}$

$g(3, \{4\}) = \min \{c_{34} + g(4, \Phi)\} = 12 + 8 = 20$

$g(4, \{3\}) = \min \{c_{43} + g(3, \Phi)\} = 9 + 6 = 15$

Therefore, $g(2, \{3, 4\}) = \min \{9 + 20, 10 + 15\} = \min \{29, 25\} = 25$

$g(3, \{2, 4\}) = \min \{(c_{32} + g(2, \{4\}), (c_{34} + g(4, \{2\})\}$

$g(2, \{4\}) = \min \{c_{24} + g(4, \Phi)\} = 10 + 8 = 18$

$g(4, \{2\}) = \min \{c_{42} + g(2, \Phi)\} = 8 + 5 = 13$

Therefore, g (3, {2, 4}) = min {13 + 18, 12 + 13} = min {41, 25} = 25

g (4, {2, 3}) = min {c_{42} + g (2, {3}), c_{43} + g (3, {2})}

g (2, {3}) = min {c_{23} + g (3, Φ} = 9 + 6 = 15

g (3, {2}) = min {c_{32} + g (2, Φ} = 13 + 5 = 18

Therefore, g (4, {2, 3}) = min {8 + 15, 9 + 18} = min {23, 27} = 23

g (1, {2, 3, 4}) = min {c_{12} + g (2, {3, 4}), c_{13} + g (3, {2, 4}), c_{14} + g (4, {2, 3})}
= min {10 + 25, 15 + 25, 20 + 23} = min {35, 40, 43} = 35

The optimal tour for the graph has length = 35

The optimal tour is: 1, 2, 4, 3, 1.

5.4. OPTIMAL BINARY SEARCH TREE

Let us assume that the given set of identifiers is {a_1, . . . , a_n} with $a_1 < a_2 < . . . . < a_n$. Let p (i) be the probability with which we search for a_i. Let q (i) be the probability that the identifier x being searched for is such that $a_i < x < a_{i+1}$, $0 \leq i \leq n$ (assume $a_0 = -\propto$ and $a_{n+1} = +\propto$). We have to arrange the identifiers in a binary search tree in a way that minimizes the expected total access time.
In a binary search tree, the number of comparisons needed to access an element at depth 'd' is d + 1, so if 'a_i' is placed at depth 'd_i', then we want to minimize:

$$\sum_{i=1}^{n} P_i \ (1 + d_i) \ .$$

Let P (i) be the probability with which we shall be searching for 'a_i'. Let Q (i) be the probability of an un-successful search. Every internal node represents a point where a successful search may terminate. Every external node represents a point where an unsuccessful search may terminate.

The expected cost contribution for the internal node for 'a_i' is:

$$P(i) \ * \ level \ (a_i) \ .$$

Unsuccessful search terminate with I = 0 (i.e at an external node). Hence the cost contribution for this node is:

$$Q \ (i) \ * \ \text{level} \ ((E_i) - 1)$$

The expected cost of binary search tree is:

$$\sum_{i=1}^{n} P(i) \ * \ level \ (a_i) + \sum_{i=0}^{n} Q(i) \ * \ level \ ((E_i) - 1)$$

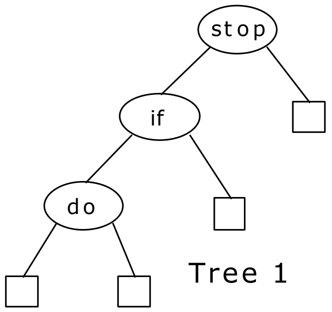
Given a fixed set of identifiers, we wish to create a binary search tree organization. We may expect different binary search trees for the same identifier set to have different performance characteristics.

The computation of each of these c(i, j)'s requires us to find the minimum of m quantities. Hence, each such c(i, j) can be computed in time O(m). The total time for all c(i, j)'s with j – i = m is therefore O(nm – m^2).

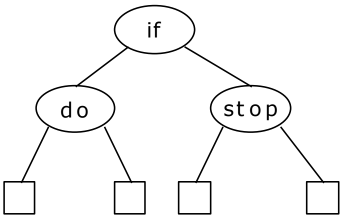
The total time to evaluate all the c(i, j)'s and r(i, j)'s is therefore:

$$\sum_{1 \leq m \leq n} \left(nm - m^2\right) = O\left(n^3\right)$$

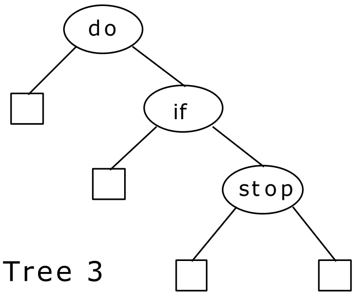
Example 1: The possible binary search trees for the identifier set (a_1, a_2, a_3) = (do, if, stop) are as follows. Given the equal probabilities p (i) = Q (i) = 1/7 for all i, we have:



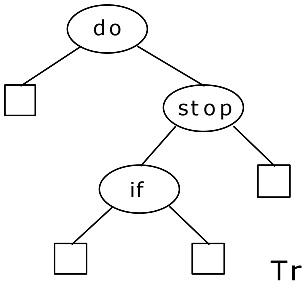
Tree 1



Tree 2



Tree 3



Tree 4

$$\text{Cost (tree \# 1)} = \left(\frac{1}{7}\ x\ 1 + \frac{1}{7}\ x\ 2 + \frac{1}{7}\ x\ 3\right) + \left(\frac{1}{7}\ x\ 1 + \frac{1}{7}\ x\ 2 + \frac{1}{7}\ x\ 3 + \frac{1}{7} x\ 3\right)$$

$$= \frac{1+2+3}{7} + \frac{1+2+3+3}{7} = \frac{6+9}{7} = \frac{15}{7}$$

$$\text{Cost (tree \# 2)} = \left(\frac{1}{7}\ x 1 + \frac{1}{7}\ x 2 + \frac{1}{7} x 2\right) + \left(\frac{1}{7} x 2 + \frac{1}{7}\ x 2 + \frac{1}{7}\ x 2 + \frac{1}{7} x 2\right)$$

$$= \frac{1+2+2}{7} + \frac{2+2+2+2}{7} = \frac{5+8}{7} = \frac{13}{7}$$

$$\text{Cost (tree \# 3)} = \left(\frac{1}{7}\ x\ 1 + \frac{1}{7}\ x\ 2 + \frac{1}{7}\ x\ 3\right) + \left(\frac{1}{7}\ x\ 1 + \frac{1}{7}\ x\ 2 + \frac{1}{7}\ x\ 3 + \frac{1}{7} x\ 3\right)$$

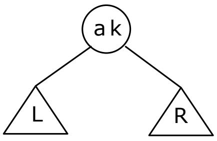
$$= \frac{1+2+3}{7} + \frac{1+2+3+3}{7} = \frac{6+9}{7} = \frac{15}{7}$$

$$\text{Cost (tree \# 4)} = \left(\frac{1}{7}\ x\ 1 + \frac{1}{7}\ x\ 2 + \frac{1}{7}\ x\ 3\right) + \left(\frac{1}{7}\ x\ 1 + \frac{1}{7}\ x\ 2 + \frac{1}{7}\ x\ 3 + \frac{1}{7} x\ 3\right)$$

$$= \frac{1+2+3}{7} + \frac{1+2+3+3}{7} = \frac{6+9}{7} = \frac{15}{7}$$

Huffman coding tree solved by a greedy algorithm has a limitation of having the data only at the leaves and it must not preserve the property that all nodes to the left of the root have keys, which are less etc. Construction of an optimal binary search tree is harder, because the data is not constrained to appear only at the leaves, and also because the tree must satisfy the binary search tree property and it must preserve the property that all nodes to the left of the root have keys, which are less.

A dynamic programming solution to the problem of obtaining an optimal binary search tree can be viewed by constructing a tree as a result of sequence of decisions by holding the principle of optimality. A possible approach to this is to make a decision as which of the a_i's be arraigned to the root node at 'T'. If we choose 'a_k' then is clear that the internal nodes for a_1, a_2, ... a_{k-1} as well as the external nodes for the classes E_o, E_1, ... E_{k-1} will lie in the left sub tree, L, of the root. The remaining nodes will be in the right subtree, R. The structure of an optimal binary search tree is:



Cost (L) = $\sum_{i=1}^{K} P(i) * level\ (a_i) + \sum_{i=0}^{K} Q(i) * \left(level(E_i)\ -1\right)$

Cost (R) = $\sum_{i=K}^{n} P(i) * level\ (a_i) + \sum_{i=K}^{n} Q(i) * \left(level(E_i)\ -1\right)$

The C (i, J) can be computed as:

$$C\ (i, J) = \min_{i<k\leq J} \{C\ (i, k-1) + C\ (k, J) + P\ (K) + w\ (i, K-1) + w\ (K, J)\}$$

$$= \min_{i<k\leq J} \{C\ (i, K-1) + C\ (K, J)\} + w\ (i, J) \qquad -- \qquad (1)$$

Where $W\ (i, J) = P\ (J) + Q\ (J) + w\ (i, J-1)$ -- (2)

Initially C (i, i) = 0 and w (i, i) = Q (i) for $0 \leq i \leq n$.

Equation (1) may be solved for C (0, n) by first computing all C (i, J) such that J - i = 1 Next, we can compute all C (i, J) such that J - i = 2, Then all C (i, J) with J - i = 3 and so on.

C (i, J) is the cost of the optimal binary search tree 'T_{ij}' during computation we record the root R (i, J) of each tree 'T_{ij}'. Then an optimal binary search tree may be constructed from these R (i, J). R (i, J) is the value of 'K' that minimizes equation (1).

We solve the problem by knowing W (i, i+1), C (i, i+1) and R (i, i+1), $0 \leq i < 4$; Knowing W (i, i+2), C (i, i+2) and R (i, i+2), $0 \leq i < 3$ and repeating until W (0, n), C (0, n) and R (0, n) are obtained.

The results are tabulated to recover the actual tree.

Example 1:

Let n = 4, and (a_1, a_2, a_3, a_4) = (do, if, need, while) Let P (1: 4) = (3, 3, 1, 1) and Q (0: 4) = (2, 3, 1, 1, 1)

Solution:

Table for recording W (i, j), C (i, j) and R (i, j):

Column Row	**0**	**1**	**2**	**3**	**4**
0	2, 0, 0	3, 0, 0	1, 0, 0	1, 0, 0,	1, 0, 0
1	8, 8, 1	7, 7, 2	3, 3, 3	3, 3, 4	
2	12, 19, 1	9, 12, 2	5, 8, 3		
3	14, 25, 2	11, 19, 2			
4	16, 32, 2				

This computation is carried out row-wise from row 0 to row 4. Initially, W (i, i) = Q (i) and C (i, i) = 0 and R (i, i) = 0, $0 \underline{<} i < 4$.

Solving for C (0, n):

First, computing all C (i, j) such that j - i = 1; j = i + 1 and as $0 \underline{<} i < 4$; i = 0, 1, 2 and 3; $i < k \leq J$. Start with i = 0; so j = 1; as $i < k \leq j$, so the possible value for k = 1

W (0, 1) = P (1) + Q (1) + W (0, 0) = 3 + 3 + 2 = 8
C (0, 1) = W (0, 1) + min {C (0, 0) + C (1, 1)} = 8
R (0, 1) = 1 (value of 'K' that is minimum in the above equation).

Next with i = 1; so j = 2; as $i < k \leq j$, so the possible value for k = 2

W (1, 2) = P (2) + Q (2) + W (1, 1) = 3 + 1 + 3 = 7
C (1, 2) = W (1, 2) + min {C (1, 1) + C (2, 2)} = 7
R (1, 2) = 2

Next with i = 2; so j = 3; as $i < k \leq j$, so the possible value for k = 3

W (2, 3) = P (3) + Q (3) + W (2, 2) = 1 + 1 + 1 = 3
C (2, 3) = W (2, 3) + min {C (2, 2) + C (3, 3)} = 3 + [(0 + 0)] = 3
R (2, 3) = 3

Next with i = 3; so j = 4; as $i < k \leq j$, so the possible value for k = 4

W (3, 4) = P (4) + Q (4) + W (3, 3) = 1 + 1 + 1 = 3
C (3, 4) = W (3, 4) + min {[C (3, 3) + C (4, 4)]} = 3 + [(0 + 0)] = 3
R (3, 4) = 4

Second, Computing all C (i, j) such that j - i = 2; j = i + 2 and as $0 \underline{<} i < 3$; i = 0, 1, 2; $i < k \leq J$. Start with i = 0; so j = 2; as $i < k \leq J$, so the possible values for k = 1 and 2.

W (0, 2) = P (2) + Q (2) + W (0, 1) = 3 + 1 + 8 = 12
C (0, 2) = W (0, 2) + min {(C (0, 0) + C (1, 2)), (C (0, 1) + C (2, 2))}
= 12 + min {(0 + 7, 8 + 0)} = 19

R (0, 2) = 1
Next, with i = 1; so j = 3; as i < k ≤ j, so the possible value for k = 2 and 3.

W (1, 3) = P (3) + Q (3) + W (1, 2) = 1 + 1+ 7 = 9
C (1, 3) = W (1, 3) + min {[C (1, 1) + C (2, 3)], [C (1, 2) + C (3, 3)]}
= W (1, 3) + min {(0 + 3), (7 + 0)} = 9 + 3 = 12
R (1, 3) = 2

Next, with i = 2; so j = 4; as i < k ≤ j, so the possible value for k = 3 and 4.

W (2, 4) = P (4) + Q (4) + W (2, 3) = 1 + 1 + 3 = 5
C (2, 4) = W (2, 4) + min {[C (2, 2) + C (3, 4)], [C (2, 3) + C (4, 4)]
= 5 + min {(0 + 3), (3 + 0)} = 5 + 3 = 8
R (2, 4) = 3

Third, Computing all C (i, j) such that J - i = 3; j = i + 3 and as 0 ≤ i < 2; i = 0, 1; i < k ≤ J. Start with i = 0; so j = 3; as i < k ≤ j, so the possible values for k = 1, 2 and 3.

W (0, 3) = P (3) + Q (3) + W (0, 2) = 1 + 1 + 12 = 14
C (0, 3) = W (0, 3) + min {[C (0, 0) + C (1, 3)], [C (0, 1) + C (2, 3)],
[C (0, 2) + C (3, 3)]}
= 14 + min {(0 + 12), (8 + 3), (19 + 0)} = 14 + 11 = 25
R (0, 3) = 2

Start with i = 1; so j = 4; as i < k ≤ j, so the possible values for k = 2, 3 and 4.

W (1, 4) = P (4) + Q (4) + W (1, 3) = 1 + 1 + 9 = 11
C (1, 4) = W (1, 4) + min {[C (1, 1) + C (2, 4)], [C (1, 2) + C (3, 4)],
[C (1, 3) + C (4, 4)]}
= 11 + min {(0 + 8), (7 + 3), (12 + 0)} = 11 + 8 = 19
R (1, 4) = 2

Fourth, Computing all C (i, j) such that j - i = 4; j = i + 4 and as 0 ≤ i < 1; i = 0; i < k ≤ J.

Start with i = 0; so j = 4; as i < k ≤ j, so the possible values for k = 1, 2, 3 and 4.

W (0, 4) = P (4) + Q (4) + W (0, 3) = 1 + 1 + 14 = 16
C (0, 4) = W (0, 4) + min {[C (0, 0) + C (1, 4)], [C (0, 1) + C (2, 4)],
[C (0, 2) + C (3, 4)], [C (0, 3) + C (4, 4)]}
= 16 + min [0 + 19, 8 + 8, 19+3, 25+0] = 16 + 16 = 32
R (0, 4) = 2

From the table we see that C (0, 4) = 32 is the minimum cost of a binary search tree for (a_1, a_2, a_3, a_4). The root of the tree 'T_{04}' is 'a_2'.

Hence the left sub tree is 'T_{01}' and right sub tree is T_{24}. The root of 'T_{01}' is 'a_1' and the root of 'T_{24}' is $a_{3.}$

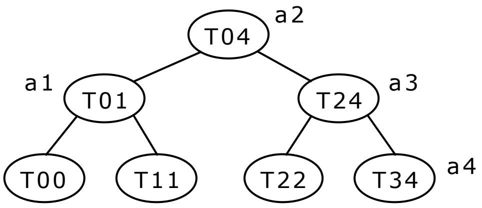
The left and right sub trees for 'T_{01}' are 'T_{00}' and 'T_{11}' respectively. The root of T_{01} is 'a_1'

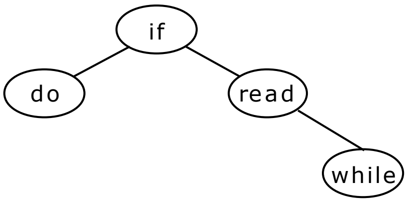
The left and right sub trees for T_{24} are T_{22} and T_{34} respectively.

The root of T_{24} is 'a_3'.

The root of T_{22} is null

The root of T_{34} is a_4.





Example 2:

Consider four elements a_1, a_2, a_3 and a_4 with Q_0 = 1/8, Q_1 = 3/16, Q_2 = Q_3 = Q_4 = 1/16 and p_1 = 1/4, p_2 = 1/8, p_3 = p_4 =1/16. Construct an optimal binary search tree.
Solving for C (0, n):

First, computing all C (i, j) such that j - i = 1; j = i + 1 and as 0 $\underline{<}$ i < 4; i = 0, 1, 2 and 3; i < k ≤ J. Start with i = 0; so j = 1; as i < k ≤ j, so the possible value for k = 1

W (0, 1) = P (1) + Q (1) + W (0, 0) = 4 + 3 + 2 = 9
C (0, 1) = W (0, 1) + min {C (0, 0) + C (1, 1)} = 9 + [(0 + 0)] = 9
R (0, 1) = 1 (value of 'K' that is minimum in the above equation).

Next with i = 1; so j = 2; as i < k ≤ j, so the possible value for k = 2

W (1, 2) = P (2) + Q (2) + W (1, 1) = 2 + 1 + 3 = 6
C (1, 2) = W (1, 2) + min {C (1, 1) + C (2, 2)} = 6 + [(0 + 0)] = 6
R (1, 2) = 2

Next with i = 2; so j = 3; as i < k ≤ j, so the possible value for k = 3

W (2, 3) = P (3) + Q (3) + W (2, 2) = 1 + 1 + 1 = 3
C (2, 3) = W (2, 3) + min {C (2, 2) + C (3, 3)} = 3 + [(0 + 0)] = 3
R (2, 3) = 3

Next with i = 3; so j = 4; as i < k ≤ j, so the possible value for k = 4

W (3, 4) = P (4) + Q (4) + W (3, 3) = 1 + 1 + 1 = 3
C (3, 4) = W (3, 4) + min {[C (3, 3) + C (4, 4)]} = 3 + [(0 + 0)] = 3
R (3, 4) = 4

Second, Computing all C (i, j) such that j - i = 2; j = i + 2 and as 0 $\underline{<}$ i < 3; i = 0, 1, 2; i < k ≤ J

Start with i = 0; so j = 2; as i < k ≤ j, so the possible values for k = 1 and 2.

W (0, 2) = P (2) + Q (2) + W (0, 1) = 2 + 1 + 9 = 12
C (0, 2) = W (0, 2) + min {(C (0, 0) + C (1, 2)), (C (0, 1) + C (2, 2))}
= 12 + min {(0 + 6, 9 + 0)} = 12 + 6 = 18
R (0, 2) = 1
Next, with i = 1; so j = 3; as i < k ≤ j, so the possible value for k = 2 and 3.

W (1, 3) = P (3) + Q (3) + W (1, 2) = 1 + 1+ 6 = 8
C (1, 3) = W (1, 3) + min {[C (1, 1) + C (2, 3)], [C (1, 2) + C (3, 3)]}
= W (1, 3) + min {(0 + 3), (6 + 0)} = 8 + 3 = 11

$R(1, 3) = 2$

Next, with $i = 2$; so $j = 4$; as $i < k \leq j$, so the possible value for $k = 3$ and 4.

$W(2, 4) = P(4) + Q(4) + W(2, 3) = 1 + 1 + 3 = 5$
$C(2, 4) = W(2, 4) + \min\{[C(2, 2) + C(3, 4)], [C(2, 3) + C(4, 4)]$
$= 5 + \min\{(0 + 3), (3 + 0)\} = 5 + 3 = 8$
$R(2, 4) = 3$

Third, Computing all $C(i, j)$ such that $J - i = 3$; $j = i + 3$ and as $0 \underline{<} i < 2$; $i = 0, 1$; $i < k \leq J$. Start with $i = 0$; so $j = 3$; as $i < k \leq j$, so the possible values for $k = 1, 2$ and 3.

$W(0, 3) = P(3) + Q(3) + W(0, 2) = 1 + 1 + 12 = 14$
$C(0, 3) = W(0, 3) + \min\{[C(0, 0) + C(1, 3)], [C(0, 1) + C(2, 3)],$
$[C(0, 2) + C(3, 3)]\}$
$= 14 + \min\{(0 + 11), (9 + 3), (18 + 0)\} = 14 + 11 = 25$
$R(0, 3) = 1$

Start with $i = 1$; so $j = 4$; as $i < k \leq j$, so the possible values for $k = 2, 3$ and 4.

$W(1, 4) = P(4) + Q(4) + W(1, 3) = 1 + 1 + 8 = 10$
$C(1, 4) = W(1, 4) + \min\{[C(1, 1) + C(2, 4)], [C(1, 2) + C(3, 4)],$
$[C(1, 3) + C(4, 4)]\}$
$= 10 + \min\{(0 + 8), (6 + 3), (11 + 0)\} = 10 + 8 = 18$
$R(1, 4) = 2$

Fourth, Computing all $C(i, j)$ such that $J - i = 4$; $j = i + 4$ and as $0 \underline{<} i < 1$; $i = 0$; $i < k \leq J$. Start with $i = 0$; so $j = 4$; as $i < k \leq j$, so the possible values for $k = 1, 2, 3$ and 4.

$W(0, 4) = P(4) + Q(4) + W(0, 3) = 1 + 1 + 14 = 16$
$C(0, 4) = W(0, 4) + \min\{[C(0, 0) + C(1, 4)], [C(0, 1) + C(2, 4)],$
$[C(0, 2) + C(3, 4)], [C(0, 3) + C(4, 4)]\}$
$= 16 + \min[0 + 18, 9 + 8, 18 + 3, 25 + 0] = 16 + 17 = 33$
$R(0, 4) = 2$

Table for recording $W(i, j)$, $C(i, j)$ and $R(i, j)$

Column Row	**0**	**1**	**2**	**3**	**4**
0	2, 0, 0	1, 0, 0	1, 0, 0	1, 0, 0,	1, 0, 0
1	9, 9, 1	6, 6, 2	3, 3, 3	3, 3, 4	
2	12, 18, 1	8, 11, 2	5, 8, 3		
3	14, 25, 2	11, 18, 2			
4	16, 33, 2				

From the table we see that $C(0, 4) = 33$ is the minimum cost of a binary search tree for (a_1, a_2, a_3, a_4)

The root of the tree 'T_{04}' is 'a_2'.

Hence the left sub tree is 'T_{01}' and right sub tree is T_{24}. The root of 'T_{01}' is 'a_1' and the root of 'T_{24}' is $a_{3.}$

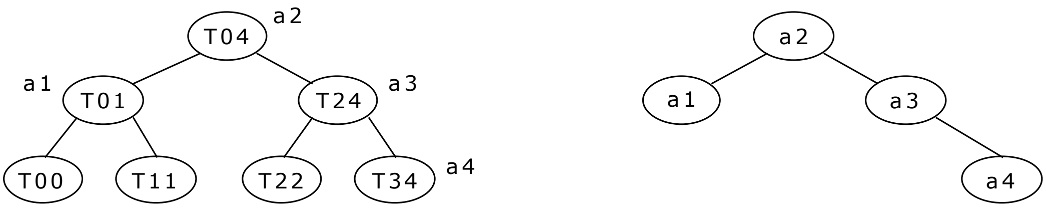
The left and right sub trees for 'T_{01}' are 'T_{00}' and 'T_{11}' respectively. The root of T_{01} is 'a_1'

The left and right sub trees for T_{24} are T_{22} and T_{34} respectively.

The root of T_{24} is 'a_3'.

The root of T_{22} is null.

The root of T_{34} is a_4.



Example 3:

WORD	PROBABILITY
A	4
B	2
C	1
D	3
E	5
F	2
G	1

and all other elements have zero probability.

Solving c(0,n):

First computing all c(i, j) such that j- i = 1;j = i +1 and as $0 \leq i < 7$; i = 0, 1, 2, 3, 4, 5 and 6; $i < k \leq j$. Start with i = 0 ; so j = 1; as $i < k \leq j$, so the possible value for k = 1

W(0, 1) = P(1) + Q(1)+W(0, 0) = 4+0+0 = 4
C(0, 1) = W(0, 1)+ min {C (0, 0) + C(1, 1) }=4 + [(0 + 0)] = 4
R(0, 1) = 1

next with i = 1 ; so j = 2; as $i < k \leq j$, so the possible value for k = 2

W(1, 2) = P(2) + Q(2)+W(1, 1) = 2+0+0 = 2
C(1, 2) = W(1, 2)+ min {C (1, 1) + C(2, 2) }=2 + [(0 + 0)] = 2
R(1, 2) = 2
next with i = 2 ; so j = 3; as $i < k \leq j$, so the possible value for k = 3

W(2, 3) = P(3) + Q(3)+W(2, 2) = 1+0+0 = 1
C(2, 3) = W(2, 3)+ min {C (2, 2) + C(3, 3) }=1 + [(0 + 0)] = 1
R(2, 3) = 3

next with i = 3 ; so j = 4; as i < k ≤ j, so the possible value for k = 4

W(3, 4) = P(4) + Q(4)+W(3, 3) = 3+0+0 = 3
C(3, 4) = W(3, 4)+ min {C (3, 4) + C(4, 4) }=3 + [(0 + 0)] = 3
R(3, 4) = 4

next with i = 4 ; so j = 5; as i < k ≤ j, so the possible value for k = 5

W(4, 5) = P(5) + Q(5)+W(4, 4) = 5+0+0 = 5
C(4, 5) = W(4, 5)+ min {C (4, 4) + C(5, 5) }=5 + [(0 + 0)] = 5
R(4, 5) = 5

next with i = 5; so j = 6; as i < k ≤ j, so the possible value for k = 6

W(5, 6) = P(6) + Q(6)+W(5, 5) = 2+0+0 = 2
C(5, 6) = W(5, 6)+ min {C (5, 5) + C(6, 6) }=2 + [(0 + 0)] = 2
R(5, 6) = 6

next with i = 6; so j = 7; as i < k ≤ j, so the possible value for k = 7

W(6, 7) = P(7) + Q(7)+W(6, 6) = 1+0+0 = 1
C(6, 7) = W(6, 7)+ min {C (6, 6) + C(7, 7) }=1 + [(0 + 0)] = 1
R(6, 7) = 7

Second, computing all c(i, j) such that j - i = 2 ;j = i + 2 and as 0 ≤ i < 6; i = 0, 1, 2, 3, 4 and 5; i < k ≤ j; Start with i = 0 ; so j = 2; as i < k ≤ j, so the possible values for k = 1 and 2.

W(0, 2) = P(2) + Q(2)+W(0, 1) = 2 + 0 + 4 = 6
C(0, 2) = W(0, 2)+ min {C (0, 0) + C(1, 2) ,C(0, 1) + C(2, 2)}
= 6 +min{ 0 + 2, 4 + 0} = 8
R(0, 2) = 1
next with i = 1 ; so j = 3; as i < k ≤ j, so the possible values for k = 2 and 3.

W(1, 3) = P(3) + Q(3) +W(1, 2) = 1+ 0 + 2 = 3
C(1, 3) = W(1, 3)+ min {C (1, 1) + C(2,3) ,C(1, 2) + C(3, 3)}
= 3 +min{ 0 + 1, 2 + 0} = 4
R(1, 3) = 2

next with i = 2 ; so j = 4; as i < k ≤ j, so the possible values for k = 3 and 4.

W(2, 4) = P(4) + Q(4) +W(2, 3) = 3+ 0 + 1 = 4
C(2, 4) = W(2, 4)+ min {C (2, 2) + C(3,4) ,C(2, 3) + C(4, 4)}
= 4 +min{ 0 + 3, 1 + 0} = 5
R(2, 4) = 4
next with i = 3 ; so j = 5; as i < k ≤ j, so the possible values for k = 4 and 5.

W(3, 5) = P(5) + Q(5)+W(3, 4) = 5+ 0 + 3 =8
C(3, 5) = W(3, 5)+ min {C (3, 3) + C(4,5) ,C(3,4) + C(5, 5)}
= 8 +min{ 0 + 5, 3 + 0} = 11
R(3, 5) = 5
next with i = 4 ; so j = 6; as i < k ≤ j, so the possible values for k = 5 and 6.

W(4, 6) = P(6) + Q(6)+W(4, 5) = 2+ 0 + 5 = 7
C(4, 6) = W(4, 6)+ min {C (4, 4) + C(5,6) ,C(4, 5) + C(6, 6)}
= 7 +min{ 0 + 2, 5 + 0} = 9
R(4, 6) = 5

next with i = 5 ; so j = 7; as i < k ≤ j, so the possible values for k = 6 and 7.

W(5, 7) = P(7) + Q(7)+W(5, 6) = 1+ 0 + 2 = 3
C(5, 7) = W(5, 7)+ min {C (5, 5) + C(6,7) ,C(5, 6) + C(7, 7)}
= 3 +min{ 0 + 1, 2 + 0} = 4
R(5, 7) = 6

Third, computing all c(i, j) such that j – i = 3 ;j = i + 3 and as 0 ≤ i < 5 ; i = 0, 1, 2, 3, 4 and I < k ≤ j.

Start with i = 0 ; so j = 3; as i < k ≤ j, so the possible values for k = 1,2 and 3.

W(0, 3) = P(3) + Q(3)+W(0, 2) = 1+ 0 + 6 = 7
C(0, 3) = W(0, 3)+ min {C (0, 0) + C(1,3) ,C(0, 1) + C(2, 3),C(0, 2) + C(3, 3)}
= 7 +min{ 0 + 4, 4 + 1, 8 + 0} = 7
R(0, 3) = 1

next with i = 1 ; so j = 4; as i < k ≤ j, so the possible values for k = 2,3 and 4.

W(1, 4) = P(4) + Q(4)+W(1, 3) = 3+ 0 + 3 = 6
C(1, 4) = W(1, 4)+ min {C (1, 1) + C(2, 4) ,C(1, 2) + C(3, 4),C(1, 3) + C(4, 4)}
= 6 +min{ 0 + 5, 2 + 3, 4 + 0} = 10
R(1, 4) = 4

next with i = 2 ; so j = 5; as i < k ≤ j, so the possible values for k = 3, 4 and 5.

W(2, 5) = P(5) + Q(5)+W(2, 4) = 5+ 0 + 4 = 9
C(2, 5) = W(2, 5)+ min {C (2, 2) + C(3, 5) ,C(2, 3) + C(4, 5),C(2, 4) + C(5, 5)}
= 9 +min{ 0 + 11, 1 + 5 ,5 + 0} = 14
R(2, 5) = 5

next with i = 3 ; so j = 6; as i < k ≤ j, so the possible values for k = 4, 5 and 6.

W(3, 6) = P(6) + Q(6)+W(3, 5) = 2+ 0 + 8 = 10
C(3, 6) = W(3, 6)+ min {C (3, 3) + C(4, 6) ,C(3 ,4) + C(5, 6),C(3, 5) + C(6, 6)}
= 10 +min{ 0 + 9 , 3 + 2 ,11 + 0} = 15
R(3, 6) = 5

next with i = 4 ; so j = 7; as i < k ≤ j, so the possible values for k = 5, 6 and 7.

W(4, 7) = P(7) + Q(7)+W(4, 6) = 1+ 0 + 7 = 8
C(4, 7) = W(4, 7)+ min {C (4, 4) + C(5, 7) ,C(4 ,5) + C(6, 7),C(4, 6) + C(7, 7)}
= 8 +min{ 0 + 4 , 5 + 1 ,9 + 0} = 12
R(4, 7) = 5

Fourth, computing all c(i, j) such that j – i = 4 ;j = i + 4 and as 0 ≤ i < 4 ; i = 0, 1, 2, 3 for i < k ≤ j. Start with i = 0 ; so j = 4; as i < k ≤ j, so the possible values for k = 1,2 ,3 and 4.

W(0, 4) = P(4) + Q(4)+W(0, 3) = 3+ 0 + 7 = 10
C(0, 4) = W(0, 4)+ min {C (0, 0) + C(1,4) ,C(0, 1) + C(2, 4),C(0, 2) + C(3, 4),
C(0, 3) + C(4, 4)}
= 10 +min{ 0 + 10, 4 + 5,8 + 3,11 + 0} = 19
R(0, 4) = 2

next with i = 1 ; so j = 5; as i < k ≤ j, so the possible values for k = 2,3 ,4 and 5.

W(1, 5) = P(5) + Q(5)+W(1, 4) = 5+ 0 + 6 = 11
C(1, 5) = W(1, 5)+ min {C (1, 1) + C(2, 5) ,C(1, 2) + C(3, 5),C(1, 3) + C(4, 5)
C(1, 4) + C(5, 5)}
= 11 +min{ 0 + 14, 2 + 11,4 + 5,10 +0} = 20
R(1, 5) = 4

next with i = 2 ; so j = 6; as i < k ≤ j, so the possible values for k = 3,4,5 and 6.

W(2, 6) = P(6) + Q(6)+W(2, 5) = 2+ 0 + 9 = 11
C(2, 6) = W(2, 6)+ min {C (2, 2) + C(3, 6) ,C(2, 3) + C(4, 6),C(2, 4) + C(5, 6)
C(2, 5) + C (6, 6)} = 11 +min{ 0 + 15, 1 + 9 ,5 + 2,14 + 0} = 18
R(2, 6) = 5

next with i = 3 ; so j = 7; as i < k ≤ j, so the possible values for k = 4,5,6 and 7.

W(3, 7) = P(7) + Q(7)+W(3, 6) = 1+ 0 +11 = 12
C(3, 7) = W(3, 7)+ min {C (3, 3) + C(4, 7) ,C(3, 4) + C(5, 7),C(3, 5) + C(6, 7)
C(3, 6) + C (7, 7)} = 12 +min{ 0 + 12, 3 +4 ,11 +1,15 + 0} = 19
R(3, 7) = 5

Fifth, computing all c(i, j) such that j – i = 5; j = i + 5 and as 0 ≤ i < 3; i = 0, 1, 2, i < k ≤ j. Start with i = 0 ; so j = 4; as i < k ≤ j, so the possible values for k = 1,2 ,3,4 and 5.

W(0, 5) = P(5) + Q(5)+W(0, 4) = 5+ 0 + 10 = 15
C(0, 5) = W(0, 5)+ min {C (0, 0) + C(1,5) ,C(0, 1) + C(2, 5),C(0, 2) + C(3, 5),
C(0, 3) + C(4, 5),C(0, 4) + C(5, 5)}
= 10 +min{ 0 + 20, 4 + 14, 8 + 11 ,19 + 0} = 28
R(0, 5) = 2

next with i = 1 ; so j = 6; as i < k ≤ j, so the possible values for k = 2, 3 ,4, 5 & 6.

W(1, 6) = P(6) + Q(6)+W(1, 5) = 2+ 0 + 11 = 13
C(1, 6) = W(1, 6)+ min {C (1, 1) + C(2, 6) ,C(1, 2) + C(3, 6),C(1, 3) + C(4, 6)
C(1, 4) + C(5, 6),C(1, 5)+C(6, 6)}
= 13 +min{ 0 + 18, 2 + 15, 4 + 9, 10 +2, 20 + 0} = 25
R(1, 6) = 5

next with i = 2 ; so j = 7; as i < k ≤ j, so the possible values for k = 3,4,5,6 and 7.

W(2, 7) = P(7) + Q(7)+W(2, 6) = 1+ 0 + 11 = 12
C(2, 7) = W(2, 7)+ min {C (2, 2) + C(3, 7) ,C(2, 3) + C(4, 7),C(2, 4) + C(5, 7)
C(2, 5) + C (6, 7),C(2, 6) + C(7,7)}
= 12 +min{ 0 + 18, 1 + 12 , 5 + 4, 14 + 1, 18 + 0} = 21
R(2, 7) = 5

Sixth, computing all c(i, j) such that j – i = 6 ;j = i + 6 and as 0 ≤ i < 2 ; i = 0, 1
i < k ≤ j. Start with i = 0; so j = 6; as i < k ≤ j, so the possible values for k = 1, 2, 3, 4 5 & 6.
W(0, 6) = P(6) + Q(6)+W(0, 5) = 2+ 0 + 15 = 17
C(0, 6) = W(0,6)+ min {C (0, 0) + C(1,6) ,C(0, 1) + C(2, 6),C(0, 2) + C(3, 6),
C(0, 3) + C(4, 6),C(0, 4) + C(5, 6),C(0, 5) + C(6, 6)}
= 17 +min{ 0 + 25, 4 + 18, 8 + 15,19 + 2, 31 + 0} = 37
R(0, 6) = 4

next with i = 1 ; so j = 7; as i < k ≤ j, so the possible values for k = 2, 3, 4, 5, 6 and 7.

W(1, 7) = P(7) + Q(7)+W(1, 6) = 1+ 0 + 13 = 14
C(1, 7) = W(1, 7)+ min {C (1, 1) + C(2, 7) ,C(1, 2) + C(3, 7),C(1, 3) + C(4, 7)
C(1, 4) + C(5, 7),C(1, 5)+C(6, 7),C(1, 6) +C(7, 7)}
= 14 +min{ 0 + 21, 2 + 18, 4 + 12, 10 + 4, 20 + 1, 21 + 0} = 28
R(1, 7) = 5

Seventh, computing all c(i, j) such that j – i = 7 ;j = i + 7 and as 0 ≤ i < 1 ; i = 0
i < k ≤ j. Start with i = 0 ; so j = 7; as i < k ≤ j, so the possible values for k = 1, 2, 3, 4, 5, 6 and 7.

W(0, 7) = P(7) + Q(7)+W(0, 6) = 1+ 0 + 17 = 18
C(0, 7) = W(0, 7)+ min {C (0, 0) + C(1, 7) ,C(0, 1) + C(2, 7),C(0, 2) + C(3, 7),
C(0, 3) + C(4, 7),C(0, 4) + C(5, 6),C(0, 5) + C (6, 7),C(0, 6) + C(7, 7)}
= 18 +min{ 0 + 28, 4 + 21, 8 + 18,19 +4, 31 + 1, 37 + 0} = 41
R(0, 7) = 4

5.5. 0/1 – KNAPSACK

We are given n objects and a knapsack. Each object i has a positive weight w_i and a positive value V_i. The knapsack can carry a weight not exceeding W. Fill the knapsack so that the value of objects in the knapsack is optimized.

A solution to the knapsack problem can be obtained by making a sequence of decisions on the variables $x_1, x_2, \ldots, x_n$. A decision on variable x_i involves determining which of the values 0 or 1 is to be assigned to it. Let us assume that decisions on the x_i are made in the order $x_n, x_{n-1}, \ldots x_1$. Following a decision on x_n, we may be in one of two possible states: the capacity remaining in m – w_n and a profit of p_n has accrued. It is clear that the remaining decisions $x_{n-1}, \ldots, x_1$ must be optimal with respect to the problem state resulting from the decision on x_n. Otherwise, $x_n, \ldots, x_1$ will not be optimal. Hence, the principal of optimality holds.

$$F_n (m) = \max \{f_{n-1} (m), f_{n-1} (m - w_n) + p_n\} \quad \text{--} \quad 1$$

For arbitrary $f_i (y)$, i > 0, this equation generalizes to:

$$F_i (y) = \max \{f_{i-1} (y), f_{i-1} (y - w_i) + p_i\} \quad \text{--} \quad 2$$

Equation-2 can be solved for $f_n (m)$ by beginning with the knowledge $f_o (y) = 0$ for all y and $f_i (y) = - \propto$, y < 0. Then $f_1, f_2, \ldots f_n$ can be successively computed using equation–2.

When the w_i's are integer, we need to compute $f_i (y)$ for integer y, $0 \leq y \leq m$. Since $f_i (y) = - \propto$ for y < 0, these function values need not be computed explicitly. Since each f_i can be computed from f_i - 1 in **Θ** (m) time, it takes **Θ** (m n) time to compute f_n. When the wi's are real numbers, $f_i (y)$ is needed for real numbers y such that $0 \leq y \leq m$. So, f_i cannot be explicitly computed for all y in this range. Even when the w_i's are integer, the explicit **Θ** (m n) computation of f_n may not be the most efficient computation. So, we explore **an alternative method for both cases.**

The $f_i (y)$ is an ascending step function; i.e., there are a finite number of y's, $0 = y_1 < y_2 < \ldots < y_k$, such that $f_i (y_1) < f_i (y_2) < \ldots < f_i (y_k)$; $f_i (y) = - \propto$, $y < y_1$; $f_i (y) = f (y_k)$, $y \geq y_k$; and $f_i (y) = f_i (y_j)$, $y_j \leq y \leq y_{j+1}$. So, we need to compute only $f_i (y_j)$, $1 \leq j \leq k$. We use the ordered set $S^i = \{(f (y_j), y_j) \mid 1 \leq j \leq k\}$ to represent $f_i (y)$. Each number of S^i is a pair (P, W), where P = f_i (y_j) and W = y_j. Notice that S^0 = {(0, 0)}. We can compute S^{i+1} from S_i by first computing:

$$S^i_1 = \{(P, W) \mid (P - p_i, W - w_i) \varepsilon S^i\}$$

Now, S^{i+1} can be computed by merging the pairs in S^i and S^i_1 together. Note that if S^{i+1} contains two pairs (P_j, W_j) and (P_k, W_k) with the property that $P_j \leq P_k$ and $W_j \geq W_k$, then the pair (P_j, W_j) can be discarded because of equation-2. Discarding or purging rules such as this one are also known as dominance rules. Dominated tuples get purged. In the above, (P_k, W_k) dominates (P_j, W_j).

Example 1:

Consider the knapsack instance n = 3, (w_1, w_2, w_3) = (2, 3, 4), (P_1, P_2, P_3) = (1, 2, 5) and M = 6.

Solution:

Initially, f_o (x) = 0, for all x and f_i (x) = - $\propto$ if x < 0.

F_n (M) = max {f_{n-1} (M), f_{n-1} (M - w_n) + p_n}

F_3 (6) = max (f_2 (6), f_2 (6 – 4) + 5} = max {f_2 (6), f_2 (2) + 5}

F_2 (6) = max (f_1 (6), f_1 (6 – 3) + 2} = max {f_1 (6), f_1 (3) + 2}

F_1 (6) = max (f_0 (6), f_0 (6 – 2) + 1} = max {0, 0 + 1} = 1

F_1 (3) = max (f_0 (3), f_0 (3 – 2) + 1} = max {0, 0 + 1} = 1

Therefore, F_2 (6) = max (1, 1 + 2} = 3

F_2 (2) = max (f_1 (2), f_1 (2 – 3) + 2} = max {f_1 (2), - $\propto$ + 2}

F_1 (2) = max (f_0 (2), f_0 (2 – 2) + 1} = max {0, 0 + 1} = 1

F_2 (2) = max {1, - $\propto$ + 2} = 1

Finally, f_3 (6) = max {3, 1 + 5} = 6

Other Solution:

For the given data we have:

S^0 = {(0, 0)}; S^0_1 = {(1, 2)}

S^1 = (S^0 U S^0_1) = {(0, 0), (1, 2)}

X - 2 = 0 => x = 2. y – 3 = 0 => y = 3
X - 2 = 1 => x = 3. y – 3 = 2 => y = 5

S^1_1 = {(2, 3), (3, 5)}

S^2 = (S^1 U S^1_1) = {(0, 0), (1, 2), (2, 3), (3, 5)}

$X - 5 = 0 \Rightarrow x = 5.$ $\quad$ $y - 4 = 0 \Rightarrow y = 4$
$X - 5 = 1 \Rightarrow x = 6.$ $\quad$ $y - 4 = 2 \Rightarrow y = 6$
$X - 5 = 2 \Rightarrow x = 7.$ $\quad$ $y - 4 = 3 \Rightarrow y = 7$
$X - 5 = 3 \Rightarrow x = 8.$ $\quad$ $y - 4 = 5 \Rightarrow y = 9$

$S^2_1 = \{(5, 4), (6, 6), (7, 7), (8, 9)\}$

$S^3 = (S^2 \cup S^2_1) = \{(0, 0), (1, 2), (2, 3), (3, 5), (5, 4), (6, 6), (7, 7), (8, 9)\}$

By applying Dominance rule,

$S^3 = (S^2 \cup S^2_1) = \{(0, 0), (1, 2), (2, 3), (5, 4), (6, 6)\}$

From (6, 6) we can infer that the maximum Profit $\sum p_i x_i = 6$ and weight $\sum x_i w_i = 6$

5.6. Reliability Design

The problem is to design a system that is composed of several devices connected in series. Let r_i be the reliability of device D_i (that is r_i is the probability that device i will function properly) then the reliability of the entire system is $\Pi\ r_i$. Even if the individual devices are very reliable (the r_i's are very close to one), the reliability of the system may not be very good. For example, if n = 10 and $r_i = 0.99$, $i \le i \le 10$, then $\Pi\ r_i = .904$. Hence, it is desirable to duplicate devices. Multiply copies of the same device type are connected in parallel.

If stage i contains mi copies of device D_i. Then the probability that all m_i have a malfunction is $(1 - r_i)^{m_i}$. Hence the reliability of stage i becomes $1 - (1 - r_i)^{m_i}$.

The reliability of stage 'i' is given by a function $\phi_i\ (m_i)$.

Our problem is to use device duplication. This maximization is to be carried out under a cost constraint. Let c_i be the cost of each unit of device i and let c be the maximum allowable cost of the system being designed.

We wish to solve:

$$\text{Maximize} \quad \pi_{1 \le i \le n} \ \phi_i \ (m_i)$$

$$\text{Subject to} \quad \sum_{1 \le i \le n} C_i \ m_i < C$$

$m_i \ge 1$ and interger, $1 \le i \le n$

Assume each $C_i > 0$, each m_i must be in the range $1 \le m_i \le u_i$, where

$$u_i = \left\lfloor \left(C + C_i - \sum_{1}^{n} C_J \right) \Big/ C_i \right\rfloor$$

The upper bound u_i follows from the observation that $m_j \geq 1$

An optimal solution $m_1, m_2 \; . \; . \; . \; . \; . \; m_n$ is the result of a sequence of decisions, one decision for each m_i.

Let f_i (x) represent the maximum value of $\underset{1 \leq j \leq i}{\pi} \; \phi \left(m_J\right)$

Subject to the constrains:

$$\sum_{1 \leq j \leq i} C_J \, m_J \leq x \quad \text{and} \quad 1 \leq m_j \leq u_J, \; 1 \leq j \leq i$$

The last decision made requires one to choose m_n from {1, 2, 3, ... u_n}

Once a value of m_n has been chosen, the remaining decisions must be such as to use the remaining funds C – C_n m_n in an optimal way.

The principle of optimality holds on

$$f_n \left(C\right) = \max_{1 \; \leq \; m_n \; \leq \; u_n} \left\{ \phi_n \left(m_n\right) f_{n-1} \left(C - C_n \; m_n\right) \right\}$$

for any f_i (x_i), i > 1, this equation generalizes to

$$f_n \; \left(x\right) = \max_{1 \; \leq \; m_i \; \leq \; u_i} \left\{ \phi_i \; \left(m_i\right) \; f_{i-1} \; \left(x - C_i \;\; m_i\right) \right\}$$

clearly, f_0 (x) = 1 for all x, $0 \leq x \leq C$ and f (x) = $-\infty$ for all x < 0.

Let S^i consist of tuples of the form (f, x), where f = f_i (x).

There is atmost one tuple for each different 'x', that result from a sequence of decisions on $m_1, m_2, \; . \; . \; . \; . \; m_n$. The dominance rule (f_1, x_1) dominate (f_2, x_2) if $f_1 \geq f_2$ and $x_1 \leq x_2$. Hence, dominated tuples can be discarded from S^i.

Example 1:

Design a three stage system with device types D_1, D_2 and D_3. The costs are \$30, \$15 and \$20 respectively. The Cost of the system is to be no more than \$105. The reliability of each device is 0.9, 0.8 and 0.5 respectively.

Solution:

We assume that if if stage I has mi devices of type i in parallel, then $\phi_i (m_i) = 1 - (1 - r_i)^{mi}$

Since, we can assume each $c_i > 0$, each m_i must be in the range $1 \leq m_i \leq u_i$. Where:

$$u_i = \left\lfloor \left(C + C_i - \sum_{1}^{n} C_J \right) \Big/ C_i \right\rfloor$$

Using the above equation compute u_1, u_2 and u_3.

$$u_1 = \frac{105+30-(30+15+20)}{30} = \frac{70}{30} = 2$$

$$u_2 = \frac{105+15-(30+15+20)}{15} = \frac{55}{15} = 3$$

$$u3 = \frac{105+20-(30+15+20)}{20} = \frac{60}{20} = 3$$

We use $S_j^i \rightarrow i: stage\ number\ and\ J: no.\ of\ devices\ in\ stage\ i = \mathrm{m_i}$

$S^o = \{f_o(x), x\}$ $\quad$ *initially* $f_o(x) = 1$ *and* $x = 0$, *so*, $S^o = \{1, 0\}$

Compute S^1, S^2 and S^3 as follows:

S^1 = depends on u_1 value, as u_1 = 2, so

$$S^1 = \left\{S_1^1, S_2^1\right\}$$

S^2 = depends on u_2 value, as u_2 = 3, so

$$S^2 = \left\{S_1^2, S_2^2, S_3^2\right\}$$

S^3 = depends on u_3 value, as u_3 = 3, so

$$S^3 = \left\{S_1^3, S_2^3, S_3^3\right\}$$

Now find, $S_1^1 = \left\{(f_1(x), x)\right\}$

$f_1(x) = \{\phi 1(1)\ f_o(\ ),\ \phi 1(2)\ f0(\ )\}$ With devices m_1 = 1 and m_2 = 2

Compute $\phi_1(1)$ and $\phi_1(2)$ using the formula: $\phi_i\ (mi)) = 1-(1-r_i)^{mi}$

$\phi_1(1) = 1-(1-r_1)^{m1} = 1-(1-0.9)^1 = 0.9$

$\phi_1(2) = 1-(1-0.9)^2 = 0.99$

$S_1^1 = \{f_1(x), x\} = = (0.9, 30)$

$S_2^1 = \{0.99, 30 + 30\} = (0.99, 60)$

Therefore, S^1 = {(0.9, 30), (0.99, 60)}

Next find $S_1^2 = \left\{(f_2(x), x)\right\}$

$f_2(x) = \{\phi_2(1) * f_1(\ ),\ \phi_2(2) * f_1(\ ),\ \phi_2(3) * f_1(\ )\}$

$\phi_2(1) = 1-(1-r_I)^{mi} = 1-(1-0.8)^1 = 1-0.2 = 0.8$

$\phi_2\ (2) = 1 - (1 - 0.8)^2 = 0.96$

$\phi_2\ (3) = 1 - (1 - 0.8)^3 = 0.992$

$S_1^2 = \{(0.8\,(0.9),\ 30+15),\ (0.8\,(0.99),\ 60+15)\} = \{(0.72,\ 45),\ (0.792,\ 75)\}$

$S_2^2 = \{(0.96\,(0.9),\ 30+15+15),\ (0.96\,(0.99),\ 60+15+15)\}$
$= \{(0.864,\ 60),\ (0.9504,\ 90)\}$

$S_3^2 = \{(0.992\,(0.9),\ 30+15+15+15),\ (0.992\,(0.99),\ 60+15+15+15)\}$
$= \{(0.8928,\ 75),\ (0.98208,\ 105)\}$

$S^2 = \left\{S_1^2,\ S_2^2,\ S_3^2\right\}$

By applying Dominance rule to S^2:

Therefore, $S^2 = \{(0.72, 45), (0.864, 60), (0.8928, 75)\}$

Dominance Rule:

If S^i contains two pairs (f_1, x_1) and (f_2, x_2) with the property that $f_1 \geq f_2$ and $x_1 \leq x_2$, then (f_1, x_1) dominates (f_2, x_2), hence by dominance rule (f_2, x_2) can be discarded. Discarding or pruning rules such as the one above is known as dominance rule. Dominating tuples will be present in S^i and Dominated tuples has to be discarded from S^i.

Case 1: if $f_1 \leq f_2$ and $x_1 > x_2$ then discard (f_1, x_1)

Case 2: if $f_1 \geq f_2$ and $x_1 < x_2$ the discard (f_2, x_2)

Case 3: otherwise simply write (f_1, x_1)

$S_2 = \{(0.72, 45), (0.864, 60), (0.8928, 75)\}$

$\phi_3\ (1) = 1 - (1 - r_I)^{mi} = 1 - (1 - 0.5)^1 = 1 - 0.5 = 0.5$

$\phi_3\ (2) = 1 - (1 - 0.5)^2 = 0.75$

$\phi_3\ (3) = 1 - (1 - 0.5)^3 = 0.875$

$S_1^3 = \{(0.5\ (0.72),\ 45 + 20),\ (0.5\ (0.864),\ 60 + 20),\ (0.5\ (0.8928),\ 75 + 20)\}$

$S_1^3 = \{(0.36,\ 65),\ (0.437,\ 80),\ (0.4464,\ 95)\}$

$S_2^3 = \{(0.75\ (0.72),\ 45 + 20 + 20),\ (0.75\ (0.864),\ 60 + 20 + 20),$
$(0.75\ (0.8928),\ 75 + 20 + 20)\}$

$= \{(0.54, 85), (0.648, 100), (0.6696, 115)\}$

$$S_3^3 = \{ (0.875\ (0.72),\ 45 + 20 + 20 + 20),\ (0.875\ (0.864),\ 60 + 20 + 20 + 20),$$
$$(0.875\ (0.8928),\ 75 + 20 + 20 + 20) \}$$

$$S_3^3 = \{(0.63,\ 105),\ (1.756,\ 120),\ (0.7812,\ 135)\}$$

If cost exceeds 105, remove that tuples

$S^3 = \{(0.36, 65), (0.437, 80), (0.54, 85), (0.648, 100)\}$

The best design has a reliability of 0.648 and a cost of 100. Tracing back for the solution through S^i 's we can determine that $m_3 = 2$, $m_2 = 2$ and $m_1 = 1$.

Other Solution:

According to the principle of optimality:

$$f_n(C) = \max_{1 \le m_n \le u_n} \{\phi_n(m_n).\ f_{n-1}(C - C_n m_n) \text{ with } f_o(x) = 1 \text{ and } 0 \le x \le C;$$

Since, we can assume each $c_i > 0$, each mi must be in the range $1 \le m_i \le u_i$. Where:

$$u_i = \left\lfloor \left(C + C_i - \sum_{i}^{n} C_J \right) / C_i \right\rfloor$$

Using the above equation compute u_1, u_2 and u_3.

$$u_1 = \frac{105 + 30 - (30 + 15 + 20)}{30} = \frac{70}{30} = 2$$

$$u_2 = \frac{105 + 15 - (30 + 15 + 20)}{15} = \frac{55}{15} = 3$$

$$u3 = \frac{105 + 20 - (30 + 15 + 20)}{20} = \frac{60}{20} = 3$$

$$f_3(105) = \max_{1 \le m3 \le u3} \{\phi_3(m_3).\ f_2(105 - 20m_3)\}$$

$$= \max \{\phi_3(1)\ f_2(105 - 20),\ \underline{\phi_3(2)\ f_2(105 - 20x2)},\ \phi_3(3)\ f_2(105 - 20x3)\}$$

$$= \max \{0.5\ f_2(85),\ 0.75\ f_2(65),\ 0.875\ f_2(45)\}$$

$$= \max \{0.5 \times 0.8928,\ 0.75 \times 0.864,\ 0.875 \times 0.72\} = 0.648.$$

$$f_2(85) = \max_{1 \le m2 \le u2} \{\phi_2(m_2).\ f_1(85 - 15m_2)\}$$

$$= \max \{\phi_2(1).f_1(85 - 15),\ \phi_2(2).f_1(85 - 15x2),\ \phi_2(3).f_1(85 - 15x3)\}$$

$$= \max \{0.8\ f_1(70),\ 0.96\ f_1(55),\ 0.992\ f_1(40)\}$$

$$= \max \{0.8 \times 0.99,\ 0.96 \times 0.9,\ 0.99 \times 0.9\} = 0.8928$$

$$f_1(70) = \max \{\phi_1(m_1).\ f_0(70 - 30m_1)\}$$

$1 \le m1 \le u1$

$= \max \{\phi_1(1)\ f_0(70 - 30),\ \phi_1(2)\ f_0\,(70 - 30x2)\}$

$= \max \{\phi_1(1) \text{ x } 1,\ \phi_1(2) \text{ x } 1\} = \max \{0.9, 0.99\} = 0.99$

$f_1\,(55) \quad = \max \{\phi_1(m_1).\ f_0(55 - 30m_1)\}$

$1 \le m1 \le u1$

$= \max \{\phi_1(1)\ f_0(50 - 30),\ \phi_1(2)\ f_0(50 - 30x2)\}$

$= \max \{\phi_1(1) \text{ x } 1,\ \phi_1(2) \text{ x } -\infty\} = \max \{0.9, -\infty\} = 0.9$

$f_1\,(40) \quad = \max \{\phi_1(m_1).\ f_0\,(40 - 30m_1)\}$

$1 \le m1 \le u1$

$= \max \{\phi_1(1)\ f_0(40 - 30),\ \phi_1(2)\ f_0(40 - 30x2)\}$

$= \max \{\phi_1(1) \text{ x } 1,\ \phi_1(2) \text{ x } -\infty\} = \max\{0.9, -\infty\} = 0.9$

$f_2\,(65) \quad = \max \{\phi_2(m_2).\ f_1(65 - 15m_2)\}$

$1 \le m2 \le u2$

$= \max \{\phi_2(1)\ f_1(65 - 15),\ \underline{\phi_2(2)\ f_1(65 - 15x2)},\ \phi_2(3)\ f_1(65 - 15x3)\}$

$= \max \{0.8\ \ f_1(50),\ 0.96\ \ f_1(35),\ 0.992\ \ f_1(20)\}$

$= \max \{0.8 \text{ x } 0.9,\ 0.96 \text{ x } 0.9,\ -\infty\} = 0.864$

$f_1\,(50) \quad = \max \{\phi_1(m_1).\ f_0(50 - 30m_1)\}$

$1 \le m1 \le u1$

$= \max \{\phi_1(1)\ f_0(50 - 30),\ \phi_1(2)\ f_0(50 - 30x2)\}$

$= \max \{\phi_1(1) \text{ x } 1,\ \phi_1(2) \text{ x } -\infty\} = \max\{0.9, -\infty\} = 0.9$

$f_1\,(35) \quad = \max\ \phi_1(m_1).\ f_0(35 - 30m_1)\}$

$1 \le m1 \le u1$

$= \max \{\underline{\phi_1(1).f_0(35-30)},\ \phi_1(2).f_0(35-30x2)\}$

$= \max \{\phi_1(1) \text{ x } 1,\ \phi_1(2) \text{ x } -\infty\} = \max\{0.9, -\infty\} = 0.9$

$f_1\,(20) \quad = \max \{\phi_1(m_1).\ f_0(20 - 30m_1)\}$

$1 \le m1 \le u1$

$= \max \{\phi_1(1)\ f_0(20 - 30),\ \phi_1(2)\ f_0(20 - 30x2)\}$

$= \max \{\phi_1(1) \text{ x } -\infty,\ \phi_1(2) \text{ x } -\infty\} = \max\{-\infty, -\infty\} = -\infty$

$f_2\,(45) \quad = \max \{\phi_2(m_2).\ f_1(45 - 15m_2)\}$

$1 \le m2 \le u2$

$= \max \{\phi_2(1)\ f_1(45 - 15),\ \phi_2(2)\ f_1(45 - 15x2),\ \phi_2(3)\ f_1(45 - 15x3)\}$

$= \max \{0.8\ f_1(30),\ 0.96\ f_1(15),\ 0.992\ f_1(0)\}$

$= \max \{0.8 \times 0.9, 0.96 \times -\infty, 0.99 \times -\infty\} = 0.72$

$$f_1(30) = \max_{1 \le m1 \le u1} \{\phi_1(m_1).\ f_0(30 - 30m_1)\}$$

$= \max \{\phi_1(1)\ f_0(30 - 30), \phi_1(2)\ f_0(30 - 30x2)\}$

$= \max \{\phi_1(1) \times 1, \phi_1(2) \times -\infty\} = \max\{0.9, -\infty\} = 0.9$

Similarly, $f_1(15) = -\infty$, $f_1(0) = -\infty$.

The best design has a reliability = 0.648 and

Cost = 30 x 1 + 15 x 2 + 20 x 2 = 100.

Tracing back for the solution through S^i 's we can determine that:

$m_3 = 2$, $m_2 = 2$ and $m_1 = 1$.

Chapter 6

BASIC TRAVERSAL AND SEARCH TECHNIQUES

Search means finding a path or traversal between a start node and one of a set of goal nodes. Search is a study of states and their transitions.

Search involves visiting nodes in a graph in a systematic manner, and may or may not result into a visit to all nodes. When the search necessarily involved the examination of every vertex in the tree, it is called the traversal.

6.1. Techniques for Traversal of a Binary Tree:

A binary tree is a finite (possibly empty) collection of elements. When the binary tree is not empty, it has a root element and remaining elements (if any) are partitioned into two binary trees, which are called the left and right subtrees.

There are three common ways to traverse a binary tree:

1. Preorder
2. Inorder
3. postorder

In all the three traversal methods, the left subtree of a node is traversed before the right subtree. The difference among the three orders comes from the difference in the time at which a node is visited.

6.1.1. Inorder Traversal:

In the case of inorder traversal, the root of each subtree is visited after its left subtree has been traversed but before the traversal of its right subtree begins. The steps for traversing a binary tree in inorder traversal are:

1. Visit the left subtree, using inorder.
2. Visit the root.
3. Visit the right subtree, using inorder.

The algorithm for preorder traversal is as follows:

```
treenode = record
{
        Type data;                              //Type is the data type of data.
        Treenode *lchild; treenode *rchild;
}
```

```
algorithm inorder (t)
// t is a binary tree. Each node of t has three fields: lchild, data, and rchild.
{
        if t ≠ 0 then
        {
                inorder (t → lchild);
                visit (t);
                inorder (t → rchild);
        }
}
```

6.1.2. Preorder Traversal:

In a preorder traversal, each node is visited before its left and right subtrees are traversed. Preorder search is also called backtracking. The steps for traversing a binary tree in preorder traversal are:

1. Visit the root.
2. Visit the left subtree, using preorder.
3. Visit the right subtree, using preorder.

The algorithm for preorder traversal is as follows:

```
Algorithm Preorder (t)
// t is a binary tree. Each node of t has three fields; lchild, data, and rchild.
{
        if t ≠ 0 then
        {
                visit (t);
                Preorder (t → lchild);
                Preorder (t → rchild);
        }
}
```

6.1.3. Postorder Traversal:

In a postorder traversal, each root is visited after its left and right subtrees have been traversed. The steps for traversing a binary tree in postorder traversal are:

1. Visit the left subtree, using postorder.
2. Visit the right subtree, using postorder
3. Visit the root.

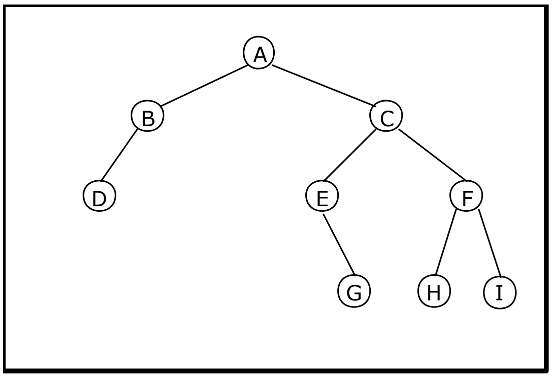
The algorithm for preorder traversal is as follows:

```
Algorithm Postorder (t)
// t is a binary tree. Each node of t has three fields : lchild, data, and rchild.
{
        if t ≠ 0 then
        {
                Postorder (t → lchild);
                Postorder (t → rchild);
                visit(t);
        }
}
```

6.1.4. Examples for binary tree traversal/search technique:

Example 1:

Traverse the following binary tree in pre, post and in-order.



Binary Tree

Preordering of the vertices:
A, B, D, C, E, G, F, H, I.

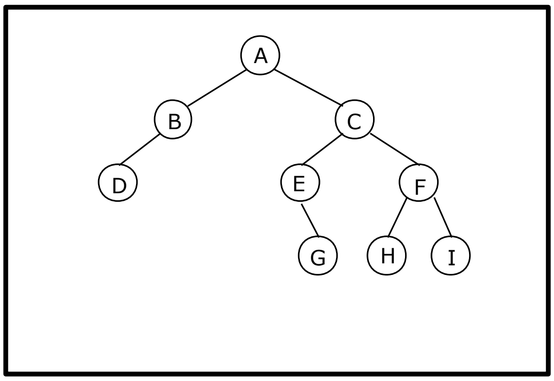
Postordering of the vertices:
D, B, G, E, H, I, F, C, A.

Inordering of the vertices:
D, B, A, E, G, C, H, F, I

Pre, Post and In-order Traversing

Example 2:

Traverse the following binary tree in pre, post, inorder and level order.



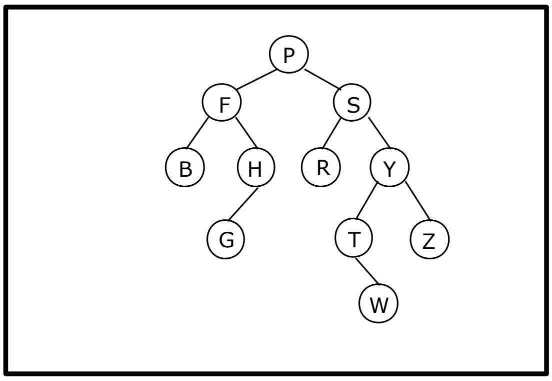
Binary Tree

- Preorder traversal yields:
  A, B, D, C, E, G, F, H, I
- Postorder traversal yields:
  D, B, G, E, H, I, F, C, A
- Inorder traversal yields:
  D, B, A, E, G, C, H, F, I
- Level order traversal yields:
  A, B, C, D, E, F, G, H, I

Pre, Post, Inorder and level order Traversing

Example 3:

Traverse the following binary tree in pre, post, inorder and level order.



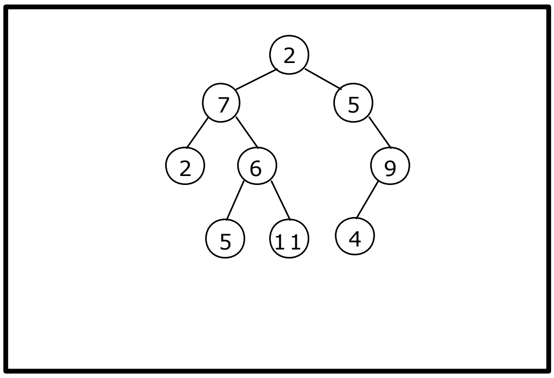
Binary Tree

- Preorder traversal yields:
  P, F, B, H, G, S, R, Y, T, W, Z
- Postorder traversal yields:
  B, G, H, F, R, W, T, Z, Y, S, P
- Inorder traversal yields:
  B, F, G, H, P, R, S, T, W, Y, Z
- Level order traversal yields:
  P, F, S, B, H, R, Y, G, T, Z, W

Pre, Post, Inorder and level order Traversing

Example 4:

Traverse the following binary tree in pre, post, inorder and level order.



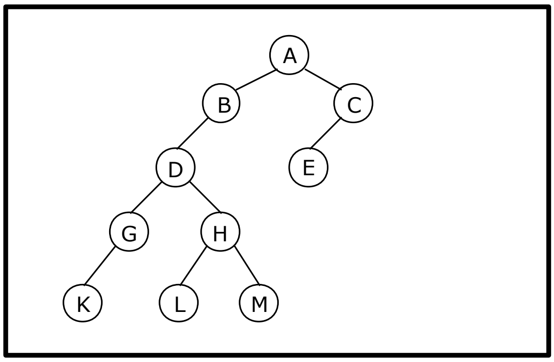
Binary Tree

- Preorder traversal yields:
  2, 7, 2, 6, 5, 11, 5, 9, 4
- Postorder travarsal yields:
  2, 5, 11, 6, 7, 4, 9, 5, 2
- Inorder travarsal yields:
  2, 7, 5, 6, 11, 2, 5, 4, 9
- Level order traversal yields:
  2, 7, 5, 2, 6, 9, 5, 11, 4

Pre, Post, Inorder and level order Traversing

Example 5:

Traverse the following binary tree in pre, post, inorder and level order.



Binary Tree

- Preorder traversal yields:
  A, B, D, G, K, H, L, M, C, E
- Postorder travarsal yields:
  K, G, L, M, H, D, B, E, C, A
- Inorder travarsal yields:
  K, G, D, L, H, M, B, A, E, C
- Level order traversal yields:
  A, B, C, D, E, G, H, K, L, M

Pre, Post, Inorder and level order Traversing

6.1.3. Non Recursive Binary Tree Traversal Algorithms:

At first glance, it appears we would always want to use the flat traversal functions since the use less stack space. But the flat versions are not necessarily better. For instance, some overhead is associated with the use of an explicit stack, which may negate the savings we gain from storing only node pointers. Use of the implicit function call stack may actually be faster due to special machine instructions that can be used.

Inorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex, pushing each vertex onto the stack and stop when there is no left son of vertex.

2. Pop and process the nodes on stack if zero is popped then exit. If a vertex with right son exists, then set right son of vertex as current vertex and return to step one.

The algorithm for inorder Non Recursive traversal is as follows:

```
Algorithm inorder()
{
          stack[1] = 0
          vertex = root
top:    while(vertex ≠ 0)
          {
                    push the vertex into the stack
                    vertex = leftson(vertex)
          }

          pop the element from the stack and make it as vertex

          while(vertex ≠ 0)
          {
                    print the vertex node
                    if(rightson(vertex) ≠ 0)
                    {
                              vertex = rightson(vertex)
                              goto top
                    }
                    pop the element from the stack and made it as vertex
          }
}
```

Preorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path by pushing the right son of vertex onto stack, if any and process each vertex. The traversing ends after a vertex with no left child exists.

2. Pop the vertex from stack, if vertex ≠ 0 then return to step one otherwise exit.

The algorithm for preorder Non Recursive traversal is as follows:

```
Algorithm preorder( )
{
          stack[1]: = 0
          vertex := root.
          while(vertex ≠ 0)
          {
                    print vertex node
```

```
                if(rightson(vertex) ≠ 0)
                        push the right son of vertex into the stack.
                if(leftson(vertex) ≠ 0)
                        vertex := leftson(vertex)
                else
                        pop the element from the stack and made it as vertex
        }
}
```

Postorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex. At each vertex of path push vertex on to stack and if vertex has a right son push –(right son of vertex) onto stack.

2. Pop and process the positive nodes (left nodes). If zero is popped then exit. If a negative node is popped, then ignore the sign and return to step one.

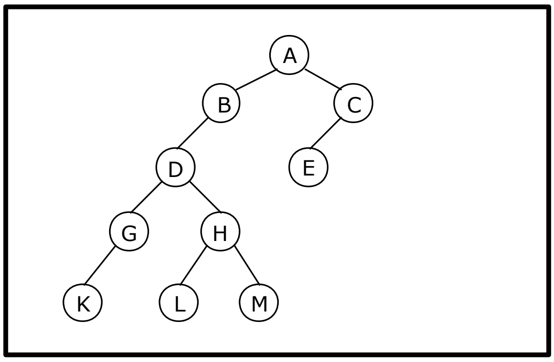
The algorithm for postorder Non Recursive traversal is as follows:

```
Algorithm postorder( )
{
        stack[1] := 0
        vertex := root

 top: while(vertex ≠ 0)
        {
                push vertex onto stack
                if(rightson(vertex) ≠ 0)
                        push -(vertex) onto stack
                vertex := leftson(vertex)
        }
        pop from stack and make it as vertex
        while(vertex > 0)
        {
                print the vertex node
                pop from stack and make it as vertex
        }
        if(vertex < 0)
        {
                vertex := -(vertex)
                goto top
        }
}
```

Example 1:

Traverse the following binary tree in pre, post and inorder using non-recursive traversing algorithm.



Binary Tree

- Preorder traversal yields:
  A, B, D, G, K, H, L, M, C, E
- Postorder travarsal yields:
  K, G, L, M, H, D, B, E, C, A
- Inorder travarsal yields:
  K, G, D, L, H, M, B, A, E, C

Pre, Post and Inorder Traversing

Inorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex, pushing each vertex onto the stack and stop when there is no left son of vertex.

2. Pop and process the nodes on stack if zero is popped then exit. If a vertex with right son exists, then set right son of vertex as current vertex and return to step one.

Current vertex	**Stack**	**Processed nodes**	**Remarks**
A	0		PUSH 0
	0 A B D G K		PUSH the left most path of A
K	0 A B D G	K	POP K
G	0 A B D	K G	POP G since K has no right son
D	0 A B	K G D	POP D since G has no right son
H	0 A B	K G D	Make the right son of D as vertex
H	0 A B H L	K G D	PUSH the leftmost path of H
L	0 A B H	K G D L	POP L
H	0 A B	K G D L H	POP H since L has no right son
M	0 A B	K G D L H	Make the right son of H as vertex
	0 A B M	K G D L H	PUSH the left most path of M
M	0 A B	K G D L H M	POP M
B	0 A	K G D L H M B	POP B since M has no right son
A	0	K G D L H M B A	Make the right son of A as vertex
C	0 C E	K G D L H M B A	PUSH the left most path of C
E	0 C	K G D L H M B A E	POP E
C	0	K G D L H M B A E C	Stop since stack is empty

Postorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex. At each vertex of path push vertex on to stack and if vertex has a right son push -(right son of vertex) onto stack.

2. Pop and process the positive nodes (left nodes). If zero is popped then exit. If a negative node is popped, then ignore the sign and return to step one.

Current vertex	**Stack**	**Processed nodes**	**Remarks**
A	0		PUSH 0
	0 A -C B D -H G K		PUSH the left most path of A with a -ve for right sons
	0 A -C B D -H	K G	POP all +ve nodes K and G
H	0 A -C B D	K G	Pop H
	0 A -C B D H -M L	K G	PUSH the left most path of H with a -ve for right sons
	0 A -C B D H -M	K G L	POP all +ve nodes L
M	0 A -C B D H	K G L	Pop M
	0 A -C B D H M	K G L	PUSH the left most path of M with a -ve for right sons
	0 A -C	K G L M H D B	POP all +ve nodes M, H, D and B
C	0 A	K G L M H D B	Pop C
	0 A C E	K G L M H D B	PUSH the left most path of C with a -ve for right sons
	0	K G L M H D B E C A	POP all +ve nodes E, C and A
	0		Stop since stack is empty

Preorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path by pushing the right son of vertex onto stack, if any and process each vertex. The traversing ends after a vertex with no left child exists.

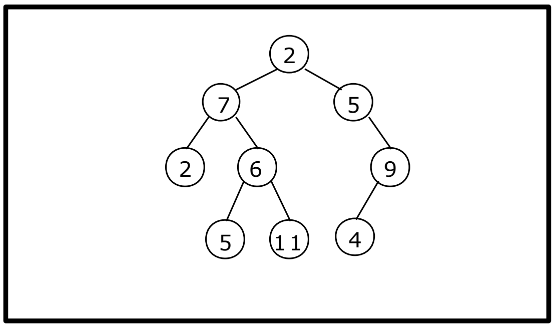
2. Pop the vertex from stack, if vertex ≠ 0 then return to step one otherwise exit.

Current vertex	**Stack**	**Processed nodes**	**Remarks**
A	0		PUSH 0
	0 C H	A B D G K	PUSH the right son of each vertex onto stack and process each vertex in the left most path
H	0 C	A B D G K	POP H

	0 C M	A B D G K H L	PUSH the right son of each vertex onto stack and process each vertex in the left most path
M	0 C	A B D G K H L	POP M
	0 C	A B D G K H L M	PUSH the right son of each vertex onto stack and process each vertex in the left most path; M has no left path
C	0	A B D G K H L M	Pop C
	0	A B D G K H L M C E	PUSH the right son of each vertex onto stack and process each vertex in the left most path; C has no right son on the left most path
	0	A B D G K H L M C E	Stop since stack is empty

Example 2:

Traverse the following binary tree in pre, post and inorder using non-recursive traversing algorithm.



Binary Tree

- Preorder traversal yields:
  2, 7, 2, 6, 5, 11, 5, 9, 4
- Postorder travarsal yields:
  2, 5, 11, 6, 7, 4, 9, 5, 2
- Inorder travarsal yields:
  2, 7, 5, 6, 11, 2, 5, 4, 9

Pre, Post and In order Traversing

Inorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex, pushing each vertex onto the stack and stop when there is no left son of vertex.

2. Pop and process the nodes on stack if zero is popped then exit. If a vertex with right son exists, then set right son of vertex as current vertex and return to step one.

Current vertex	**Stack**	**Processed nodes**	**Remarks**
2	0		
	0 2 7 2		
2	0 2 7	2	
7	0 2	2 7	
6	0 2 6 5	2 7	
5	0 2 6	2 7 5	
11	0 2	2 7 5 6 11	
5	0 5	2 7 5 6 11 2	
9	0 9 4	2 7 5 6 11 2 5	
4	0 9	2 7 5 6 11 2 5 4	

	0	2 7 5 6 11 2 5 4 9	Stop since stack is empty

Postorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex. At each vertex of path push vertex on to stack and if vertex has a right son push –(right son of vertex) onto stack.
2. Pop and process the positive nodes (left nodes). If zero is popped then exit. If a negative node is popped, then ignore the sign and return to step one.

Current vertex	**Stack**	**Processed nodes**	**Remarks**
2	0		
	0 2 -5 7 -6 2		
2	0 2 -5 7 -6	2	
6	0 2 -5 7	2	
	0 2 -5 7 6 -11 5	2	
5	0 2 -5 7 6 -11	2 5	
11	0 2 -5 7 6 11	2 5	
	0 2 -5	2 5 11 6 7	
5	0 2 5 -9	2 5 11 6 7	
9	0 2 5 9 4	2 5 11 6 7	
	0	2 5 11 6 7 4 9 5 2	Stop since stack is empty

Preorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path by pushing the right son of vertex onto stack, if any and process each vertex. The traversing ends after a vertex with no left child exists.

2. Pop the vertex from stack, if vertex ≠ 0 then return to step one otherwise exit.

Current vertex	**Stack**	**Processed nodes**	**Remarks**
2	0		
	0 5 6	2 7 2	
6	0 5 11	2 7 2 6 5	
11	0 5	2 7 2 6 5	
	0 5	2 7 2 6 5 11	
5	0 9	2 7 2 6 5 11	
9	0	2 7 2 6 5 11 5	
	0	2 7 2 6 5 11 5 9 4	Stop since stack is empty

6.2. Techniques for graphs:

Given a graph G = (V, E) and a vertex V in V (G) traversing can be done in two ways.

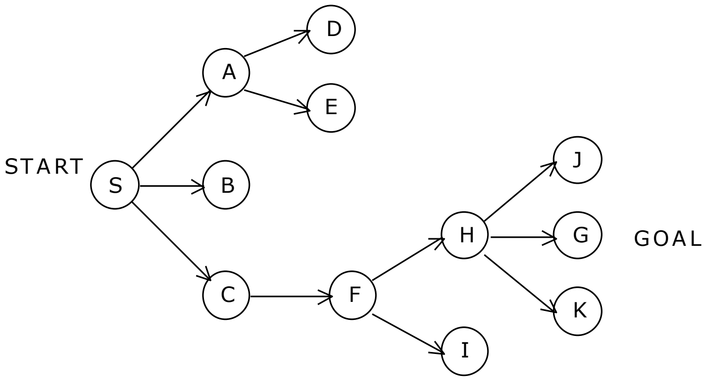
1. Depth first search
2. Breadth first search
3. D-search (Depth Search)

6.2.1. Depth first search:

With depth first search, the start state is chosen to begin, then some successor of the start state, then some successor of that state, then some successor of that and so on, trying to reach a goal state.

If depth first search reaches a state S without successors, or if all the successors of a state S have been chosen (visited) and a goal state has not get been found, then it "backs up" that means it goes to the immediately previous state or predecessor formally, the state whose successor was 'S' originally.

For example consider the figure. The circled letters are state and arrows are branches.



Suppose S is the start and G is the only goal state. Depth first search will first visit S, then A then D. But D has no successors, so we must back up to A and try its second successor, E. But this doesn't have any successors either, so we back up to A again. But now we have tried all the successors of A and haven't found the goal state G so we must back to 'S'. Now 'S' has a second successor, B. But B has no successors, so we back up to S again and choose its third successor, C. C has one successor, F. The first successor of F is H, and the first of H is J. J doesn't have any successors, so we back up to H and try its second successor. And that's G, the only goal state. So the solution path to the goal is S, C, F, H and G and the states considered were in order S, A, D, E, B, C, F, H, J, G.

Disadvantages:

1. It works very fine when search graphs are trees or lattices, but can get struck in an infinite loop on graphs. This is because depth first search can travel around a cycle in the graph forever.

   To eliminate this keep a list of states previously visited, and never permit search to return to any of them.

2. One more problem is that, the state space tree may be of infinite depth, to prevent consideration of paths that are too long, a maximum is often placed on the depth of nodes to be expanded, and any node at that depth is treated as if it had no successors.

3. We cannot come up with shortest solution to the problem.

Time Complexity:

Let n = |V| and e = |E|. Observe that the initialization portion requires Φ (n) time. Since we never visit a vertex twice, the number of times we go through the loop is at most n (exactly n assuming each vertex is reachable from the source). As, each vertex is visited at most once. At each vertex visited, we scan its adjacency list once. Thus, each edge is examined at most twice (once at each endpoint). So the total running time is O (n + e).

Alternatively,

If the average branching factor is assumed as 'b' and the depth of the solution as 'd', and maximum depth m ≥ d.

The worst case time complexity is $O(b^m)$ as we explore b^m nodes. If many solutions exists DFS will be likely to find faster than the BFS.

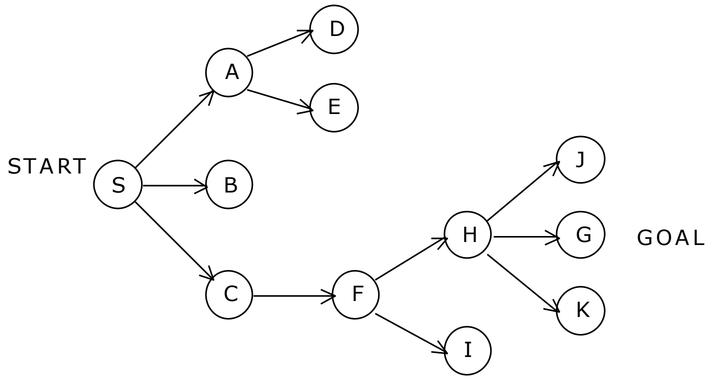
Space Complexity:

We have to store the nodes from root to current leaf and all the unexpanded siblings of each node on path. So, We need to store bm nodes.

6.2.2. Breadth first search:

Given an graph G = (V, E), breadth-first search starts at some source vertex S and "discovers" which vertices are reachable from S. Define the distance between a vertex V and S to be the minimum number of edges on a path from S to V. Breadth-first search discovers vertices in increasing order of distance, and hence can be used as an algorithm for computing shortest paths (where the length of a path = number of edges on the path). Breadth-first search is named because it visits vertices across the entire breadth.

To illustrate this let us consider the following tree:



Breadth first search finds states level by level. Here we first check all the immediate successors of the start state. Then all the immediate successors of these, then all the immediate successors of these, and so on until we find a goal node. Suppose S is the start state and G is the goal state. In the figure, start state S is at level 0; A, B and C are at level 1; D, e and F at level 2; H and I at level 3; and J, G and K at level 4. So breadth first search, will consider in order S, A, B, C, D, E, F, H, I, J and G and then stop because it has reached the goal node.

Breadth first search does not have the danger of infinite loops as we consider states in order of increasing number of branches (level) from the start state.

One simple way to implement breadth first search is to use a queue data structure consisting of just a start state. Any time we need a new state, we pick it from the front of the queue and any time we find successors, we put them at the end of the queue. That way we are guaranteed to not try (find successors of) any states at level 'N' until all states at level 'N – 1' have been tried.

Time Complexity:

The running time analysis of BFS is similar to the running time analysis of many graph traversal algorithms. Let n = |V| and e = |E|. Observe that the initialization portion requires Φ (n) time. Since we never visit a vertex twice, the number of times we go through the loop is at most n (exactly n, assuming each vertex is reachable from the source). So, Running time is O (n + e) as in DFS. For a directed graph the analysis is essentially the same.

Alternatively,

If the average branching factor is assumed as 'b' and the depth of the solution as 'd'.

In the worst case we will examine $1 + b + b^2 + b^3 + . . . + b^d = (b^{d+1} - 1) / (b - 1) = O(b^d)$.

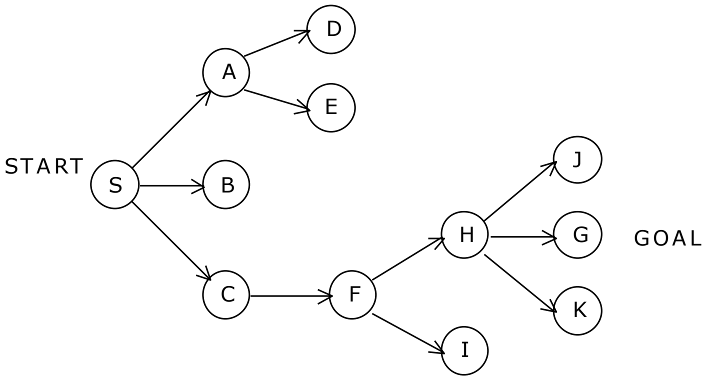
In the average case the last term of the series would be $b^d / 2$. So, the complexity is still $O(b^d)$

Space Complexity:

Before examining any node at depth d, all of its siblings must be expanded and stored. So, space requirement is also $O(b^d)$.

6.2.3. Depth Search (D-Search):

The exploration of a new node cannot begin until the node currently being explored is fully explored. D-search like state space search is called LIFO (Last In First Out) search which uses stack data structure. To illustrate the D-search let us consider the following tree:



The search order for goal node (G) is as follows: S, A, B, C, F, H, I, J, G.

Time Complexity:

The time complexity is same as Breadth first search.

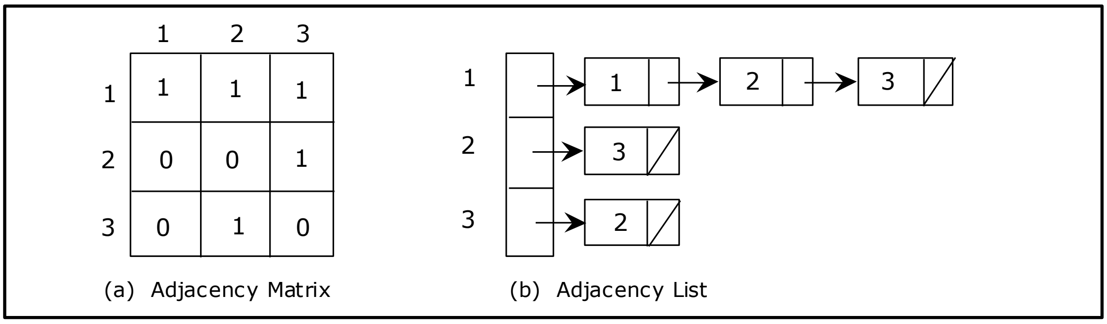
6.3. Representation of Graphs and Digraphs by Adjacency List:

We will describe two ways of representing digraphs. We can represent undirected graphs using exactly the same representation, but we will double each edge, representing the undirected edge {v, w} by the two oppositely directed edges (v, w) and (w, v). Notice that even though we represent undirected graphs in the same way that we represent digraphs, it is important to remember that these two classes of objects are mathematically distinct from one another.

Let G = (V, E) be a digraph with n = |V| and let e = |E|. We will assume that the vertices of G are indexed {1, 2, ... , n}.

$$A\,[v,\,w] = \begin{cases} 1 & \text{if } (v,w) \in E \\ 0 & \text{otherwise} \end{cases}$$

Adjacency List: An array Adj [1 . . . . n] of pointers where for 1 $\leq$ v $\leq$ n, Adj [v] points to a linked list containing the vertices which are adjacent to v (i.e. the vertices that can be reached from v by a single edge). If the edges have weights then these weights may also be stored in the linked list elements.



$$\begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline 1 & 1 & 1 & 1 \\ 2 & 0 & 0 & 1 \\ 3 & 0 & 1 & 0 \end{array}$$

(a) Adjacency Matrix

(b) Adjacency List

Adjacency matrix and adjacency list

An adjacency matrix requires **Θ** (n^2) storage and an adjacency list requires **Θ** (n + e) storage.

Adjacency matrices allow faster access to edge queries (for example, is (u, v) $\in$ E) and adjacency lists allow faster access to enumeration tasks (for example, find all the vertices adjacent to v).

6.4. Depth First and Breadth First Spanning Trees:

BFS and DFS impose a tree (the BFS/DFS tree) along with some auxiliary edges (cross edges) on the structure of graph. So, we can compute a spanning tree in a graph. The computed spanning tree is not a minimum spanning tree. Trees are much more structured objects than graphs. For example, trees break up nicely into subtrees, upon which subproblems can be solved recursively. For directed graphs the other edges of the graph can be classified as follows:

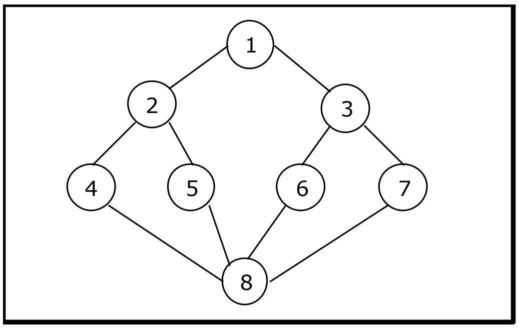
Back edges: (u, v) where v is a (not necessarily proper) ancestor of u in the tree. (Thus, a self-loop is considered to be a back edge).
Forward edges: (u, v) where v is a proper descendent of u in the tree.

Cross edges: (u, v) where u and v are not ancestors or descendents of one another (in fact, the edge may go between different trees of the forest).

6.4.1. Depth first search and traversal:

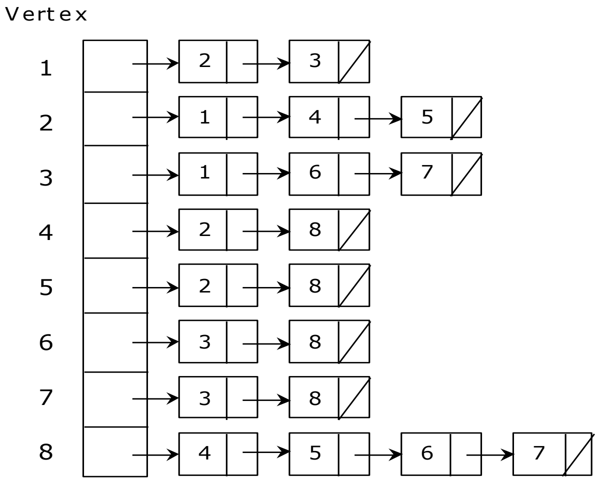
Depth first search of undirected graph proceeds as follows. The start vertex V is visited. Next an unvisited vertex 'W' adjacent to 'V' is selected and a depth first search from 'W' is initiated. When a vertex 'u' is reached such that all its adjacent vertices have been visited, we back up to the last vertex visited, which has an unvisited vertex 'W' adjacent to it and initiate a depth first search from W. The search terminates when no unvisited vertex can be reached from any of the visited ones.

Let us consider the following Graph (G):

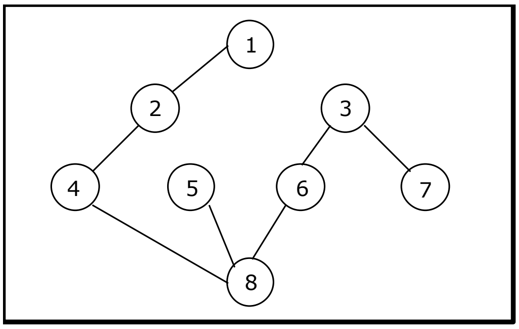


Graph

The adjacency list for G is:



If the depth first is initiated from vertex 1, then the vertices of G are visited in the order: 1, 2, 4, 8, 5, 6, 3, 7. The depth first spanning tree is as follows:

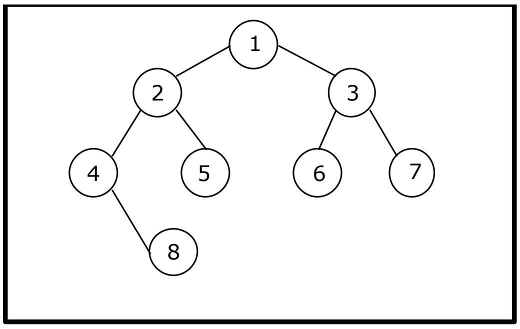


Depth First Spanning Tree

The spanning trees obtained using depth first searches are called depth first spanning trees. The edges rejected in the context of depth first search are called a back edges. Depth first spanning tree has no cross edges.

6.4.2. Breadth first search and traversal:

Starting at vertex 'V' and marking it as visited, BFS differs from DFS in that all unvisited vertices adjacent to V are visited next. Then unvisited vertices adjacent to there vertices are visited and so on. A breadth first search beginning at vertex 1 of the graph would first visit 1 and then 2 and 3.



Breadth First Spanning Tree

Next vertices 4, 5, 6 and 7 will be visited and finally 8. The spanning trees obtained using BFS are called Breadth first spanning trees. The edges that were rejected in the breadth first search are called cross edges.

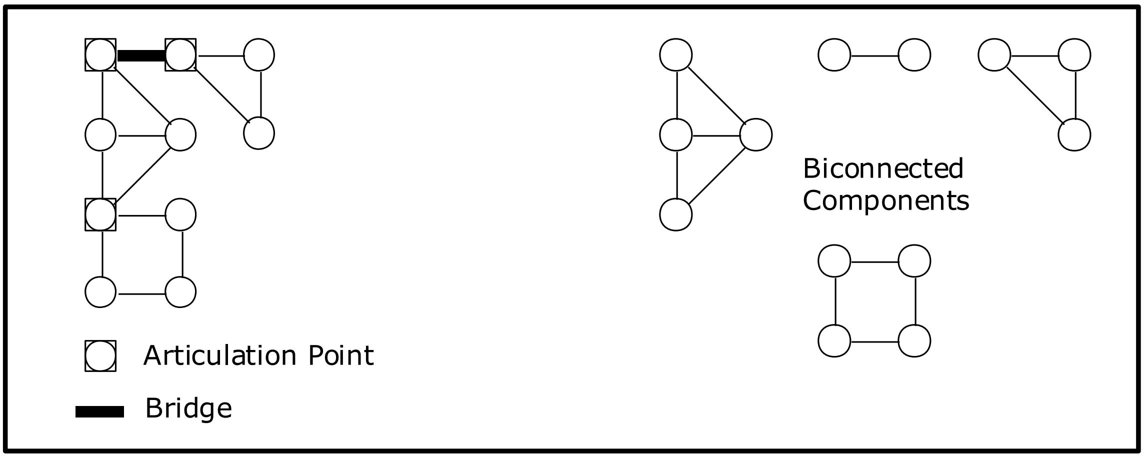
6.5. Articulation Points and Biconnected Components:

Let G = (V, E) be a connected undirected graph. Consider the following definitions:

Articulation Point (or Cut Vertex): An articulation point in a connected graph is a vertex (together with the removal of any incident edges) that, if deleted, would break the graph into two or more pieces..

Bridge: Is an edge whose removal results in a disconnected graph.

Biconnected: A graph is biconnected if it contains no articulation points. In a biconnected graph, two distinct paths connect each pair of vertices. A graph that is not biconnected divides into biconnected components. This is illustrated in the following figure:



Articulation Points and Bridges

B

iconnected graphs and articulation points are of great interest in the design of network algorithms, because these are the “critical" points, whose failure will result in the network becoming disconnected.

Let us consider the typical case of vertex v, where v is not a leaf and v is not the root. Let w_1, w_2, ... w_k be the children of v. For each child there is a subtree of the DFS tree rooted at this child. If for some child, there is no back edge going to a proper ancestor of v, then if we remove v, this subtree becomes disconnected from the rest of the graph, and hence v is an articulation point.

On the other hand, if every one of the subtree rooted at the children of v have back edges to proper ancestors of v, then if v is removed, the graph remains connected (the back edges hold everything together). This leads to the following:

Observation 1: An internal vertex v of the DFS tree (other than the root) is an articulation point if and only if there is a subtree rooted at a child of v such that there is no back edge from any vertex in this subtree to a proper ancestor of v.

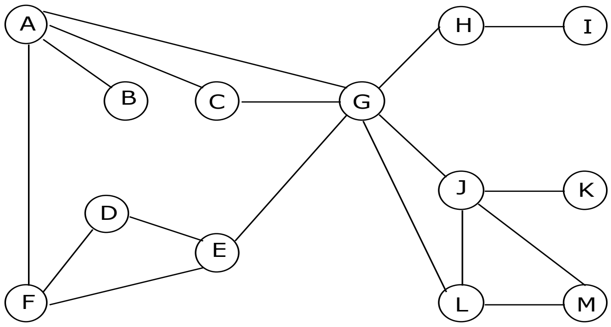
Observation 2: A leaf of the DFS tree is never an articulation point, since a leaf will not have any subtrees in the DFS tree.

Thus, after deletion of a leaf from a tree, the rest of the tree remains connected, thus even ignoring the back edges, the graph is connected after the deletion of a leaf from the DFS tree.

Observation 3: The root of the DFS is an articulation point if and only if it has two or more children. If the root has only a single child, then (as in the case of leaves) its removal does not disconnect the DFS tree, and hence cannot disconnect the graph in general.

6.6. Articulation Points by Depth First Search:

Determining the articulation turns out to be a simple extension of depth first search. Consider a depth first spanning tree for this graph.

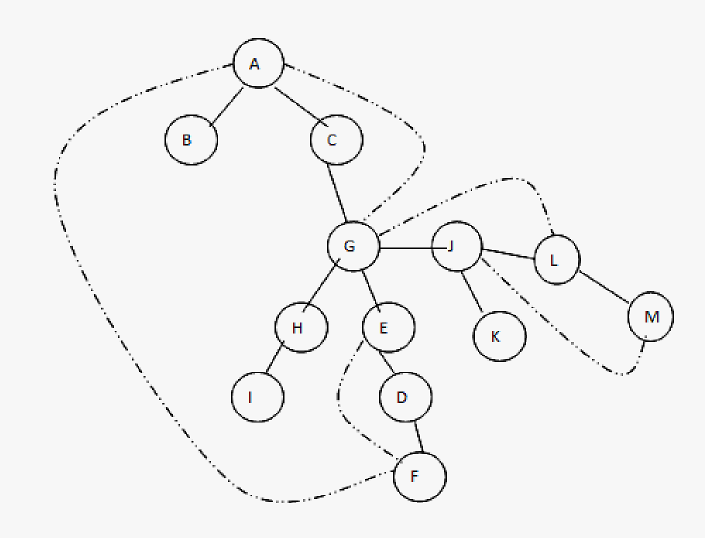


Observations 1, 2, and 3 provide us with a structural characterization of which vertices in the DFS tree are articulation points.

Deleting node E does not disconnect the graph because G and D both have dotted links (back edges) that point above E, giving alternate paths from them to F. On the other hand, deleting G does disconnect the graph because there are no such alternate paths from L or H to E (G's parent).

A vertex 'x' is not an articulation point if every child 'y' has some node lower in the tree connect (via a dotted link) to a node higher in the tree than 'x', thus providing an alternate connection from 'x' to 'y'. This rule will not work for the root node since there are no *nodes higher in the tree*. The root is an articulation point if it has two or more children.

Depth First Spanning Tree for the above graph is:



By using the above observations the articulation points of this graph are:

A : because it connects B to the rest of the graph.
H : because it connects I to the rest of the graph.
J : because it connects K to the rest of the graph.
G : because the graph would fall into three pieces if G is deleted.

Biconnected components are: {A, C, G, D, E, F}, {G, J, L, M}, B, H, I and K
This observation leads to a simple rule to identify articulation points. For each is define L (u) as follows:

$$L(u) = \min \{DFN(u), \min \{L(w) \mid w \text{ is a child of } u\}, \min \{DFN(w) \mid (u, w) \text{ is a back edge}\}\}.$$

L (u) is the lowest depth first number that can be reached from 'u' using a path of descendents followed by at most one back edge. It follows that, If 'u' is not the root then 'u' is an articulation point iff 'u' has a child 'w' such that:

L (w) ≥ DFN (u)

6.6.2. Algorithm for finding the Articulation points:

Pseudocode to compute DFN and L.

Algorithm Art (u, v)

```
// u is a start vertex for depth first search. V is its parent if any  in the depth first
// spanning tree. It is assumed that the global array dfn is initialized to zero and that // the global variable num is initialized to 1. n is the number of vertices in G.
{
    dfn [u] := num; L [u] := num; num := num + 1;
    for each vertex w adjacent from u do
    {
        if (dfn [w] = 0) then
        {
            Art (w, u);                         // w is unvisited.
            L [u] := min (L [u], L [w]);
        }
    else if (w ≠ v) then L [u] := min (L [u], dfn [w]);
    }
}
```

6.6.1. Algorithm for finding the Biconnected Components:

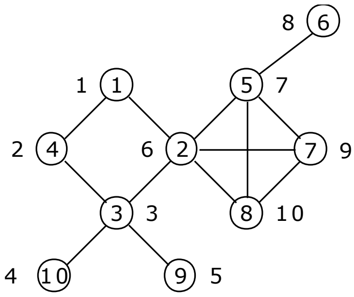
Algorithm BiComp (u, v)

```
// u is a start vertex for depth first search. V is its parent if any in the depth first
// spanning tree. It is assumed that the global array dfn is initially zero and that the
// global variable num is initialized to 1. n is the number of vertices in G.
{
    dfn [u] := num; L [u] := num; num := num + 1;
    for each vertex w adjacent from u do
    {
        if ((v ≠ w) and (dfn [w] ≤ dfn [u])) then
            add (u, w) to the top of a stack s;
        if (dfn [w] = 0) then
        {
            if (L [w] ≥ dfn [u]) then
            {
                write ("New bicomponent");
                repeat
                {
                    Delete an edge from the top of stack s;
                    Let this edge be (x, y);
                    Write (x, y);
                } until (((x, y) = (u, w)) or ((x, y) = (w, u)));
            }
```

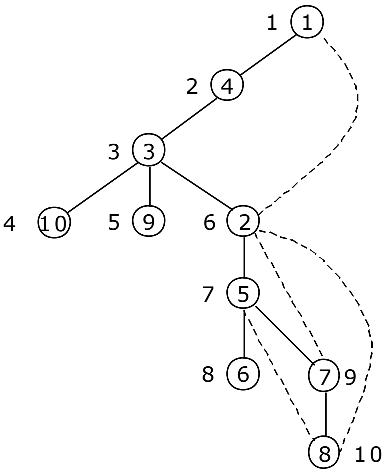
```
                        BiComp (w, u);                          // w is unvisited.
                        L [u] := min (L [u], L [w]);
                }
                else if (w ≠ v) then L [u] : = min (L [u], dfn [w]);
        }
}
```

6.7.1. Example:

For the following graph identify the articulation points and Biconnected components:



Graph



Depth First Spanning Tree

To identify the articulation points, we use:

L (u) = min {DFN (u), min {L (w) | w is a child of u}, min {DFN (w) | w is a vertex to which there is back edge from u}}

L (1) = min {DFN (1), min {L (4)}} = min {1, L (4)} = min {1, 1} = 1

L (4) = min {DFN (4), min {L (3)}} = min {2, L (3)} = min {2, 1} = 1

L (3) = min {DFN (3), min {L (10), L (9), L (2)}} =
= min {3, min {L (10), L (9), L (2)}} = min {3, min {4, 5, 1}} = 1

L (10) = min {DFN (10)} = 4

L (9) = min {DFN (9)} = 5

L (2) = min {DFN (2), min {L (5)}, min {DFN (1)}}
= min {6, min {L (5)}, 1} = min {6, 6, 1} = 1

L (5) = min {DFN (5), min {L (6), L (7)}} = min {7, 8, 6} = 6

L (6) = min {DFN (6)} = 8
L (7) = min {DFN (7), min {L (8}, min {DFN (2)}}
= min {9, L (8) , 6} = min {9, 6, 6} = 6

L (8) = min {DFN (8), min {DFN (5), DFN (2)}}

= min {10, min (7, 6)} = min {10, 6} = 6

Therefore, L (1: 10) = (1, 1, 1, 1, 6, 8, 6, 6, 5, 4)

Finding the Articulation Points:

Vertex 1: Vertex 1 is not an articulation point. It is a root node. Root is an articulation point if it has two or more child nodes.

Vertex 2: is an articulation point as child 5 has L (5) = 6 and DFN (2) = 6,
So, the condition L (5) = DFN (2) is true.

Vertex 3: is an articulation point as child 10 has L (10) = 4 and DFN (3) = 3,
So, the condition L (10) > DFN (3) is true.

Vertex 4: is not an articulation point as child 3 has L (3) = 1 and DFN (4) = 2,
So, the condition L (3) ≥ DFN (4) is false.

Vertex 5: is an articulation point as child 6 has L (6) = 8, and DFN (5) = 7,
So, the condition L (6) > DFN (5) is true.

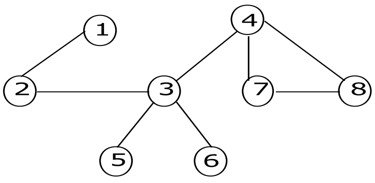
Vertex 7: is not an articulation point as child 8 has L (8) = 6, and DFN (7) = 9,
So, the condition L (8) ≥ DFN (7) is false.

Vertex 6, Vertex 8, Vertex 9 and Vertex 10 are leaf nodes.

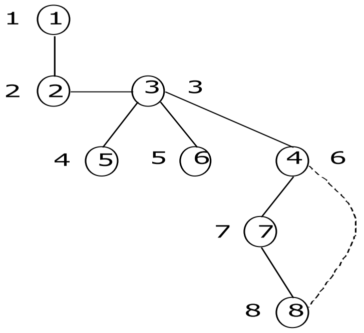
Therefore, the articulation points are {2, 3, 5}.

6.7.2. Example:

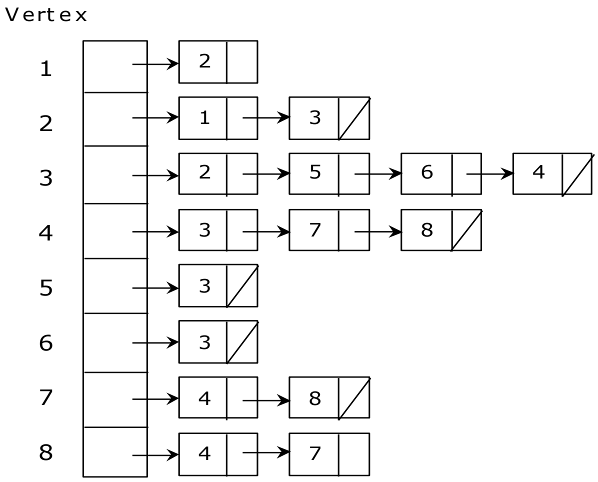
For the following graph identify the articulation points and Biconnected components:



Graph



DFS spanning Tree



Adjacency List

L (u) = min {DFN (u), min {L (w) | w is a child of u}, min {DFN (w) | w is a vertex to which there is back edge from u}}

L (1) = min {DFN (1), min {L (2)}} = min {1, L (2)} = min {1, 2} = 1

L (2) = min {DFN (2), min {L (3)}} = min {2, L (3)} = min {2, 3} = 2

L (3) = min {DFN (3), min {L (4), L (5), L (6)}} = min {3, min {6, 4, 5}} = 3

L (4) = min {DFN (4), min {L (7)} = min {6, L (7)} = min {6, 6} = 6

L (5) = min {DFN (5)} = 4

L (6) = min {DFN (6)} = 5

L (7) = min {DFN (7), min {L (8)}} = min {7, 6} = 6

L (8) = min {DFN (8), min {DFN (4)}} = min {8, 6} = 6

Therefore, L (1: 8) = {1, 2, 3, 6, 4, 5, 6, 6}

Finding the Articulation Points:

Check for the condition if L (w) ≥ DFN (u) is true, where w is any child of u.

Vertex 1: Vertex 1 is not an articulation point.

It is a root node. Root is an articulation point if it has two or more child nodes.

Vertex 2: is an articulation point as L (3) = 3 and DFN (2) = 2.
So, the condition is true

Vertex 3: is an articulation Point as:

I. L (5) = 4 and DFN (3) = 3
II. L (6) = 5 and DFN (3) = 3 and
III. L (4) = 6 and DFN (3) = 3

So, the condition true in above cases

Vertex 4: is an articulation point as L (7) = 6 and DFN (4) = 6.
So, the condition is true

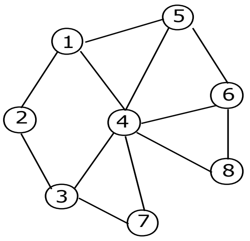
Vertex 7: is not an articulation point as L (8) = 6 and DFN (7) = 7.
So, the condition is False

Vertex 5, Vertex 6 and Vertex 8 are leaf nodes.

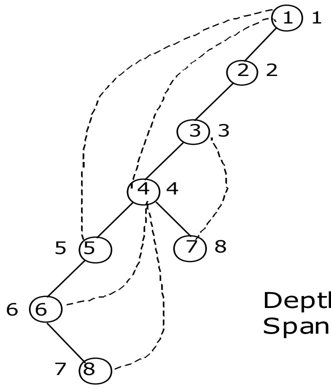
Therefore, the articulation points are {2, 3, 4}.

6.7.3. Example:

For the following graph identify the articulation points and Biconnected components:

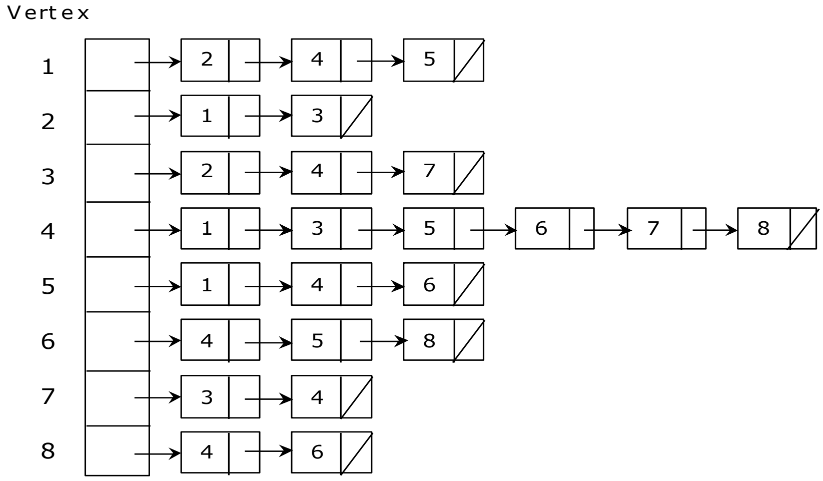


Graph



Depth First Spanning Tree

DFN (1: 8) = {1, 2, 3, 4, 5, 6, 8, 7}



Adjacency List

L (u) = min {DFN (u), min {L (w) | w is a child of u}, min {DFN (w) | w is a vertex to which there is back edge from u}}

L (1) = min {DFN (1), min {L (2)}}
= min {1, L (2)} = 1

L (2) = min {DFN (2), min {L (3)}} = min {2, L (3)} = min{2, 1}= 11

L (3) = min {DFN (3), min {L (4)}} = min {3, L (4)} = min {3, L (4)}
= min {3, 1} = 1

L (4) = min {DFN (4), min {L (5), L (7)}, min {DFN (1)}}
= min {4, min {L (5), L (7)}, 1} = min {4, min {1, 3}, 1}
= min {4, 1, 1} = 1

L (5) = min {DFN (5), min {L (6)}, min {DFN (1)}} = min {5, L (6), 1}
= min {5, 4, 1} = 1

L (6) = min {DFN (6), min {L (8)}, min {DFN (4)}} = min(6, L (8), 4}
= min(6, 4, 4} = 4

L (7) = min {DFN (7), min {DFN (3)}} = min {8, 3} = 3

L (8) = min {DFN (8), min {DFN (4)}} = min {7, 4} = 4

Therefore, L (1: 8) = {1, 1, 1, 1, 1, 4, 3, 4}

Finding the Articulation Points:

Check for the condition if L (w) $\geq$ DFN (u) is true, where w is any child of u.

Vertex 1: is not an articulation point.

It is a root node. Root is an articulation point if it has two or more child nodes.

Vertex 2: is not an articulation point. As L (3) = 1 and DFN (2) = 2.
So, the condition is False.

Vertex 3: is not an articulation Point as L (4) = 1 and DFN (3) = 3.
So, the condition is False.

Vertex 4: is not an articulation Point as:
L (3) = 1 and DFN (2) = 2 and
L (7) = 3 and DFN (4) = 4
So, the condition fails in both cases.

Vertex 5: is not an Articulation Point as L (6) = 4 and DFN (5) = 6.
So, the condition is False

Vertex 6: is not an Articulation Point as L (8) = 4 and DFN (6) = 7.
So, the condition is False

Vertex 7: is a leaf node.

Vertex 8: is a leaf node.

So they are no articulation points.

6.8. GAME PLAYING

In game-playing literature, the term play is used for a move. The two major components of game-playing program are plausible move generator, and a static evaluation function generator.
In a game of chess, for every move a player makes, the average branching factor is 35, that is the opponent can make 35 different moves. It may not be possible to examine all the states because the amount of time given for a move is limited and the amount of computational power available for examining various states is also limited. Hence it is essential that only very selected moves or paths are examined. For this purpose only, one has a plausible move generator, that expands or generates only selected moves.

Static Evaluation function generator (SEF) is the most important component of the game-playing program. The process of computing a number that reflects board quality is called static evaluation. The procedure that does the computation is called a static evaluator. A static evaluation function, is defined as the one that estimates the value of the board position without looking any of that position's successors.

The SEF gives a snapshot of a particular move. More the SEF value, more is the probability for a victory. Games can be classified as either a single person playing or multi-person playing. For single person games (for example, Rubik's cube, 8-tile puzzle etc.) search strategies such as Best-first branch and bound algorithm can be used.

On the other hand, in a two-person games like chess, checkers etc., each player tries to outsmart the opponent. Each has their own way of evaluating the situation and since each player tries to obtain the maximum benefits, best first search branch and bound algorithm do not serve the purpose. The basic methods available for game playing are:

1. Minimax search.
2. Minimax search with alpha beta cutoffs.

6.8.1. MINIMAX SEARCH

The standard algorithm for two-player games is called minimax search with static evaluation. We have to add more knowledge to improve search and to reduce the complexity. Heuristic search adds a small amount of knowledge to a problem space, giving, surprisingly, a fairly dramatic effect of the efficiency of search algorithm. A good heuristic is to select roads that go in the direction of the goal. In this case we call it as heuristic static evaluation function that takes a board position and returns a number that indicates how favorable that position is to one player or other. We call our players as **MAX** and **MIN**.

The large positive values would correspond to strong positions for one player whereas large negative values would represent advantageous situations for the opponent.

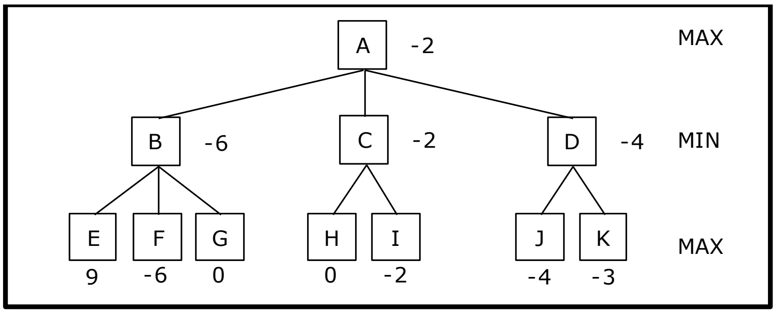
The player for whom large positive values are advantageous is called the MAX, and conversely the opponent is referred to as MIN. The value of a node where it is MAX's turn to move is maximum of the values of its children, while the value of a node where MIN is to move is the minimum of the values of its children.

MAX represents the player trying to win, or to MAXimize his/her advantage. MIN, the opponent attempts to MINimize MAX's score. (i.e. we can assume that MIN uses the same information and always attempts to move to a state that is worst for MAX).

At alternate levels of tree, the minimum and the maximum values of the children are backedup. The backingup values is done as follows: The (MAX node) parent of MIN tip nodes is assigned a backedup value equal to the maximum of the evaluations of the tip nodes, on the other hand, if MIN were to choose among tip nodes, he will choose that having the smallest evaluation (the most negative). Therefore, the (MIN node), parent of

MAX tip nodes is assigned a backedup value equal to the minimum of the evaluation of the tip nodes.
After the parents of all tip nodes have been assigned backedup values, we backup values another level, assuming that MAX would choose that node with largest backedup value, while MIN would choose that node with smallest backedup value.

We continue the procedure level by level, until finally, the successors of the start node are assigned backedup values. The search algorithm then uses these derived values to select among possible next moves.

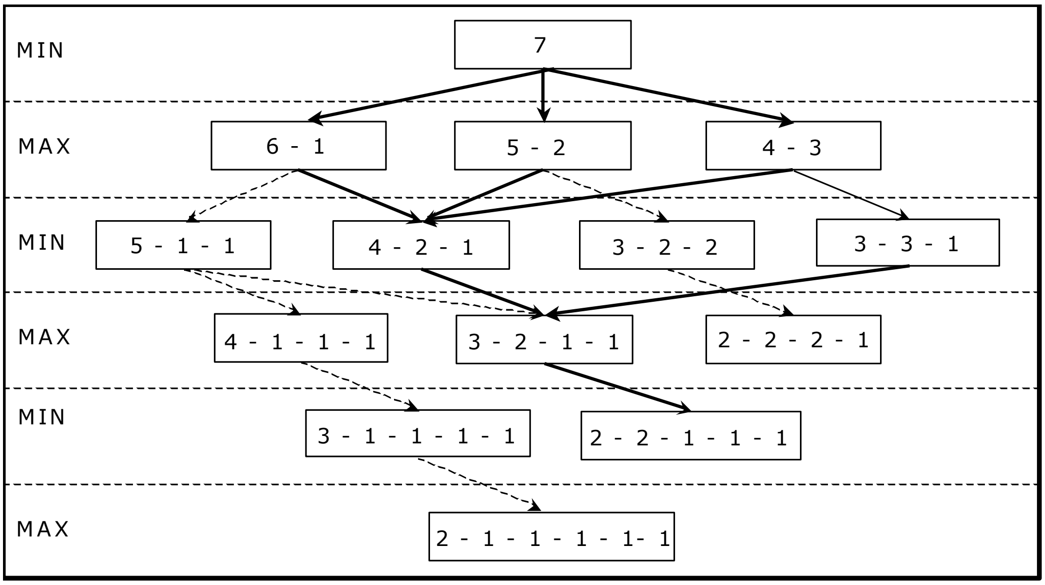


Backingup the values of a 2 - ply search

The best first move can be extracted from minimax procedure after search terminates.

6.8.2. EXAMPLE

To illustrate these ideas, let us consider a simple game called "Grundy's game". The rules of the game are as follows: Two players have in front of them a single pile of 7 matches. At each move the player must divide a pile of matches into two non-empty piles with different numbers of matches in each pile. Thus, 6 matches may be divided into piles of 5 and 1 or 4 and 2, but not 3 and 3. The first player who can no longer make a move loses the game. The following figure shows the space for a game with 7 matches and let MIN play first:



A GAME GRAPH FOR GRUNDYS GAME

We can also use the game tree to show that, no matter what min does max can always win. A winning strategy for max is shown by heavy lines. Max, can always force the game to a win, regardless of min's first move. Min could win only if max played foolishly.

6.9. ALPHA-BETA PRUNING

The minimax pursues all branches in the space, including many that can be ignored or pruned by a more intelligent algorithm. Researchers in game playing have developed a class of search technique called Alpha-beta pruning to improve the efficiency of search in two-person games.

The idea for alpha-beta search is simple: Two values called **alpha** and **beta** are created during the search. The alpha value, associated with MAX nodes, can never decrease, and the beta value, associated with MIN nodes, can never increase.

Suppose for example, MAX node's alpha value is 6, then MAX need not consider any backedup value less than or equal to 6 that is associated with any MIN node below it. Similarly, if MIN has a beta value of 6, it need not consider further any MAX node below it that has a value of 6 or more.

Because of these constraints, we can state the following two rules for terminating the search:

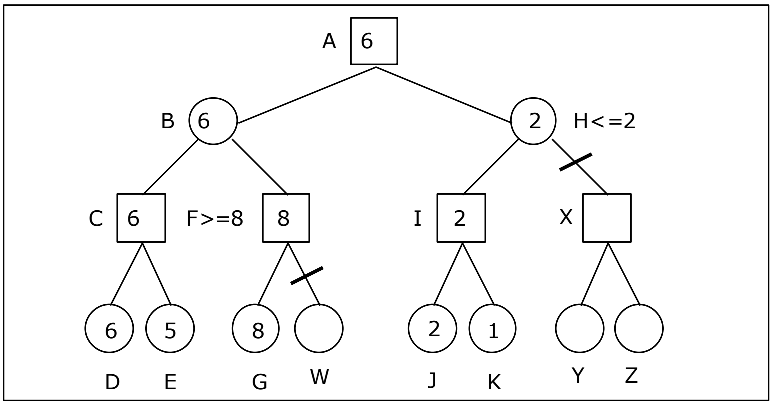
1. Search can be stopped below any MIN node having a beta value less than or equal to the alpha value of any of its MAX node ancestors. The final backedup value of this MIN node can then the set to its beta value.

2. Search can be stopped below any MAX node having an alpha value greater than or equal to the beta value of any of its MIN node ancestors. The final backedup value of this MAX node can then be set to its alpha value.

Figure given below shows an example of alpha-beta pruning. The search proceeds depth-first to minimise the memory requirement, and only evaluates a node when necessary. After statically evaluating nodes D and E to 6 and 5, respectively, we back up their maximum value, 6 as the value of node C. After statically evaluating node G as 8, we know that the backed up value of node F must be greater than or equal to 8, since it is the maximum of 8 and the unknown value node W. The value of node B must be 6 then, because it is the maximum of 6 and a value that must be greater than or equal to 8. Since we have exactly determined the value of node B, we do not need to evaluate or even generate node W. This is called an **ALPHA CUTOFF**.

Similarly, after statically evaluating nodes J and K to 2 and 1, the backed up value is their maximum or 2. This tells that the backed up value of node H must be less than or equal to 2, since it the minimum of 2 and the unknown value of node X. Since the value of node A is the maximum of 6 and a value that must be less than or equal to 2, it must be 6, and hence we have evaluated the root of the tree without generating or evaluating the nodes X, Y or Z. This is called **BETA CUTOFF**.

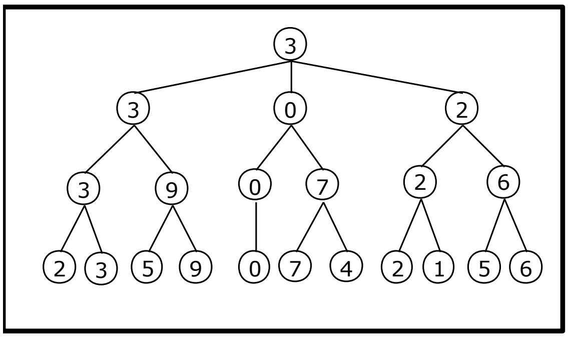
The whole process of keeping track of alpha and beta values and making cutoff's when possible is called as ***alpha-beta procedure***.



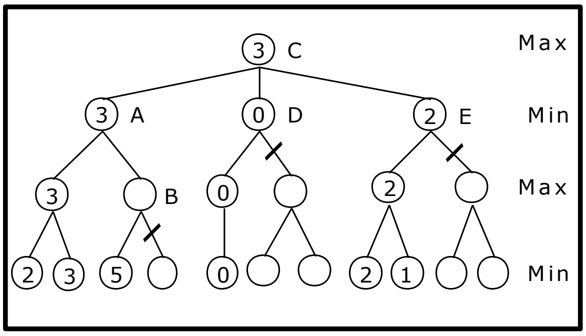
Alpha - Beta Pruning

6.9.1. EXAMPLE 2:

Taking the space of figure as shown below and when alpha-beta pruning applied is on this problem, is as follows:



Hypothetical state space to minimax



Alpha - Beta Pruning

A has β = 3 (A will be no larger than 3).
B is β pruned, since 5 > 3.
C has α = 3 (C will be no smaller than 3).
D is α pruned, since 0 < 3.
E is α pruned, since 2 < 3.
C is 3.

6.10. AND/OR GRAPH:

And/or graph is a specialization of hypergraph which connects nodes by sets of arcs rather than by a single arcs. A hypergraph is defined as follows:

A hypergraph consists of:

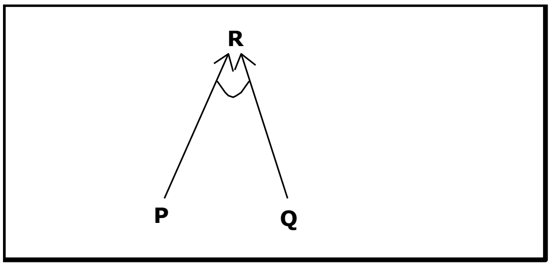
N, a set of nodes,

H, a set of hyperarcs defined by ordered pairs, in which the first implement of the pair is a node of N and the second implement is the subset of N.

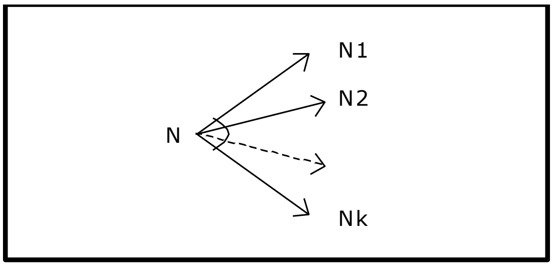
An ordinary graph is a special case of hypergraph in which all the sets of decendent nodes have a cardinality of 1.

Hyperarcs also known as K-connectors, where K is the cardinality of the set of decendent nodes. If K = 1, the descendent may be thought of as an OR nodes. If K > 1, the elements of the set of decendents may be thought of as AND nodes. In this case the connector is drawn with individual edges from the parent node to each of the decendent nodes; these individual edges are then joined with a curved link.

And/or graph for the expression P and Q -> R is follows:



Expression for P and Q -> R



A K-Connector

The K-connector is represented as a fan of arrows with a single tie is shown above.

The and/or graphs consists of nodes labelled by global databases. Nodes labelled by compound databases have sets of successor nodes. These successor nodes are called AND nodes, in order to process the compound database to termination, all the compound databases must be processed to termination.

For example consider, consider a boy who collects stamps (M). He has for the purpose of exchange a winning conker (C), a bat (B) and a small toy animal (A). In his class there are friends who are also keen collectors of different items and will make the following exchanges.

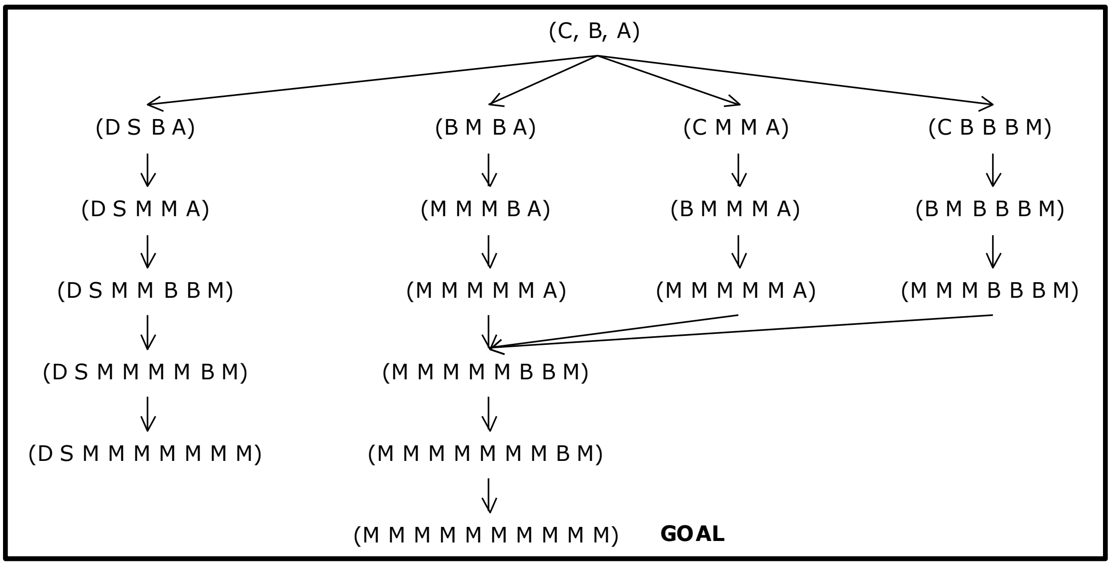
1. 1 winning conker (C) for a comic (D) and a bag of sweets (S).
2. 1 winning conker (C) for a bat (B) and a stamp (M).
3. 1 bat (B) for two stamps (M, M).
4. 1 small toy animal (A) for two bats (B, B) and a stamp (M).

The problem is how to carry out the exchanges so that all his exchangable items are converted into stamps (M). This task can be expressed more briefly as:

1. Initial state = (C, B, A)

2. Transformation rules:
    a. If C then (D, S)
    b. If C then (B, M)
    c. If B then (M, M)
    d. If A then (B, B, M)

3. The goal state is to left with only stamps (M, ... , M)



Expansion for the exchange problem using OR connectors only

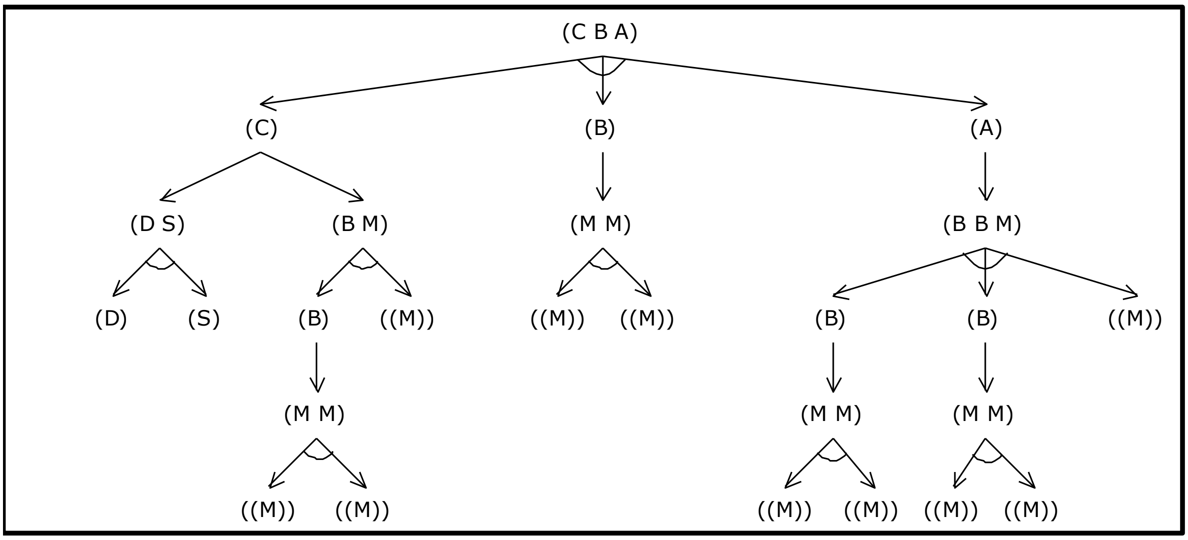
The figure shows that, a lot of extra work is done by redoing many of the transformations. This repetition can be avoided by decomposing the problem into subproblems. There are two major ways to order the components:

1. The components can either be arranged in some fixed order at the time they are generated (or).
2. They can be dynamically reordered during processing.

The more flexible system is to reorder dynamically as the processing unfolds. It can be represented by and/or graph. The solution to the exchange problem will be:

> Swap conker for a bat and a stamp, then exchange this bat for two stamps. Swap his own bat for two more stamps, and finally swap the small toy animal for two bats and a stamp. The two bats can be exchanged for two stamps.

The previous exchange problem, when implemented as an and/or graph looks as follows:

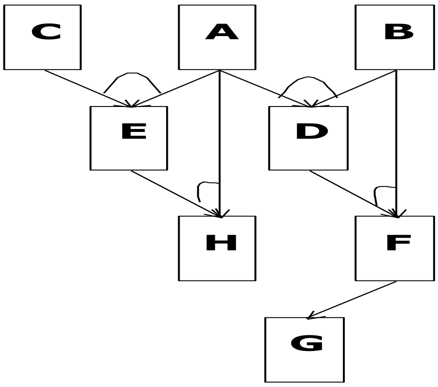


The exchange problem as an AND/OR graph

6.10.1. Example 1:

Draw an And/Or graph for the following prepositions:

1. A
2. B
3. C
4. A ^ B -> D
5. A ^ C -> E
6. B ^ D -> F
7. F -> G
8. A ^ E -> H



Chapter 7

BACKTRACKING

7.1. General Method:

Backtracking is used to solve problem in which a sequence of objects is chosen from a specified set so that the sequence satisfies some criterion. The desired solution is expressed as an n-tuple (x1, ... , x_n) where each x_i Є S, S being a finite set.

The solution is based on finding one or more vectors that maximize, minimize, or satisfy a criterion function P (x_1, ... , x_n). Form a solution and check at every step if this has any chance of success. If the solution at any point seems not promising, ignore it. All solutions requires a set of constraints divided into two categories: explicit and implicit constraints.

Definition 1: Explicit constraints are rules that restrict each x_i to take on values only from a given set. Explicit constraints depend on the particular instance I of problem being solved. All tuples that satisfy the explicit constraints define a possible solution space for I.

Definition 2: Implicit constraints are rules that determine which of the tuples in the solution space of I satisfy the criterion function. Thus, implicit constraints describe the way in which the x_i's must relate to each other.

- For 8-queens problem:

Explicit constraints using 8-tuple formation, for this problem are S= {1, 2, 3, 4, 5, 6, 7, 8}.

The implicit constraints for this problem are that no two queens can be the same (i.e., all queens must be on different columns) and no two queens can be on the same diagonal.

Backtracking is a modified depth first search of a tree. Backtracking algorithms determine problem solutions by systematically searching the solution space for the given problem instance. This search is facilitated by using a tree organization for the solution space.

Backtracking is the procedure where by, after determining that a node can lead to nothing but dead end, we go back (backtrack) to the nodes parent and proceed with the search on the next child.

A backtracking algorithm need not actually create a tree. Rather, it only needs to keep track of the values in the current branch being investigated. This is the way we implement backtracking algorithm. We say that the state space tree exists implicitly in the algorithm because it is not actually constructed.

7.2. Terminology:

Problem state is each node in the depth first search tree.

Solution states are the problem states 'S' for which the path from the root node to 'S' defines a tuple in the solution space.

Answer states are those solution states for which the path from root node to s defines a tuple that is a member of the set of solutions.

State space is the set of paths from root node to other nodes. *State space* tree is the tree organization of the solution space. The state space trees are called static trees. This terminology follows from the observation that the tree organizations are independent of the problem instance being solved. For some problems it is advantageous to use different tree organizations for different problem instance. In this case the tree organization is determined dynamically as the solution space is being searched. Tree organizations that are problem instance dependent are called dynamic trees.

Live node is a node that has been generated but whose children have not yet been generated.

E-node is a live node whose children are currently being explored. In other words, an E-node is a node currently being expanded.

Dead node is a generated node that is not to be expanded or explored any further. All children of a dead node have already been expanded.

Branch and Bound refers to all state space search methods in which all children of an E-node are generated before any other live node can become the E-node.

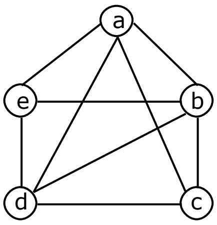
Depth first node generation with bounding functions is called ***backtracking***. State generation methods in which the E-node remains the E-node until it is dead, lead to branch and bound methods.

Planar Graphs:

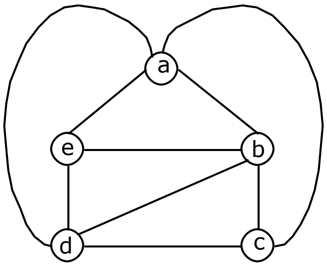
When drawing a graph on a piece of a paper, we often find it convenient to permit edges to intersect at points other than at vertices of the graph. These points of interactions are called crossovers.

A graph G is said to be planar if it can be drawn on a plane without any crossovers; otherwise G is said to be non-planar i.e., A graph is said to be planar iff it can be drawn in a plane in such a way that no two edges cross each other.

Example:



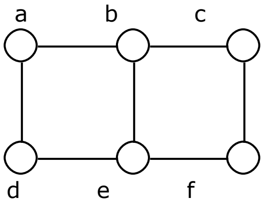
the following graph can be redrawn without crossovers as follows:

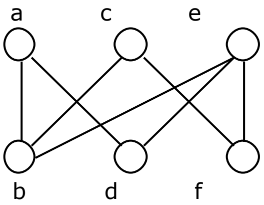


Bipartite Graph:

A bipartite graph is a non-directed graph whose set of vertices can be portioned into two sets V_1 and V_2 (i.e. $V_1 \cup V_2 = V$ and $V_1 \cap V_2 = ø$) so that every edge has one end in V_1 and the other in V_2. That is, vertices in V_1 are only adjacent to those in V_2 and vice-versa.

Example:





The graph is bipartite. We can redraw it as

The vertex set V = {a, b, c, d, e, f} has been partitioned into V_1 = {a, c, e} and V_2 = {b, d, f}. The complete bipartite graph for which V_1 = n and V_2 = m is denoted $K_{n,m}$.

7.3. N-Queens Problem:

Let us consider, N = 8. Then 8-Queens Problem is to place eight queens on an 8 x 8 chessboard so that no two "attack", that is, no two of them are on the same row, column, or diagonal.

All solutions to the 8-queens problem can be represented as 8-tuples (x_1, ... , x_8), where x_i is the column of the i^{th} row where the i^{th} queen is placed.

The explicit constraints using this formulation are S_i = {1, 2, 3, 4, 5, 6, 7, 8}, $1 \leq i \leq 8$. Therefore the solution space consists of 8^8 8-tuples.

The implicit constraints for this problem are that no two x_i's can be the same (i.e., all queens must be on different columns) and no two queens can be on the same diagonal.

This realization reduces the size of the solution space from 8^8 tuples to 8! Tuples.

The promising function must check whether two queens are in the same column or diagonal:

Suppose two queens are placed at positions (i, j) and (k, l) Then:

- Column Conflicts: Two queens conflict if their x_i values are identical.
- Diag 45 conflict: Two queens i and j are on the same 45^0 diagonal if:

i – j = k – l.

This implies, $j - l = i - k$

- Diag 135 conflict:

$i + j = k + l$.

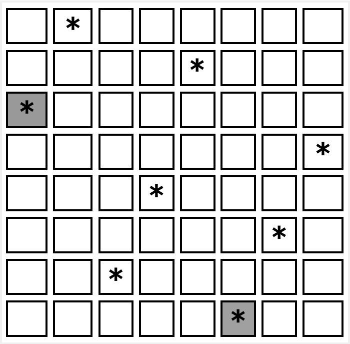
This implies, $j - l = k - i$

Therefore, two queens lie on the same diagonal if and only if:

$$|j - l| = |i - k|$$

Where, j be the column of object in row i for the ith queen and l be the column of object in row 'k' for the kth queen.

To check the diagonal clashes, let us take the following tile configuration:



In this example, we have:

i	1	2	3	4	5	6	7	8
x_i	2	5	1	8	4	7	3	6

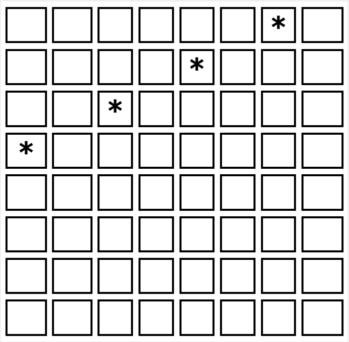
Let us consider for the case whether the queens on 3rd row and 8th row are conflicting or not. In this case (i, j) = (3, 1) and (k, l) = (8, 6). Therefore:

$$|j - l| = |i - k| \Rightarrow |1 - 6| = |3 - 8|$$
$$\Rightarrow 5 = 5$$

In the above example we have, $|j - l| = |i - k|$, so the two queens are attacking. This is not a solution.

Example:

Suppose we start with the feasible sequence 7, 5, 3, 1.



Step 1:

Add to the sequence the next number in the sequence 1, 2, . . . , 8 not yet used.

Step 2:

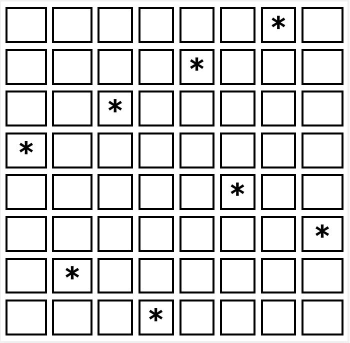
If this new sequence is feasible and has length 8 then STOP with a solution. If the new sequence is feasible and has length less then 8, repeat Step 1.

Step 3:
If the sequence is not feasible, then *backtrack* through the sequence until we find the *most recent* place at which we can exchange a value. Go back to Step 1.

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**Remarks**
7	5	3	1					
7	5	3	1*	2*				$\lvert j - l \rvert = \lvert 1 - 2 \rvert = 1$ $\lvert i - k \rvert = \lvert 4 - 5 \rvert = 1$
7	5	3	1	4				
7*	5	3	1	4	2*			$\lvert j - l \rvert = \lvert 7 - 2 \rvert = 5$ $\lvert i - k \rvert = \lvert 1 - 6 \rvert = 5$
7	5	3*	1	4	6*			$\lvert j - l \rvert = \lvert 3 - 6 \rvert = 3$ $\lvert i - k \rvert = \lvert 3 - 6 \rvert = 3$
7	5	3	1	4	8			
7	5	3	1	4*	8	2*		$\lvert j - l \rvert = \lvert 4 - 2 \rvert = 2$ $\lvert i - k \rvert = \lvert 5 - 7 \rvert = 2$
7	5	3	1	4*	8	6*		$\lvert j - l \rvert = \lvert 4 - 6 \rvert = 2$ $\lvert i - k \rvert = \lvert 5 - 7 \rvert = 2$
7	5	3	1	4	8			*Backtrack*
7	5	3	1	4				*Backtrack*
7	5	3	1	6				
7*	5	3	1	6	2*			$\lvert j - l \rvert = \lvert 1 - 2 \rvert = 1$ $\lvert i - k \rvert = \lvert 7 - 6 \rvert = 1$
7	5	3	1	6	4			
7	5	3	1	6	4	2		
7	5	3*	1	6	4	2	8*	$\lvert j - l \rvert = \lvert 3 - 8 \rvert = 5$ $\lvert i - k \rvert = \lvert 3 - 8 \rvert = 5$
7	5	3	1	6	4	2		*Backtrack*
7	5	3	1	6	4			*Backtrack*
7	5	3	1	6	8			
7	5	3	1	6	8	2		
7	5	3	1	6	8	2	4	**SOLUTION**

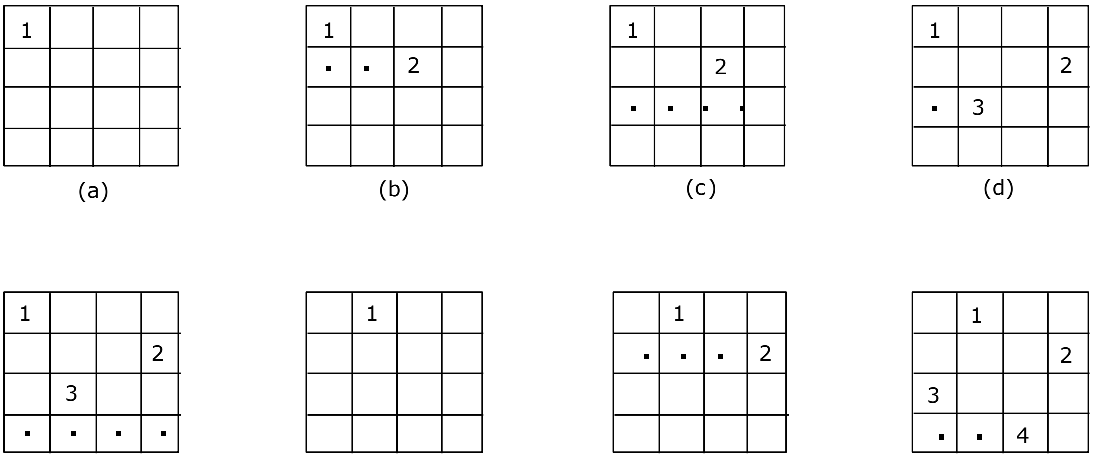
* indicates conflicting queens.

On a chessboard, the **solution** will look like:



4 – Queens Problem:

Let us see how backtracking works on the 4-queens problem. We start with the root node as the only live node. This becomes the E-node. We generate one child. Let us assume that the children are generated in ascending order. Let us assume that the children are generated in ascending order. Thus node number 2 of figure is generated and the path is now (1). This corresponds to placing queen 1 on column 1. Node 2 becomes the E-node. Node 3 is generated and immediately killed. The next node generated is node 8 and the path becomes (1, 3). Node 8 becomes the E-node. However, it gets killed as all its children represent board configurations that cannot lead to an answer node. We backtrack to node 2 and generate another child, node 13. The path is now (1, 4). The board configurations as backtracking proceeds is as follows:

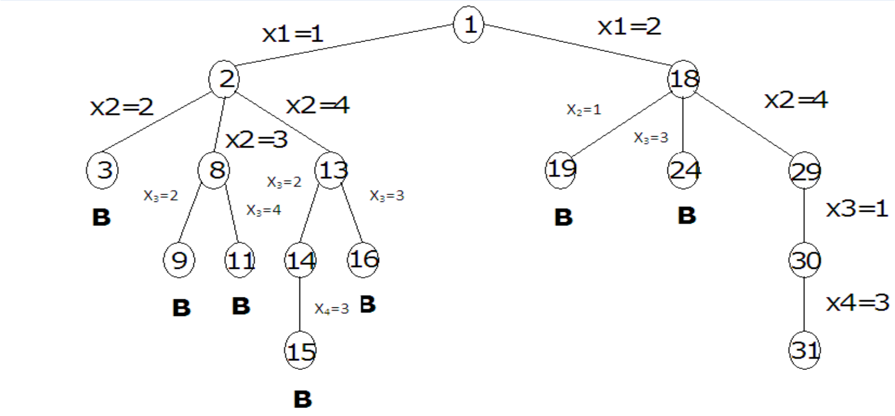


(a) (b) (c) (d)

(e) (f) (g) (h)

The above figure shows graphically the steps that the backtracking algorithm goes through as it tries to find a solution. The dots indicate placements of a queen, which were tried and rejected because another queen was attacking.

In Figure (b) the second queen is placed on columns 1 and 2 and finally settles on column 3. In figure (c) the algorithm tries all four columns and is unable to place the next queen on a square. Backtracking now takes place. In figure (d) the second queen is moved to the next possible column, column 4 and the third queen is placed on column 2. The boards in Figure (e), (f), (g), and (h) show the remaining steps that the algorithm goes through until a solution is found.



Portion of the tree generated during backtracking

Complexity Analysis:

$$1 + n + n^2 + n^3 + \ldots\ldots\ldots\ldots + n^n = \frac{n^{n+1} - 1}{n - 1}$$

For the instance in which n = 8, the state space tree contains:

$$\frac{8^{8+1} - 1}{8 - 1} = 19,\ 173,\ 961 \text{ nodes}$$

Program for N-Queens Problem:

```
# include <stdio.h>
# include <conio.h>
# include <stdlib.h>

int x[10] = {5, 5, 5, 5, 5, 5, 5, 5, 5, 5};

place (int k)
{
                int i;
                for (i=1; i < k; i++)
{
                if ((x [i] == x [k]) || (abs (x [i] – x [k]) == abs (i - k)))
                return (0);
}
return (1);
}
nqueen (int n)
{
                int m, k, i = 0;
                x [1] = 0;
                k = 1;
                while (k > 0)
                {
                        x [k] = x [k] + 1;
                        while ((x [k] <= n) && (!place (k)))
                                x [k] = x [k] +1;
                        if(x [k] <= n)
                        {
                                if (k == n)
                                {
                                        i++;
                                        printf ("\ncombination; %d\n",i);
                                        for (m=1;m<=n; m++)
        printf("row = %3d\t  column=%3d\n", m, x[m]);
                                getch();
                        }
                        else
                        {
                                k++;
                                x [k]=0;
                        }
}

else
                                k--;
}
return (0);
}
main ()
{
int n;
clrscr ();
printf ("enter value for N: ");
scanf ("%d", &n);
nqueen (n);
}
```

Output:

```
Enter the value for N: 4

Combination: 1                          Combination: 2

Row = 1         column = 2                      3
Row = 2         column = 4                      1
Row = 3         column = 1                      4
Row = 4         column = 3                      2
```

For N = 8, there will be 92 combinations.

7.4. Sum of Subsets:

Given positive numbers wi, $1 \leq i \leq n$, and m, this problem requires finding all subsets of w_i whose sums are 'm'.

All solutions are k-tuples, $1 \leq k \leq n$.

Explicit constraints:

- $x_i \in \{j \mid j$ is an integer and $1 \leq j \leq n\}$.

Implicit constraints:

- No two x_i can be the same.

- The sum of the corresponding w_i's be m.

- $x_i < x_{i+1}$, $1 \leq i < k$ (total order in indices) to avoid generating multiple instances of the same subset (for example, (1, 2, 4) and (1, 4, 2) represent the same subset).

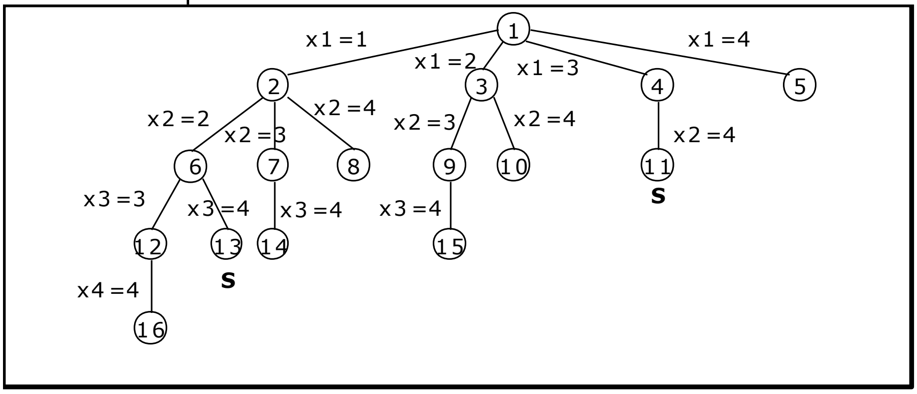
A better formulation of the problem is where the solution subset is represented by an n-tuple (x_1, ... , x_n) such that x_i **Є** {0, 1}.

The above solutions are then represented by (1, 1, 0, 1) and (0, 0, 1, 1).

For both the above formulations, the solution space is 2^n distinct tuples.

For example, n = 4, w = (11, 13, 24, 7) and m = 31, the desired subsets are (11, 13, 7) and (24, 7).

The following figure shows a possible tree organization for two possible formulations of the solution space for the case n = 4.



A possible solution space organisation for the sum of the subsets problem.

The tree corresponds to the variable tuple size formulation. The edges are labeled such that an edge from a level i node to a level i+1 node represents a value for x_i. At each node, the solution space is partitioned into sub - solution spaces. All paths from the root node to any node in the tree define the solution space, since any such path corresponds to a subset satisfying the explicit constraints.

The possible paths are (1), (1, 2), (1, 2, 3), (1, 2, 3, 4), (1, 2, 4), (1, 3, 4), (2), (2, 3), and so on. Thus, the left mot sub-tree defines all subsets containing w_1, the next sub-tree defines all subsets containing w_2 but not w_1, and so on.

7.5. Graph Coloring (for planar graphs):

Let G be a graph and m be a given positive integer. We want to discover whether the nodes of G can be colored in such a way that no two adjacent nodes have the same color, yet only m colors are used. This is termed the m-colorabiltiy decision problem. The m-colorability optimization problem asks for the smallest integer m for which the graph G can be colored.

Given any map, if the regions are to be colored in such a way that no two adjacent regions have the same color, only four colors are needed.

For many years it was known that five colors were sufficient to color any map, but no map that required more than four colors had ever been found. After several hundred years, this problem was solved by a group of mathematicians with the help of a computer. They showed that in fact four colors are sufficient for planar graphs.

The function m-coloring will begin by first assigning the graph to its adjacency matrix, setting the array x [] to zero. The colors are represented by the integers 1, 2, . . . , m and the solutions are given by the n-tuple (x_1, x_2, . . ., x_n), where x_i is the color of node i.

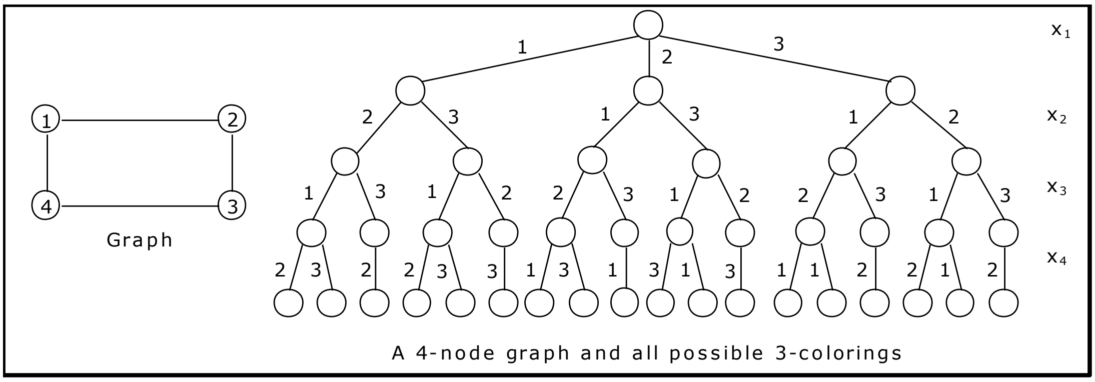
A recursive backtracking algorithm for graph coloring is carried out by invoking the statement mcoloring(1);

```
Algorithm mcoloring (k)
// This algorithm was formed using the recursive backtracking schema. The graph is
// represented by its Boolean adjacency matrix G [1: n, 1: n].  All assignments of
// 1, 2, . . . . . , m to the vertices of the graph such that adjacent vertices are assigned
// distinct integers are printed. k is the index  of the next vertex to color.
{
      repeat
      {                                   // Generate all legal assignments for x[k].
            NextValue (k);          // Assign to x [k] a legal color.
            If (x [k] = 0) then return;    // No new color possible
            If (k = n) then         // at most m colors have been
                                          // used to color the n vertices.
                  write (x [1: n]);
                  else mcoloring (k+1);
            } until (false);
      }

Algorithm NextValue (k)
// x [1] , . . . . x [k-1] have been assigned integer values in the range [1, m] such that
// adjacent vertices have distinct integers. A value for x [k] is determined in the range
// [0, m].x[k] is assigned the next highest numbered color while maintaining distinctness
// from the adjacent vertices of vertex k. If no such color exists, then x [k] is 0.
{
repeat
{
x [k]: = (x [k] +1) mod (m+1)             // Next highest color.
If (x [k] = 0) then return;               // All colors have been used
for j := 1 to n do
{     // check if this color is distinct from adjacent colors
            if ((G [k, j] ≠ 0) and (x [k] = x [j]))
                  // If (k, j) is and edge and if adj. vertices have the same color.
            then break;
}
if (j = n+1) then return;           // New color found
} until (false);                    // Otherwise try to find another color.
}
```

Example:

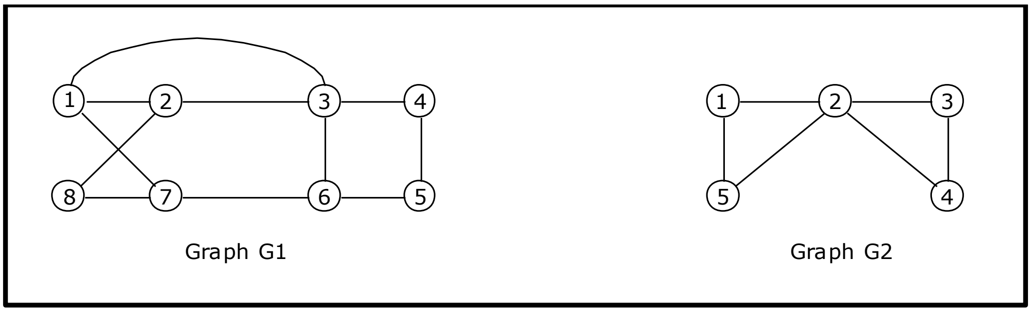
Color the graph given below with minimum number of colors by backtracking using state space tree.



A 4-node graph and all possible 3-colorings

7.6. Hamiltonian Cycles:

Let G = (V, E) be a connected graph with n vertices. A Hamiltonian cycle (suggested by William Hamilton) is a round-trip path along n edges of G that visits every vertex once and returns to its starting position. In other vertices of G are visited in the order v_1, v_2, ... , v_{n+1}, then the edges (v_i, v_{i+1}) are in E, $1 \leq i \leq n$, and the v_i are distinct expect for v_1 and v_{n+1}, which are equal. The graph G_1 contains the Hamiltonian cycle 1, 2, 8, 7, 6, 5, 4, 3, 1. The graph G_2 contains no Hamiltonian cycle.



Two graphs to illustrate Hamiltonian cycle

The backtracking solution vector (x_1, ... x_n) is defined so that x_i represents the i^{th} visited vertex of the proposed cycle. If k = 1, then x_1 can be any of the n vertices. To avoid printing the same cycle n times, we require that $x_1 = 1$. If $1 < k < n$, then x_k can be any vertex v that is distinct from x_1, x_2, . . . , x_{k-1} and v is connected by an edge to k_{x-1}. The vertex x_n can only be one remaining vertex and it must be connected to both x_{n-1} and x_1.

Using NextValue algorithm we can particularize the recursive backtracking schema to find all Hamiltonian cycles. This algorithm is started by first initializing the adjacency matrix G[1: n, 1: n], then setting x[2: n] to zero and x[1] to 1, and then executing Hamiltonian(2).

The traveling salesperson problem using dynamic programming asked for a tour that has minimum cost. This tour is a Hamiltonian cycles. For the simple case of a graph all of whose edge costs are identical, Hamiltonian will find a minimum-cost tour if a tour exists.

```
Algorithm NextValue (k)
// x [1: k-1] is a path of k – 1 distinct vertices . If x[k] = 0, then no vertex has as yet been
// assigned to x [k]. After execution, x[k] is assigned to the next highest numbered vertex
// which does not already appear in x [1 : k – 1] and is connected by an edge to x [k – 1].
// Otherwise x [k] = 0. If k = n, then in addition x [k] is connected to x [1].
{
repeat
{
```

```
          x [k] := (x [k] +1) mod (n+1);              // Next vertex.
          If (x [k] = 0) then return;
          If (G [x [k – 1], x [k]] ≠ 0) then
          {                                           // Is there an edge?
                    for j := 1 to k – 1 do if (x [j] = x [k]) then break;
                                                      // check for distinctness.
If (j = k) then                     // If true, then the vertex is distinct.
                    If ((k < n) or ((k = n) and G [x [n], x [1]] ≠ 0))
then return;
}
} until (false);
}
```

Algorithm Hamiltonian (k)
// This algorithm uses the recursive formulation of backtracking to find all the Hamiltonian
// cycles of a graph. The graph is stored as an adjacency matrix G [1: n, 1: n]. All cycles begin // at node 1.

```
{
          repeat
          {                                                     // Generate values for x [k].
                    NextValue (k);                    //Assign a legal Next value to x [k].
                    if (x [k] = 0) then return;
                    if (k = n) then write (x [1: n]);
                    else Hamiltonian (k + 1)
          } until (false);
}
```

7.7. 0/1 Knapsack:

Given n positive weights w_i, n positive profits p_i, and a positive number m that is the knapsack capacity, the problem calls for choosing a subset of the weights such that:

$$\sum_{1 \leq i \leq n} w_i \ \ x_i \leq m \ \ and \quad \sum_{1 \leq i \leq n} p_i \ \ x_i \ \ is \max imized.$$

The x_i's constitute a zero–one-valued vector.

The solution space for this problem consists of the 2^n distinct ways to assign zero or one values to the x_i's.

Bounding functions are needed to kill some live nodes without expanding them. A good bounding function for this problem is obtained by using an upper bound on the value of the best feasible solution obtainable by expanding the given live node and any of its descendants. If this upper bound is not higher than the value of the best solution determined so far, than that live node can be killed.

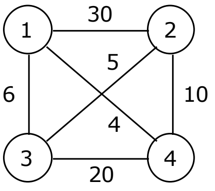
We continue the discussion using the fixed tuple size formulation. If at node Z the values of x_i, $1 \leq i \leq k$, have already been determined, then an upper bound for Z can be obtained by relaxing the requirements $x_i = 0$ or 1.

(Knapsack problem using backtracking is solved in branch and bound chapter)

7.8 Traveling Sale Person (TSP) using Backtracking:

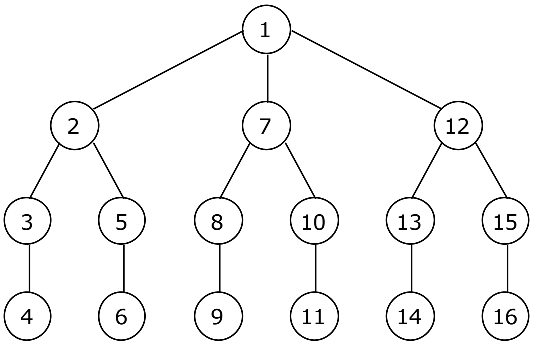
We have solved TSP problem using dynamic programming**.** In this section we shall solve the same problem using backtracking.

Consider the graph shown below with 4 vertices.



A graph for TSP

The solution space tree, similar to the n-queens problem is as follows:



We will assume that the starting node is 1 and the ending node is obviously 1. Then 1, {2, … ,4}, 1 forms a tour with some cost which should be minimum. The vertices shown as {2, 3, …. ,4} forms a permutation of vertices which constitutes a tour. We can also start from any vertex, but the tour should end with the same vertex.

Since, the starting vertex is 1, the tree has a root node R and the remaining nodes are numbered as depth-first order. As per the tree, from node 1, which is the live node, we generate 3 braches node 2, 7 and 12. We simply come down to the left most leaf node 4, which is a valid tour {1, 2, 3, 4, 1} with cost 30 + 5 + 20 + 4 = 59. Currently this is the best tour found so far and we backtrack to node 3 and to 2, because we do not have any children from node 3. When node 2 becomes the E-node, we generate node 5 and then node 6. This forms the tour {1, 2, 4, 3, 1} with cost 30 + 10 + 20 + 6 = 66 and is discarded, as the best tour so far is 59.

Similarly, all the paths from node 1 to every leaf node in the tree is searched in a depth first manner and the best tour is saved. In our example, the tour costs are shown adjacent to each leaf nodes. The optimal tour cost is therefore 25.

Chapter 8

Branch and Bound

8.1. General method:

Branch and Bound is another method to systematically search a solution space. Just like backtracking, we will use bounding functions to avoid generating subtrees that do not contain an answer node. However branch and Bound differs from backtracking in two important manners:

1. It has a branching function, which can be a depth first search, breadth first search or based on bounding function.

2. It has a bounding function, which goes far beyond the feasibility test as a mean to prune efficiently the search tree.

Branch and Bound refers to all state space search methods in which all children of the E-node are generated before any other live node becomes the E-node

Branch and Bound is the generalisation of both graph search strategies, BFS and D-search.

- A BFS like state space search is called as FIFO (First in first out) search as the list of live nodes in a first in first out list (or queue).

- A D search like state space search is called as LIFO (Last in first out) search as the list of live nodes in a last in first out (or stack).

Definition 1: **Live node is a node that has been generated but whose children have not yet been generated.**

Definition 2: E-node is a live node whose children are currently being explored. In other words, an E-node is a node currently being expanded.

Definition 3: **Dead node is a generated node that is not to be expanded or explored any further. All children of a dead node have already been expanded.**

Definition 4: **Branch-an-bound refers to all state space search methods in which all children of an E-node are generated before any other live node can become the E-node.**

Definition 5: **The adjective "heuristic", means" related to improving problem solving performance". As a noun it is also used in regard to "any method or trick used to improve the efficiency of a problem solving problem". But imperfect methods are not necessarily heuristic or vice versa. "A heuristic (heuristic rule, heuristic method) is a rule of thumb, strategy, trick simplification or any other kind of device which drastically limits search for solutions in large problem spaces. Heuristics do not guarantee optimal solutions, they do not guarantee any solution at all. A useful heuristic offers solutions which are good enough most of the time.**

8.2. Least Cost (LC) search:

In both LIFO and FIFO Branch and Bound the selection rule for the next E-node in rigid and blind. The selection rule for the next E-node does not give any preference to a node that has a very good chance of getting the search to an answer node quickly.

The search for an answer node can be speeded by using an "intelligent" ranking function $\hat{c}(\cdot)$ **for live nodes. The next E-node is selected on the basis of this ranking function. The node x is assigned a rank using:**

$$\hat{c}(x) = \mathbf{f(h(x))} + \hat{g}(x)$$

where, $\hat{c}(x)$ **is the cost of x.**

h(x) is the cost of reaching x from the root and f(.) is any non-decreasing function.

$\hat{g}(x)$ **is an estimate of the additional effort needed to reach an answer node from x.**

A search strategy that uses a cost function $\hat{c}(x)$ **= f(h(x) +** $\hat{g}(x)$ **to select the next E-node would always choose for its next E-node a live node with least** $\hat{c}(.)$ **is called a LC–search (Least Cost search)**

BFS and D-search are special cases of LC-search. If $\hat{g}(x)$ **= 0 and f(h(x)) = level of node x, then an LC search generates nodes by levels. This is eventually the same as a BFS. If f(h(x)) = 0 and** $\hat{g}(x) > \hat{g}(y)$ **whenever y is a child of x, then the search is essentially a D-search.**

An LC-search coupled with bounding functions is called an LC-branch and bound search

We associate a cost c(x) with each node x in the state space tree. It is not possible to easily compute the function c(x). So we compute a estimate $\hat{c}(x)$ **of c(x).**

8.3. Control Abstraction for LC-Search:

Let t be a state space tree and c() a cost function for the nodes in t. If x is a node in t, then c(x) is the minimum cost of any answer node in the subtree with root x. Thus, c(t) is the cost of a minimum-cost answer node in t.

A heuristic $\hat{c}(.)$ is used to estimate c(). This heuristic should be easy to compute and generally has the property that if x is either an answer node or a leaf node, then c(x) = $\hat{c}(x)$.

LC-search uses $\hat{c}$ to find an answer node. The algorithm uses two functions Least() and Add() to delete and add a live node from or to the list of live nodes, respectively.

Least() finds a live node with least c(). This node is deleted from the list of live nodes and returned.

Add(x) adds the new live node x to the list of live nodes. The list of live nodes be implemented as a min-heap.

Algorithm LCSearch outputs the path from the answer node it finds to the root node t. This is easy to do if with each node x that becomes live, we associate a field *parent* which gives the parent of node x. When the answer node g is found, the path from g to t can be determined by following a sequence of *parent* values starting from the current E-node (which is the parent of g) and ending at node t.

```
Listnode = record
{
        Listnode * next, *parent; float cost;
}

Algorithm LCSearch(t)
{       //Search t for an answer node
        if *t is an answer node then output *t and return;
        E := t;         //E-node.
        initialize the list of live nodes to be empty;
        repeat
        {
                for each child x of E do
                {
                        if x is an answer node then output the path from x to t and return;
                        Add (x);                        //x is a new live node.
                        (x → parent) := E;              // pointer for path to root
                }
                if there are no more live nodes then
                {
                        write ("No answer node");
                        return;
                }
                E := Least();
        } until (false);
}
```

The root node is the first, E-node. During the execution of LC search, this list contains all live nodes except the E-node. Initially this list should be empty. Examine all the children of the E-node, if one of the children is an answer node, then the algorithm outputs the path from x to t and terminates. If the child of E is not an answer node, then it becomes a live node. It is added to the list of live nodes and its parent field set to E. When all the children of E have been generated, E becomes a dead node. This happens only if none of E's children is an answer node. Continue the search further until no live nodes found. Otherwise, Least(), by definition, correctly chooses the next E-node and the search continues from here.

LC search terminates only when either an answer node is found or the entire state space tree has been generated and searched.

8.4. Bounding:

A branch and bound method searches a state space tree using any search mechanism in which all the children of the E-node are generated before another node becomes the E-node. We assume that each answer node x has a cost c(x) associated with it and that a minimum-cost answer node is to be found. Three common search strategies are FIFO, LIFO, and LC. The three search methods differ only in the selection rule used to obtain the next E-node.

A good bounding helps to prune efficiently the tree, leading to a faster exploration of the solution space.

A cost function $\hat{c}(.)$ such that $\hat{c}(x) \leq c(x)$ is used to provide lower bounds on solutions obtainable from any node x. If upper is an upper bound on the cost of a minimum-cost solution, then all live nodes x with $c(x) \geq \hat{c}(x) >$ upper. The starting value for upper can be obtained by some heuristic or can be set to ∞.

As long as the initial value for upper is not less than the cost of a minimum-cost answer node, the above rules to kill live nodes will not result in the killing of a live node that can reach a minimum-cost answer node. Each time a new answer node is found, the value of upper can be updated.

Branch-and-bound algorithms are used for optimization problems where, we deal directly only with minimization problems. A maximization problem is easily converted to a minimization problem by changing the sign of the objective function.

To formulate the search for an optimal solution for a least-cost answer node in a state space tree, it is necessary to define the cost function c(.), such that c(x) is minimum for all nodes representing an optimal solution. The easiest way to do this is to use the objective function itself for c(.).

- **For nodes representing feasible solutions, c(x) is the value of the objective function for that feasible solution.**

- **For nodes representing infeasible solutions, $c(x) = \propto$.**

- **For nodes representing partial solutions, c(x) is the cost of the minimum-cost node in the subtree with root x.**

Since, c(x) is generally hard to compute, the branch-and-bound algorithm will use an estimate $\hat{c}(x)$ such that $\hat{c}(x) \leq c(x)$ for all x.

8.5. The 15 – Puzzle Problem:

The 15 puzzle is to search the state space for the goal state and use the path from the initial state to the goal state as the answer. There are 16! (16! ≈ 20.9 x 10^{12}) different arrangements of the tiles on the frame.

As the state space for the problem is very large it would be worthwhile to determine whether the goal state is reachable from the initial state. Number the frame positions 1 to 16.

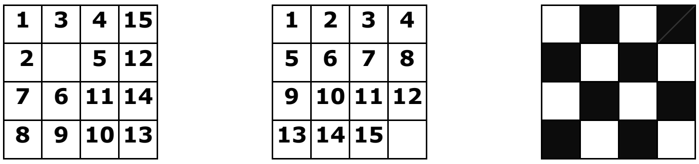
Position i is the frame position containing title numbered i in the goal arrangement of Figure 8.1(b). Position 16 is the empty spot. Let position(i) be the position number in the initial state of the title number i. Then position(16) will denote the position of the empty spot.

For any state let:

less(i) be the number of tiles j such that j < i and position(j) > position(i).

The goal state is reachable from the initial state iff: $\sum_{i=1}^{16} less(i) + x$ **is even.**

Here, x = 1 if in the initial state the empty spot is at one of the shaded positions of
figure 8.1(c) and x = 0 if it is at one of the remaining positions.



1	3	4	15
2		5	12
7	6	11	14
8	9	10	13

(a) An arrangement

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

(b) Goal arrangement

(c)

Figure 8.1. 15-puzzle arrangement

Example 1:

For the state of Figure 8.1(a) we have less(i) values as follows:

less(1) = 0	**less(2) = 0**	**less(3) = 1**	**less(4) = 1**
less(5) = 0	**less(6) = 0**	**less(7) = 1**	**less(8) = 0**
less(9) = 0	**less(10) = 0**	**less(11) = 3**	**less(12) = 6**
less(13) = 0	**less(14) = 4**	**less(15) = 11**	**less(16) = 10**

Therefore, $\sum_{i=1}^{16} less(i) + x =$ **(0 + 0 + 1 + 1 + 0 + 0 + 1 + 0 + 0 + 0 + 3 + 6 + 0 + 4 + 11 + 10) + 0 = 37 + 0 = 37.**

Hence, goal is *not reachable*.

Example 2:

For the root state of Figure 8.2 we have less(i) values are as follows:

less(1) = 0	**less(2) = 0**	**less(3) = 0**	**less(4) = 0**
less(5) = 0	**less(6) = 0**	**less(7) = 0**	**less(8) = 1**
less(9) = 1	**less(10) = 1**	**less(11) = 0**	**less(12) = 0**
less(13) = 1	**less(14) = 1**	**less(15) = 1**	**less(16) = 9**

Therefore, $\sum_{i=1}^{16} less(i) + x =$ **(0 + 0 + 0 + 0 + 0 + 0 + 0 + 1 + 1 + 1 + 0 + 0 + 1 + 1 + 1 + 9) + 1 = 15 + 1 = 16.**

Hence, goal is *reachable*.

8.6. LC Search for 15 Puzzle Problem:

A depth first state space tree generation will result in the subtree of Figure 8.3 when the next moves are attempted in the order: move the empty space up, right, down and left. The search of the state space tree is blind. It will take the leftmost path from the root regardless of the starting configuration. As a result, the answer node may never be found.

A breadth first search will always find a goal node nearest to the root. However, such a search is also blind in the sense that no matter what the initial configuration, the algorithm attempts to make the same sequence of moves.

We need a more intelligent search method. We associate a cost c(x) with each node x in the state space tree. The cost c(x) is the length of a path from the root to a nearest goal node in the subtree with root x. The easy to compute estimate $\hat{c}$ (x) of c(x) is as follows:

$$\hat{c}(x) = f(x) + \hat{g}(x)$$

where, f(x) is the length of the path from the root to node x and

> **$\hat{g}$ (x) is an estimate of the length of a shortest path from x to a goal node in the subtree with root x. Here, $\hat{g}$ (x) is the number of nonblank tiles not in their goal position.**

An LC-search of Figure 8.2, begin with the root as the E-node and generate all child nodes 2, 3, 4 and 5. The next node to become the E-node is a live node with least $\hat{c}$ (x).

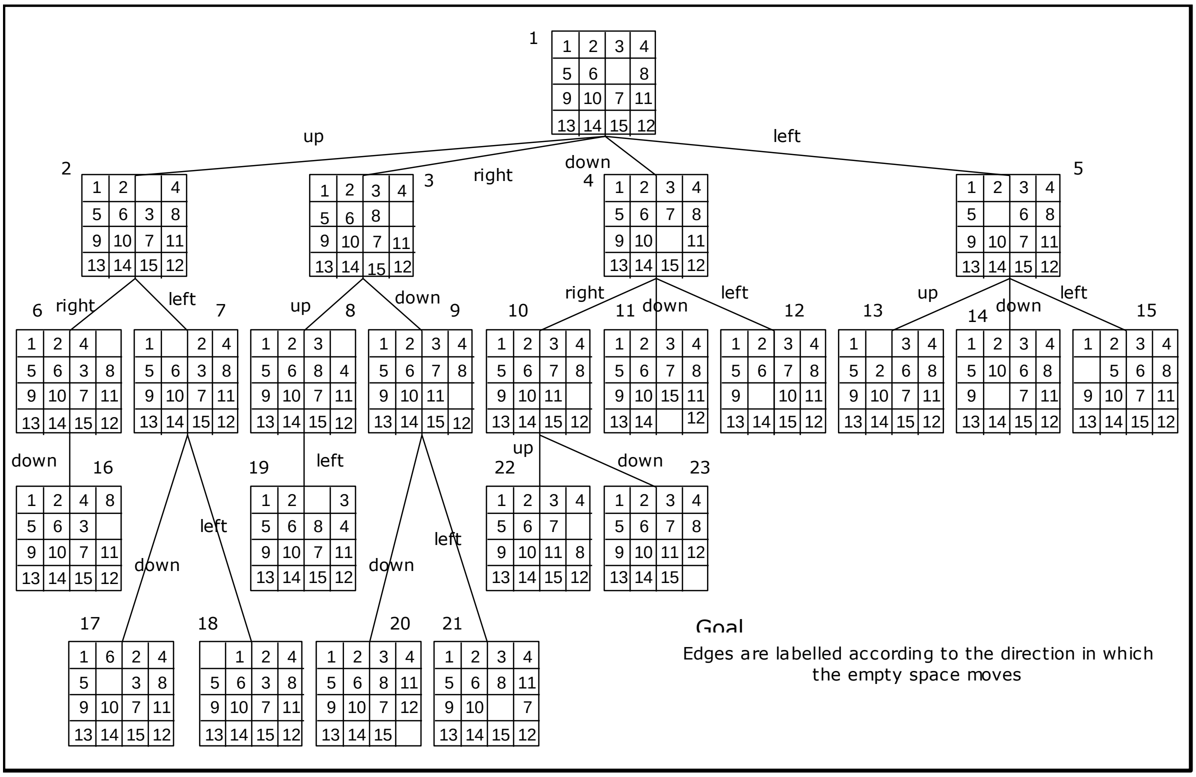
$\hat{c}$ (2) = 1 + 4 = 5
$\hat{c}$ (3) = 1 + 4 = 5
$\hat{c}$ (4) = 1 + 2 = 3 and
$\hat{c}$ (5) = 1 + 4 = 5.

Node 4 becomes the E-node and its children are generated. The live nodes at this time are 2, 3, 5, 10, 11 and 12. So:

$\hat{c}(10) = 2 + 1 = 3$
$\hat{c}(11) = 2 + 3 = 5$ and
$\hat{c}(12) = 2 + 3 = 5$.

The live node with least $\hat{c}$ is node 10. This becomes the next E-node. Nodes 22 and 23 are generated next. Node 23 is the goal node, so search terminates.

LC-search was almost as efficient as using the exact function c(), with a suitable choice for $\hat{c}$(), LC-search will be far more selective than any of the other search methods.



Edges are labelled according to the direction in which the empty space moves

Figure 8.2. Part of the state space tree for 15-puzzle problem

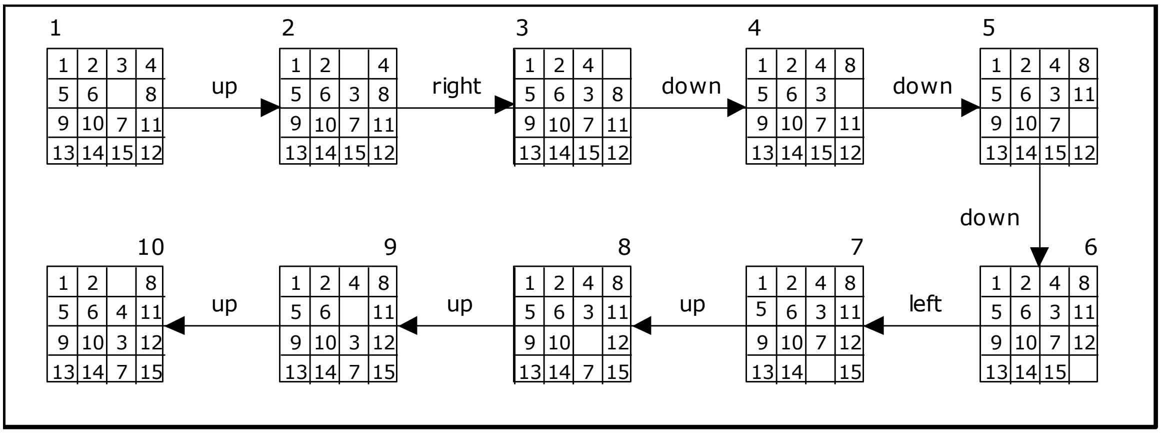


Figure 8.3. First ten steps in a depth first search

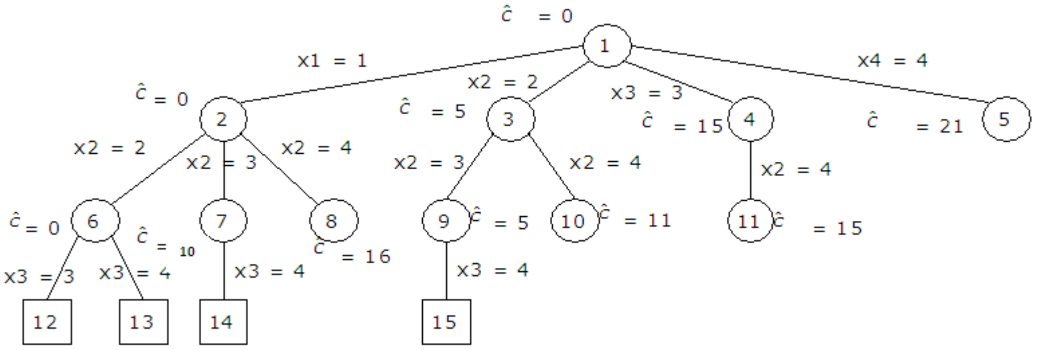
8.7. Job sequencing with deadlines:

We are given n jobs and one processor. Each job i is associated with it a three tuple (p_i, d_i, t_i). Job i requires t_i units of processing time. If its processing is not completed by the deadline d_i, then a penalty p_i is incurred. The objective is to select a subset J of the n jobs such that all jobs in J can be completed by their deadlines. Hence, a penalty can be incurred only on those jobs not in J. The subset J should be such that the penalty incurred is minimum among all possible subsets J. Such a J is optimal.

Example:

Consider the following instance with n = 4, (p_1, d_1, t_1) = (5, 1, 1), (p_2, d_2, t_2) = (10, 3, 2), (p_3, d_3, t_3) = (6, 2, 1) and (p_4, d_4, t_4) = (3, 1, 1).

The solution space for this instance consists of all possible subsets of the job index set {1, 2, 3, 4}. The solution space can be organized into a tree. Figure 8.4 corresponds to the variable tuple size formulation. In figure square nodes represent infeasible subsets and all non-square nodes are answer nodes. Node 9 represents an optimal solution and is the only minimum-cost answer node. For this node J={2, 3} and the penalty is 8.



The cost function c(x) for any circular node x is the minimum penalty corresponding to any node in the subtree with root x.

The value of c(x) = ∝ for a square node.

Let S_x be the subset of jobs selected for J at node x.

If m = max $\{ i \;/\; i \in S_x \}$, then $\hat{c}(x) = \sum_{\substack{i<m \\ i \neq s_x}} p_i$ is an estimate for c(x) with the property $\hat{c}$ (x) ≤ c(x).

An upper bound u(x) on the cost of a minimum-cost answer node in the subtree x is u(x) = $\sum_{i \notin S_x} p_i$. The u(x) is the cost of the solution S_x corresponding to node x.

S(2) = {1}	**M =1**	$\hat{c}(2) = \sum_{\substack{i<m \\ i \neq sx}} p_i = 0$

$S_{(3)}$ = {2}	**m = 2**	$\hat{c}(3) = \sum_{i<2} p_i = \sum_{i=1} p_i = 5$
S(4)= {3}	**m = 3.**	$\hat{c}(4) = \sum_{i<3} p_i = \sum_{i=1,2} p_i = p_1 + p_2 = 5 + 10 = 15$
S(5) = {4}	**m = 4**	$\hat{c}(5) = \sum_{i<4} p_i = \sum_{i=1,2,3} p_i = p_1 + p_2 + p_3 = 5 + 10 + 6 = 21$
S(6) = {1, 2}	**m = 2**	$\hat{c}(6) = \sum_{\substack{i=1,2 \\ i \in s_x}} p_i = \sum_{\substack{i<1 \\ i \neq s(6)}} p_i = 0$
S(7) = {1, 3}	**m = 3**	$\hat{c}(7) = \sum_{\substack{i<3 \\ i \notin s(7)}} p_i = p_2$ **= 10**
S(8) = {1, 4}	**m = 4**	$\hat{c}(8) = \sum_{\substack{i<4 \\ i \notin s(8)}} p_i = p_2 + p_3 = 10 + 6 = 16$
S(9)={2,3}	**m=3**	$\hat{c}$ **(9)= 5**
S(10)={2,4}	**m=3**	$\hat{c}$ **(10)=11**
S(11)={3,4}	**m=4**	$\hat{c}$ **(11)=15**

Calculation of the Upper bound u(x) = $\sum_{i \notin s_x} p_i$

U(1) = 0

U(2) = $\sum_{i \notin s(2)} p_i = p_2 + p_3 + p_4 = 10 + 6 + 3 = 19$ **eliminate job 1**

U(3) = p_1 + p_3 + p_4 = 5 + 6 + 3 = 14 **eliminate job 2**

U(4) = p_1 + p_2 + p_4 = 5 + 10 + 3 = 18 **eliminate job 3**

U(5) = p_1 + p_2 + p_4 = 5 + 10 + 6= 21 **eliminate job 4**

U(6) = p_3 + p_4 = 6 + 3 = 9 **eliminate jobs 1, 2**

U(7) = p_2 + p_4 = 10 + 3 = 13 **eliminate jobs 1, 3**

U(8) = p_2 + p_3 = 10 + 6 = 16 **eliminate jobs 1, 4**

U(9) = p_1 + p_4 = 5 + 3 = 8 **eliminate jobs 2, 3**

U(10) = p_1 + p_3 = 5 + 6 = 11 **eliminate jobs 2, 4**

U(11) = p_1 + p_2 = 5 + 10 = 15 **eliminate jobs 3, 4**

FIFO Branch and Bound:

A FIFO branch-and-bound algorithm for the job sequencing problem can begin with upper = ∝ as an upper bound on the cost of a minimum-cost answer node.

Starting with node 1 as the E-node and using the variable tuple size formulation of Figure 8.4, nodes 2, 3, 4, and 5 are generated. Then u(2) = 19, u(3) = 14, u(4) = 18, and u(5) = 21.

The variable upper is updated to 14 when node 3 is generated. Since $\hat{c}$ (4) and $\hat{c}(5)$ are greater than upper, nodes 4 and 5 get killed. Only nodes 2 and 3 remain alive.

Node 2 becomes the next E-node. Its children, nodes 6, 7 and 8 are generated. Then u(6) = 9 and so upper is updated to 9. The cost $\hat{c}(7) =$ 10 > upper and node 7 gets killed. Node 8 is infeasible and so it is killed.

Next, node 3 becomes the E-node. Nodes 9 and 10 are now generated. Then u(9) = 8 and so upper becomes 8. The cost $\hat{c}(10) =$ 11 > upper, and this node is killed.

The next E-node is node 6. Both its children are infeasible. Node 9's only child is also infeasible. The minimum-cost answer node is node 9. It has a cost of 8.

When implementing a FIFO branch-and-bound algorithm, it is not economical to kill live nodes with $\hat{c}(x)$ > upper each time upper is updated. This is so because live nodes are in the queue in the order in which they were generated. Hence, nodes with $\hat{c}(x)$ > upper are distributed in some random way in the queue. Instead, live nodes with $\hat{c}(x)$ > upper can be killed when they are about to become E-nodes.

The FIFO-based branch-and-bound algorithm with an appropriate $\hat{c}(.)$ and u(.) is called FIFOBB.

LC Branch and Bound:

An LC Branch-and-Bound search of the tree of Figure 8.4 will begin with upper = $\propto$ and node 1 as the first E-node.

When node 1 is expanded, nodes 2, 3, 4 and 5 are generated in that order.

As in the case of FIFOBB, upper is updated to 14 when node 3 is generated and nodes 4 and 5 are killed as $\hat{c}(4)$ > upper and $\hat{c}(5)$ > upper.

Node 2 is the next E-node as $\hat{c}(2) =$ 0 and $\hat{c}(3) =$ 5. Nodes 6, 7 and 8 are generated and upper is updated to 9 when node 6 is generated. So, node 7 is killed as $\hat{c}(7) =$ 10 > upper. Node 8 is infeasible and so killed. The only live nodes now are nodes 3 and 6.

Node 6 is the next E-node as $\hat{c}(6) =$ 0 < $\hat{c}(3)$. Both its children are infeasible.

Node 3 becomes the next E-node. When node 9 is generated, upper is updated to 8 as u(9) = 8. So, node 10 with $\hat{c}(10) =$ 11 is killed on generation.

Node 9 becomes the next E-node. Its only child is infeasible. No live nodes remain. The search terminates with node 9 representing the minimum-cost answer node.

The path = $1 \xrightarrow{2} 3 \xrightarrow{3} 9 = 5 + 3 = 8$

8.8. Traveling Sale Person Problem:

By using dynamic programming algorithm we can solve the problem with time complexity of $O(n^2 2^n)$ for worst case. This can be solved by branch and bound technique using efficient bounding function. The time complexity of traveling sale person problem using LC branch and bound is $O(n^2 2^n)$ which shows that there is no change or reduction of complexity than previous method.

We start at a particular node and visit all nodes exactly once and come back to initial node with minimum cost.

Let G = (V, E) is a connected graph. Let C(i, J) be the cost of edge <i, j>. $c_{ij} = \propto$ if <i, j> $\notin$ E and let |V| = n, the number of vertices. Every tour starts at vertex 1 and ends at the same vertex. So, the solution space is given by S = {1, π, 1 | π is a permutation of (2, 3, . . . , n)} and |S| = (n – 1)!. The size of S can be reduced by restricting S so that $(1, i_1, i_2, \ldots\ i_{n-1}, 1) \in S$ iff $<i_j, i_{j+1}> \in E$, $0 \leq j \leq n - 1$ and $i_0 = i_n = 1$.

Procedure for solving traveling sale person problem:

1. **Reduce the given cost matrix. A matrix is reduced if every row and column is reduced. A row (column) is said to be reduced if it contain at least one zero and all-remaining entries are non-negative. This can be done as follows:**

    a) ***Row reduction:* Take the minimum element from first row, subtract it from all elements of first row, next take minimum element from the second row and subtract it from second row. Similarly apply the same procedure for all rows.**

    b) **Find the sum of elements, which were subtracted from rows.**

    c) **Apply column reductions for the matrix obtained after row reduction.**

    ***Column reduction:* Take the minimum element from first column, subtract it from all elements of first column, next take minimum element from the second column and subtract it from second column. Similarly apply the same procedure for all columns.**

    d) **Find the sum of elements, which were subtracted from columns.**

    e) **Obtain the cumulative sum of row wise reduction and column wise reduction.**

    Cumulative reduced sum = Row wise reduction sum + column wise reduction sum.

    Associate the cumulative reduced sum to the starting state as lower bound and $\propto$ as upper bound.

2. **Calculate the reduced cost matrix for every node R. Let A is the reduced cost matrix for node R. Let S be a child of R such that the tree edge (R, S) corresponds to including edge <i, j> in the tour. If S is not a leaf node, then the reduced cost matrix for S may be obtained as follows:**

a) Change all entries in row i and column j of A to ∝.

b) Set A (j, 1) to ∝.

c) Reduce all rows and columns in the resulting matrix except for rows and column containing only ∝. Let r is the total amount subtracted to reduce the matrix.

c) Find $\hat{c}(S) = \hat{c}(R) + A(i,\ j) + r,$ where 'r' is the total amount subtracted to reduce the matrix, $\hat{c}(R)$ indicates the lower bound of the i^{th} node in (i, j) path and $\hat{c}(S)$ is called the cost function.

3. Repeat step 2 until all nodes are visited.

Example:

Find the LC branch and bound solution for the traveling sale person problem whose cost matrix is as follows:

The cost matrix is $\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$

Step 1: Find the reduced cost matrix.

Apply row reduction method:

Deduct 10 (which is the minimum) from all values in the 1st row.
Deduct 2 (which is the minimum) from all values in the 2nd row.
Deduct 2 (which is the minimum) from all values in the 3rd row.
Deduct 3 (which is the minimum) from all values in the 4th row.
Deduct 4 (which is the minimum) from all values in the 5th row.

The resulting row wise reduced cost matrix = $\begin{bmatrix} \infty & 10 & 20 & 0 & 1 \\ 13 & \infty & 14 & 2 & 0 \\ 1 & 3 & \infty & 0 & 0 \\ 16 & 3 & 15 & \infty & 0 \\ 12 & 0 & 3 & 12 & \infty \end{bmatrix}$

Row wise reduction sum = 10 + 2 + 2 + 3 + 4 = 21

Now apply column reduction for the above matrix:

Deduct 1 (which is the minimum) from all values in the 1st column.
Deduct 3 (which is the minimum) from all values in the 3rd column.

The resulting column wise reduced cost matrix (A) = $\begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$

Column wise reduction sum = 1 + 0 + 3 + 0 + 0 = 4

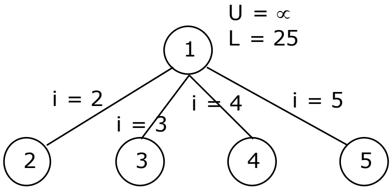
Cumulative reduced sum = row wise reduction + column wise reduction sum.
= 21 + 4 = 25.

This is the cost of a root i.e., node 1, because this is the initially reduced cost matrix.

The lower bound for node is 25 and upper bound is ∝.

Starting from node 1, we can next visit 2, 3, 4 and 5 vertices. So, consider to explore the paths (1, 2), (1, 3), (1, 4) and (1, 5).

The tree organization up to this point is as follows:



Variable 'i' indicates the next node to visit.

Step 2:

2.1. Consider the path (1, 2):

Change all entries of row 1 and column 2 of A to ∝ and also set A(2, 1) to ∝.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∝.

Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix}$

Row reduction sum = 0 + 0 + 0 + 0 = 0
Column reduction sum = 0 + 0 + 0 + 0 = 0
Cumulative reduction (r) = 0 + 0 = 0

Therefore, as $\hat{c}(S) = \hat{c}(R) + A(1, 2) + r$
$\hat{c}(S)$ **= 25 + 10 + 0 = 35**

2.2. Consider the path (1, 3):

Change all entries of row 1 and column 3 of A to $\propto$ and also set A(3, 1) to $\propto$.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 15 & 3 & \infty & \infty & 0 \\ 11 & 0 & \infty & 12 & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely $\propto$.

Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 4 & 3 & \infty & \infty & 0 \\ 0 & 0 & \infty & 12 & \infty \end{bmatrix}$

Row reduction sum = 0
Column reduction sum = 11
Cumulative reduction (r) = 0 + 11 = 11

Therefore, as $\hat{c}(S) = \hat{c}(R) + A(1, 3) + r$

$\widehat{c}(S)$ **= 25 + 17 + 11 = 53**

2.3. Consider the path (1, 4):

Change all entries of row 1 and column 4 of A to ∝ and also set A(4, 1) to ∝.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∝.

Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$

Row reduction sum = 0
Column reduction sum = 0
Cumulative reduction (r) = 0 + 0 = 0

Therefore, as $\widehat{c}(S) = \widehat{c}(R) + A\,(1,\ 4) + r$

$\widehat{c}(S)$ **= 25 + 0 + 0 = 25**

2.4. Consider the path (1, 5):

Change all entries of row 1 and column 5 of A to ∝ and also set A(5, 1) to ∝.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & 2 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 15 & 3 & 12 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∝.

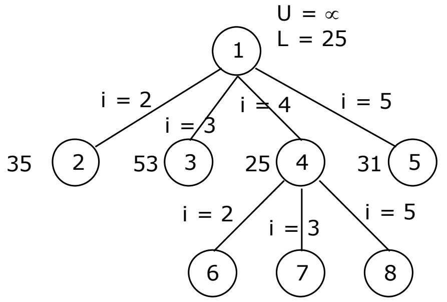
Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 10 & \infty & 9 & 0 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 12 & 0 & 9 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix}$

Row reduction sum = 5
Column reduction sum = 0
Cumulative reduction (r) = 5 + 0 = 0

Therefore, as $\widehat{c}(S) = \widehat{c}(R) + A(1, 5) + r$

$\widehat{c}(S)$ **= 25 + 1 + 5 = 31**

The tree organization up to this point is as follows:



The cost of the paths between (1, 2) = 35, (1, 3) = 53, (1, 4) = 25 and (1, 5) = 31. The cost of the path between (1, 4) is minimum. Hence the matrix obtained for path (1, 4) is considered as reduced cost matrix.

A = $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$

The new possible paths are (4, 2), (4, 3) and (4, 5).

2.5. Consider the path (4, 2):

Change all entries of row 4 and column 2 of A to ∝ and also set A(2, 1) to ∝.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∝.

Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$

Row reduction sum = 0
Column reduction sum = 0
Cumulative reduction (r) = 0 + 0 = 0

Therefore, as $\hat{c}(S) = \hat{c}(R) + A\,(4,2) + r$

$\hat{c}(S)$ **= 25 + 3 + 0 = 28**

2.6. Consider the path (4, 3):

Change all entries of row 4 and column 3 of A to ∝ and also set A(3, 1) to ∝.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty & 0 \\ \infty & 3 & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & 0 & \infty & \infty & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely $\propto$.

Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & \infty & 0 \\ \infty & 1 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & 0 & \infty & \infty & \infty \end{bmatrix}$

Row reduction sum = 2
Column reduction sum = 11
Cumulative reduction (r) = 2 + 11 = 13

Therefore, as $\hat{c}(S) = \hat{c}(R) + A\,(4,\,3) + r$

$\hat{c}(S)$ **= 25 + 12 + 13 = 50**

2.7. Consider the path (4, 5):

Change all entries of row 4 and column 5 of A to $\propto$ and also set A(5, 1) to $\propto$.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely $\propto$.

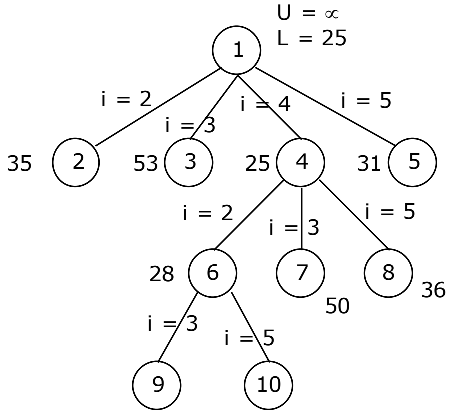
Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix}$

Row reduction sum = 11
Column reduction sum = 0
Cumulative reduction (r) = 11+0 = 11

Therefore, as $\hat{c}(S) = \hat{c}(R) + A(4, 5) + r$

$\hat{c}(S)$ **= 25 + 0 + 11 = 36**

The tree organization up to this point is as follows:



The cost of the paths between (4, 2) = 28, (4, 3) = 50 and (4, 5) = 36. The cost of the path between (4, 2) is minimum. Hence the matrix obtained for path (4, 2) is considered as reduced cost matrix.

$$\mathbf{A} = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

The new possible paths are (2, 3) and (2, 5).

2.8. *Consider the path (2, 3):*

Change all entries of row 2 and column 3 of A to ∝ and also set A(3, 1) to ∝.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & \infty & \infty & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∝.

Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \end{bmatrix}$

Row reduction sum = 2
Column reduction sum = 11
Cumulative reduction (r) = 2 + 11 = 13

Therefore, as $\widehat{c}(S) = \widehat{c}(R) + A\,(2,\,3) + r$

$\widehat{c}(S)$ **= 28 + 11 + 13 = 52**

2.9. *Consider the path (2, 5):*

Change all entries of row 2 and column 5 of A to ∝ and also set A(5, 1) to ∝.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely $\propto$.

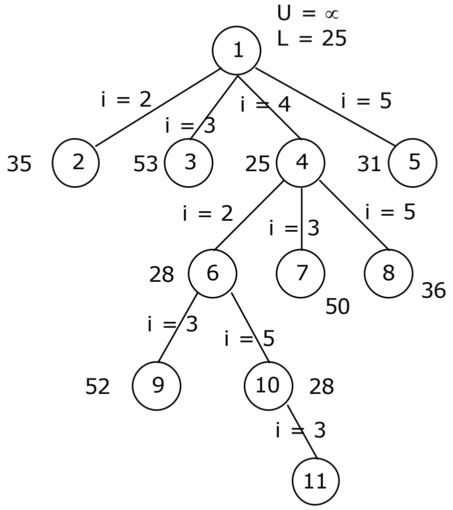
Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix}$

Row reduction sum = 0
Column reduction sum = 0
Cumulative reduction (r) = 0 + 0 = 0

Therefore, as $\hat{c}(S) = \hat{c}(R) + A(2, 5) + r$

$\hat{c}(S)$ **= 28 + 0 + 0 = 28**

The tree organization up to this point is as follows:



The cost of the paths between (2, 3) = 52 and (2, 5) = 28. The cost of the path between (2, 5) is minimum. Hence the matrix obtained for path (2, 5) is considered as reduced cost matrix.

A = $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix}$

The new possible paths is (5, 3).

2.10. Consider the path (5, 3):

Change all entries of row 5 and column 3 of A to ∝ and also set A(3, 1) to ∝. Apply row and column reduction for the rows and columns whose rows and columns are not completely ∝.

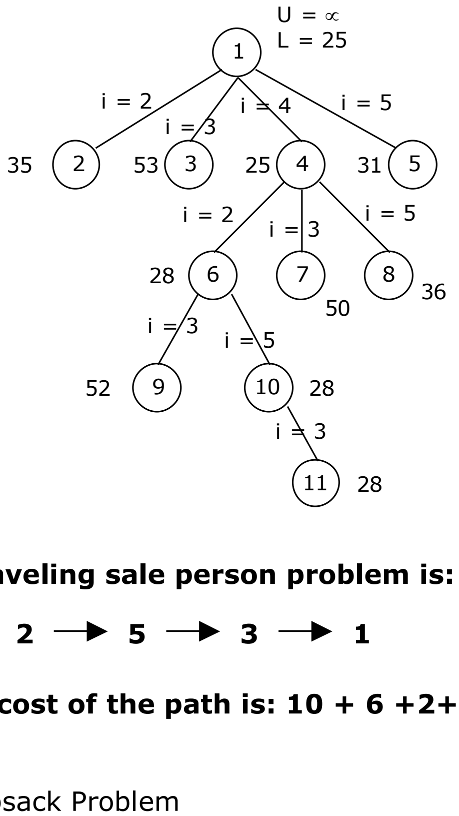
Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$

Row reduction sum = 0
Column reduction sum = 0
Cumulative reduction (r) = 0 + 0 = 0

Therefore, as $\hat{c}(S) = \hat{c}(R) + A(5,\ 3) + r$

$\hat{c}(S)$ **= 28 + 0 + 0 = 28**

The overall tree organization is as follows:



The path of traveling sale person problem is:

1 ⟶ 4 ⟶ 2 ⟶ 5 ⟶ 3 ⟶ 1

The minimum cost of the path is: 10 + 6 +2+ 7 + 3 = 28.

8.9. 0/1 Knapsack Problem

Consider the instance: M = 15, n = 4, (P_1, P_2, P_3, P_4) = (10, 10, 12, 18) and (w_1, w_2, w_3, w_4) = (2, 4, 6, 9).

0/1 knapsack problem can be solved by using branch and bound technique. In this problem we will calculate lower bound and upper bound for each node.

Place first item in knapsack. Remaining weight of knapsack is 15 – 2 = 13. Place next item w_2 in knapsack and the remaining weight of knapsack is 13 – 4 = 9. Place next item w_3 in knapsack then the remaining weight of knapsack is 9 – 6 = 3. No fractions are allowed in calculation of upper bound so w_4 cannot be placed in knapsack.

Profit = $P_1 + P_2 + P_3$ = 10 + 10 + 12

So, Upper bound = 32

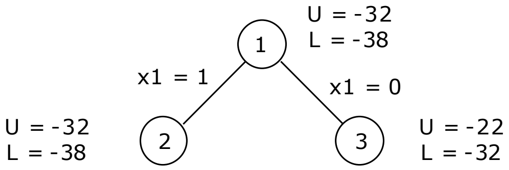
To calculate lower bound we can place w_4 in knapsack since fractions are allowed in calculation of lower bound.

Lower bound = 10 + 10 + 12 + ($\frac{3}{9}$ X 18) = 32 + 6 = 38

Knapsack problem is maximization problem but branch and bound technique is applicable for only minimization problems. In order to convert maximization problem into minimization problem we have to take negative sign for upper bound and lower bound.

Therefore, Upper bound (U) = -32
Lower bound (L) = -38

We choose the path, which has minimum difference of upper bound and lower bound. If the difference is equal then we choose the path by comparing upper bounds and we discard node with maximum upper bound.



Now we will calculate upper bound and lower bound for nodes 2, 3.

For node 2, x_1= 1, means we should place first item in the knapsack.

U = 10 + 10 + 12 = 32, make it as -32

L = 10 + 10 + 12 + $\frac{3}{9}$ x 18 = 32 + 6 = 38, make it as -38

For node 3, x_1 = 0, means we should not place first item in the knapsack.

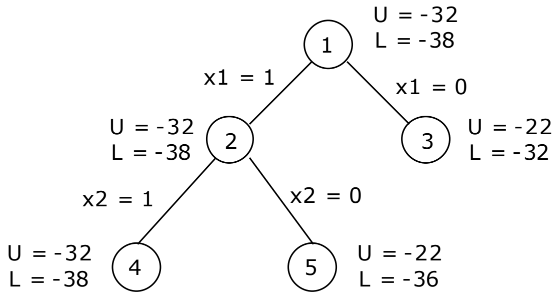
U = 10 + 12 = 22, make it as -22

L = 10 + 12 + $\frac{5}{9}$ x 18 = 10 + 12 + 10 = 32, make it as -32

Next, we will calculate difference of upper bound and lower bound for nodes 2, 3

For node 2, U – L = -32 + 38 = 6
For node 3, U – L = -22 + 32 = 10

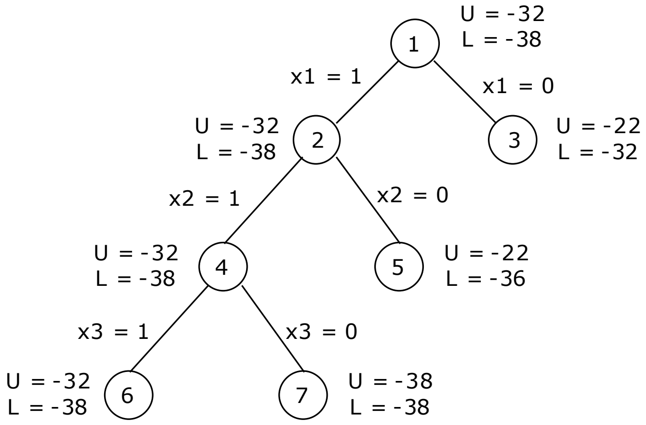
Choose node 2, since it has minimum difference value of 6.



Now we will calculate lower bound and upper bound of node 4 and 5. Calculate difference of lower and upper bound of nodes 4 and 5.

For node 4, U – L = -32 + 38 = 6
For node 5, U – L = -22 + 36 = 14

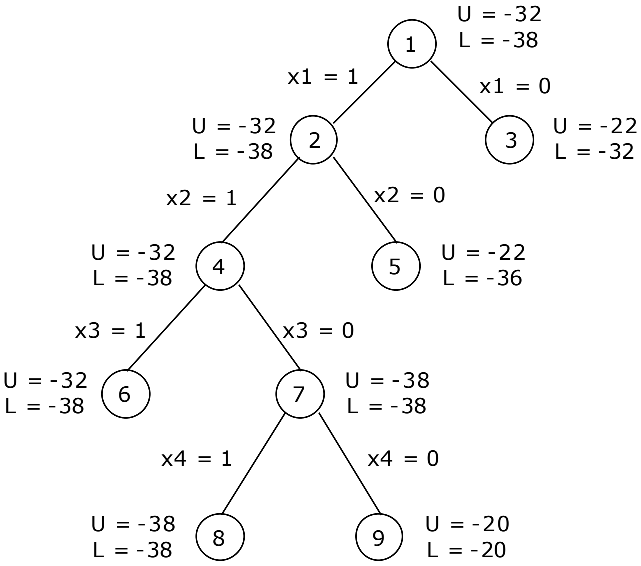
Choose node 4, since it has minimum difference value of 6.



Now we will calculate lower bound and upper bound of node 8 and 9. Calculate difference of lower and upper bound of nodes 8 and 9.

For node 6, U – L = -32 + 38 = 6
For node 7, U – L = -38 + 38 = 0

Choose node 7, since it is minimum difference value of 0.



Now we will calculate lower bound and upper bound of node 4 and 5. Calculate difference of lower and upper bound of nodes 4 and 5.

For node 8, U – L = -38 + 38 = 0
For node 9, U – L = -20 + 20 = 0

Here the difference is same, so compare upper bounds of nodes 8 and 9. Discard the node, which has maximum upper bound. Choose node 8, discard node 9 since, it has maximum upper bound.

Consider the path from 1 → 2 → 4 → 7 → 8

X_1 = 1

X_2 = 1

X_3 = 0

X_4 = 1

The solution for 0/1 Knapsack problem is (x_1, x_2, x_3, x_4) = (1, 1, 0, 1)

Maximum profit is:

$$\sum P_i x_i = 10 \times 1 + 10 \times 1 + 12 \times 0 + 18 \times 1$$
$$= 10 + 10 + 18 = 38.$$

Chapter 9

Lower bound theory

Searching ordered lists with Comparison–Based Algorithms

Comparison-Based Algorithms: Information can be gained only by comparing key–to–element, or element–to–element (in some problems).

Given: An integer n, a key, and an ordered list of n values.
Question: Is the key in the list and, if so, at what index?

We have an algorithm. We don't know what it is, or how it works. It accepts n, a key and a list on n values. That's it.

It MUST, though, work pretty much as follows:

1) It must calculate an index for the first compare based solely upon n since it has not yet compared the key against anything, i.e., it has not yet obtained any additional information. Notice, this means for a fixed value of n, the position of the first compare is fixed for all data sets (of size n).

2) The following is repeated until the key is found, or until it is determined that no location contains the key:
The key is compared against the item at the specified index.
a) If they are equal, the algorithm halts.
b) If the key is less, then it incorporates this information and computes a new index.
c) If the key is greater, then it incorporates this information and computes a new index

There are no rules about how this must be done. In fact we want to leave it wide open so that we are not eliminating any possible algorithm.

After the first compare, there are two possible second compare locations (indexes). Neither depends upon the key or any item in the list: Just upon the result of the first compare. Every second compare on every set of n items will be one of these two locations.

Every third compare will be one of four locations. Every fourth compare will be one of eight locations. And, so on. In fact, we may look at an algorithm (for a given n) as being described by (or, possibly, describing) a binary tree in which the root corresponds to the first comparison, it's children to the possible second comparisons, their four children represent the possible third comparison, etc. This binary tree, called in this context a "decision tree," then depicts for this algorithm every possible path of comparisons that could be forced by any particular key and set of n values.

Observation 0: Every comparison–based search algorithm has it's own set of decision tree's (one for each value of n) – even if we don't know what or how it does its task, we know it has one for each n and are pretty much like the one described above.
Observation 1: For any decision tree and any root–leaf path, there is a set of date which will force the algorithm to take that path. The number of compares with a given data set (key and n values) is the number of nodes in the "forced" root–leaf path.

Observation 2: The longest root–leaf path is the "worst case" running time of the algorithm.
Observation 3. For any position i of {1, 2, . . . , n}, some data set contains the key in that position. So every algorithm must have a compare for every index, that is, the decision tree must have at least one node for each position.

Therefore, all decision trees–for the search problem–must have at least n nodes in them.

All binary trees with n nodes have a root–leaf path with at least $\lceil \log_2(n+1) \rceil$ nodes (you can verify this by induction).

Thus, all decision trees defined by search algorithms on n items, have a path requiring $\lceil \log_2(n+1) \rceil$ compares.

Therefore, **the best any comparison–based search algorithm can hope to do** is $\log_2 n \approx \lceil \log_2(n+1) \rceil$.

This is the *comparison–based lower bound for the problem of searching an ordered list* of n items for a given key.

Comparison Based Sorting

Here, there is no "key." Typically, in comparison–based sorting, we will compare values in two locations and, depending upon which is greater, we might

1) do nothing,
2) exchange the two values,
3) move one of the values to a third position,
4) leave a reference (pointer) at one of the positions,
5) etc.

As with searching, above, each sorting algorithm has its own decision tree. Some differences occur: Leaf nodes are the locations which indicate "the list is now sorted." Internal nodes simply represent comparisons on the path to a leaf node.

As with searching, we will determine the minimum number of nodes any sorting algorithm (comparison–based) must have. Then, from that, the minimum height of all decision trees (for sorting) can be determined providing the proof that all comparison–based sorting algorithms must use at least this many comparisons.

Actually, it is quite simple now. Every decision tree starts with an unordered list and ends up at a leaf node with a sorted list. Suppose you have two lists, one is a rearrangement of the other. Then, in sorting them, something must be done differently to one of the lists (done at a different time). Otherwise, if the same actions are performed to both lists in exactly the same sequence, then one of them can not end up sorted. Therefore, they must go through different paths from the root of the decision tree. By the same reasoning, all n! different permutations of the integers 1, 2, . . ., n (these are valid things to sort, too, you know) must go through distinct paths. Notice that distinct paths end in distinct leaf nodes. Thus, there must be n! leaf nodes in every decision tree for sorting. That is, their height is at least $\log_2(n!)$. By a common result (one of Sterlings formula's) $\log_2(n!) \approx n\log_2(n)$.

Therefore, **all comparison–based sorting algorithms require $n\log_2(n)$ time**.

NP-COMPLETE PROBLEMS

A polynomial algorithm is “faster” than an exponential algorithm. As n grows a^n (exponential) always grows faster than n^k (polynomial),
i.e. for any values of a and k, after n> certain integer n_0, it is true that $a^n > n^k$.
Even 2^n grows faster than n^{1000} at some large value of n. The former functions are exponential and the later functions are polynomial.

It seems that for some problems we just may not have any polynomial algorithm at all (as in the *information theoretic* bound)! The theory of NP-completeness is about this issue, and in general the computational complexity theory addresses it.

Solvable Problems

Some problems are even unsolvable/ undecidable algorithmically, so you cannot write a computer program for them.

Example. *Halting problem*: Given an algorithm as input, determine if it has an infinite loop.

There does not exist any general-purpose algorithm for this problem.
Suppose (contradiction) there is an algorithm H that can tell if any algorithm X halts or not, i.e., H(X) scans X and returns True iff X does halt.
Then, write

```
P(X):                  // X is any “input algorithm” to P
while (H(X)) { };              // condition is true only when X terminates
return;                // P terminates
End.
```

Then, provide P itself as the input X to the algorithm [i.e., P(P)]: what happens?!
H cannot return T/F for input P, there cannot exist such H.
It is equivalent to the truth value of the sentence “*This sentence is a lie*.”

Note (1), we are considering **Problems** (i.e., for all instances of the problem) as opposed to some **instances of the problem**. For some sub-cases you may be able to solve the *halting problem*, but you cannot have an algorithm, which would solve the halting problem for ALL input.

<u>Different **types of problems**</u>: decision problems, optimization problems, …
Decision problems: Output is *True/False*.

Decision Problem <-> corresponding optimization problem.
Example of 0-1 *Knapsack Decision* (KSD) problem:
Input: KS problem + a profit value p
Output: Answer the question "does there exist a knapsack with profit $\geq p$?"

Algorithms are also inter-operable between a problem and its corresponding decision problem.

Solve a KSD problem K, using an optimizing algorithm:
Algorithm-Optimization-KS (first part of K without given profit p) return optimal profit o
if $p \geq o$ then return *TRUE* else return *FALSE*.

Solve a KS problem *N*, using a decision algorithm:

```
For (o = Sum-of-all-objects-profit;   o>0;   o = o - delta)  do  // do a linear search, for a small delta
  If (Algorithm-Decision-KS (N, o) ) then continue
   Else return  (the last value of o, before failure, in the above step);
Endfor.   // a binary search would be faster!!
```

Note (2), The complexity theory is developed over *Decision problems*, but valid for other problems as well. We will often use other types of problems as examples.

NP-class of solvable problems

Deterministic algorithms are where no step is random.

If the program/algorithm chooses a step non-deterministically (by some extraneous influence to the algorithm!) such that it is always the *right choice*,
then such an algorithm is called **non-deterministic algorithm**.
Example, suppose a 0-1 KS backtrack-algorithm always knows which object to pick up next in order to find an optimal profit-making knapsack!

If one has a *polynomial deterministic algorithm* for a problem (e.g., the sorting problem), then the problem is called a **P-class** problem.
And, the set of all such problems constitute the ***P-class.***

If one has a *polynomial non-deterministic algorithm* for a problem, then the problem is called a **NP-class** problem.
And, the set of all such problems constitute the *N**P-class.***

It is impossible to check for a problem's being in NP-class this way, because non-deterministic algorithms are impossible to develop, as defined above.
So, how can one check for polynomial complexity of such non-existent algorithms?

However, an equivalent way of developing non-deterministic polynomial algorithm is:
when a solution to the problem is provided (as if someone knows what the solution could be!),
then that *proposed solution* is checked by that algorithm in *polynomial* time.
Such a *proposed solution* is called a certificate to the input problem-instance.

For example: in a KSD problem, given a certificate *knapsack content* check if the total profit is ≥ *p* or not.
Complexity: calculation of total profit of the given knapsack content is worst case O(*n*), for *n* objects.

For example: for a Hamiltonian circuit problem instance (*find a cycle in a graph over all nodes but without any node being repeated in the cycle*),
a *path* is given as a certificate.
An algorithm would go over the certificate path and check if the first and the last nodes are *same*,
and the rest of the path is *simple* (no node is covered twice),
then it will check if all nodes in the graph are *covered* in the path,
and then, check if all the *arcs* in the path do actually from the input graph.
This takes polynomial time with respect to *N* and *E*. Hence HC is a NP-class problem.

Two important points to note:

(1) NP-class problems are sub-set of the *Solvable* problems,

(2) P-class problems are sub-set of NP-class problems (because if you have a deterministic algorithm to solve any problem instance,
then that algorithm can be used to check any certificate in polynomial time also).

[DRAW SETS]

Problems belonging to the NP-class have at least exponential algorithms.

A related question is: does there exist any solvable problem that is not NP?
Answer: yes.
Example: non-HC problem (is there *no HC* in a given input graph) does not have any polynomial non-deterministic algorithm.

If a problem is in NP-class its complement (negative problem as the non-HC problem is) is in *Co-NP*.
All Co-NP problems constitute the Co-NP class of problems.
P-class is a subset of the intersection of NP and Co-NP sets.

An IMPORTANT question is: can we write a polynomial algorithm for *every* NP-class problem?
The answer is: *we do not know*.

From a reverse point of view, we would like to find an example problem that is in the NP-class and whose information-theoretic lower bound is exponential.
Then, we would at least know that P is a proper subset of NP. We do not yet have any such example either.

All we know now is that **P $\subseteq$ NP**.

Question remains: (1) P $\subset$ NP? or, P $\neq$ NP? [One should find counter-example(s)]
Or, (2) NP $\subseteq$ P, so that P = NP? [One should prove the theorem]

NP-completeness

Summary: Apparently some problems are "harder" in a relative sense than the other problems within the NP-class of problems. If you can write polynomial algorithm for any one problem in this group of apparently hard problems, then it can be shown that every NP-class problem will have a polynomial algorithm for each. This group of problems is called *NP-complete* problems.

The secret lies in the fact that they are all connected to *all NP-class problems* (!!!) by directed chains of *polynomial transformations* (explained below).

We will explain polynomial transformations first and then come back to the issue of NP-completeness.

Polynomial Transformation

Problem Transformation: some algorithms which take a decision problem X (or rather ANY instance of the problem of type X), and output a corresponding instance of the decision problem of type Y, in such a way that if the input has answer *True*, then the output (of type *Y*) is also *True* and vice versa.

For example, you can write a problem transformation algorithm from *3-SAT* problem to *3D-Matching* problem (will see later).

Note that the problem transformations are directed.

When a problem transformation algorithm is polynomial-time we call it a **polynomial transformation.**

Existence of a polynomial transformation algorithm has a great **significance** for the complexity issues.

Suppose you have (1) a poly-transformation from a (source) problem X to another (target) problem Y,
and (2) Y has a poly algorithm, then
you can solve any instance of the source problem X polynomially, by the following method.
Just transform any instance of X into another instance of Y first, and then use Y's poly-algorithm. Both of these steps are polynomial, and the output (*T/F*) from Y's algorithm is valid for the source instance (of X) as well. Hence, the *True/False* answer for the original instance of X would be obtained in poly-time. This constitutes an indirect poly-algorithm for X, thus making X also belonging to the P-class.

Once again, **note the direction**.
(*You will be amazed with how many practicing computer scientists get confused with this direction*!)

Cook's theorem.

Cook modeled all NP-problems (an infinite set) to an abstract Turing machine. Then he developed a poly-transformation from this machine (i.e., all NP-class problems) to a particular decision problem, namely, the *Boolean Satisfiability* (*SAT*) problem.

Significance of Cook's theorem: if one can find a poly-algorithm for SAT, then by using Cook's poly-transformation one can solve all NP-class problems in poly-time (consequently, P-class = NP-class would be proved).

SAT is the first identified *NP-hard problem*!

Further significance of Cook's theorem: if you find a poly-transformation from SAT to another problem Z, then Z becomes another NP-hard problem. That is, if anyone finds a poly algorithm for Z, then by using your poly-transformation from SAT-to-Z anyone would be able to solve any SAT problem-instance in poly-time, and hence would be able to solve all NP-class problems in poly-time (by Cook's theorem).

These problems, which have a chain of poly-transformation from SAT, are called **NP-hard problems**.

If an NP-hard problem also belongs to the NP-class it is called an NP-complete problem, and
the group of such problems are called **NP-complete problems**.
[DRAW SETS]

Significance of NP-hard problems

As stated before, if one finds any poly-algorithm for any NP-hard problem, then
We would be able to write polynomial algorithm for each of NP-class problems, or
NP-class = P-class will be proved (in other words, poly-algorithms would be found for all NP-class problems).

Unfortunately, **neither** anyone could find any poly algorithm for any NP-hard problem (which would signify that P-class = NP-class),
nor anyone could prove an exponential information-theoretic bound for any NP-complete problem, say L (which would signify that L is in NP but not in P, or in other words that would prove that P-class $\subset$ NP-class). The NP-hard problems are the best candidate for finding such counter-examples. [*There is a claim in Summer 2010 of a lower bound-proof of some NP-hard problem, from IBM research!*]

As a result, when we say a problem X (say, the KSD problem) is NP-complete all we mean is that
IF one finds a poly-alg for X, THEN all the NP-class problems would have poly algorithm.
We also mean, that it is UNLIKELY that there would be any poly-algorithm found for X.

$P \neq NP$ is a mathematical **conjecture** currently (YET to be proved as a theorem, *see above though*).
Based on this conjecture many new results about the complexity-structure of computational problems have been obtained.
Note this very carefully: NP (as yet in history) does NOT stand for "non-polynomial."

[Also, note that NP-complete problems do have solutions, but they are all exponential algorithms, so far! A common student-mistake is to confuse NP-complete problems with unsolvable problems.]

There exist other hierarchies. For example, all problems, which need polynomial memory-space (rather than time) form PSPACE problems. Poly-time algorithm will not need more than poly-space, but the reverse may not be necessarily true. Answer to this question is not known. There exists a chain of poly-transformations from all PSPACE problems to the elements of a subset of it. That subset is called *PSPACE-complete*. PSPACE-complete problems may lie outside NP-class, and may be harder than the NP-complete problems. Example: *Quantified Boolean Formula* (first-order logic) is PSPACE-complete.

Chapter
1

An Overview of C

1.0. Quick History of C

- Developed at Bell Laboratories in the early seventies by Dennis Ritchie.
- Born out of two other languages – BCPL(Basic Control Programming Language) and B.
- C introduced such things as character types, floating point arithmetic, structures, unions and the preprocessor.
- The principal objective was to devise a language that was easy enough to understand to be "high-level" – i.e. understood by general programmers, but low-level enough to be applicable to the writing of systems-level software.
- The language should abstract the details of how the computer achieves its tasks in such a way as to ensure that C could be portable across different types of computers, thus allowing the UNIX operating system to be compiled on other computers with a minimum of re-writing.
- C as a language was in use by 1973, although extra functionality, such as new types, was introduced up until 1980.
- In 1978, Brian Kernighan and Dennis M. Ritchie wrote the seminal work *The C Programming Language,* which is now the standard reference book for C.
- A formal ANSI standard for C was produced in 1989.
- In 1986, a descendant of C, called C++ was developed by Bjarne Stroustrup, which is in wide use today. Many modern languages such as C#, Java and Perl are based on C and C++.
- Using C language scientific, business and system-level applications can be developed easily.

1.1. The Compile Process

You type in your program as C source code, compile it (or try to run it from your programming editor, which will typically compile and build automatically) to turn it into object code, and then build the program using a linker program to create an executable program that contains instructions written in machine code, which can be understood and run on the computer that you are working on.

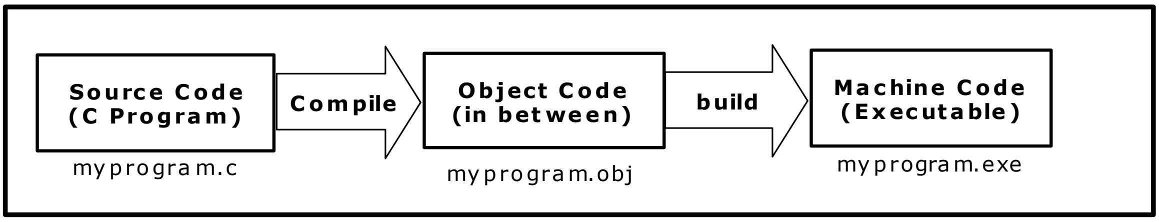


Fig 1.0. Compilation Process

To compile a C program that you have typed in using C/C++ Editor, do one of the following:

- Click on the menu and choose **Compile**.
- Hold down the **Alt** key and press **F9**

To run the program, do one of the following:

- Click on the menu and choose **Run**.
- Press Ctrl and F9 together.

1.2. Guidelines on Structure:

When creating your programs, here are a few guidelines for getting a program that can be easily read and debugged by you (when you come back to it later) and others (who may have to change your program later).

- A well-structured program should indent (i.e. move to the right by pressing the tab key once) all instructions inside each block (i.e. following an opening curly brace – {). Therefore, a block inside another block will be indented twice etc. This helps to show which instructions will be executed, and also helps to line up opening and closing curly braces.

- It is often helpful to separate stages of a program by putting an extra blank line (and maybe a comment explaining what the next stage does) between each stage.

- Another useful technique that helps the readability inside each instruction is to put spaces after commas when listing parameters (i.e. data) to be given to a function. This helps identify where each parameter starts and makes the program easier to read. Spaces before and after brackets, assignments (=) and operators (+ - * / < > == != etc.) can also help make things more readable – but this is up to you.

- Start a program with a **/* comment */** that explains the purpose of the program, and who wrote it. This is useful in case you ever have to work on somebody else's program, and you need to ask them for advice on how it works.

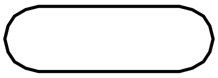
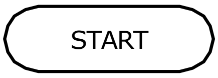
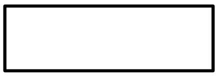
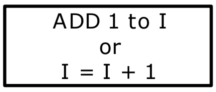
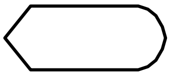
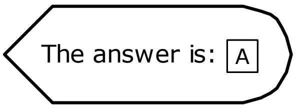
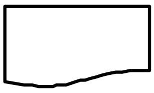
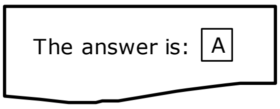
- If you create a new function, put a comment before it showing what data goes in, and what comes out, and the purpose of the function.

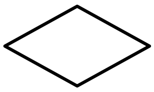
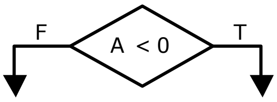
- If a part of your code is quite difficult to follow, or you wish to explain why you have written it the way you have, or even if you wish to say that it could be improved (and how) but you haven't the time to do it better yet, then use a comment.

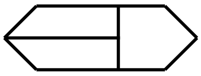
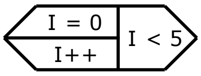
1.3. FLOWCHARTING:

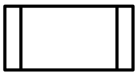
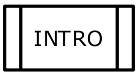
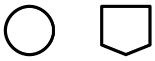
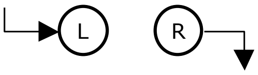
Note: In the examples below, assume that variables **A** and **I** are **integers**

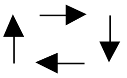
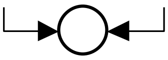
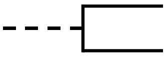
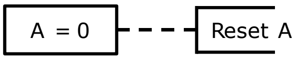
Symbol	**Type of operation/C Code Example**	**Example of symbol usage**

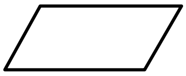
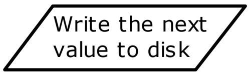
	Terminal Activity - Start, Stop or End {	
	Assignment of a value to a variable, either directly or as the result of a calculation. I = I + 1;	
	Softcopy - screen output to a video display. printf ("The answer is: %d", A);	
	Hardcopy - document output from a printer.	

	Decision based on a condition.
	
	if (A < 0)
	{
	statements;
	}
	else
	{
	statements;
	}
		

	To repeat a statement/s a known number of times.
	
	for (I = 0; I < 5; I++)
	{
	statements;
	}
		

	Sub-routine (function) used to indicate a process, which is defined elsewhere. INTRO (); /* Call Intro*/	
	Connectors: On-page (left) & Off-page (right). Used to either:	

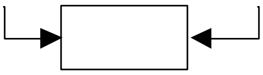
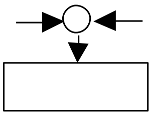
	1. Continue a flowchart at a different place either on or off the same piece of paper. 2. Close a selection branch or loop.	
	Flow of Control Arrows indicating the sequence of steps ("flow of control").	
	Annotation for placing comments in logic. A = 0; /* Reset A */	

	General Input/Output of Data
	fprintf (filename, data);
	printf("Message");
	scanf("%d", &I);
		

1.3.1. General Flowcharting Guidelines

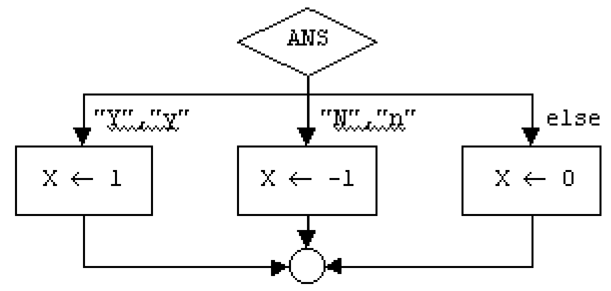
Symbols can be of any size (height or width) necessary to hold the symbol's contents. The shapes of most symbols imply the process. It is redundant to put the word "print" in a hardcopy symbol for example.

Always put an arrowhead on each line connecting symbols to indicate the "flow of control". The only symbols that may receive more than one incoming arrow are connectors. Never enter any other symbols using more than one arrow. If you want to do that, put a connector in front of the symbol and let the multiple arrows enter the connector.

Don't do this:	Instead, do this:
	

The switch statement involves a special use of the diamond symbol. A flowchart for the following switch statement is shown to its right. Notice that the diamond contains only the name of the single variable to be evaluated. The "legs" that exit the decision diamond are each labeled with the unique values from a limited set of possible values for the variable answer, including the "else" (default) option.

```
switch (ANS)
{
     case 'Y':
     case 'y':
          X = 1;
          break;
     case 'N':
     case 'n':
          X =-1;
          break;
     default:
          X = 0;
}
```



1.4. Algorithm

Algorithm is a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time. No matter what the input values may be, an algorithm terminates after executing a finite number of instructions.

We represent an algorithm using a pseudo language that is a combination of the constructs of a programming language together with informal English statements.

The ordered set of instructions required to solve a problem is known as an *algorithm*.

1.5. Using the Turbo C/C++ Editor:

Most of the commands can be activated directly without going through the main menu. These operations are activated by using hot keys and can be used whenever needed.

Hot Key	**Meaning**
F1	On line help
F2	Save
F3	Select a file to load
F4	Execute program until cursor is reached
F5	Zoom window
F6	Switches between windows
F7	Traces into function calls
F8	Trace but skip function calls
F9	Compile and link program
F10	Toggle between main menu and editor
Alt F1	Shows the previous help screen
Alt F3	Close an open file
Alt F6	Switch between watch and message windows

Alt F7	Previous error
Alt F8	Next error
Alt F9	Compiles file to .obj
Alt spacebar	Activates the main menu
Alt C	Compile menu
Alt D	Debug menu
Alt E	Editor
Alt F	File menu
Alt O	Options menu
Alt P	Project menu
Alt R	Run menu
Alt X	Quit TC
Ctrl F1	C Help about the item upon which the cursor is present
Ctrl F2	Reset program
Ctrl F3	Show function call stack
Ctrl F4	Evaluate an expression
Ctrl F7	Set a watch expression
Ctrl F8	Set or clear a break point
Ctrl F9	Execute program

1.5.1. Moving, Deleting and Copying Blocks of Text:

Start of block	CTRL – KB	
End of block	CTRL – KK	
Move block	CTRL – KV	Removes previous defined block and places it at the new location.
Copy block	CTRL – KC	
Print a block	CTRL – KP	
Delete block mark	CTRL - KY	

1.5.2. Deleting Characters, Words and Lines:

To delete entire word to the right of cursor: CTRL-T

Entire line: CTRL-Y

Current cursor position to the end of line: CTRL-QY

1.5.3. Cursor Movement:

HOME	Move cursor to the start of the line
END	Cursor to the end of the line
CTRL-PGUP	Cursor to the top of the file
CTRL-PGDN	Cursor to the bottom of the file
CTRL-HOME	Cursor to the top of the screen
CTRL-END	Cursor to the bottom of the screen
CTRL - ←	Left one word
CTRL - →	Right one word

1.5.4. Moving Blocks of Text to and from Disk Files:

1. Define the Block:

   Start of block: CTRL – KB

   End of Block: CTRL – KK

2. Then type: CTRL - KW

   It asks for file name to move the block

3. To Read a block in: CTRL – KR

   You are prompted for the file name and the contents come to the current cursor location.

1.6. Information Representation:

The two common standards for representing characters:

1. ASCII (American standard code for information interchange).
2. EBCDIC (Extended Binary coded decimal interchange code)

Character	ASCII	EBCDIC
A	65	193
1	49	241
+	43	78

C allows a character constant to be specified as a character enclosed in single quote marks (‘ ’) which the compiler will then translate to a value in the code (ASCII or otherwise).

1.7. Characteristics of C:

We briefly list some of C's characteristics that define the language and also have lead to its popularity as a programming language. Naturally we will be studying many of these aspects throughout the course.

- Small size

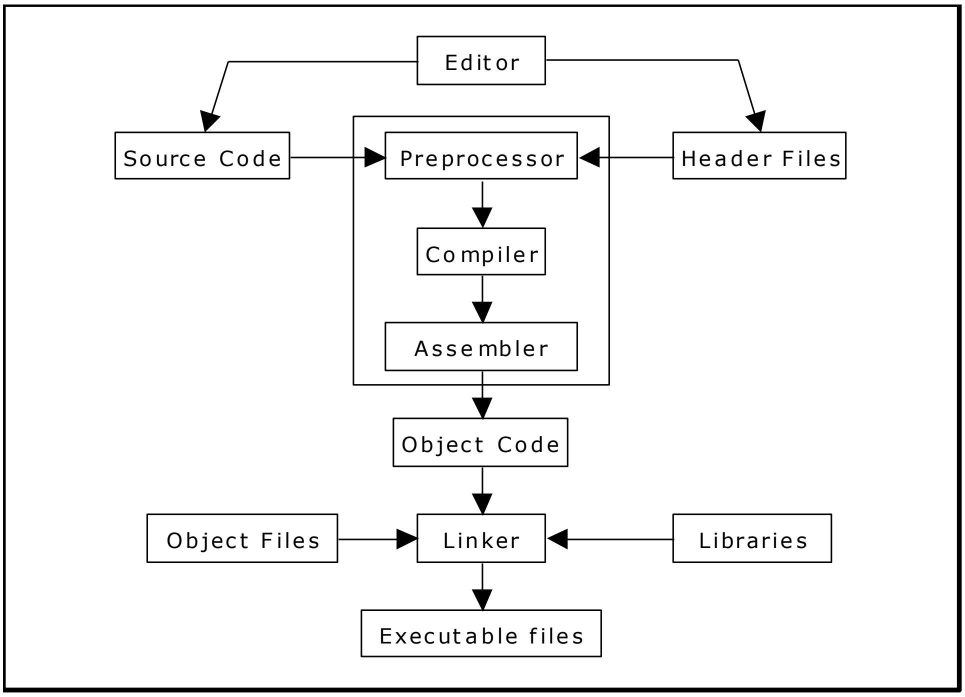
- Extensive use of function calls
- Loose typing
- Structured language
- Low level (Bitwise) programming readily available
- ***Pointer implementation - extensive use of pointers for memory, array, structures and functions*.**

C has now become a widely used professional language for various reasons.

- It has high-level constructs.
- It can handle low-level activities.
- It produces efficient programs.
- It can be compiled on a wide variety of computers.

Its main drawback is that it has poor error detection, which can make it off putting to the beginner. However diligence in this matter can pay off handsomely since having learned the rules of C we can break them. Not many languages allow this. This if done properly and carefully leads to the power of C programming.

1.8. Compiling & Executing C Program:



Stages of Compilation and Execution

C language program is passed through the following steps:

1. C preprocessor: This program accepts C source files as input and produces a file that is passed to the compiler. The preprocessor is responsible for removing comments and interpreting special C language preprocessor

directives, which can be easily identified as they begin with the # symbol. These special preprocessor statements libraries include the following.

a) # include: The preprocessor adds the information about the object code used in the body of the main function. These files are called header files.

b) # define: this directive assigns a symbolic name to a constant. Symbolic names are used to make programs more readable and maintainable.

c) # ifdef: This directive is called conditional compilation and allows the programmer to write programs that are adaptable to different operating environment.

2. C compiler: This program translates the C language source code into the machine assembly language.

3. Assembler: The assembler accepts the C – compiler output and creates object code. If the program does not contain any external function calls, this code is directly executable.

4. Linker: If a source file references library functions, which is defined in other source files, the linker combines these functions with the main() function to create an executable program file.

1.9. Keywords in C:

Here are the 32 keywords:

Flow control (6) – if, else, return, switch, case, default

Loops (5) – for, do, while, break, continue

Common *types* (5) – int, float, double, char, void

For dealing with *structures* (3) – struct, typedef, union

Counting and sizing things (2) – enum, sizeof

Rare but still useful *types* (7) – extern, signed, unsigned, long, short, static, const

Keywords that is undiscouraged and which we NEVER use (1) – goto

We don't use unless we're doing something strange (3) – auto, register, volatile

Total keywords: **32**

1.10. Simple C program structure:

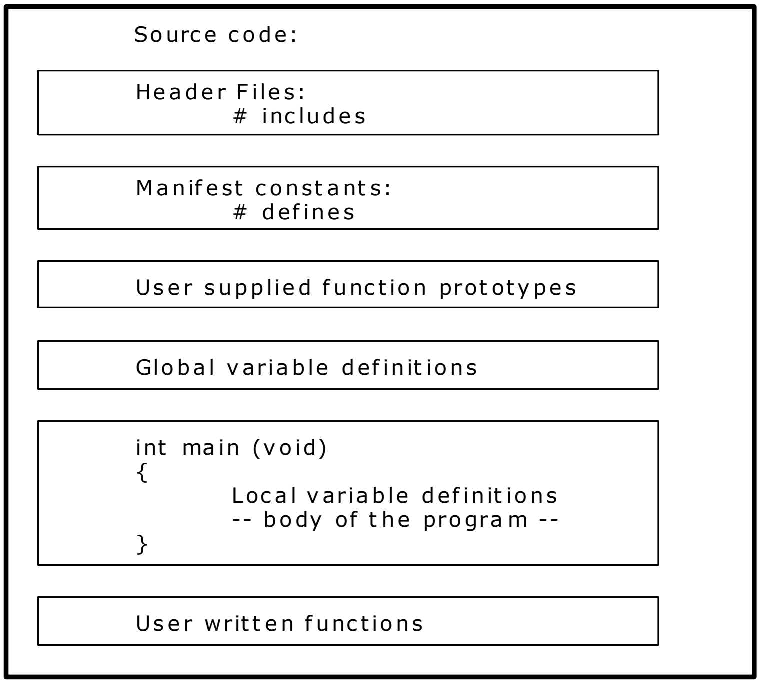
In C, comments begin with the sequence /* and are terminated by */. Any thing that is between the beginning and ending comments symbols is ignored by the compiler.

Example:

/* sample program */

In C, blank lines are permitted and have no effect on the program. A non-trivial C application typically has following general portions on it:

The structure of a C program looks as follows:



```
Source code:

Header Files:
        # includes

Manifest constants:
        # defines

User supplied function prototypes

Global variable definitions

int main (void)
{
        Local variable definitions
        -- body of the program --
}

User written functions
```

Header Files (.h):

Header files contains declaration information for function or constants that are referred in programs. They are used to keep source-file size to a minimum and to reduce the amount of redundant information that must be coded.

# includes:

An include directive tells the preprocessor to include the contents of the specified file at the point in the program. Path names must either be enclosed by double quotes or angle brackets.

Example:

1: # include <stdio.h>

2: # include “mylib.h”

3: # include “mine\include\mylib.h”

In the example (1) above, the <> tells the preprocessor to search for the included file in a special known \include directory or directories.

In the example (2), the double quotes (“ ”) indicate that the current directory should be checked for the header file first. If it is not found, the special directory (or directories) should be checked.

The example (3) is similar, but the named relative directory \mine\include is checked for the header file mylib.h.

Relative paths can also be proceeded by the .\ or ..\ notation; absolute paths always begin with a \.

# defines:

ANSI C allows you to declare *constants.* The # define directive is used to tell the preprocessor to perform a search-and-replace operation.
Example:

```
# define Pi 3.14159

# define Tax-rate 0.0735
```

In the example above, the preprocessor will search through the source file and replace every instance of the token Pi with 3.14159

After performing the search and replace operation, the preprocessor removes the
# define line.

User supplied function prototypes:

Declares the user-written functions actually defined later in the source file. A function prototype is a statement (rather than an entire function declaration) that is placed a head of a calling function in the source code to define the function's label before it is used. A function prototype is simply a reproduction of a function header (the first line of a function declaration) that is terminated with a semi-colon.

Global variable definitions:

Create the Global variables before main ().

The main function:

main () The main function is the entry point in the C program. All C programs begin execution by calling the main () function. When the main function returns, your program terminates execution and control passes back to the operating system.

Every C/C++ program must have one and only one main function.

{ The next line consists of a single curly brace which signifies the start of main () function.

int age ; The first line of code inside function main () is declaration of variables. In C all variables must be declared before they are used.

}   which signifies the end of main () function.

User-written functions:

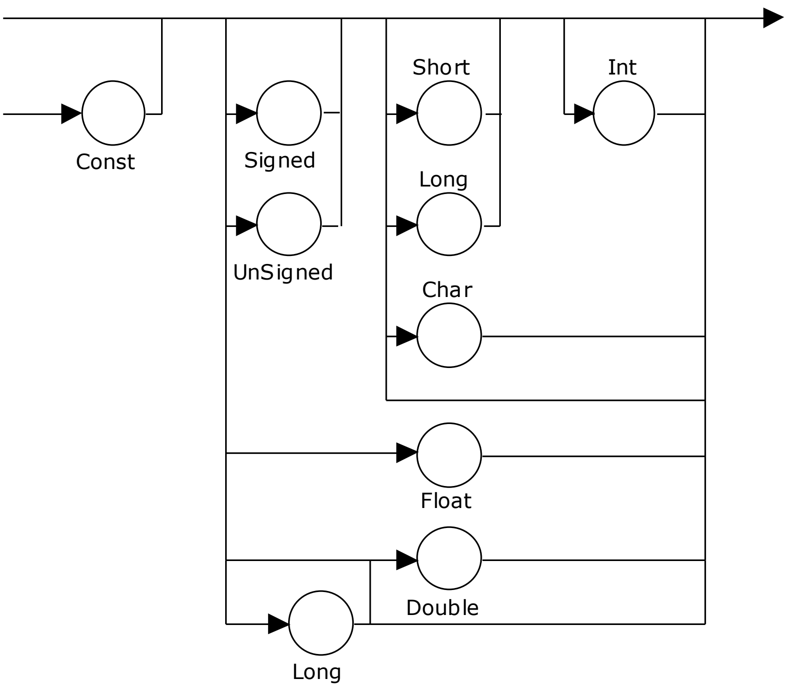
Divide the application into logical procedural units and factor out commonly used code to eliminate repetition.

1.11. Fundamental/Primary Data Types:

C has the following simple data types (16-bit implementation):

Type	**Size**	**Range**	**Precision for real numbers**
char	1 byte	-128 to 127	
unsigned char	1 byte	0 to 255	
signed char	1 byte	-128 to 127	
short int or short	2 bytes	-32,768 to 32,767	
unsigned short or unsigned short int	2 bytes	0 to 65535	
int	2 bytes	-32,768 to 32,767	
unsigned int	2 bytes	0 to 65535	
Long or long int	4 bytes	-2147483648 to 2147483647 (2.1 billion)	
unsigned long or unsigned long int	4 bytes	0 to 4294967295	
float	4 bytes	3.4 E–38 to 3.4 E+38	6 digits of precision
double	8 bytes	1.7 E-308 to 1.7 E+308	15 digits of precision
long double	10 bytes	+3.4 E-4932 to 1.1 E+4932	provides between 16 and 30 decimal places

The syntax of the simple type specifier:



Currently the three char data types are guaranteed to be 1 byte in length, but the other data types are machine architecture dependent. Unsigned can be used with all char and int types.

1.12. Identifier names (Declaring Variables):

The C language defines the names that are used to reference variables, functions, labels, and various other user-defined objects as identifiers.

An identifier can vary from one to several characters. The first character must be a letter or an underscore with subsequent characters being letters, numbers or under-score.

Example:

Correct:	Count, test23, High_balances, _name
In-correct:	1 count, hil there, high..balance

In turbo C, the first 32 characters of an identifier name are significant. If two variables have first 32 characters in common and differ only on the 33rd, TC will not be able to tell then apart. In C++ identifiers may be any length.

In C, upper and lower case characters are treated as different and distinct.

Example:

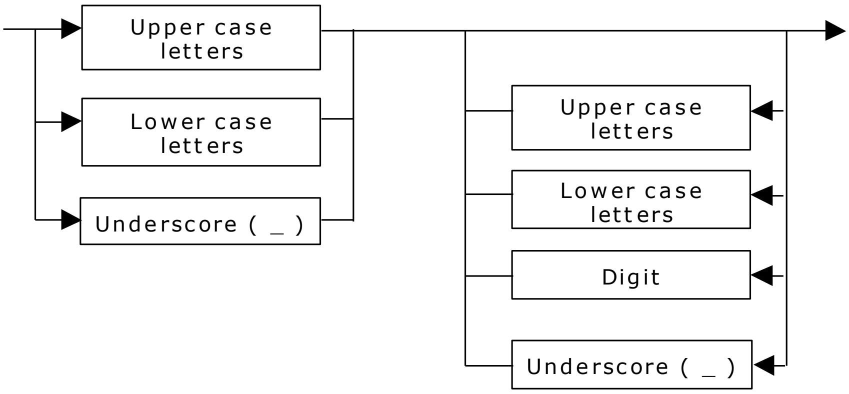
count, Count, COUNT are three separate identifiers.

To declare a variable in C, do:

var_type *list variables*;

Example:

```
int i, j, k;
float x, y, z;
char ch;
```



1.13. Declaring Variables:

Variables are declared in three basic places: inside functions, in the definition of function parameters and outside of the functions. If they are declared inside functions, they are called as *local variables*, if in the definition of functions then formal parameters and out side of all functions, then global variables.

1.13.1. Global variables:

Global variables are known throughout the program. The variables hold their values throughout the programs execution. Global variables are created by declaring them outside of any function.

Global variables are defined above main () in the following way:

```
short number, sum;
int bignumber, bigsum;
char letter;

main()
{

}
```

It is also possible to pre-initialise global variables using the = operator for assignment.

Example:

```
float sum = 0.0;
int bigsum = 0;
char letter =`A';
main()
{

}
```

This is the same as:

```
float sum;
int bigsum;
```

```
char letter;
main()
{
        sum = 0.0;
        bigsum = 0;
        letter =`A';
}
```

is more efficient.

C also allows multiple assignment statements using =, for example:

```
a = b = c = d = 3;
```

which is the same as, but more efficient than:

```
a = 3;
b = 3;
c = 3;
d = 3;
```

This kind of assignment is only possible if all the variable types in the statement are the same.

1.13.2. Local variables:

Variable that are declared inside a function are called “local variables”. These variables are referred to as “automatic variables”. Local variables may be referenced only by statements that are inside the block in which the variables are declared.

Local variables exist only while the block of code in which they are declared is executing, i.e. a local variable is created upon entry into its block & destroyed upon exit.

Example:

Consider the following two functions:

```
void func1 (void)
{
        int x;
        x = 10;
}

void func2 (void)
{
        int x;
        x = -199;
}
```

The integer variable x is declared twice, once in func1 () & once in func2 (). The x in func1 () has no bearing on or relationship to the x in func2 (). This is because each x is only known to the code with in the same block as variable declaration.

1.14. Manifest Constants:

The # define directive is used to tell the preprocessor to perform a search-and-replace operation.

Example: # define Pi 3.14159
# define Tax-rate 0.0735

In the example above, the preprocessor will search through the source file and replace every instance of the token Pi with 3.14159

After performing the search and replace operation, the preprocessor removes the # define line.

The following are two purposes for defining and using manifest constants:

(1) They improve source-code readability
(2) They facilitate program maintenance.

1.15. Constants:

ANSI C allows you to declare *constants*. When you declare a constant it is a bit like a variable declaration except the value cannot be changed.

Syntax for initializing a const data type:

Const data-type variable = initial value.

Example:

```
int const a = 1;
const int a =2;
```

You can declare the const before or after the type. Choose one and stick to it. It is usual to initialise a const with a value as it cannot get a value *any other way*.

The difference between a const variable and a manifest constant is that the # define causes the preprocessor to do a search-and—replace operation through out code. This sprinkles the literal through your code wherever it is used. On the other hand, a const variable allows the compiler to optimize its use. This makes your code run faster (compiler optimization is outside the scope of this course).

1.16. Escape sequences:

C provides special backslash character constants as shown below:

Code	Meaning
\n	New line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\b	Backspace

\’	Single quote
\”	Double quote
\\	Back slash
\a	Alert (bell)
\f	Form feed

1.17. Operators:

There are three general classes of operators: arithmetic, relational and logical and bitwise.

1.17.1. Arithmetic Operators:

Operation	Symbol	Meaning
Add	+	
Subtract	-	
Multiply	*	
Division	/	Remainder lost
Modulus	%	Gives the remainder on integer division, so 7 % 3 is 1.
--	Decrement	
++	Increment	

As well as the standard arithmetic operators (+ - * /) found in most languages, C provides some more operators.

Assignment is = *i.e. i* = 4; ch = `y';

Increment ++, Decrement -- which are more efficient than their long hand equivalents.

For example, X = X + 1 can be written as ++X or as X++. There is however a difference when they are used in expression.

The ++ and -- operators can be either in post-fixed or pre-fixed. A pre-increment operation such as ++a, increments the value of a by 1, before a is used for computation, while a post increment operation such as a++, uses the current value of a in the calculation and then increments the value of a by 1. Consider the following:

```
X = 10;
Y = ++X;
```

In this case, Y will be set to 11 because X is first incremented and then assigned to Y. However if the code had been written as

```
X = 10;
Y = X++;
```

Y would have been set to 10 and then X incremented. In both the cases, X is set to 11; the difference is when it happens.

The % (modulus) operator only works with integers.

Division / is for both integer and float division. So be careful.

The answer to: *x* = 3 / 2 is 1 even if *x* is declared a float!!

RULE: If both arguments of / are integer then do integer division. So make sure you do this. The correct (for division) answer to the above is *x* = 3.0 / 2 or *x*= 3 / 2.0 or (better) *x* = 3.0 / 2.0.

There is also a convenient **shorthand** way to express computations in C.

It is very common to have expressions like: *i* = *i* + 3 or *x* = *x* * (*y* + 2)

This can written in C (generally) in a *shorthand* form like this:

We can rewrite *i* = *i* + 3 as *i* += 3

and *x* = *x* * (*y* + 2) as *x* *= *y* + 2.

NOTE: that *x* *= *y* + 2 means *x* = *x* * (*y* + 2) and NOT *x* = *x* * *y* + 2.

1.17.2. Relational Operators:

The relational operators are used to determine the relationship of one quantity to another. They always return 1 or 0 depending upon the outcome of the test. The relational operators are as follows:

Operator	Action
= =	equal to
! =	not equal to
<	Less than
<=	less than or equal to
>	Greater than
>=	Greater than or equal to

To test for equality is ==

If the values of x and y, are 1 and 2 respectively then the various expressions and their results are:

Expression	**Result**	**Value**
X != 2	False	0
X == 2	False	0
X == 1	True	1
Y != 3	True	1

A warning: Beware of using "=" instead of "= =", such as writing accidentally

```
if (i = j) .....
```

This is a perfectly **LEGAL** C statement (syntactically speaking) which copies the value in "j" into "i", and delivers this value, which will then be interpreted as TRUE if j is non-zero. This is called **assignment by value** -- a key feature of C.

Not equals is: !=

Other operators < (less than), > (grater than), <= (less than or equals), >= (greater than or equals) are as usual.

1.17.3. Logical (Comparison) Operators:

Logical operators are usually used with conditional statements. The three basic logical operators are && for logical AND, || for logical OR and ! for not.

The truth table for the logical operators is shown here using one's and zero's. (the idea of true and false under lies the concepts of relational and logical operators). In C true is any value other than zero, false is zero. Expressions that use relational or logical operators return zero for false and one for true.

P	Q	P && q	P || q	! p
0	0	0	0	1
0	1	0	1	1
1	1	1	1	0
1	0	0	1	0

Example:

(i) x == 6 && y == 7

This while expression will be TRUE (1) if both x equals 6 and y equals 7, and FALSE (0) otherwise.

(ii) x < 5 || x > 8

This whole expression will be TRUE (1) if either x is less than 5 or x is greater than 8 and FALSE (0) other wise.

1.17.4. Bit wise Operators:

The *bit wise* operators of C are summarised in the following table:

Bitwise operators	
&	AND
|	OR
^	XOR

~	One's Compliment
<<	Left shift
>>	Right Shift

The truth table for Bitwise operators AND, OR, and XOR is shown below. The table uses 1 for true and 0 for false.

P	Q	P AND q	P OR q	P XOR q
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

DO NOT confuse & with &&: & is bit wise AND, && logical AND. Similarly for | and ||.

~ is a unary operator: it only operates on one argument to right of the operator. It finds 1's compliment (unary). It translates all the 1 bits into O's and all O's into 1's

Example:

```
12 = 00001100
~12 = 11110011 = 243
```

The shift operators perform appropriate shift by operator on the right to the operator on the left. The right operator must be positive. The vacated bits are filled with zero (*i.e.* when shift operation takes places any bits shifted off are lost).
Example:

X << 2 shifts the bits in X by 2 places to the left.
So:

if X = 00000010 (binary) or 2 (decimal)
then:
X >>= 2 implies X = 00000000 or 0 (decimal)

Also: if X = 00000010 (binary) or 2 (decimal)

X <<= 2 implies X = 00001000 or 8 (decimal)

Therefore a shift left is equivalent to a multiplication by 2.

Similarly, a shift right is equal to division by 2.

NOTE: Shifting is much faster than actual multiplication (*) or division (/) by 2. So if you want fast multiplications or division by 2 *use shifts*.

The bit wise AND operator (&) returns 1 if both the operands are one, otherwise it returns zero. For example, if y = 29 and z = 83, x = y & z the result is:

```
0  0  0  1  1  1  0  1   29 in binary
                                    &
0  1  0  1  0  0  1  1   83 in binary

0  0  0  1  0  0  0  1   Result
```

The bit wise or operator (|) returns 1 if one or more bits have a value of 1, otherwise it returns zero. For example if, y = 29 and z = 83, x = y | z the result is:

```
0  0  0  1  1  1  0  1   29 in binary
                                    |
0  1  0  1  0  0  1  1   83 in binary

0  1  0  1  1  1  1  1   Result
```

The bit wise XOR operator (^) returns 1 if one of the operand is 1 and the other is zero, otherwise if returns zero. For example, if y = 29 and z = 83, x = y ^ z the result is:

```
0  0  0  1  1  1  0  1   29 in binary
                                    ^
0  1  0  1  0  0  1  1   83 in binary

0  1  0  0  1  1  1  0   Result
```

1.17.5. Conditional Operator:

Conditional expression use the operator symbols question mark (?)

(x > 7) ? 2 : 3

What this says is that if x is greater than 7 then the expression value is 2. Otherwise the expression value is 3.

In general, the format of a conditional expression is:

```
    a ? b : c
```

Where, a, b & c can be any C expressions.

Evaluation of this expression begins with the evaluation of the sub-expression ‘a’. If the value of ‘a’ is true then the while condition expression evaluates to the value of the sub-expression ‘b’. If the value of ‘a’ is FALSE then the conditional expression returns the value of the sub-expression ‘C’.

1.18. sizeof Operator:

In situation where you need to incorporate the size of some object into an expression and also for the code to be portable across different machines the size of unary operator will be useful. The size of operator computes the size of any object at compile time. This can be used for dynamic memory allocation.

Usage: sizeof (object)

The object itself can be the name of any sort of variable or the name of a basic type (like int, float, char etc).

Example:

```
        sizeof (char) = 1
        sizeof (int) = 2
        sizeof (float) = 4
        sizeof (double) = 8
```

1.19. Special Operators:

Some of the special operators used in C are listed below. These are reffered as separators or punctuators.

Ampersand (&)
Ellipsis (…)
Brackets ([ ])
Semicolon (;)
Comma (,)
Braces ({ })
Parenthesis (())
Asterick (*)
Hash (#)
Colon (:)

Ampersand:

Ampersand (&) also referred as address of operator usually precedes the identifier name, which indicates the memory allocation (address) of the identifier.

Comma:

Comma (,) operator is used to link the related expressions together. Comma used expressions are linked from left to right and the value of the right most expression is the value of the combined expression. The comma operator has the lowest precedence of all operators. For example:

```
Sum = (x = 12, y = 8, x + y);
```

The result will be sum = 20.

The comma operator is also used to separate variables during declaration. For example:

```
int a, b, c;
```

Asterick:

Asterick (*) also referred as an indirection operator usually precedes the identifier name, which identifies the creation of the pointer operator. It is also used as an unary operator.

Ellipsis:

Ellipsis (…) are three successive periods with no white space in between them. It is used in function prototypes to indicate that this function can have any number of arguments with varying types. For example:

```
void fun (char c, int n, float f, . . . . )
```

The above declaration indicates that fun () is a function that takes at least three arguments, a char, an int and a float in the order specified, but can have any number of additional arguments of any type.

Hash:

Hash (#) also referred as pound sign is used to indicate preprocessor directives, which is discussed in detail already.

Parenthesis:

Parenthesis () also referred as function call operator is used to indicate the opening and closing of function prototypes, function calls, function parameters, etc., Parenthesis are also used to group expressions, and there by changing the order of evaluation of expressions.

Semicolon:

Semicolon (;) is a statement terminator. It is used to end a C statement. All valid C statements must end with a semicolon, which the C compiler interprets as the end of the statement. For example:

```
c = a + b;
b = 5;
```

1.20. Order of Precedence of C Operators:

It is necessary to be careful of the meaning of such expressions as a + b * c.

We may want the effect as either

```
(a + b) * c
    or
a + (b * c)
```

All operators have a priority, and high priority operators are evaluated before lower priority ones. Operators of the same priority are evaluated from left to right, so that:

```
a - b - c
```

is evaluated as: (a - b) - c

as you would expect.

From high priority to low priority the order for all C operators (we have not met all of them yet) is:

<table>
<tr><td><b>Highest</b></td><td>( ) [ ] -> .</td></tr>
<tr><td></td><td>! ~ - (type) * & sizeof ++ --</td></tr>
<tr><td></td><td>* / %</td></tr>
<tr><td></td><td>+ -</td></tr>
<tr><td></td><td><< >></td></tr>
<tr><td></td><td>< <= >= ></td></tr>
<tr><td></td><td>== !=</td></tr>
<tr><td></td><td>&</td></tr>
<tr><td></td><td>^</td></tr>
<tr><td></td><td>|</td></tr>
<tr><td></td><td>&&</td></tr>
<tr><td></td><td>||</td></tr>
<tr><td></td><td>?:</td></tr>
<tr><td></td><td>= += -= *= /= etc.</td></tr>
<tr><td><b>Lowest</b></td><td>, (comma)</td></tr>
</table>

Thus: a < 10 && 2 * b < c is interpreted as: (a < 10) && ((2 * b) < c)

1.21. Pair Matching:

Turbo C++ editor will find the companion delimiters for the following pairs:

{ }, [ ], ( ), < >, /* */, " " and ' '.

To find the matching delimiter, place the cursor on the delimiter you wish to match and press CTRL – Q [for a forward match and CTRL – Q] for a reverse match. The editor will move the cursor to the matching delimiter.

1.22. Casting between types:

Mixing *types* can cause problems. For example:

```
int a = 3;
int b = 2;
float c;
c = b * (a / b);
printf ("2 * (3 / 2) = %f\n", c);
```

doesn't behave as you might expect. Because the first (a/b) is performed with integer arithmetic it gets the answer 1 not 1.5. Therefore the program prints:

```
2 * (3/2) = 2.000
```

The best way round this is what is known as a *cast*. We can *cast* a variable of one type to another type like so:

```
int a = 3;
int b = 2;
float c;
c = b * ( (float) a / b);
```

The (float) a construct tells the compiler to switch the type of variable to be a float. The value of the expression is automatically cast to float . The main use of *casting* is when you have written a routine which takes a variable of one type and you want to call it with a variable of another type. For example, say we have written a power function with a prototype like so:

```
int pow (int n, int m);
```

We might well have a float that we want to find an approximate power of a given number ***n***. Your compiler should complain bitterly about your writing:

```
float n= 4.0;
int squared;
squared= pow (n, 2);
```

The compiler will not like this because it expects ***n*** to be of type int but not float.

However, in this case, we want to tell the compiler that we do know what we're doing and have good reason for passing it a float when it expects an int (whatever that reason might be). Again, a cast can rescue us:

```
float n = 4.0;
int squared;
squared = pow ((int) n, 2);      /* We cast the float down to an int*/
```

IMPORTANT RULE: To move a variable from one type to another then we use a *cast* which has the form (*variable_type*) *variable_name*.

CAUTION: It can be a problem when we *downcast* – that is cast to a type which has less precision than the type we are casting from. For example, if we cast a double to a float we will lose some bits of precision. If we cast an int to a char it is likely to overflow [recall that a char is basically an int which fits into 8 binary bits].

1.23. Console I/O:

Console I/O refers to operations that occur at the keyboard and screen of your computer.

1.23.1. Reading and writing Characters:

The simplest of the console I/O functions are getche (), which reads a character from the keyboard, and putchar (), which prints a character to the screen. The getche () function waits until a key is pressed and then returns its value. The key pressed is also echoed to the screen automatically. The putchar () function will write its character argument to the screen at the current cursor position. The prototypes for getche () and putchar () are shown here:

```
        int getche (void);
        int putchar (int c);
```

The header file for getche () and putchar () is in CONIO.H.
The following programs inputs characters from the keyboard and prints them in reverse case. That is, uppercase prints as lowercase, the lowercase prints as uppercase. The program halts when a period is typed. The header file CTYPE.H is required by the islower() library function, which returns true if its argument is lowercase and false if it is not.

```
# include <stdio.h>
# include <conio.h>
# include <ctype.h>

main(void)
{
        char ch;
        printf (“enter chars, enter a period to stop\n”);
        do
        {
                ch = getche ();
                if ( islower (ch) )
                        putchar (toupper (ch));
                else
                        putchar (tolower (ch));
        } while (ch! = ‘.’);                    /* use a period to stop */
        return 0;
}
```

There are two important variations on getche().

- The first is getchar(), which is the original, UNIX-based character input function.

- ❑ The trouble with getchar() is that it buffers input until a carriage return is entered. The reason for this is that the original UNIX systems line-buffered terminal input, i.e., you had to hit a carriage return for anything you had just typed to actually be sent to the computer.

- ❑ The getchar() function uses the STDIO.H header file.

- A second, more useful, variation on getche() is getch(), which operates precisely like getche () except that the character you type is not echoed to the screen. It uses the CONIO.H header.

1.23.2. Reading and writing Strings:

On the next step, the functions gets() and puts() enable us to read and write strings of characters at the console.

The gets() function reads a string of characters entered at the keyboard and places them at the address pointed to by its argument. We can type characters at the keyboard until we strike a carriage return. The carriage return does not become part of the string; instead a null terminator is placed at the end and gets() returns. Typing mistakes can be corrected prior to striking ENTER. The prototype for gets() is:

```
char* gets (char *str);
```

Where, str is a character array that receives the characters input by the user. Its prototype is found in STDIO.H. The following program reads a string into the array str and prints its length.

```
# include <stdio.h>
# include <string.h>

main(void)
{
        char str[80];
        gets (str);
        printf (“length is %d”, strlen(str));
        return 0;
}
```

The puts() function writes its string argument to the screen followed by a newline. Its prototype is.

```
int puts (char str);
```

It recognizes the same backslash codes as printf(), such as “\t” for tab. It cannot output numbers or do format conversions. Therefore, puts() takes up less space and runs faster than printf(). Hence, the puts() function is often used when it is important to have highly optimized code. The puts() function returns a non negative value if successful, EOF otherwise. The following statement writes “hello” on the screen.

```
puts (“hello”);
```

The puts() function uses the STDIO.H header file.

Basic console I/O functions:

Function	**Operation**
getchar()	Reads a character from the keyboard and waits for carriage return
getche()	Reads a character with echo and does not waits for carriage return
getch()	Reads a character from the keyboard with out echo and not waits for carriage return
Putchar()	Writes a character to the screen
gets()	Reads a string from the keyboard
puts()	Writes a string to the screen

Distinguishion between getchar() and gets() functions:

getchar()	gets()
Used to receive a single character.	Used to receive a single string, white spaces and blanks.
Does not require any argument.	It requires a single argument.

1.23.3. Reading Character Data in a C Program

All data that is entered into a C program by using the scanf function enters the computer through a special storage area in memory called the **standard input buffer** (or **stdin**). A user's keystrokes (including the new line character \n generated when the Enter key is pressed) are stored in this buffer, which can then be read using functions such as scanf. When numbers are read from this area, the function converts the keystrokes stored in the buffer (except for the \n) into the type of data specified by the control string (such as "%f") and stores it in the memory location indicated by the second parameter in the scanf function call (the variable's address). The \n remains in the buffer. This can cause a problem if the next scanf statement is intended to read a *character* from the buffer. The program will mistakenly read the remaining \n as the character pressed by the user and then proceed to the next statement, never even allowing the user to enter a character of their own.

You can solve this problem when writing C statements to read *character* data from the keyboard by adding a call to a special function named **fflush** that clears all characters (including \n's) from the given input buffer. The statement would be place ahead of each statement in your program used to input characters, such as:

```
fflush(stdin); scanf("%c", &A);
        or
fflush(stdin); A=getchar();
```

1.24. Formatted Console I/O (printf() and scanf()):

The pre-written function printf() is used to output most types of data to the screen and to other devices such as disks. The C statement to display the word "Hello" would be:

```
printf("Hello");
```

The printf () function can be found in the stdio.h header file.

1.24.1. Displaying Prompts:

The printf() function can be used to display "prompts" (messages urging a user to enter some data) as in:

```
printf ("How old are you? ");
```

When executed, this statement would leave the cursor on the same line as the prompt and allow the user to enter a response following it on the same line. A space was included at the end of the prompt (before the closing quote mark) to separate the upcoming response from the prompt and make the prompt easier to read during data entry.

1.24.2. Carriage Returns:

If you want a carriage return to occur after displaying a message, simply include the special *escape sequence* \n at the end of the of the message before the terminating quote mark, as in the statement.

```
printf ("This is my program.\n");
```

1.24.3. Conversion Specifiers:

All data output by the printf() function is actually produced as a string of characters (one symbol after another). This is because display screens are character-based devices. In order to send any other type of data (such as integers and floats) to the screen, you must add special symbols called conversion specifiers to the output command to convert data from its stored data format into a string of characters. The C statement to display the floating point number 123.456 would be:

```
printf ("%f",123.456);
```

The "%f" is a conversion specifier. It tells the printf() function to convert the floating point data into a string of characters so that it can be sent to the screen. When outputting non-string data with the printf() function you must include two parameters (items of data) within the parentheses following printf(). The first parameter is a quoted control string containing the appropriate conversion specifier for the type of non-string data being displayed *and optionally* any text that should be displayed with it. The second parameter (following a comma) should be the value or variable containing the value. The C statement to display the floating point number 123.456 preceded by the string "The answer is: " and followed by a carriage return would be:

```
printf ("The answer is: %f\n",123.456);
```

Notice that the value does not get typed inside of the quoted control string, but rather as a separate item following it and separated by a comma. The conversion specifier acts as a place holder for the data within the output string.

The following table lists the most common conversion specifiers and the types of data that they convert:

Specifier	**Data Type**
%c	char
%f	float
%d or %i	signed int (decimal)

%h	short int
%p	Pointer (Address)
%s	String of Characters
Qualified Data Types	
%lf	long float or double
%o	unsigned int (octal)
%u	unsigned int (decimal)
%x	unsigned int (hexadecimal)
%X	Unsigned Hexadecimal (Upper Case)
%e	Scientific Notation (Lower case e)
%E	Scientific Notation (Upper case E)
%g	Uses %e or %f which ever is shorter
%ho	short unsigned int (octal)
%hu	short unsigned int (decimal)
%hx	short unsigned int (hexadecimal)
%lo	long unsigned int (octal)
%lu	long unsigned int (decimal)
%lx	long unsigned int (hexadecimal)
%Lf	long double
%n	The associated argument is an integer pointer into which the number of characters written so far is placed.
%%	Prints a % sign

1.24.4. Output Field Width and Rounding:

When displaying the contents of a variable, we seldom know what the value will be. And yet, we can still control the format of the output field (area), including the:

amount of space provided for output (referred to as the output's "*field width*")

alignment of output (left or right) within a specified field and rounding *of floating point numbers* to a fixed number of places right of the decimal point

Output formatting is used to define specific alignment and rounding of output, and can be performed in the printf() function by including information within the conversion specifier. For example, to display the floating point number 123.456 right-aligned in an eight character wide output field and rounded to two decimal places, you would expand basic conversion specifier from "%f" to "%8.2f". The revised statement would read:

```
printf ("The answer is:%8.2f\n",123.456);
```

The 8 would indicate the width, and the .2 would indicating the rounding. *Keep in mind that the addition of these specifiers as no effect on the value itself, only on the appearance of the output characters.*

The value 123.456 in the statement above could be replaced by a variable or a symbolic constant of the same data type. The 8 in the statement above specifies the width of the output field. If you include a width in the conversion specifier, the function will attempt to display the number in that width, aligning it against the rightmost character position. Unused positions will appear as blank spaces (padding) on the left of the output field. If you want the value to be left-aligned within the field, precede the width with a minus sign (-). If no width is specified, the number will be displayed using only as many characters as are necessary without padding. When a values is too large to fit in a specified field width, the function will expand the field to use as many characters as necessary.

The .2 in the statement above specifies the decimal precision (i.e., the number of place that you want to appear to the right of the decimal point) *and would be used only in situations where you are outputting floating point values*. The function will round the output to use the specified number of places (adding zeros to the end if necessary). Any specified field width *includes* the decimal point and the digits to its right. The default decimal precision (if none is specified) is 6 places.

In situations where you want floating point values displayed in scientific notation, formatting also is used to define specific alignment and rounding of output in the printf function by including information within the conversion specifier %e. For example, to display the floating point number 123.456 in scientific notation in a twelve character wide output field with its mantissa (significant digits) rounded to two decimal places, you would expand basic conversion specifier from "%e" to "%12.2e". The resulting output would be:

```
The answer is: 1.23e+002
```

Notice that values displayed in scientific notation always place the decimal point after the first significant digit and use the exponent (digits shown following the letter e) to express the power of the number. The C statement to produce the output above would be:

```
printf ("The answer is:%12.2e\n",123.456);
```

The 12 would indicate the *overall* field width (following any message) *including* the decimal point and the exponential information. The .2 would indicating the rounding of the digits following the decimal point. The exponential information is always expressed in 5 characters: the first one an "e", then a sign (- or +), followed by the power in three characters (with leading zeros if needed).

Review the following examples of formatted output statements paying close attention to the format of the resulting output beside them. Each box indicates one character position on the screen. All output starts in the leftmost box, although some output might be "padded" with blank spaces to align it to the right edge of the field. "X"'s indicated unused positions.

C command using printf () with various conversion specifiers:	Output Produced on Screen																			
	Position:									1	1	1	1	1	1	1	1	1	1	2
	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0
printf ("%3c",'A');			**A**	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
printf ("%-3c",'A');	**A**			X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

printf ("%8s","ABCD");					A	B	C	D	X	X	X	X	X	X	X	X	X	X	X	X
printf ("%d",52);	5	2	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
printf ("%8d",52);							5	2	X	X	X	X	X	X	X	X	X	X	X	X
printf ("%-8d",52);	5	2							X	X	X	X	X	X	X	X	X	X	X	X
printf ("%f",123.456);	1	2	3	.	4	5	6	0	0	0	X	X	X	X	X	X	X	X	X	X
printf ("%10f",123.456);	1	2	3	.	4	5	6	0	0	0	X	X	X	X	X	X	X	X	X	X
printf ("%10.2f",123.456);					1	2	3	.	4	6	X	X	X	X	X	X	X	X	X	X
printf ("%-10.2f",123.456);	1	2	3	.	4	6					X	X	X	X	X	X	X	X	X	X
printf ("%.2f",123.456);	1	2	3	.	4	6	X	X	X	X	X	X	X	X	X	X	X	X	X	X
printf ("%10.3f",-45.8);				-	4	5	.	8	0	0	X	X	X	X	X	X	X	X	X	X
printf ("%10f",0.00085);			0	.	0	0	0	8	5	0	X	X	X	X	X	X	X	X	X	X
printf ("%10.2e",123.89);		1	.	2	4	e	+	0	0	2	X	X	X	X	X	X	X	X	X	X

Distinguishing between printf() and puts() functions:

puts()	printf()
They can display only one string at a time.	They can display any number of characters, integers or strings a time.
All data types of considered as characters.	Each data type is considered separately depending upon the conversion specifications.

1.23. scanf () function:

scanf() is the general purpose console input routine. It can read all the built-in data types of automatically convert numbers into the proper internal format. It is much like the reverse of printf(). The f in scanf stands for formatted.

```
# include <stdio.h>
int main()
{
        int num;
        printf (“Enter a number: “);
        scanf(“%d”, &num);
        printf(“The number you have entered was %d\n”, num);
        return 0;
}
```

The above program requires a variable in which to store the value for num. The declaration, int num; provides a temporary storage area called num that has a data type of integer (whole number). The scanf function requires a Format String, which is provided by the %d character pair. The percent sign is an introducer and is followed by a conversion character, d, which specifies that the input is to be an integer. The input is stored in the variable, num. The scanf function requires the address of the variable, and this is achieved by prefixing the variable name with an ampersand (eg. &num).

The printf statement also uses the format string to relay the information. The printf function does not require the address of the variable and so the ampersand is not required. The prototype for scanf() is in STDIO.H.

The format specifiers for scanf () are as follows:

Code	**Meaning**
%c	Read a single character
%d	Read a decimal integer
%i	Read a decimal integer, hexa decimal or octal integer
%h	Read a short integer
%e	Read a floating-point number
%f	Read a floating-point number
%g	Read a floating-point number
%o	Read an octal number
%s	Read a string
%x	Read a hexadecimal number
%p	Read a pointer
%n	Receives an integer value equal to the number of characters read so far
%u	Read an unsigned integer
%[..]	Scan for a string of words

Distinguishing between scanf() and gets() functions:

scanf()	**gets()**
Strings with spaces cannot be accessed until ENTER key is pressed.	Strings with any number of spaces can be accessed.
All data types can be accessed.	Only character data type can be accessed.
Spaces and tabs are not acceptable as a part of the input string.	Spaces and tabs are perfectly acceptable of the input string as a part.
Any number of characters, integers.	Only one string can be received at a time. Strings, floats can be received at a time.

Key Terms & Concepts

The list of terms below is provided to supplement or elaborate on the boldfaced terms and definitions provided in the course textbook. Students are advised to also review the textbook to develop a fuller understanding of these and other important terms related.

Text is a form of data consisting of characters such as letters, numerals, and punctuation.
ASCII **is the American Standard Code for Information Interchange; a language for representing text on computers.**

Binary is a word with two meanings in programming. The first and most common meaning relates to the numbering system known as "Base-2" in which all values are represented using only "two (bi) numerals (nary) ". The second meaning relates to the use of operators such as the minus sign (-) in a formula. When the symbol appears between *two* items (such as the values 5 and 2 in the formula 5-2), it is referred to as a "*binary* operator". When the symbol appears preceding only *one* item (such as the value 2 in the formula -2), it is referred to as a "*unary* operator".

Bits are binary digits (0 or 1) that represent the settings of individual switches or circuits inside a computer (OFF or ON). Data is represented as standardized patterns of bits.

A **byte** is the unit of measure of storage space used to represent a single character.
There are typically 8 bits in a byte.

Decimal is another word with two meanings in programming. The first and most common meaning relates to the numbering system known as "Base-10" in which all values are represented using ten digits (0-9). The second meaning relates to the use of a decimal point when writing numbers with fractional parts, such as one half in 8.5 or one tenth in 4.1. Numbers written containing a decimal point are often referred to as "decimal numbers", however this is technically incorrect, as it implies that the numbers are also written using the Base-10 numbering system (which may not be the case). A more precise way to talk about numbers that contain fractional parts is to call them "floating point numbers" or more simply "floats".

Bug: A bug is an error in a program. These can be split into syntax errors and semantic errors. A Syntax error is where the language rules of the C programming language have been broken, and can be easily picked up by the C Compiler – e.g. a missing semi-colon. A semantic error is harder-to find, and refers to an error where the syntax of the program is correct, but the meaning of the program is not what was intended – e.g. `int three_cubed=3*3;` when you actually meant `int three_cubed=3*3*3;`

Breakpoint: This is a line of source code, marked in some way, where the program execution stops temporarily for you to examine what is happening – e.g. the contents of variables etc. Once stopped, you can usually step through the execution of the program line-by-line to see how variables change, and to follow the flow of execution in the program.

Build: This is the step following compilation. It takes one or a number of object code files generated by one or more compiled programs, and links them together to form an executable file that comprises instructions that are understood by the computer on which the application is to run.

Execution: This means running a program – i.e. starting the executable file going so that we can see the program working.

Debugging: Debugging refers to the task of removing bugs from a program. This is often attained with the help of tools in the IDE, such as creating breakpoints, stepping through line-by-line etc.

Executable: The name given to a file on your computer that contains instructions written in machine code. These are typically applications that do some task on your computer – e.g. *Microsoft Word* has an executable file (winword.exe) that you can execute in order to start the application.

IDE: Integrated Development Environment. The application used to edit your source code, perform debugging tasks, compile and build your application, and organize your projects. It acts as a complete set of tools in which to create your C programs.

Program: This is a generic name for an application on your computer that performs a task. An application such as *Microsoft Word* may be referred to as a program. Sometimes also used to refer rather vaguely as the source code that, when compiled, will generate the executable application.

Run: When we refer to running a program, we are referring to the execution (starting) of the executable version of the program so that we can see the program working.

Machine Code: The language that each type of computer understands. All PCs understand instructions based on microprocessors such as the Intel 8086 through to the latest Pentium processors, Macintoshes understand instructions based on the 68000 and upwards processors, and Sun servers understand instructions based on Sun's Sparc processor.

Syntax: The syntax refers to the structure of a C program. It refers to the rules that determine what is a correct C program. A Syntax Error is where a part of the program breaks these rules in some way – e.g. a semi-colon omitted from the end of a C instruction.

A **compiler** is a program that translates high-level language (such as C) instructions into machine language instructions. It also often provides an editor for writing the program code in the first place. Compilers often are packaged with other software known as programming environments that include features for testing and debugging programs.

Source Code: The C Program that you type in using a text editor, containing a list of C instructions that describe the tasks you wish to perform. The source code is typically fairly independent of the computer that you write it on – the compile and build stages convert it to a form that is understood by your computer.

Object code is machine language that resulted from a compiling operation being performed. Files containing C object code typically end with a filename extension of ".obj".

Text Editor: An application program, rather like a word-processor, that you use to type out your C source code instructions to create a C program. These can be as simple as the notepad text editor supplied with Microsoft Windows, to the sophisticated editor that comes part of an IDE, typically including features such as bookmarks, integrated debugging, and syntax highlighting.

A **header file** is a file containing useful blocks of pre-written C source code that can be added to your source code using the "# include" compiler directive. Header files typically end with a filename extension of ".h".

White space is any character or group of characters that a program normally interprets as a separator of text, including: spaces ( ), form feeds (\f), new-lines (\n), carriage returns (\r), horizontal tabs (\t), and vertical tabs (\v). In C source code, all of these characters are interpreted as the same unless they are quoted. In other words, one space is interpreted the same as three blank lines.

The **declaration** part of a program defines its identifiers, such as: symbolic constants and variable names and data types.

The **body of a program** contains the statements that represent the steps in the main algorithm.

A **constant** (also called as literal) is an actual piece of data (also often referred to as a "value") such as the number 5 or the character 'A'.

Symbolic **Constants** are aliases (nicknames) used when coding programs in place of values that are expected to be the same during each execution of a program, but might need to be changed someday. Symbolic constants are defined within C source code using the "#define" compiler directive at the top of the program to allow easy location and revision later.

Variable: A temporary storage location in a C program, used to hold data for calculations (or other uses) further in on in the program. For identification purposes, a variable is given a name, and to ensure we hold the right sort of information in the variable, we give it a type (e.g. int for whole numbers, float for decimal numbers). An example would be **int count, total;** or **char name[21];** or **float wage_cost, total_pay;**

A **keyword** is a reserved word within a program that can NOT be redefined by the programmer.

Identifiers are labels used to represent such items such as: constants, variables, and functions. Identifiers are case-sensitive in C, meaning that upper and lowercase appearances of the same name are treated as being different identifiers. Identifiers that are defined within a function (see below) are local to that function, meaning that they will not be recognized outside of it. Identifiers that are defined outside *of* all functions are global, meaning that they will be recognized within all functions.

Addresses are numeric designations (like the numbers on mailboxes) that distinguish one data storage location from another. Prior to the use of identifiers, programmers had to remember the address of stored data rather than its identifier. In C, addresses are referred to by preceding an identifier with an ampersand symbol (&) as in &X which refers to the address of storage location X as opposed to its contents.

Integer data is a numeric type of data involving a whole number (i.e. it CANNOT HAVE a fractional portion). An example of an integer constant is 5 (written without a decimal point). The most common type identifier (reserved word used in a declaration) for integer data is int.

Floating point data is a numeric type of data that CAN HAVE a fractional portion. Mathematicians call such data a real number. An example of a floating point **constant is** 12.567 (written without a decimal point). The most common type identifiers for floating point data are float (for typical numbers) and double (for numbers such as 1.2345678901234E+205 that involve extreme precision or magnitude).

Character data is a type of data that involves only a single symbol such as: 'A', '4', '!', or ' ' (a blank space). The type identifier for character data is char.

String data is a type of data involving multiple symbols such as words or sentences. The C language stores strings as a collection of separate *characters* (see above).

Boolean data is a logical type of data involving only two values: True or False. The identifier used in C to declare Boolean data is bool.

A **problem statement** is program documentation that defines the purpose and restrictions of a program in sufficient detail that the program can be analyzed and designed without more facts.

A sample **softcopy** is program documentation that precisely demonstrates all video (or other intangible) output required of a program.

A sample **hardcopy** is program documentation that precisely demonstrates all printed (tangible) output required of a program.

A **structure diagram** is graphic documentation that helps to describe the hierarchical relationship between a module and its various sub-modules.

An **algorithm** is a finite list of steps to follow in solving a well-defined task, written in the language of the user (i.e. in human terms).

Pseudocode is an algorithm written in English, but as clearly stated logical items that can be easily translated by a programmer into C or other high-level programming languages.

A **desk check** is a manual test of a program algorithm that is performed prior to writing the source code. It must follow the algorithm exactly and typically produces two items of documentation: the tracing chart showing what values are stored in memory during program execution, and any test outputs (softcopy and hardcopy) showing that the algorithm will produce the output specified earlier in the samples.

A **logic error** is caused by a mistake in the steps used to design the program algorithm.

A **syntax error** is caused by a grammatical error in the language of the source code.

A **run-time error** occurs when a program encounters commands it cannot execute.

Comments can be included within C source code, enclosed in the symbols /* and */. The inclusion of comments in source code neither causes the program to run more slowly, nor causes the object code to take up more space, since comments are not translated.

Blank spaces in C code act as separators and are not allowed in an identifier. Multiple blank spaces are treated the same as one, except in string constants (text inside of double quotes).

Character constants must be enclosed in single quote marks (').

String constants must be enclosed in double quote marks (").

Escape sequences are special strings typed within C output statements to produce characters that would otherwise be interpreted as having special meaning to the compiler.

Semicolon (**;**) Each statement is C is normally terminated with a semi-colon (;) except for: include statements.

A **compiler directive** (or preprocessor directive) is an instruction in a C source code file that is used to give commands to the compiler about how the compilation should be performed (as distinguished from C language statements that will be translated into machine code). Compiler directives are not *statements*, therefore they are not terminated with semi-colons. Compiler directives are written starting with a # symbol immediately in front of (touching) the command word, such as #include <stdio.h>

include is a compiler directive in C that is used to indicate that a unit of pre-defined program code (such as the header file stdio.h) should be linked to your program when it is compiled.

define is a compiler directive in C that is used to indicate that a symbolic constant is being used in your program in place of a constant value and that the value should be used in place of the symbolic constant when translating the source code into machine language.

{ (open brace) is the symbol used in C to start a group of executable statements.

} (close brace) is the symbol used in C to end a group of executable statements.

printf is the name of a function in C that will display formatted output on the screen.

scanf is the name of a function in C that will store input from a keyboard into a variable and then advance the cursor to the next line when the user presses the Enter key.

Assignment is the action of storing a value in a memory location and is accomplished using the symbol =.

An **expression** is another name for a formula. Expressions consist of operands (such as constants or variables) and operators (such as + or -). Operators that involve data from *only one* operand are called unary operators. In the statement X = -5; the minus sign acts as a unary operator and operates on a single operand (the constant 5). Operators that involve data from *two* operands are called binary operators. In the statement X = A/B; the slash acts as a binary operator and operates on a pair of operands (the variables A and B).

An **arithmetic expression** is one which uses numeric operands (such as constants or variables) and mathematical operations (such as addition, subtraction, multiplication, or division) to produce a numeric result. An example of an arithmetic expression is X+3.

Order of Precedence is a term used to describe which operations precede others when groups of operators appear in an expression. For example, C compilers see the expression A+B/C as A+(B/C) as opposed to (A+B)/C. The division operation will preceed the addition because division has a higher order of precedence.

Casting is the process of converting data from one data type to another. For example, if you try to divide two integer variables A and B and store the result in a floating point variable C, the result will have any decimal fraction truncated (chopped-off) because both operands are integers. To prevent this, *cast* either of the operands as floats *before* the division operation is performed, as in either of the following examples:

```
C = (float) A / B;
C = A / (float) B;
```

Trying to cast the result instead of the operands would be pointless because the truncation would have already taken place. So it would be ineffective to try:

```
C = (float) (A / B);
```

Top-Down Design is an analysis method in which a major task is sub-divided into smaller, more manageable, tasks called functions (see definition below). Each sub-task is then treated as a completely new analysis project. These sub-tasks may be sufficiently large and complex that they also require sub-division, and so on. The document that analysts use to represent their thinking related to this activity is referred to as a structure diagram. For more information and an example of such a diagram, see the web notes on Analysis & coding of a task involving multiple functions.

Boolean data is a logical type of data with only two values: True and False. Boolean data can be represented in two ways in C. When you want to store Boolean data for fututre use, a variable can be declared to have data type of _Bool (notice the leading underscore and capitalization). A common alternative approach is to represent the true and false values using the integer values of 1 for true and 0 for false. Many functions in C use the latter approach and return a 1 or a 0 to indicate if a condition is true or false. In C, *any* non-zero integer value is interpreted as true.

Ordinal data is a type of data in which all of the values within the set are known and in a predictable order. Ordinal data types include: all integer data types and char data, but <u>not</u> floating point or string data.

Relational operators are those used to evaluate the relationships between items of data, including:

== for Equal to, the opposite of which is written in C as != (or not equal to).
> - Greater than, the opposite of which is written in C as <= (less than or equal to).

< - Less than, the opposite of which is written in C as >= (greater than or equal to).

Relational expressions (also known as relational tests) describe conditions using formulae such as X==A+B that compare items of data (possibly including constants, symbolic constants, variables or arithmetic expressions) and produce a boolean (true or false) result.

Logical operators such as && for "and", || for "or" and ! for "not" are those used to combine conditions into logical expressions such as (X==0 && Y>10) to produce a single boolean result.

A **condition** is any expression that produces a boolean result.

Branching is the act of breaking out of the normal sequence of steps in an algorithm to allow an alternative process to be performed or to allow the repetition of process(es).

Chapter 2

Control Statements, Arrays and Strings

2.0. Control Statements:

This deals with the various methods that C can control the *flow* of logic in a program. Control statements can be classified as un-conditional and conditional branch statements and loop or iterative statements. The Branch type includes:

1. Un-conditional:

- goto
- break
- return
- continue

2. Conditional:

- if
- if – else
- Nested if
- switch case statement

3. Loop or iterative:

- for loop
- while loop
- do-while loop

2.1. Conditional Statements:

Sometimes we want a program to select an action from two or more alternatives. This requires a deviation from the basic sequential order of statement execution. Such programs must contain two or more statements that might *be* executed, but have some way to select only one of the listed options each time the program is run. This is known as conditional execution.

2.1.1. if statement:

Statement or set of statements can be conditionally executed using if statement. Here, logical condition is tested which, may either true or false. If the logical test is true (non zero value) the statement that immediately follows if is executed. If the logical condition is false the control transfers to the next executable statement.

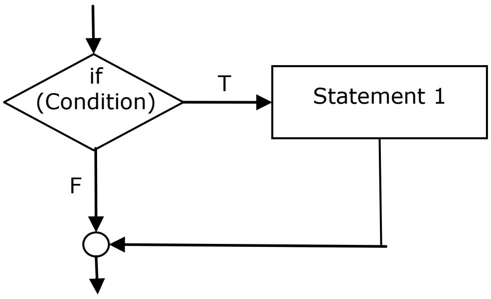
The general syntax of simple **if** statement is:

```
if (condition)
      statement_to_execute_if_condition_is_true;
```

or

```
if (condition)
{
      statement 1;
      statement 2;
        _ _ _ _;
}
```

Flowchart Segment:



2.1.2. if – else statement:

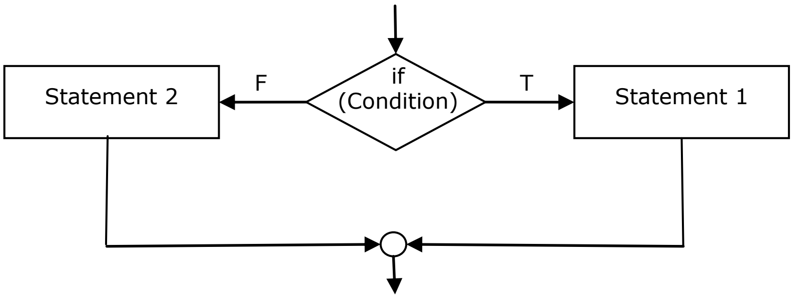
The if statement is used to execute only one action. If there are two statements to be executed alternatively, then if-else statement is used. The if-else statement is a two way branching. The general syntax of simple **if - else** statement is:

```
if (condition)
            statement_to_execute_if_condition_is_true;
else
            statement_to_execute_if_condition_is_false;
```

Where, *statement* may be a single statement, a block, or nothing, and the else statement is optional. The conditional statement produces a scalar result, i.e., an integer, character or floating point type.

It is important to remember that an if statement in C can execute only one statement on each branch (T or F). If we desire that multiple statements be executed on a branch, we must **block** them inside of a **{** and **}** pair to make them a single **compound statement**. Thus, the C code for the flowchart segment above would be:

Flowchart Segment:



Example:

```
main()
{
          int num;
          printf(" Enter a number : ");
          scanf("%d",&num);
          if (num % 2  == 0)
                    printf(" Even Number ");
          else
                    printf(" Odd Number ");
}
```

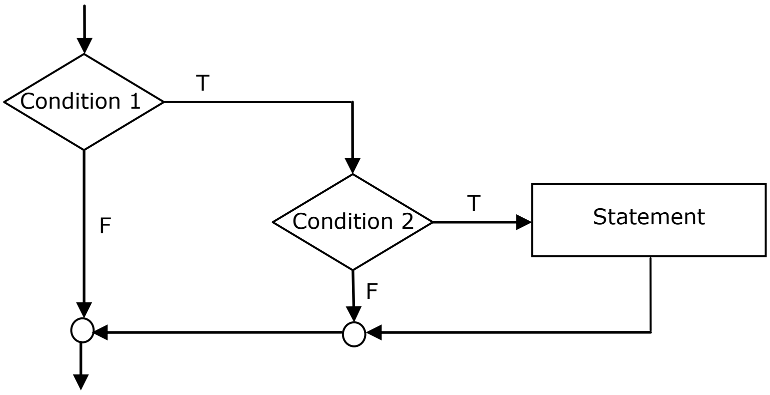
2.1.3. Nested if statement:

The ANSI standard specifies that 15 levels of nesting must be supported. In C, an else statement always refers to the nearest if statement in the same block and not already associated with if.

Example:

```
main()
{
          int num;
          printf(" Enter a number : ");
          scanf("%d",&num);
          if( num > 0 )
          {
                    if( num % 2  == 0)
                              printf("Even Number");
                    else
                              printf("Odd Number");
          }
          else
          {
                    if( num < 0 )
                              printf("Negative Number");
                    else
                              printf(" Number is Zero");
          }
}
```

Flowchart Segment:



2.1.4. if-else-if Ladder:

When faced with a situation in which a program must select from *many* processing alternatives based on the value of a single variable, an analyst must expand his or her use of the basic selection structure beyond the standard two processing branches offered by the `if` statement to allow for multiple branches. One solution to this is to use an approach called **nesting** in which one (or both) branch(es) of a selection contain another selection. This approach is applied to each branch of an algorithm until enough additional branches have been created to handle each alternative. The general syntax of a nested if statement is:

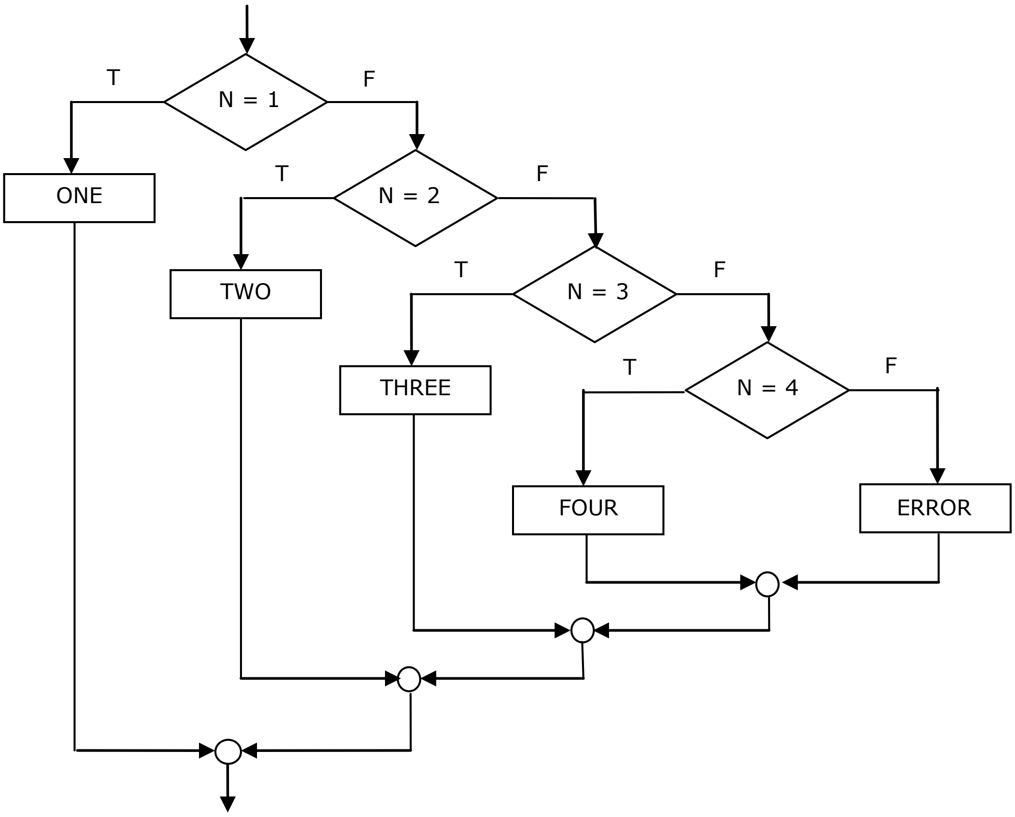
```
if (expression)
        statement1

else if (expression)
        statement2
        ..
        ..
else
        statement3
```

Example:

```
#include <stdio.h>
void main (void)
{
        int N;                         /* Menu Choice */
        printf ("MENU OF TERMS\n\n");
        printf ("1. Single\n");
        printf ("2. Double\n");
        printf ("3. Triple\n");
        printf ("4. Quadruple\n\n");
        printf ("Enter the numbe (1-4): ");
        scanf ("%d", &N);
        if (N == 1) printf ("one");
                else if (N == 2) printf ("two");
                        else if (N == 3) printf ("three");
                                else if (N == 4) printf ("four");
                                        else printf ("ERROR");

}
```



2.2. The ? : operator (ternary):

The ? (*ternary condition*) operator is a more efficient form for expressing simple if statements. It has the following form:

expression$_1$? *expression*$_2$: *expression*$_3$

It simply states as:

if *expression*$_1$ **then** *expression*$_2$ **else** *expression*$_3$

Example:

Assign the maximum of a and b to z:

```
main()
{
        int a,b,z;
        printf("\n Enter  a and b ");
        scanf("%d%d",&a,&b);
        z = (a > b) ? a : b;
        printf("Maximum number: %d", z);
}
```

which is the same as:

```
        if (a > b)
                z = a;
        else
                z = b;
```

2.3. The switch case statement:

The switch-case statement is used when an expression's value is to be checked against several values. If a match takes place, the appropriate action is taken. The general form of switch case statement is:

```
switch (expression)
{
        case constant$_1$:
        statement;
        break;

        case constant$_2$:
        statement;
        break;

        default:
        statement;
        break;
}
```

In this construct, the expression whose value is being compared may be any valid expression, including the value of a variable, an arithmetic expression, a logical comparison rarely, a bit wise expression, or the return value from a function call, but not a floating-point expression. The expression's value is checked against each of the specified cases and when a match occurs, the statements following that case are executed. When a break statement is encountered, control proceeds to the end of the switch - case statement.

The break statements inside the switch statement are optional. If the break statement is omitted, execution will continue on into the next case statements even though a match has already taken place until either a break or the end of the switch is reached.

The keyword case may only be constants, they cannot be expressions. They may be integers or characters, but not floating point numbers or character string.

Case constants may not be repeated within a switch statement.

The last case is a special keyword default. The default statement is executed if no matches are found. The default is optional and if it is not present, no action takes place if all matches fail.

Three important things to know about switch statement:

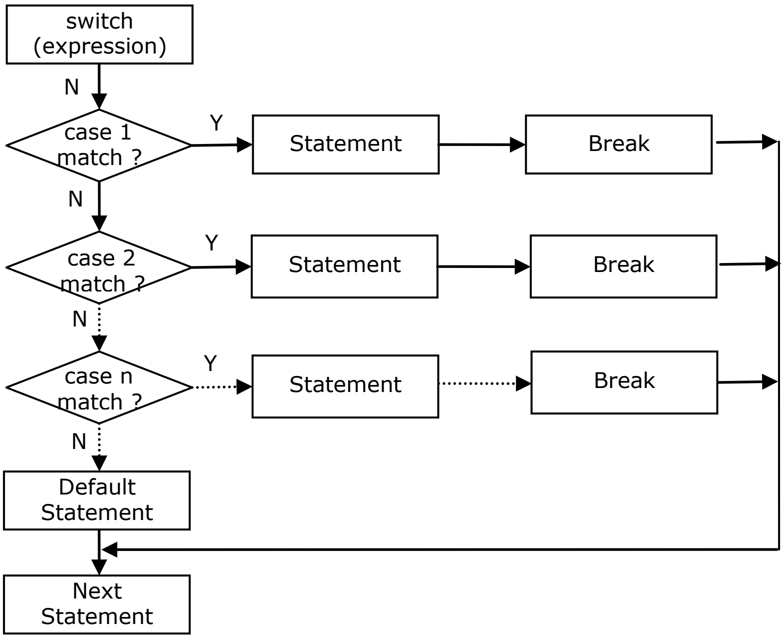
1. The switch differs from the if in that switch can only test for equality whereas if can evaluate any type of relational or logical expression.

2. No two case constants in the same switch can have identical values. But, a switch statement enclosed by an outer switch may have case constants and either same.

3. If character constants are used in the switch statement, they are automatically converted to integers.

Flowchart Segment - Case Selection:

In the example below, five possible paths might be followed depending on the value stored in the character storage location X. Each path is selected based on the individual value(s) that might be stored in X.



Example 1:

```
main()
{
        char gender;
        printf (“Enter Gender code:(M/F)“);
        scanf (“%c”, &gender);
        switch (gender)
        {
                case ‘M’ : printf (“ Male“);
                             break;
                case ‘F’  : prrintf (“Female”);
                             break;
                default :  printf (“Wrong code“);
        }
}
```

We can also have null statements by just including a “**;**” or let the switch statement *fall through* by omitting any statements (see *example* below).

Example 2:

```
        switch (letter)
        {
                case `A':
                case `E':
                case `I' :
                case `O':
```

```
            case `U':
                    numberofvowels++;
                    break;
            case `  ':
                    numberofspaces++;
                    break;
            default:
                    numberofconstants++;
                    break;
    }
```

In the above example if the value of letter is `A', `E', `I', `O' or `U' then numberofvowels is incremented. If the value of letter is ` ' then numberofspaces is incremented. If none of these is true then the default condition is executed, that is numberofconstants is incremented.

2.4. Looping and Iteration:

Looping is a powerful programming technique through which a group of statements is executed repeatedly, until certain specified condition is satisfied. Looping is also called a repetition or iterative control mechanism.

C provides three types of loop control structures. They are:

- for statement
- while statement
- do-while statement

2.4.1. The for statement:

The for loop statement is useful to repeat a statement/s a known number of times.
The general syntax is as follows:

```
for (initialization; condition; operation)
        statement;
```

The **initialization** is generally an assignment statement that is used to set the loop control variable.

The **condition** is an expression(relational/logical/arithmetic/bitwise ….) that determines when the loop exists.

The **Operation** defines how the loop control variable changes each time the loop is repeated.
We must separate these three major sections by semicolon.

The for loop continues to execute as long as the condition is true. Once the condition becomes false, program execution resumes on the statement following the for. The control flow of the for statement is as follows:

Example 1:

```
// printing all odd and even numbers between 1 to 5
        int x;
        main ()
        {
                for (x=1; x <=5 ; x++)
                {
                        if( x % 2 == 0 )
                                printf( " %d is EVEN \n",x);
                        else
                                printf(" %d is ODD \n",x);
                }
        }
```

Output to the screen:

```
1 is ODD
2 is EVEN
3 is ODD
4 is EVEN
5 is EVEN
```

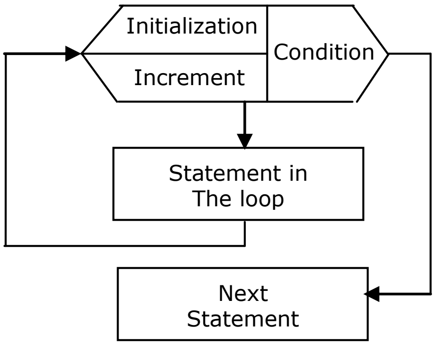
Example 2:

```
//  sum the squares of all the numbers between 1 to 5
main()
{
        int x, sum = 0;
        for (x = 1; x  <= 5; x ++)
        {
                sum = sum +  x * x;
        }
        printf ("\n Sum of squares of all the numbers between 1 to 5 = %d ", sum);
}
```

Output to the screen:

```
Sum of squares of all the numbers between 1 to 5 = 55
```

Flowchart Segment - for Statement:



The **comma (,) operator** is used to extend the flexibility of the for loop. It allows the general form to be modified as follows:

```
for (initialization_1, initialization_2; condition; operation_1, operation_2)
        statement;
```

All the following are legal for statements in C. The practical application of such statements is not important here, we are just trying to illustrate peculiar features that may be useful:

```
1.      for (x=0; ((x>3) && (x<9)); x++)
2.      for (x=0,y=4; ((x>3) && (y<9)); x++, y+=2)
3.      for (x=0, y=4, z=4000; z; z/=10)
```

The second example shows that multiple expressions can be separated by a , (comma).

Example:

```
main()
{
        int j ;
        double  degC, degF;
        clrscr ();
        printf ("\n Table of Celsius  and Fahrenheit  degrees \n\n");
        printf ("Celsius Degree \t  Fahrenheit Degree \n")
        degC = -20.0;
        for (j = 1; j <= 6; j++)
        {
                degC = degC + 20.0;
                degF = (degC * 9.0/5.0) + 32.0;
                printf ("\n %7.2lf\t\ %7.2lf ", degC, degF);
        }
}
```

Output:

```
   Table of Celsius  and Fahrenheit  degrees
   Celsius Degree       Fahrenheit Degree
   0.00                 32.00
   20.00                68.00
   40.00                104.00
   60.00                140.00
   80.00                176.00
   100.00               212.00
```

2.4.2. Nested for loop:

Nested loops consist of one loop placed inside another loop. An example of a nested for loop is:

```
        for (initialization; condition; operation)
        {
                for (initialization; condition; operation)
                {
                        statement;
                }
                statement;
        }
```

In this example, the inner loop runs through its full range of iterations for each single iteration of the outer loop.

Example:

Program to show table of first four powers of numbers 1 to 9.

```
#include <stdio.h >

void main()
{
        int i, j, k, temp;
        printf("I\tI^2\tI^3\tI^4 \n");
        printf("--------------------------------\n");
        for ( i = 1; i < 10; i ++)              /* Outer loop */
        {
                for (j = 1; j < 5; j ++)                /* 1st  level of nesting */
                {
                        temp = 1;
                        for(k = 0; k < j; k ++)
                                temp = temp * I;
                        printf ("%d\t", temp);
                }
                printf ("\n");
        }
}
```

Output to the screen:

```
        I       I ^ 2     I ^ 3    I ^ 4
        -----------------------------------
        1       1         1        1
        2       4         8        16
        3       9         27       81
        4       16        64       256
        5       25        125      625
        6       36        216      1296
        7       49        343      2401
        8       64        512      4096
        9       81        729      6561
```

2.4.3. Infinite for loop:

We can make an endless loop by leaving the conditional expression empty as given below:

```
        for( ; ; )
                printf("This loop will run for ever");
```

To terminate the infinite loop the break statement can be used anywhere inside the body of the loop. A sample example is given below:

```
        for(; ;)
        {
                ch = getchar ();
                if(ch == 'A')
                      break;
          }
          printf("You typed an A");
```

This loop will run until the user types an A at the keyboard.

2.4.4. for with no bodies:

A C-statement may be empty. This means that the body of the for loop may also be empty. There need not be an expression present for any of the sections. The expressions are optional.

Example 1:

```
/*   The loop will run until the user enters 123  */

      for( x = 0; x != 123; )
            scanf ("%d", &x);
```

This means that each time the loop repeats, 'x' is tested to see if it equals 123, but no further action takes place. If you type 123, at the keyboard, however the loop condition becomes false and the loop terminates.

The initialization some times happens when the initial condition of the loop control variable must be computed by some complex means.

Example 2:

```
/* Program to print the name in reverse order. */

#include<conio.h>
#include<string.h>
#include<stdio.h>

void main()
{
      char s[20];
      int x;
      clrscr ();
      printf ("\nEnter your name: ");
      gets (s);
      x = strlen (s);
      for ( ; x > 0 ; )
      {
            --x;
            printf ("%c\t", s[x]);
      }
}
```

Output to the screen:

```
Enter your name: KIRAN

N       A       R       I       K
```

2.5. The while statement:

The second loop available in 'C' is while loop.

The general format of while loop is:

```
        while (expression)
                statement
```

A while statement is useful to repeat a statement execution as long as a condition remains true or an error is detected. The while statement tests the condition before executing the statement.

The condition, can be any valid C languages expression including the value of a variable, a unary or binary expression, an arithmetic expression, or the return value from a function call.

The statement can be a simple or compound statement. A compound statement in a while statement appears as:

```
        while (condition)
        {
                statement1;
                statement2;
        }
```

With the if statement, it is important that no semicolon follow the closing parenthesis, otherwise the compiler will assume the loop body consists of a single null statement. This usually results in an infinite loop because the value of the condition will not change with in the body of the loop.

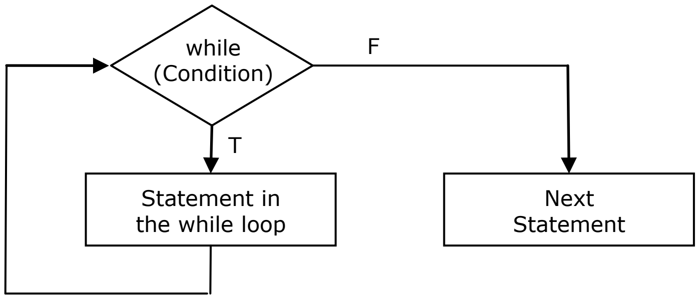
Example:

```
main()
{
        int j = 1;
        double  degC, degF;
        clrscr ();
        printf ("\n Table of Celsius  and Fahrenheit  degrees \n\n");
        printf ("Celsius Degree \t  Fahrenheit Degree \n")
        degC = -20.0;
        while (j <= 6)
        {
                degC = degC + 20.0;
                degF = (degC * 9.0/5.0) + 32.0;
                printf ("\n %7.2lf\t\ %7.2lf ", degC, degF);
                j++;
        }
}
```

Output:

```
Table of Celsius  and Fahrenheit  degrees
Celsius Degree      Fahrenheit Degree
0.00                32.00
20.00               68.00
40.00               104.00
60.00               140.00
80.00               176.00
100.00              212.00
```

Flowchart Segment - while Statement:



Because the while loop can accept expressions, not just conditions, the following are all legal:

```
        while(x--);
        while(x = x+1);
        while(x += 5);
```

Using this type of expression, only when the result of x--, x=x+1, or x+=5, evaluates to 0 will the while condition fail and the loop be exited.

We can go further still and perform complete operations within the while *expression*:

```
        while(i++ < 10);
```

The counts i up to 10.

```
        while((ch = getchar()) !=`q')
                putchar(ch);
```

This uses C standard library functions: getchar () to reads a character from the keyboard and putchar () to writes a given char to screen. The while loop will proceed to read from the keyboard and echo characters to the screen until a 'q' character is read.

Nested while:

Example:

Program to show table of first four powers of numbers 1 to 9.

```
#include <stdio.h >

void main()
{
        int i, j, k, temp;
        printf("I\tI^2\tI^3\tI^4 \n");
        printf("--------------------------------\n");
        i = 1;
        while (i < 10)                              /* Outer loop */
        {
                j = 1;
                while (j < 5)                  /* 1st  level of nesting */
                {
                        temp = 1;
                        k = 1;
                        while (k < j)
                        {
                                temp = temp * i;
                                k++;
                        }
                        printf ("%d\t", temp);
                        j++;
                }
                printf ("\n");
                i++;
        }
}
```

Output to the screen:

```
        I       I ^ 2    I ^ 3    I ^ 4
        ----------------------------------
        1       1        1        1
        2       4        8        16
        3       9        27       81
        4       16       64       256
        5       25       125      625
        6       36       216      1296
        7       49       343      2401
        8       64       512      4096
        9       81       729      6561
```

2.5.1. The do-while statement:

The third loop available in C is do – while loop.

The general format of do-while is:

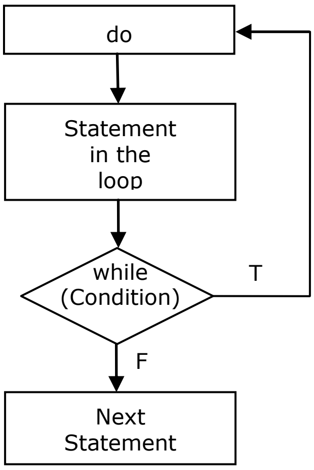
```
        do
                statement;
        while (expression);
```

Unlike for and while loops, which tests the condition at the top of the loop. The do – while loop checks its condition at the bottom of the loop. This means that the do – while loop always executes first and then the condition is tested. Unlike the while construction, the do – while requires a semicolon to follow the statement's conditional part.

If more than one statement is to be executed in the body of the loop, then these statements may be formed into a compound statement as follows:

```
        do
        {
                statement1;
                statement2;
        } while (condition);
```

Flowchart Segment of do-while Statement:



Example 1:

```
# include <stdio.h>
main()
{
        do
        {
                printf("x = %d\n", x--);
        } while(x > 0);
}
```

Output to the screen:

```
X = 3
X = 2
X = 1
```

Example 2:

```
#include <stdio.h>
void main()
{
        char ch;
        printf("T: Train\n");
        printf("C: Car\n");
        printf("S: Ship\n");
        do
        {
                printf("\nEnter your choice: ");
                fflush(stdin);
                ch = getchar();
                switch(ch)
                {
                        case 'T' :
                                        printf("\nTrain");
                                        break;
                        case 'C' :
                                        printf("\nCar");
                                        break;
                        case 'S':
                                        printf("\nShip");
                                        break;
                        default:
                                        printf("\n Invalid Choice");
                }
         } while(ch == 'T' || ch == 'C' || ch == 'S');
}
```

Output to the screen:

```
        T: Train
        C: Car
        S: Ship

Enter your choice: T
        Train
```

Distinguishing between while and do-while loops:

While loop	Do-while loop
The while loop tests the condition before each iteration.	The do-while loop tests the condition after the first iteration.
If the condition fails initially the loop is skipped entirely even in the first iteration.	Even if the condition fails initially the loop is executed once.

2.6. Un-conditional (Jump) statements:

C has four jump statements to perform an unconditional branch:

- return
- goto
- break and
- continue

2.6.1 return statement:

A return statement is used to return from a function. A function can use this statement as a mechanism to return a value to its calling function. If now value is specified, assume that a garbage value is returned (some compilers will return 0).

The general form of return statement is:

```
return expression;
```

Where expression is any valid rvalue expression.

Example:

```
return x; or return(x);
return x + y or return(x + y);
return rand(x); or return(rand(x));
return 10 * rand(x); or return (10 * rand(x));
```

We can use as many return statements as we like within a function. However, the function will stop executing as soon as it encounters the first return. The } that ends a function also causes the function to return. It is same way as return without any specified value.

A function declared as void may not contain a return statement that specifies a value.

2.6.2. goto statement:

goto statement provides a method of unconditional transfer control to a labeled point in the program. The goto statement requires a destination label declared as:

```
label:
```

The label is a word (permissible length is machine dependent) followed by a colon. The goto statement is formally defined as:

```
goto label;
        `
        `
        `
label:
        target statement
```

Since, C has a rich set of control statements and allows additional control using break and continue, there is a little need for goto. The chief concern about the goto is its tendency to render programs unreachable. Rather, it a convenience, it used wisely, can be a benefit in a narrow set of programming situation. So the usage of goto is highly discouraged.

Example:

```
Void main()
{
        int x = 6, y = 12;

        if( x == y)
                x++;
        else
```

```
                goto error;
        error:
                printf (“Fatal error; Exiting”);
}
```

The compiler doesn’t require any formal declaration of the label identifiers.

2.6.3. break statement:

We can use it to terminate a case in a switch statement and to terminate a loop.

Consider the following example where we read an integer values and process them according to the following conditions. If the value we have read is negative, we wish to print an error message and abandon the loop. If the value read is greater than 100, we wish to ignore it and continue to the next value in the data. If the value is zero, we wish to terminate the loop.

Example:

```
void main()
{
        int value;

        while (scanf(“%d'', &value ) == 1 && value != 0)
        {
                if(value < 0)
                {
                        printf (“Illegal value\n'');
                        break;                  /* Terminate the loop */
                }
                if(value > 100)
                {
                        printf(“Invalid value\n'');
                        continue;               /* Skip to start loop again */
                }
        }                                       /* end while value != 0 */
}
```

2.6.4. Continue statement:

The continue statement forces the next iteration of the loop to take place, skipping any code in between. But the break statement forces for termination.

Example 1:

```
/* Program to print the even numbers below 100 */

#include<stdio.h>

void main()
{
        int x;
        for(x = 1; x < 10; x++)
        {
                if (x % 2)
                        continue;
                printf (“%d\t”, x)
        }
}
```

An odd number causes continue to execute and the next iteration to occur, by passing the printf () statement. A continue statement is used within a loop (i.e for, while, do – while) to end an iteration in while and do-while loops, a continue statement will cause control to go directly to the conditional test and then continue the looping process. In the case of for, first the increment part of the loop is performed, next the conditional test is executed and finally the loop continues.

Example 2:

```
main()
{
        char ch;
        while (1)
        {
                ch = getchar();
                if (ch == EOF)
                        break;
                if (iscntrl (ch))
                        continue;
                else
                        printf (“\n not a control character”);
}
```

Distinguishing between break and continue statement:

Break	Continue
Used to terminate the loops or to exist loop from a switch.	Used to transfer the control to the start of loop.
The break statement when executed causes immediate termination of loop containing it.	The continue statement when executed cause immediate termination of the current iteration of the loop.

2.6.5. The exit () function:

Just as we can break out of a loop, we can break out of a program by using the standard library function exit(). This function causes immediate termination of the entire program, forcing a return to the operation system.

The general form of the exit() function is:

```
void exit (int return_code);
```

The value of the return_code is returned to the calling process, which is usually the operation system. Zero is generally used as a return code to indicate normal program termination.

Example:

```
Void menu(void)
{
        char ch;
        printf("B: Breakfast");
        printf("L: Lunch");
        printf("D: Dinner");
        printf("E: Exit");
        printf("Enter your choice: ");
        do
        {
                ch = getchar();
                switch (ch)
                {
                        case 'B' :
                                printf ("time for breakfast");
                                break;
                        case 'L' :
                                printf ("time for lunch");
                                break;
                        case 'D' :
                                printf ("time for dinner");
                                break;
                        case 'E' :
                                exit (0);          /* return to operating system */
                }
        } while (ch != 'B' && ch != 'L' && ch != 'D');
}
```

2.7. ARRAYS:

An array is a collection of variables of the same type that are referenced by a common name. In C, all arrays consists of contiguous memory locations. The lowest address corresponds to the first element, and the highest address to the last element. Arrays may have from one to several dimensions. A specific element in an array is accessed by an index.

One Dimensional Array:

The general form of single-dimension array declaration is:

```
Type variable-name[size];
```

Here, type declares the base type of the array, size defines how many elements the array will hold.

For example, the following declares as integer array named sample that is ten elements long:

```
int sample[10];
```

In C, all arrays have zero as the index of their first element. This declares an integer array that has ten elements, sample[0] through sample[9]

Example 1:

```
/* load an inter array with the number 0 through 9 */

main()
{
        int x [10], t;
        for (t=; t<10; t++)
                x [t] = t;
}
```

Example 2:

```
/* To find the average of 10 numbers */

# include <stdio.h>
main()
{
        int i, avg, sample[10];

        for (i=0; i<10; i++)
        {
                printf (“\nEnter number: %d “, i);
                scanf (“%d”, &sample[i]);
        }
        avg = 0;
        for (i=0; i<10; i++)
                avg = avg + sample[i];
        printf (“\nThe average is: %d\n”, avg/10);
}
```

Two-dimensional arrays:

To declare two-dimensional integer array num of size (3,4),we write:

```
        int num[3][4];
```

Left Index determines row   Right index determines column

Two dimensional arrays are stored in a row-column matrix where the first index indicates the row and the second indicates the column. This means that the right most index changes faster than the leftmost when accessing the elements in the array in the order in which they are actually stored in memory.

Num[t][I]	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12

Example:

```
main ()
{
        int t, i, num [3][4];

        for (t=0; t<3; t++)
                for (i=0; i<4; ++i)
                        num [t][i] = (t * 4) + i + 1;
        for (t=0; t<3; t++)
        {
                for (i=0; i<4; ++i)
                        printf (“%3d”, num[t][i]);
                printf (“\n”);
        }
}
```

the graphic representation of a two-dimensional array in memory is:

byte = sizeof 1st Index * sizeof 2nd Index * sizeof (base type)

Size of the base type can be obtained by using **size of operation**.

returns the size of memory (in terms of bytes) required to store an integer object.

sizeof (unsigned short)	=	2
sizeof (int)	=	4
sizeof (double)	=	8
sizeof (float)	=	4

assuming 2 byte integers as integer with dimension 4, 3 would have

4 * 3 * 2 = 24 bytes

Given: char ch[4][3]

Ch [0][0]	Ch [0][1]	Ch [0][2]
Ch [1][0]	Ch [1][1]	Ch [1][2]
Ch [2][0]	Ch [2][1]	Ch [2][2]
Ch [3][0]	Ch [3][1]	Ch [3][2]

N - dimensional array or multi dimensional array:

This type of array has n size of rows, columns and spaces and so on. The syntax used for declaration of this type of array is as follows:

Data type   array name[s1] [s2] … … … [sn];

In this sn is the n^{th} size of the array.

Array Initialization:

The general form of array initialization is:

```
Type_specifier array_name[size1]…. [sizeN] = { value_list};
```

The value list is a comma_separated list of constants whose type is compatible with type_specifier.

Example 1:

10 element integer array is initialized with the numbers 1 through 10 as:

```
int I[10] ={1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

i.e., I[0] will have the value 1 and

……….

……….

I[9] will have the value 10

Character arrays that hold strings allow a shorthand initialization that takes the form:

```
char array_name[size] = “string” ;
```

Example 2:

```
char str[9] = “I  like  C”;
```

this is same as writing:

```
char str[9] = {‘I’, ‘ ‘, ‘l’, ’i’, ’k’, ’e’, ’ ‘, ’C’, ’\0’};
```

2.8. STRINGS:

In C, In order to allow variable length strings the \0 character is used to indicate the end of a string. A null is specified using ‘\0’ and is zero. Because of the null terminator, It is necessary to declare character array to be one character longer than the largest string that they are to hold. For example, to declare an array str that can hold a 10 character string, we write:

```
char str[11];
```

this makes room for the null at the end of the string.

A string constant is a list of characters enclosed by double quotation marks.
For example:

```
“hello, this is a test”
```

It is not necessary to manually add the null onto end of string constants – the compiler does this automatically.

T	U	R	B	O	C	‘\0’

Reading a string from the keyboard:

The easiest way to input a string from the keyboard is with the gets() library function.
The general form gets() is:

```
        gets(array_name);
```

To read a string, call gets() with the name of the array, with out any index, as its arguments. Upon return form gets() the array will hold the string input. The gets() function will continue to read characters until you enter a carriage return. The header file used for gets() is stdio.h

Example:

```
# include <stdio.h>
main()
{
        char str[80];
        printf ("\nEnter a string:");
        gets (str);
        printf ("%s", str);
}
```

the carriage return does not become part of the string instead a null terminator is placed at the end.

Writing strings:

The puts() functions writes its string argument to the screen followed by a newline. Its prototype is:

```
        puts(string);
```

It recognizes the same back slash code as printf(), such as "\t" for tab. As puts() can output a string of characters – It cannot output numbers or do format conversions it required faster overhead than printf(). Hence, puts() function is often used when it is important to have highly optimized code.

For example, to display "hello" on the screen:

```
        puts("hello");
```

Array of strings:

To create an array of strings, a two dimensional character array is used with the size of the left-Index determining the number of strings and the size of the right Index specifying the maximum length of each string.

For example, to declare an array of 30 strings each having a max length of 80 characters.

```
        char str_array[30][80];
```

To access an individual string is quite easy: you simply specify only the left Index.

Example:

```
/* to accept lines of text and redisplay them when a blank line is entered */

main()
{
        int t, i;
        char text [100][80];
        for (t=0; t<100; t++)
        {
                printf ("%d Enter Text: ", t);
                gets (text [t]);
                if (! text [t][0])              /* quit on blank line */
                        break;
        }
        for (i=0; i<t; i++)
                printf ("%s\n", text [i]);
}
```

2.8.1 Basic String Handling Functions:

As string data type is not present. A string constant is a list of characters enclosed in double quotes.

For example, "Hello"

C- supports a wide range of string manipulation functions,

Name	Function
From to strcpy(s1, s2)	Copies s2 into s1 (the array forming to must be large enough to hold the string content in form)
strcat(s1, s2)	Append s2 onto the end of s1
strlen(s1)	Returns the length of s1
strcmp(s1, s2)	Returns 0 if s1 and s2 are the same to determine alphabetic order. Less than 0 if s1 < s2; greater than 0 if s1 > s2
strchr(s1, ch)	Return a pointer to the first occurrence of ch in s1
strstr(s1, s2)	Return a pointer to the first occurrence of s2 in s1
strrev(s1)	Reverses the string s1.

All the string handling functions are prototyped in: # include <string.h>

strcat () and strcopy () both return a copy of their first argument which is the destination array. Note the order of the arguments is ***destination array*** followed by ***source array*** which is sometimes easy to get the wrong around when programming.

The strcmp () function ***lexically*** compares the two input strings and returns:

Less than zero: if string1 is lexically less than string2

Zero: if string1 and string2 are lexically equal

Greater than zero: if string1 is lexically greater than string2
This can also confuse beginners and experience programmers forget this too.

The strcat (), strcmp () and strcpy () copy functions are string restricted version of their more general counterparts. They perform a similar task but only up to the first n characters. Note the NULL terminated requirement may get violated when using these functions.

Example 1:

```
# include <stdio.h>
# include <string.h>

void main(void)
{
        char s1 [80], s2 [80];
        gets (s1);
        gets (s2);
        printf (“lengths: %d %d\n”, strlen (s1), strlen (s2));
        if (! Strcmp (s1, s2))          /* strcmp () returns false if the strings are equal, */
                                                */ use ! to reverse the condition*/
                printf(“two strings are equal \n”);
        strcat (s1, s2);
        printf (“%s\n”, s1);
        strcpy (s1,”this is a test\n”);
        printf (s1);
        if (strchr (“hello”, ’e’)
                printf(“e is in hello\n”);
        if (strstr (“hi these”, ”hi”);
                printf (“found hi”);
}
```

Output to the screen:

```
enter: hello hello
lengths: 5 5
two strings are equal
hellohello
this is a test
e is in hello
found hi
```

Example 2:

```
/* to reverse a string * /

# include <stdio.h>
# include <string.h>
main ()
{
        char str[80];
        int i;
        printf (“enter a string: “);
        gets (str);
        for (i = strlen (str)-1; i > 0; i--)
                printf (“%c”, str[i]);
}
```

2.8.2. Character conversions and testing: ctype.h

We conclude this chapter with a related library #include <ctype.h> which contains many useful functions to convert and test ***single*** characters. The common functions are prototypes as follows:

Character testing:

int isalnum (int c) -- True if c is alphanumeric.

int isalpha (int c) -- True if c is a letter.

int isascii (int c) -- True if c is ASCII .

int iscntrl (int c) -- True if c is a control character.

int isdigit (int c) -- True if c is a decimal digit

int isgraph (int c) -- True if c is a graphical character.

int islower (int c) -- True if c is a lowercase letter

int isprint (int c) -- True if c is a printable character

int ispunct (int c) -- True if c is a punctuation character.

int isspace (int c) -- True if c is a space character.

int isupper (int c) -- True if c is an uppercase letter.

int isxdigit (int c) -- True if c is a hexadecimal digit

2.8.3. Character Conversion:

int toascii (int c) -- Convert c to ASCII .

tolower (int c) -- Convert c to lowercase.

int toupper (int c) -- Convert c to uppercase.

The use of these functions is straightforward and we do not give examples here.

Glossary and Keywords

The ANSI standard categorizes C's program control statements as follows:

Category	Relevant Keywords
Selection (or Conditional) Statements	if switch
Iteration	for while do-while
Jump	break goto return continue
Label	case default

Conditional Expressions in C:

Many C statements rely on the outcome of a conditional expression, which evaluates to either **true** or **false**. In C, unlike many other languages, true is any non-zero value and false is zero. This approach facilitates efficient coding.

The programmer is not restricted to conditional expressions involving only the relational and logical operators. Any valid C expression may be used to control the if. All that is required is an evaluation to either zero or no-zero. For example:

```
b = a * c;

if (b)
        printf (&quot%d\n&quot,b);
else
        printf ("b evaluated to zero\n&quot);
```

The if Statement:

The general form of an if statement is:

```
if (expression)
        statement;
else
        statement;
```

where, *statement* may be a single statement, a block, or nothing, and the else statement is optional. The conditional statement produces a scalar result, ie, an integer, character or floating point type.

Nested ifs. The ANSI standard specifies that 15 levels of nesting must be supported. In C, an else statement always refers to the nearest if statement in the same block and not already associated with an if. For example:

```
if(i)
{
        if(j)
                statement 1;
        if(k)                   // this if is associated with
                statement 2;
        else                    // this else
```

```
                    statement 3;
        }
        else                  // this is associated with if(i)
                statement 4;
```

The if-else-if Ladder. To avoid excessive indentation, if-else-if ladders are normally written like this:

```
        if(expression)
                statement;
        else if(expression)
                statement;
        else if(expression)
                statement;
                ...
        else
                statement;
```

The switch Statement:

switch successively tests the value of an expression against a list of integer or character constants. When a match is found, the statements associated with that constant are executed.

The general form is:

```
        switch (expression)
        {
                case constant1:
                        statement sequence;
                        break;
                case constant2:
                        statement sequence;
                        break;
                        ...

                default:
                        statement sequence;
        }
```

default: is executed if no matches are found.

Although case is a label statement, it cannot exist outside a switch.
Some important aspects of switch are as follows:

- A switch can only test for equality, whereas if can evaluate an expression.
- No two case constants can have identical values.
- If character constants are used, they will automatically be converted to integers.

If the break statement is omitted, execution continues on into the next case statement until either a break or the end of the switch is reached.

Nested switch Statements. A switch statement may be part of the statement sequence of an outer switch.

Iteration Statements:

The for Loop:

The general form of the for statement is:

```
For (initialization; condition; operation)
        statement;
```

initialization is generally an assignment statement used to set the loop control variable. *condition* is a relational expression that determines when the loop exits. operation defines how the loop variable changes each time the loop is repeated.

In for loops, the conditional test is always performed at the top of the loop, which means that the code inside the loop may not be executed at all if the condition is false, to begin with as in:

```
x = 10;
for (y=10; y != x; ++y)
        printf (" %d", y);
```

Variation 1 - The Comma Operator. A variant of the for loop is made possible by the comma operator, as in:

```
for(x=0, y=0; x+y < 10; ++x);
```

in which both x and y control the loop.

Variation 2 - Missing Pieces of the Loop Definition. An interesting trait of the for loop is that pieces of the loop definition need not be there. For example, in:

```
for (x=0; x!=123; )
        scanf ("%d", &x);
```

each time the loop repeats, x is tested to see if it equals 123. The loop condition only becomes false, terminating the loop, when 123 is entered.

Variation 3 - The Infinite Loop. If all of the pieces in the loop definition are missing, an infinite loop is created. However, the break statement may be used to break out of the loop, as in:

```
for(;;)
{
        ch = getchar();
        if(ch == 'A')
                break;
}
```

Variation 4 - for Loops with No Bodies. The body of the for loop may also be empty. This improves the efficiency of some code. For example, the following removes leading spaces from the stream pointed to by str:

```
for ( ; *str==' '; str++) ;
```

Time delay loops are another application of a loop with an empty body, eg:

```
for (t=0; t<1000; t++);
```

The while Loop:

The general form of the while loop is:

```
while (condition)
        statement;
```

where *statement* is either an empty statement, a single statement or a block of statements. The loop iterates while the condition, which is executed at the top of the loop, is true.

Variation 1 - No Body. The following, which is an example of a while loop with no body, loops until the user types A:

```
while ((ch = getchar ()) !='A') ;
```

The do-while Loop:

The do-while loop tests the condition at the bottom of the loop rather than at the top. This ensures that the loop will always execute at least once. In the following example, the loop will read numbers from the keyboard until it finds a number equal to or less than 100:

```
do
{
        scanf("%d", &num);
} while (num>100);
```

Jump Statements:

The return Statement:

The return statement is used to return from a function. If return has a value associated with it, that value is returned by the function. If no return value is specified, either zero or a garbage value will be returned, depending on the compiler. The general form of the return statement is:

```
return expression;
```

The **}** which ends a function also causes the function to return. A function declared as void may not contain a return statement that specifies a value

The goto Statement:

The goto statement requires a label (an identifier followed by a colon), which must be in the same function as the goto.

The break Statement:

The break statement has two uses:

- It can be used to terminate a case in a switch.
- It can be used to force immediate termination of a loop

The exit () Function:

The standard library function exit () causes immediate termination of the entire program. The general form of the exit function is:

```
void exit (int return_code);
```

The value of *return_code* is returned to the operating system. Zero is generally used as a return code to indicate normal program termination. Other arguments are used to indicate some sort of error.

The continue Statement:

The continue statement forces the next iteration of the loop, skipping any code in between. For the for loop, continue causes the conditional test and then the increment portion of the loop to execute. For the while and do-while loops, program control passes to the conditional test.

ARRAYS AND STRINGS:

An **array** is a collection of variables of the same type which are referenced by a common name. A specific element in an array is referenced by an **index**. The most common array in C is the string, which is simply an array of characters terminated by a null.

In C, arrays and pointers are closely related; a discussion of one usually refers to the other.

C has no bounds checking on arrays.

Single Dimension Arrays:

The general form for declaring a **single-dimension array** is:

```
type var_name[size];
```

In C, all arrays have zero as the index of their first element. Therefore a declaration of char p[10]; declares a ten-element array (p[0] through p[9]).

Generating a Pointer to an Array:

A pointer to the first element in an array may be generated by simply specifying the array name, without any index. For example, given:

```
int sample[10];
```

a pointer to the first element may be generated by simply using the name sample. For example, the following code fragment assigns p the address of the first element of sample:

```
int *p;
int sample[10];

p = sample;
```

The address of the first element may also be specified using the & operator. For example, sample and &sample[0] produce the same result. The former is usually used.

Passing Single-Dimension Arrays to Functions:

In C, an entire array cannot be passed as an argument to a function. However, a *pointer* to an array may be passed by specifying the array's name without an index. If a function receives a single dimension array, the formal parameter may be declared as either a pointer, a sized array, or an unsized array. Examples are:

```
function(int x)          /* A pointer */
function(int x[10])      /* A sized array */
function(int x[])        /* An unsized array  */
```

Strings:

A string is actually an array of characters. Because strings are terminated by a null ('\0'), character arrays must be declared with one extra element (to hold the null).

Although C does not have a string data type, it allows string constants. For example, "hello there" is a string constant.

C supports a wide range of string manipulation functions, including:

Function	Description
strcpy (s1, s2)	Copies s2 into s1.
strcat (s1, s2)	Concatenates s2 to s1.
strlen (s1)	Returns the length of s1.
strcmp (s1, s2)	Returns 0 (false) if s1 and s2 are the same. Returns less than 0 if s1&lts2 Returns greater than 0 if s1>s2
strchr (s1, ch)	Returns pointer to first occurrence ch in s1.
strstr (s1, s2)	Returns pointer to first occurrence s2 in s1.
strrev (s1)	Reverses the string s1.

Since strcmp () returns false if the strings are equal, it is best to use the ! operator to reverse the condition if the test is for equality.

Two-Dimensional Arrays:

In C, a **two-dimensional array** is declared as shown in the following example:

```
int d[10][20];
```

Two-dimensional arrays are stored in a row-column matrix. The first index indicates the row. The second index indicates the column.

When a two-dimensional array is used as an argument to a function, only a pointer to the first element is passed. However, the receiving function must define at least the length of the second dimension.

Example:

```
        function (int x[][20]);
```

Arrays of Strings:

Arrays of strings are created using a two-dimensional array. The left index determines the number of strings. Each string is accessed using only the left index.

Multi-Dimensional Arrays:

C allows **arrays of more than two dimensions**, the exact limit depending on the individual compiler.

Indexing Pointers:

In C, pointers and arrays are closely related. As previously stated, an array name without an index is a pointer to the first element. For example, given the array char

my_array[10], my_array and &my_array[0] are identical.

Conversely, any pointer variable may be indexed as if it were declared as an array. For example, in this program fragment:

```
        int *p, i[10];

        p = i;
        p[5] = 100;                     /* assignment using index */
        (p+5) = 100                     /* assignment using pointer arithmetic  */
```

both assignments achieve the same thing.

Pointers are sometimes used to access arrays because pointer arithmetic is faster than array indexing.

In a sense, a two-dimensional array is like an array of pointers that point to arrays of rows. Therefore, using a separate pointer variable is one easy way to access elements.

For example, the following prints the contents of the specified row for the global array num:

```
        int num[10][10];
                ...
        void print_row(int j)
        {
                int *p, t;
                p = &num[j][0]; // get address of first element in row j
                for(t=0;t<10;++t)
                printf("%d ", *(p+t));
        }
```

Array Initialization:

Arrays may be initialized at the time of declaration. The following example initializes a ten-element integer array:

```
int i[10] = { 1,2,3,4,5,6,7,8,9,10 };
```

Character arrays which hold strings allow a shorthand initialization, e.g.:

```
char str[9] = "I like C";
```

which is the same as:

```
char str[9] = { 'I',' ','l', 'i', 'k', 'e',' ','C','\0' };
```

When the string constant method is used, the compiler automatically supplies the null terminator.

Multi-dimensional arrays are initialized in the same way as single-dimension arrays, e.g.:

```
        int sgrs[6][2] =
        {
                1, 1,
                2, 4,
                3, 9,
                4, 16,
                5, 25
                6, 36
        };
```

Unsized Array Initializations:

If unsized arrays are declared, the C compiler automatically creates an array big enough to hold all the initializes. This is called an unsized array.

Example:

declaration/initializations are as follows:

```
        char e1[] = "read error\n";
        char e2[] = "write error\n";

        int sgrs[][2] =
        {
                1, 1,
                2, 4,
                3, 9
                4, 16,
        };
```

SAMPLE C – PROGRAMS

1. Program to find whether a given year is leap year or not.

```
 # include <stdio.h>

 main()
 {
        int year;

        printf("Enter a year:\n");
        scanf("%d", &year);

        if ( (year % 4) == 0)
               printf("%d is a leap year", year);
        else
               printf("%d is not a leap year\n", year);
 }
```

Output:

```
Enter a year: 2000
2000 is a leap year
```

RUN2:

```
Enter a year: 1999
1999 is not a leap year
```

2. Program to multiply given number by 4 using bitwise operators.

```
 # include <stdio.h>

 main()
 {
        long number, tempnum;
        printf("Enter an integer:\n");
        scanf("%ld", &number);
        tempnum = number;
        number = number << 2;   /*left shift by two bits*/

        printf("%ld x 4 = %ld\n", tempnum, number);
}
```

Output:

```
Enter an integer: 15
15 x 4 = 60
```

RUN2:

```
Enter an integer: 262
262 x 4 = 1048
```

3. Program to compute the value of X ^ N given X and N as inputs.

```
 #include <stdio.h>
 #include <math.h>

 void main()
 {
         long int x, n, xpown;
         long int power(int x, int n);

         printf("Enter the values of X and N: \n");
         scanf("%ld %ld", &x, &n);

         xpown = power (x, n);

         printf("X to the power N = %ld\n");
 }

 /*Recursive function to computer the X to power N*/

 long int power(int x, int n)
 {
         if (n==1)
                 return(x);
          else if ( n%2 == 0)
                 return (pow(power(x,n/2),2));              /*if n is even*/
         else
                 return (x*power(x, n-1));                  /* if n is odd*/
}
```

Output:

```
Enter the values of X and N: 2 5
X to the power N = 32

RUN2:
Enter the values of X and N: 4 4
X to the power N ==256

RUN3:
Enter the values of X and N: 5 2
X to the power N = 25

RUN4:
Enter the values of X and N: 10 5
X to the power N = 100000
```

4. Program to swap the contents of two numbers using bitwise XOR operation. Don't use either the temporary variable or arithmetic operators.

```
 # include <stdio.h>

 main()
 {
        long i, k;
        printf("Enter two integers: \n");
        scanf("%ld %ld", &i, &k);
        printf("\nBefore swapping i= %ld and k = %ld", i, k);
        i = i^k;
        k = i^k;
        i = i^k;
        printf("\nAfter swapping i= %ld and k = %ld", i, k);
 }
```

Output:

```
Enter two integers: 23 34
Before swapping i= 23 and k = 34
After swapping i= 34 and k = 23
```

5. Program to find and output all the roots of a quadratic equation, for non-zero coefficients. In case of errors your program should report suitable error message.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

main()
{
        float A, B, C, root1, root2;
        float realp, imagp, disc;
        clrscr();
        printf("Enter the values of A, B and C\n");
        scanf("%f %f %f", &A,&B,&C);

        if( A==0 || B==0 || C==0)
        {
                printf("Error: Roots cannot be determined\n");
                exit(1);
        }
        else
        {
                disc = B*B - 4.0*A*C;
                if(disc < 0)
                {
                        printf("Imaginary Roots\n");
                        realp = -B/(2.0*A) ;
                        imagp = sqrt(abs(disc))/(2.0*A);
                        printf("Root1 = %f  +i %f\n", realp, imagp);
                        printf("Root2 = %f  -i %f\n", realp, imagp);
                }
                else if(disc == 0)
                {
                        printf("Roots are real and equal\n");
                        root1 = -B/(2.0*A);
                        root2 = root1;
                        printf("Root1 = %f  \n",root1);
                        printf("Root2 = %f  \n",root2);
                }
                else if(disc > 0 )
                {
                        printf("Roots are real and distinct\n");
                        root1 =(-B+sqrt(disc))/(2.0*A);
                        root2 =(-B-sqrt(disc))/(2.0*A);
                        printf("Root1 = %f  \n",root1);
                        printf("Root2 = %f  \n",root2);
                }
        }

    }
```

Output:

```
RUN 1
Enter the values of A, B and C: 3 2 1
Imaginary Roots
Root1 = -0.333333  +i 0.471405
Root2 = -0.333333  -i 0.471405

RUN 2
Enter the values of A, B and C: 1 2 1
Roots are real and equal
Root1 = -1.000000
Root2 = -1.000000

RUN 3
Enter the values of A, B and C: 3 5 2
Roots are real and distinct
Root1 = -0.666667
Root2 = -1.000000
```

6. Write a C programme to accept a list of data items & find the II largest & II smallest in it & take average of both & search for that value. Display appropriate message on successful search.

```
main ()
{
    int i,j,a,n,counter,ave,number[30];
    printf ("Enter the value of N\n");
    scanf ("%d", &n);
    printf ("Enter the numbers \n");
    for (i=0; i<n; ++i)
    scanf ("%d",&number[i]);
    for (i=0; i<n; ++i)
    {
        for (j=i+1; j<n; ++j)
        {
            if (number[i] < number[j])
            {
                a = number[i];
                number[i] = number[j];
                number[j] = a;
            }
        }
    }
    printf ("The numbers arranged in descending order are given below\n");
```

```
        for (i=0; i<n; ++i)
                printf ("%10d\n",number[i]);
        printf ("The 2nd largest number is  = %d\n", number[1]);
        printf ("The 2nd smallest number is = %d\n", number[n-2]);
        ave = (number[1] +number[n-2])/2;
        counter = 0;
        for (i=0; i<n; ++i)
        {
                if (ave==number[i])
                ++counter;
        }
        if (counter==0)
                printf("The average of 2nd largest & 2nd smallest is not in the array\n");
        else
               printf("The average of 2nd largest & 2nd smallest in array is %d in
                      numbers\n", counter);
  }
```

7. Write a C programme to arrange the given numbers in ascending order.

```
main ()
{
        int i,j,a,n,number[30];
        printf ("Enter the value of N\n");
        scanf ("%d", &n);
        printf ("Enter the numbers \n");
        for (i=0; i<n; ++i)
                scanf ("%d",&number[i]);
        for (i=0; i<n; ++i)
        {
                        for (j=i+1; j<n; ++j)
                        {
                                if (number[i] > number[j])
                                {
                                        a= number[i];
                                        number[i] = number[j];
                                        number[j] = a;
                                }
                        }
        }
        printf("Number in Asscending order:\n");
        for(i=0;i<n;i++)
                printf("\t%d\n",number[i]);
}
```

8. Program to convert the given binary number into decimal.

```
# include <stdio.h>
main()
{
        int   num, bnum, dec = 0, base = 1, rem ;
        printf("Enter the binary number(1s and 0s)\n");
        scanf("%d", &num);           /*maximum five digits */
        bnum = num;
        while( num > 0)
        {
                        rem = num % 10;
                        if(rem>1)
                        {
                                printf("\nError in input");
                                break;
                        }
                        dec = dec + rem * base;
                        num = num / 10 ;
                        base = base * 2;
        }
        if(num==0)
        {
                printf("The Binary number is = %d\n", bnum);
                printf("Its decimal equivalent is =%d\n", dec);
        }
}
```

9.	Program to generate the fibonacci sequence.

```
#include <stdio.h>
main()
{
        int   fib1=0, fib2=1, fib3, limit, count=0;

        printf("Enter the limit to generate the fibonacci sequence\n");
        scanf("%d", &limit);

        printf("Fibonacci sequence is ...\n");
        printf("%d\n",fib1);
        printf("%d\n",fib2);
        count = 2;                      /* fib1 and fib2 are already used */

        while( count < limit)
        {
                fib3 = fib1 + fib2;
                count ++;
                printf("%d\n",fib3);
                fib1 = fib2;
                fib2 = fib3;
        }
}
```

10.	Program to reverse the given integer (palindrome).

```
#include <stdio.h>
main()
{
        int   num, rev = 0, found = 0, temp, digit;

        printf("Enter the number\n");
        scanf("%d", &num);

        temp = num;
        while(num > 0)
        {
                digit = num % 10;
                rev = rev * 10 + digit;
                num /= 10;
        }
        printf("Given number =%d\n", temp);
        printf("Its reverse is =%d\n", rev);

        if(temp == rev )
                printf("Number is a palindrome\n");
        else
                printf("Number is not a palindrome\n");
}
```

11. Program to determine the given number is odd.

```
#include <stdio.h>

main()
{
        int numb;
        printf(" Enter the number\n");
        scanf("%d", &numb);

        if((numb%2)!=0)
                printf(" %d , is an odd number\n", numb);

}
```

12. Program to find the largest among three numbers.

```
#include <stdio.h>

main()
{
        int a, b, c;
        printf(" Enter  the values for A,B,C\n");
        scanf("%d %d %d", &a, &b, &c);
        if( a > b )
        {
                if ( a > c)
                        printf(" A is the Largest\n");
                else
                        printf("C is the largest\n");
        }
        else if ( b > c)
                printf(" B is the Largest\n");
        else
                printf("C is the Largest\n");
}
```

13. Program to find the areas of different geometrical figures using switch statement.

```
#include <stdio.h>
main()
{
        int  fig_code;
        float  side, base, length, bredth, height, area, radius;
        printf("-------------------------\n");
        printf(" 1 --> Circle\n");
        printf(" 2 --> Rectangle\n");
        printf(" 3 --> Triangle\n");
        printf(" 4 --> Square\n");
        printf("-------------------------\n");

        printf("Enter the Figure code\n");
        scanf("%d", &fig_code);

        switch(fig_code)
        {
                case 1:
                        printf(" Enter the radius\n");
                        scanf("%f",&radius);
                        area=3.142*radius*radius;
                        printf("Area of a circle=%f\n", area);
                        break;
                case 2:
                        printf(" Enter the bredth and length\n");
                        scanf("%f %f",&bredth, &length);
                        area=bredth *length;
                        printf("Area of a Reactangle=%f\n", area);
                        break;
                case 3:
                        printf(" Enter the base and height\n");
                        scanf("%f %f", &base, &height);
                        area=0.5 *base*height;
                        printf("Area of a Triangle=%f\n", area);
                        break;
                case 4:
                        printf(" Enter the side\n");
                        scanf("%f", &side);
                        area=side * side;
                        printf("Area of a Square=%f\n", area);
                        break;
                default:
                        printf(" Error in figure code\n");
                        break;
        }

}
```

14. Program to find the factorial of a number.

```
#include <stdio.h>
```

```
main()
{
        int  i,fact=1,num;
        printf("Enter the number\n");
        scanf("%d",&num);
        if( num <0)
                printf(“Factorial is not there for –ve numbers”);
        else if(num==0 || num==1)
                fact=1;
        else
        {
                for(i=1;i<=num; i++)
                        fact *= i;
        }
        printf(" Factorial of %d =%5d\n", num,fact);
}
```

15. Program to illustrate for loop without initial and increment/decrement expressions.

```
#include <stdio.h>
main()
{
        int  i=0,limit=5;
        printf(" Values of I\n");
        for(  ; i<limit;   )
        {
                i++;
                printf("%d\n", i);
        }
}
```

16. Program to accept a string and find the sum of all digits in the string.

```
#include <stdio.h>
main()
{
        char string[80];
        int count, nc=0, sum=0;
        printf("Enter the string containing both digits and alphabet\n");
        scanf("%s", string);
        for(count=0; string[count]!='\0'; count++)
        {
                if((string[count]>='0') && (string[count]<='9'))
                {
                        nc += 1;
                        sum += (string[count] - '0');
                }
        }
        printf("NO. of Digits in the string= %d\n",nc);
        printf("Sum of all digits= %d\n",sum);

}
```

17. Program to find the sum of the sine series.

```
#include <stdio.h>
```

```
#include <math.h>
#define pi 3.142

main()
{
        int  i,n,k,sign;
        float sum=0,num,den,xdeg,xrad,xsqr,term;
        printf("Enter the angle( in degree): \n");
        scanf("%f",&xdeg);
        printf("Enter the no. of terms: \n");
        scanf("%d",&n);
        xrad=xdeg * (pi/180.0);    /* Degrees to radians*/
        xsqr= xrad*xrad;
        sign=1; k=2; num=xrad; den=1;

        for(i=1;i<=n; i++)
        {
                term=(num/den)* sign;
                sum += term;
                sign *= -1;
                num *= xsqr;
                den *= k*(k+1);
                k += 2;
        }
        printf("Sum of sine series of %d terms =%8.3f\n",n,sum);
}
```

18. Program to find the sum of cos(x) series.

```
#include<stdio.h>
#include<math.h>
main()
{
        float x, sign, cosx, fact;
        int n,x1,i,j;
        printf("Enter the number of the terms in a series\n");
        scanf("%d", &n);
        printf("Enter the value of x(in degrees)\n");
        scanf("%f", &x);
        x1=x;
        x=x*(3.142/180.0); /* Degrees to radians*/
        cosx=1;
        sign=-1;
        for(i=2; i<=n; i=i+2)
        {
                fact=1;
                for(j=1;j<=i;j++)
                {
                        fact=fact*j;
                }
                cosx=cosx+(pow(x,i)/fact)*sign;
                sign=sign*(-1);
        }
        printf("Sum of the cosine series=%f\n", cosx);
        printf("The value of cos(%d) using library function=%f\n",x1,cos(x));
}
```

19.	Program to reverse the given integer.

```
#include <stdio.h>

main()
{
        int   num, rev = 0, found = 0, temp, digit;

        printf("Enter the number\n");
        scanf("%d", &num);

        temp = num;
        while(num > 0)
        {
                digit = num % 10;
                rev = rev * 10 + digit;
                num /= 10;
        }
        printf("Given number =%d\n", temp);
        printf("Its reverse is =%d\n", rev);
}
```

20.	Program to accept a decimal number and convert to binary and count the number of 1's in the binary number.

```
#include <stdio.h>
main()
{
        long   num, dnum, bin = 0, base = 1;
        int rem, no_of_1s = 0 ;
        printf("Enter a decimal integer:\n");
        scanf("%ld", &num);                          /*maximum five digits */
        dnum = num;
        while( num > 0)
        {
                rem = num % 2;
                if (rem==1)                           /*To count number of 1s*/
                {
                        no_of_1s++;
                }
                bin = bin + rem * base;
                num = num / 2 ;
                base = base * 10;
        }
        printf("Input number is = %ld\n", dnum);
        printf("Its Binary equivalent is =%ld\n", bin);
        printf("No. of 1's in the binary number is = %d\n", no_of_1s);
}
```

Output:

```
Enter a decimal integer: 75
```

```
Input number is = 75
Its Binary equivalent is =1001011
No. of 1's in the binary number is = 4

RUN2
Enter a decimal integer: 128
Input number is = 128
Its Binary equivalent is=10000000
No. of 1's in the binary number is = 1
```

21. Program to find the number of characters, words and lines.

```
#include<conio.h>
#include<string.h>
#include<stdio.h>
void main()
{
        int count=0,chars,words=0,lines,i;
        char text[1000];
        clrscr();
        puts("Enter text:");
        gets(text);
        while (text[count]!='\0')
                count++;
        chars=count;
        for (i=0;i<=count;i++)
        {
                if ((text[i]==' '&&text[i+1]!=' ')||text[i]=='\0')
                   words++;
        }
        lines=chars/80+1;
        printf("no. of characters: %d\n", chars);
        printf("no. of words: %d\n", words);
        printf("no. of lines: %d\n", lines);
        getch();
}
```

22. Program to find the GCD and LCM of two integers output the results along with the given integers. Use Euclids' algorithm.

```
#include <stdio.h>
main()
{
        int  num1, num2, gcd, lcm, remainder, numerator, denominator;
        clrscr();
        printf("Enter two numbers: \n");
        scanf("%d %d", &num1,&num2);
        if (num1 > num2)
        {
                numerator = num1;
                denominator = num2;
        }
        else
        {
                numerator = num2;
                denominator = num1;
        }
        remainder = numerator % denominator;
        while(remainder !=0)
        {
                numerator   = denominator;
                denominator = remainder;
                remainder   = numerator % denominator;
        }
        gcd = denominator;
        lcm = num1 * num2 / gcd;
        printf("GCD of %d and %d = %d \n", num1,num2,gcd);
        printf("LCM of %d and %d = %d \n", num1,num2,lcm);
}
```

Output:

```
Enter two numbers: 5 15
GCD of 5 and 15 = 5
LCM of 5 and 15 = 15
```

23. Program to find the sum of odd numbers and sum of even numbers from 1 to N. Output the computed sums on two different lines with suitable headings.

```
#include <stdio.h>
main()
{
                int i, N, oddsum = 0, evensum = 0;
                printf("Enter the value of N: \n");
                scanf ("%d", &N);
                for (i=1; i <=N; i++)
                {
                                if (i % 2 == 0)
                                        evensum = evensum + i;
                                else
                                        oddsum = oddsum + i;
                }
                printf ("Sum of all odd numbers  = %d\n", oddsum);
                printf ("Sum of all even numbers = %d\n", evensum);
}
```

Output:

```
RUN1
Enter the value of N: 10
Sum of all odd numbers  = 25
Sum of all even numbers = 30

RUN2
Enter the value of N: 50
Sum of all odd numbers  = 625
Sum of all even numbers = 650
```

24. Program to check whether a given number is prime or not and output the given number with suitable message.

```
#include <stdio.h>
```

```
#include <stdlib.h>
main()
{
        int num, j, flag;
        clrscr();

        printf("Enter a number: \n");
        scanf("%d", &num);
        if ( num <= 1)
        {
                printf("%d is not a prime numbers\n", num);
                exit(1);
        }
        flag = 0;
        for ( j=2; j<= num/2; j++)
        {
                if( ( num % j ) == 0)
                {
                        flag = 1;
                        break;
                }
        }
        if(flag == 0)
                printf("%d is a prime number\n",num);
        else
                printf("%d is not a prime number\n", num);
}
```

Output:

```
RUN 1
Enter a number: 34
34 is not a prime number

RUN 2
Enter a number: 29
29 is a prime number
```

25. Program to generate and print prime numbers in a given range. Also print the number of prime numbers.

```
#include <stdio.h>
#include <math.h>
main()
{
        int M, N, i, j, flag, temp, count = 0;
        clrscr();
        printf("Enter the value of M and N: \n");
        scanf("%d %d", &M,&N);
        if(N < 2)
        {
                printf("There are no primes upto %d\n", N);
                exit(0);
        }
        printf("Prime numbers are\n");
        temp = M;
        if ( M % 2 == 0)
        {
                M++;
        }
        for (i=M; i<=N; i=i+2)
        {
                flag = 0;
                for (j=2; j<=i/2; j++)
                {
                        if( (i%j) == 0)
                        {
                                flag = 1;
                                break;
                        }
                }
                if(flag == 0)
                {
                        printf("%d\n",i);
                        count++;
                }
        }
        printf("Number of primes between %d  and %d = %d\n",temp,N,count);
}
```

Output:

```
Enter the value of M and N: 15 45
Prime numbers are
17
19
23
29
31
37
41
43
Number of primes between 15 and 45 = 8
```

26. Write to accept a 1-dimensional array of N elements & split into 2 halves & sort 1st half in ascending order & 2nd into descending order.

```
#include<stdio.h>
main ()
{
        int i,j,a,n,b,number[30];
        printf ("Enter the value of N\n");
        scanf ("%d", &n);
        b = n/2;
        printf ("Enter the numbers \n");
        for (i=0; i<n; ++i)
                scanf ("%d",&number[i]);
        for (i=0; i<b; ++i)
        {
                for (j=i+1; j<b; ++j)
                {
                        if (number[i] > number[j])
                        {
                                a = number[i];
                                number[i] = number[j];
                                number[j] = a;
                        }
                }
        }
        for (i=b; i<n; ++i)
        {
                        for (j=i+1; j<n; ++j)
                        {
                                if (number[i] < number[j])
                                {
                                        a = number[i];
                                        number[i] = number[j];
                                        number[j] = a;
                                }
                        }
         }
        printf (" The 1st half numbers\n");
        printf (" arranged in asc\n");
        for (i=0; i<b; ++i)
                printf ("%d ",number[i]);
        printf("\nThe 2nd half Numbers\n");
        printf("order arranged in desc.order\n");
        for(i=b;i<n;i++)
                printf("%d ",number[i]);
}
```

27.	Program to delete the desired element from the list.

```
# include <stdio.h>
main()
{
        int  vectx[10];
        int  i, n, found = 0, pos, element;
        printf("Enter how many elements\n");
        scanf("%d", &n);
        printf("Enter the elements\n");
        for(i=0; i<n; i++)
        {
                scanf("%d", &vectx[i]);
        }
        printf("Input array elements are\n");
        for(i=0; i<n; i++)
        {
                printf("%d\n", vectx[i]);
        }
         printf("Enter the element to be deleted\n");
         scanf("%d",&element);
        for(i=0; i<n; i++)
        {
                if ( vectx[i] == element)
                {
                        found = 1;
                        pos = i;
                break;
            }
        }
        if (found == 1)
        {
            for(i=pos; i< n-1; i++)
            {
                vectx[i] = vectx[i+1];
            }
            printf("The resultant vector is \n");
            for(i=0; i<n-1; i++)
            {
                printf("%d\n",vectx[i]);
            }
        }
        else
            printf("Element %d is not found in the vector\n", element);
}
```

28. Write a "C" program to Interchange the main diagonal elements with the scondary diagonal elements.

```
#include<stdio.h>
main ()
{
        int i,j,m,n,a;
        static int ma[10][10];
        printf ("Enetr the order of the matix \n");
        scanf ("%dx%d",&m,&n);
        if (m==n)
        {
            printf ("Enter the co-efficients of the matrix\n");
            for (i=0;i<m;++i)
            {
                for (j=0;j<n;++j)
                {
                    scanf ("%d",&ma[i][j]);
                }
            }
            printf ("The given matrix is \n");
            for (i=0;i<m;++i)
            {
                for (j=0;j<n;++j)
                {
                    printf (" %d",ma[i][j]);
                }
                printf ("\n");
            }
            for (i=0;i<m;++i)
            {
                a = ma[i][i];
                ma[i][i]   = ma[i][m-i-1];
                ma[i][m-i-1] = a;
            }
            printf ("THe matrix after changing the \n");
            printf ("main diagonal & secondary diagonal\n");
```

```
                for (i=0;i<m;++i)
                {
                        for (j=0;j<n;++j)
                        {
                                printf (" %d",ma[i][j]);
                        }
                        printf ("\n");
                }
        }
        else
                printf ("The given order is not square matrix\n");
}
```

29	Program to insert an element at an appropriate position in an array.

```
#include <stdio.h>
#include <conio.h>
main()
{
        int  x[10];
        int  i, j, n, m, temp, key, pos;
        clrscr();
        printf("Enter how many elements\n");
        scanf("%d", &n);
        printf("Enter the elements\n");
        for(i=0; i<n; i++)
        {
                scanf("%d", &x[i]);
        }
        printf("Input array elements are\n");
        for(i=0; i<n; i++)
        {
                printf("%d\n", x[i]);
        }
        for(i=0; i< n; i++)
        {
                for(j=i+1; j<n; j++)
                {
                        if (x[i] > x[j])
                        {
                                temp = x[i];
                                x[i] = x[j];
                                x[j] = temp;
                        }
                }
        }
        printf("Sorted list is:\n");
        for(i=0; i<n; i++)
        {
                printf("%d\n", x[i]);
        }
        printf("Enter the element to be inserted\n");
        scanf("%d",&key);
        for(i=0; i<n; i++)
        {
                if ( key < x[i] )
                {
                        pos = i;
                        break;
                }
        }
        m = n - pos + 1 ;
        for(i=0; i<= m ; i++)
        {
                x[n-i+2] = x[n-i+1] ;
        }
        x[pos] = key;

        printf("Final list is:\n");
        for(i=0; i<n+1; i++)
        {
                printf("%d\n", x[i]);
        }
}
```

30. Program to compute mean, varience and standard deviation.

```
main()
{
                float x[10];
                int  i, n;
                float avrg, var, SD, sum=0, sum1=0;

                printf("Enter how many elements\n");
                scanf("%d", &n);
                printf("Enter %d numbers:",n);
                for(i=0; i<n; i++)
                {
                        scanf("%f", &x[i]);
                }
                                        /* Compute the sum of all elements */
                for(i=0; i<n; i++)
                        sum = sum + x[i];

                avrg = sum /(float) n;
                                        /* Compute varaience and standard deviation  */
                for(i=0; i<n; i++)
                {
                        sum1 = sum1 + pow((x[i] - avrg),2);
                }
                var = sum1 / (float) n;
                SD = sqrt(var);
                printf("Average of all elements =%.2f\n", avrg);
                printf("Variance of all elements =%.2f\n", avrg);
                printf("Standard Deviation of all elements =%.2f\n", avrg);
}
```

31. Program to find the frequency of odd numbers & even numbers in the input of a matrix.

```
#include<stdio.h>
```

```
main ()
{
        int i,j,m,n,even=0,odd=0;
        static int ma[10][10];
        printf ("Enter the order ofthe matrix \n");
        scanf ("%d %d",&m,&n);
        printf ("Enter the coefficients if matrix \n");
        for (i=0;i<m;++i)
        {
                for (j=0;j<n;++j)
                {
                        scanf ("%d", &ma[i][j]);
                        if ((ma[i][j]%2) == 0)
                        {
                                ++even;
                        }
                        else
                                ++odd;
                }
        }
        printf ("The given matrix is\n");
        for (i=0;i<m;++i)
        {
                for (j=0;j<n;++j)
                        printf (" %d",ma[i][j]);
                printf ("\n");
        }
        printf ("The frequency of odd number occurrence = %d\n",odd);
        printf ("The frequency of even number occurrence = %d\n",even);
}
```

32. Program to sort all rows of matrix in ascending order & all columns in descending order.

```
#include <stdio.h>
main ()
{
        int i,j,k,a,m,n;
        static int ma[10][10],mb[10][10];
        printf ("Enter the order of the matrix \n");
        scanf ("%d %d", &m,&n);
        printf ("Enter co-efficients of the matrix \n");

        for (i=0;i<m;++i)
        {
                for (j=0;j<n;++j)
                {
                        scanf ("%d",&ma[i][j]);
                        mb[i][j] = ma[i][j];
                }
        }
        printf ("The given matrix is \n");
        for (i=0;i<m;++i)
        {
                for (j=0;j<n;++j)
                {
```

```
            printf (" %d",ma[i][j]);
        }
        printf ("\n");
    }
    printf ("After arranging rows in ascending order\n");
    for (i=0;i<m;++i)
    {
        for (j=0;j<n;++j)
        {
            for (k=(j+1);k<n;++k)
            {
                 if (ma[i][j] > ma[i][k])
                {
                    a = ma[i][j];
                    ma[i][j] = ma[i][k];
                    ma[i][k] = a;
                }
            }
        }
    }
    for (i=0;i<m;++i)
    {
        for (j=0;j<n;++j)
        {
            printf (" %d",ma[i][j]);
        }
        printf ("\n");
    }
    printf ("After arranging the columns in descending order \n");
    for (j=0;j<n;++j)
    {
        for (i=0;i<m;++i)
        {
            for (k=i+1;k<m;++k)
            {
                if (mb[i][j] < mb[k][j])
                {
                    a = mb[i][j];
                    mb[i][j] = mb[k][j];
                    mb[k][j] = a;
                }
            }
        }
    }
    for (i=0;i<m;++i)
    {
        for (j=0;j<n;++j)
        {
             printf (" %d",mb[i][j]);
        }
        printf ("\n");
    }
}
```

33. Program to convert lower case letters to upper case vice-versa.

```
#include <stdio.h>
#include <ctype.h>
```

```
#include <conio.h>
main()
{
        char sentence[100];
        int count, ch, i;
        clrscr();
        printf("Enter a sentence\n");
        for(i=0; (sentence[i] = getchar())!='\n'; i++);
        count = i;
        sentence[count]='\0';
        printf("Input sentence is : %s",sentence);
        printf("\nResultant sentence is\n");
        for(i=0; i < count; i++)
        {
                ch = islower(sentence[i]) ? toupper(sentence[i]) : tolower(sentence[i]);
                putchar(ch);
        }
}
```

34. Program to find the sum of the rows & columns of a matrix.

```
main ()
{
        int i,j,m,n,sum=0;
        static int m1[10][10];
        printf ("Enter the order of the matrix\n");
        scanf ("%d%d", &m,&n);
        printf ("Enter the co-efficients of the matrix\n");
        for (i=0;i<m;++i)
        {
                for (j=0;j<n;++j)
                {
                        scanf ("%d",&m1[i][j]);
                }
        }
        for (i=0;i<m;++i)
        {
                for (j=0;j<n;++j)
                {
                        sum = sum + m1[i][j] ;
                }
                printf ("   Sum of the %d row is = %d\n",i,sum);
                sum = 0;
        }
        sum=0;
        for (j=0;j<n;++j)
        {
                for (i=0;i<m;++i)
                {
                        sum = sum+m1[i][j];
                }
                 printf ("Sum of the %d column is = %d\n", j,sum);
                sum = 0;
        }
 }
```

35. Program to find the transpose of a matrix.

```
#include<stdio.h>
main ()
{
        int i,j,m,n;
        static int ma[10][10];
        printf ("Enter the order of the matrix \n");
        scanf ("%d %d",&m,&n);
        printf ("Enter the coefiicients of the matrix\n");
        for (i=0;i<m;++i)
        {
                for (j=0;j<n;++j)
                {
                        scanf ("%d",&ma[i][j]);
                }
        }
        printf ("The given matrix is \n");
        for (i=0;i<m;++i)
        {
                for (j=0;j<n;++j)
                {
                        printf (" %d",ma[i][j]);
                }
                printf ("\n");
        }
        printf ("Transpose of matrix is \n");
        for (j=0;j<n;++j)
        {
                for (i=0;i<m;++i)
                {
                        printf (" %d",ma[i][j]);
                }
                printf ("\n");
        }
  }
```

36. Program to accepts two strings and compare them. Finally print whether, both are equal, or first string is greater than the second or the first string is less than the second string without using string library.

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int count1=0,count2=0,flag=0,i;
        char str1[10],str2[10];
        clrscr();
        puts("Enter a string:");
        gets(str1);
        puts("Enter another string:");
        gets(str2);
                                        /*Count the number of characters in str1*/
        while (str1[count1]!='\0')
                count1++;
```

```
                                        /*Count the number of characters in str2*/
        while (str2[count2]!='\0')
                count2++;
        i=0;

/*The string comparison starts with the first character in each string and continues with
subsequent characters until the corresponding characters differ or until the end of the
strings is reached.*/

        while ( (i < count1) && (i < count2))
        {
                if (str1[i] == str2[i])
                {
                        i++;
                        continue;
                }
                if (str1[i]<str2[i])
                {
                        flag = -1;
                        break;
                }
                if (str1[i] > str2[i])
                {
                        flag = 1;
                        break;
                }
        }
        if (flag==0)
                printf("Both strings are equal\n");
        if (flag==1)
                printf("String1 is greater than string2\n", str1, str2);
        if (flag == -1)
                printf("String1 is less than string2\n", str1, str2);
        getch();
}
```

Output:

```
Enter a string: happy
Enter another string: HAPPY
String1 is greater than string2

RUN2:
Enter a string: Hello
Enter another string: Hello
Both strings are equal

RUN3:
Enter a string: gold
Enter another string: silver
String1 is less than string2
```

37. Program to accept N integer number and store them in an array AR. The odd elements in the AR are copied into OAR and other elements are copied into EAR. Display the contents of OAR and EAR.

```
 #include <stdio.h>
main()
 {
        long int ARR[10], OAR[10], EAR[10];
        int i,j=0,k=0,n;
        printf("Enter the size of array AR:\n");
        scanf("%d",&n);
        printf("Enter the elements of the array:\n");
        for(i=0;i<n;i++)
        {
                scanf("%ld",&ARR[i]);
                fflush(stdin);
        }
                        /*Copy odd and even elemets into their respective arrays*/
        for(i=0;i<n;i++)
        {
                if (ARR[i]%2 == 0)
                {
                        EAR[j] = ARR[i];
                        j++;
                }
                else
                {
                        OAR[k] = ARR[i];
                        k++;
                }
        }
        printf("The elements of OAR are:\n");
        for(i=0;i<j;i++)
        {
                printf("%ld\n",OAR[i]);
        }

        printf("The elements of EAR are:\n");
        for(i=0;i<k;i++)
        {
                printf("%ld\n", EAR[i]);
        }
 }
```

Output:

```
 Enter the size of array AR: 6
 Enter the elements of the array:
 12
 345
 678
 899
 900
 111

The elements of OAR are:
 345
 899
 111

The elements of EAR are:
 12
 678
 900
```

38. Program to find the sub-string in a given string.

```
#include<stdio.h>
#include<conio.h>
void main()
{
	int count1=0,count2=0,i,j,flag;
	char str[80],search[10];
	clrscr();
	puts("Enter a string:");
	gets(str);
	puts("Enter search substring:");
	gets (search);
	while (str[count1]!='\0')
		count1++;
	while (search[count2]!='\0')
		count2++;
	for(i=0;i<=count1-count2;i++)
	{
		for(j=i;j<i+count2;j++)
		{
			flag=1;
			if (str[j]!=search[j-i])
			{
				flag=0;
				break;
			}
		}
		if (flag==1)
			break;
	}
	if (flag==1)
		puts("SEARCH SUCCESSFUL!");
	else
		puts("SEARCH UNSUCCESSFUL!");
	getch();
}
```

39. Program to accept a string and find the number of times the word 'the' appears in it.

```
#include<stdio.h>
#include<conio.h>

void main()
{
	int count=0,i,times=0,t,h,e,space;
	char str[100];
	clrscr();
	puts("Enter a string:");
	gets(str);
```

```
                                /*Traverse the string to count the number of characters*/
        while (str[count]!='\0')
        {
                count++;
        }
                                /*Finding the frequency of the word 'the'*/
        for(i=0;i<=count-3;i++)
        {
                t=(str[i]=='t'||str[i]=='T');
                h=(str[i+1]=='h'||str[i+1]=='H');
                e=(str[i+2]=='e'||str[i+2]=='E');
                space=(str[i+3]==' '||str[i+3]=='\0');
                if ((t&&h&&e&&space)==1)
                   times++;
        }
        printf("Frequency of the word \'the\' is %d\n",times);
        getch();
}
```

Output:

```
Enter a string: The Teacher's day is the birth day of Dr.S.Radhakrishnan
Frequency of the word 'the' is 2
```

40. Program to find the length of a string without using the built-in function, also check whether it is a palindrome or not.

```
 #include <stdio.h>
 #include <string.h>

 main()
 {
        char string[25], revString[25]={'\0'};
        int  i,length = 0, flag = 0;
        clrscr();
        fflush(stdin);
        printf("Enter a string\n");
        gets(string);
        for (i=0; string[i] != '\0'; i++) /*keep going through each */
        {                             /*character of the string */
                length++;                 /*till its end */
        }
        printf("The length of the string: \'%s\' = %d\n", string, length);

        for (i=length-1; i >= 0 ; i--)
        {
           revString[length-i-1] = string[i];
        }
                        /*Compare the input string and its reverse. If both are equal
                                then the input string is palindrome. Otherwise it is
                                        not a palindrome */
        for (i=0; i < length ; i++)
        {
                if (revString[i] == string[i])
                        flag = 1;
                else
```

```
                    flag = 0;
        }
        if (flag == 1)
                printf ("%s is a palindrome\n", string);
        else
                printf("%s is not a palindrome\n", string);
}
```

Output:

```
Enter a string: madam
The length of the string 'madam' = 5
madam is a palindrome

RUN2:
Enter a string: good
The length of the string 'good' = 4
good is not a palindrome
```

41. Program to accept two strings and concatenate them i.e. The second string is appended to the end of the first string.

```
#include <stdio.h>
#include <string.h>
main()
 {
    char string1[20], string2[20];
    int i,j,pos;
    strset(string1, '\0');              /*set all occurrences in two strings to NULL*/
    strset(string2,'\0');
    printf("Enter the first string:");
    gets(string1);
    fflush(stdin);
    printf("Enter the second string:");
    gets(string2);
    printf("First string  = %s\n", string1);
    printf("Second string = %s\n", string2);
                                /*To concate the second stribg to the end of the string
     travserse the first to its end and attach the second string*/
     for (i=0; string1[i] != '\0'; i++)
     {
        ;                       /*null statement: simply trvsering the string1*/
     }
     pos = i;
     for (i=pos,j=0; string2[j]!='\0'; i++)
     {
          string1[i] = string2[j++];
     }
     string1[i]='\0';             /*set the last character of string1 to NULL*/
     printf("Concatenated string = %s\n", string1);
}
```

Output:

```
Enter the first string: CD-
Enter the second string: ROM

First string  = CD-
Second string = ROM

Concatenated string = CD-ROM
```

42.   Program for Comparing two matrices for equality.

```
#include <stdio.h>
#include <conio.h>
main()
{
        int A[10][10], B[10][10];
        int i, j, R1, C1, R2, C2, flag =1;
        printf("Enter the order of the matrix A\n");
        scanf("%d %d", &R1, &C1);
        printf("Enter the order of the matrix B\n");
        scanf("%d %d", &R2,&C2);
        printf("Enter the elements of matrix A\n");
        for(i=0; i<R1; i++)
        {
                for(j=0; j<C1; j++)
                {
                        scanf("%d",&A[i][j]);
                }
        }
        printf("Enter the elements of matrix B\n");
        for(i=0; i<R2; i++)
        {
                for(j=0; j<C2; j++)
                {
                        scanf("%d",&B[i][j]);
                }
        }
        printf("MATRIX A is\n");
        for(i=0; i<R1; i++)
        {
                for(j=0; j<C1; j++)
                {
                        printf("%3d",A[i][j]);
                }
                printf("\n");
        }
        printf("MATRIX B is\n");
        for(i=0; i<R2; i++)
        {
                for(j=0; j<C2; j++)
                {
                        printf("%3d",B[i][j]);
                }
                printf("\n");
        }
                                /* Comparing two matrices for equality */
        if(R1 == R2 && C1 == C2)
        {
```

```
        printf("Matrices can be compared\n");
        for(i=0; i<R1; i++)
        {
            for(j=0; j<C2; j++)
            {
                if(A[i][j] != B[i][j])
                {
                    flag = 0;
                    break;
                }
            }
        }
    }
    else
    {
        printf(" Cannot be compared\n");
        exit(1);
    }
    if(flag == 1 )
        printf("Two matrices are equal\n");
    else
        printf("But,two matrices are not equal\n");
}
```

43. Program to illustrate how user authentication is made before allowing the user to access the secured resources. It asks for the user name and then the password. The password that you enter will not be displayed, instead that character is replaced by '*'.

```
#include <stdio.h>
void main()
{
    char pasword[10],usrname[10], ch;
    int i;
    clrscr();
    printf("Enter User name: ");
    gets(usrname);
    printf("Enter the password <any 8 characters>: ");
    for(i=0;i<8;i++)
    {
        ch = getch();
        pasword[i] = ch;
        ch = '*' ;
        printf("%c",ch);
    }
    pasword[i] = '\0';
    printf("\n\n\nYour password is :");
    for(i=0;i<8;i++)
    {
         printf("%c",pasword[i]);
    }
}
```

44. Program to find the length of a string without using the built-in function.

```
 # include <stdio.h>
 main()
 {
        char string[50];
        int  i, length = 0;
        printf("Enter a string\n");
        gets(string);
        for (i=0; string[i] != '\0'; i++) /*keep going through each */
        {                               /*character of the string */
           length++;                    /*till its end */
        }
        printf("The length of a string is the number of characters in it\n");
        printf("So, the length of %s =%d\n", string, length);
 }
```

Output:

```
Enter a string
hello
The length of a string is the number of characters in it
So, the length of hello = 5

RUN2
Enter a string
E-Commerce is hot now
The length of a string is the number of characters in it
So, the length of E-Commerce is hot now =21
```

45. Program to read N integers (zero, +ve and -ve) into an array A and to
    a) Find the sum of negative numbers
    b) Find the sum of positive numbers and
    c) Find the average of all input numbers

Output the various results computed with proper headings.

```
#include <stdio.h>
#define MAXSIZE 10
main()
{
        int array[MAXSIZE];
        int i, N, negsum=0, posum=0, count1=0, count2=0;
        float total=0.0, averg;
        clrscr();
        printf ("Enter the value of N\n");
        scanf("%d", &N);
        printf("Enter %d numbers (-ve, +ve and zero)\n", N);
        for(i=0; i< N ; i++)
        {
                scanf("%d",&array[i]);
                fflush(stdin);
        }
        printf("Input array elements\n");
        for(i=0; i< N ; i++)
        {
                printf("%+3d\n",array[i]);
        }
```

```
                    /* Summing  begins */
        for(i=0; i< N ; i++)
        {
                if(array[i] < 0)
                {
                        negsum = negsum +  array[i];
                }
                else if(array[i] > 0)
                {
                        posum = posum + array[i];
                }
                else if( array[i] == 0)
                {
                         ;
                }
                total = total + array[i] ;
        }
        averg = total / N;
        printf("\nSum of all negative numbers    = %d\n",negsum);
        printf("Sum of all positive numbers    = %d\n", posum);
        printf("\nAverage of all input numbers   = %.2f\n", averg);
}
```

Output:

```
Enter the value of N: 5
Enter 5 numbers (-ve, +ve and zero)
5
-3
0
-7
6

Input array elements
 +5
 -3
 +0
 -7
 +6

Sum of all negative numbers    = -10
Sum of all positive numbers    = 11
Average of all input numbers   = 0.20
```

46. Program to sort N numbers in ascending order using Bubble sort and print both the given and the sorted array with suitable headings.

```
#include <stdio.h>
#define MAXSIZE 10
main()
{
        int array[MAXSIZE];
        int i, j, N, temp;
        clrscr();
        printf("Enter the value of N: \n");
        scanf("%d",&N);
        printf("Enter the elements one by one\n");
        for(i=0; i<N ; i++)
        {
                scanf("%d",&array[i]);
        }
        printf("Input array is\n");
        for(i=0; i<N ; i++)
        {
                printf("%d\n",array[i]);
        }
                        /* Bubble sorting begins */
        for(i=0; i< N ; i++)
        {
                for(j=0; j< (N-i-1) ; j++)
                {
                        if(array[j] > array[j+1])
                        {
                                temp      = array[j];
                                array[j]   = array[j+1];
                                array[j+1] = temp;
                        }
                }
        }
        printf("Sorted array is...\n");
        for(i=0; i<N ; i++)
        {
                printf("%d\n",array[i]);
        }
}
```

Output:

```
Enter the value of N: 5
Enter the elements one by one
390
234
111
876
345

Input array is
390
234
111
876
345

Sorted array is...
111
234
345
390
876
```

47. Program to accept N numbers sorted in ascending order and to search for a given number using binary search. Report success or failure in the form of suitable messages.

```
#include <stdio.h>
```

```
main()
{
        int array[10];
        int i, j, N, temp, keynum, ascending = 0;
        int low,mid,high;
        clrscr();
        printf("Enter the value of N: \n");
        scanf("%d",&N);
        printf("Enter the elements one by one\n");
        for(i=0; i<N ; i++)
        {
                scanf("%d",&array[i]);
        }
        printf("Input array elements\n");
        for(i=0; i<N ; i++)
        {
                printf("%d\n",array[i]);
        }
                                  /* Bubble sorting begins */
        for(i=0; i< N ; i++)
        {
                for(j=0; j< (N-i-1) ; j++)
                {
                        if(array[j] > array[j+1])
                        {
                                temp = array[j];
                                array[j] = array[j+1];
                                array[j+1] = temp;
                        }
                }
        }
        printf("Sorted array is...\n");
        for(i=0; i<N ; i++)
        {
                printf("%d\n",array[i]);
        }
        printf("Enter the element to be searched\n");
        scanf("%d", &keynum);
                        /* Binary searching begins */
        low=1;
        high=N;
        do
        {
                mid= (low + high) / 2;
                if ( keynum < array[mid] )
                        high = mid - 1;
                else if ( keynum > array[mid])
                        low = mid + 1;
        } while( keynum!=array[mid] && low <= high);            /* End of do- while */
        if( keynum == array[mid] )
        {
                printf("SUCCESSFUL SEARCH\n");
        }
        else
                printf("Search is FAILED\n");
}
```

Output:

```
Enter the value of N: 4
Enter the elements one by one
3
1
4
2


Input array elements
3
1
4
2

Sorted array is...
1
2
3
4

Enter the element to be searched
4

SUCCESSFUL SEARCH
```

48. Program read a sentence and count the number of number of vowels and consonants in the given sentence. Output the results on two lines with suitable headings.

```
#include <stdio.h>

main()
{
        char sentence[80];
        int i, vowels=0, consonants=0, special = 0;
        clrscr();
        printf("Enter a sentence\n");
        gets(sentence);
        for(i=0; sentence[i] != '\0'; i++)
        {
                if((sentence[i] == 'a'||sentence[i] == 'e'||sentence[i] == 'i'||
                sentence[i] == 'o'||sentence[i] == 'u') ||(sentence[i] == 'A'||
                sentence[i] == 'E'||sentence[i] == 'I'|| sentence[i] == 'O'||
                sentence[i] == 'U'))
                {
                        vowels = vowels + 1;
                }
                else
                {
                        consonants = consonants + 1;
                }
                 if (sentence[i] =='\t' ||sentence[i] =='\0' || sentence[i] ==' ')
                {
                        special = special + 1;
                }
        }
        consonants = consonants - special;
        printf("No. of vowels in %s = %d\n", sentence, vowels);
        printf("No. of consonants in %s = %d\n", sentence, consonants);
}
```

Output:

```
Enter a sentence
Good Morning
No. of vowels in Good Morning = 4
No. of consonants in Good Morning = 7
```

49. Program to sort names in alphabetical order using structures.

```
#include<stdio.h>
#include<string.h>
#include<conio.h>

struct tag
{
        char name[10];
        int rno;
};

typedef struct tag node;

node  s[5];

sort(int no)
{
        int i,j;
        node temp;
        for(i=0;i<no-1;i++)
                for(j=i+1;j<no;j++)
                        if(strcmp(s[i].name,s[j].name)>0)
                        {
                                temp=s[i];
                                s[i]=s[j];
                                s[j]=temp;
                        }
}

main()
{
        int no,i;
        clrscr();
        fflush(stdin);
        printf("Enter The Number Of Students:");
        scanf("%d",&no);
        for(i=0;i<no;i++)
        {
                printf("Enter The Name:");
                fflush(stdin);
                gets(s[i].name);
                printf("Enter the Roll:");
                scanf("%d",&s[i].rno);
        }
        sort(no);
        for(i=0;i<no;i++)
        {
                printf("%s\t",s[i].name);
                printf("%d\n",s[i].rno);
        }
        getche();
}
```

50. Program to illustrate the unions.

```
#include <stdio.h>
#include <conio.h>

main()
{
        union number
        {
                int  n1;
                float n2;
        };
        union number x;
        clrscr() ;
        printf("Enter the value of n1: ");
        scanf("%d", &x.n1);
        printf("Value of n1 =%d", x.n1);
        printf("\nEnter the value of n2: ");
        scanf("%d", &x.n2);
        printf("Value of n2 = %d\n",x.n2);
}
```

51. Program to store the inventory information system.

```
#include<stdio.h>
#include<conio.h>

void main()
{       struct date
        {
                int day;
                int month;
                int year;
        };
        struct details
        {
                char name[20];
                int price;
                int code;
                int qty;
                struct date mfg;
        };
        struct details item[50];
        int n,i;
        clrscr();
        printf("Enter number of items:");
        scanf("%d",&n);
        fflush(stdin);
        for(i=0;i<n;i++)
        {
                fflush(stdin);
                printf("Item name:");
                scanf("%s",item[i].name);
                fflush(stdin);
                printf("Item code:");
                scanf("%d",&item[i].code);
                fflush(stdin);
                printf("Quantity:");
                scanf("%d",&item[i].qty);
                fflush(stdin);
                printf("price:");
                scanf("%d",&item[i].price);
                fflush(stdin);
                printf("Manufacturing date(dd-mm-yyyy):");
                scanf("%d-%d-%d",&item[i].mfg.day,&item[i].mfg.month,&item[i].mfg.year);
        }
        printf("            *****  INVENTORY *****\n");
        printf("----------------------------------------------------------------\n");
        printf("S.N.|    NAME           |   CODE   |  QUANTITY |  PRICE  |MFG.DATE\n");
        printf("----------------------------------------------------------------\n");
        for(i=0;i<n;i++)
       printf("%d     %-15s        %-d          %-5d     %-5d
                %d/%d/%d\n",i+1,item[i].name,item[i].code,item[i].qty,item[i].price,item
                [i].mfg.day,item[i].mfg.month,item[i].mfg.year);
        printf("----------------------------------------------------------------\n");
        getch();
}
```

52. Program to find the sum and difference of two matrices.

```
#include <stdio.h>

 int A[10][10], B[10][10], sumat[10][10], diffmat[10][10];
 int i, j, R1, C1, R2, C2;

main()
{
        void readmatA();
        void printmatA();
        void readmatB();
        void printmatB();
        void sum();
        void diff();
        printf("Enter the order of the matrix A\n");
        scanf("%d %d", &R1, &C1);
        printf("Enter the order of the matrix B\n");
        scanf("%d %d", &R2,&C2);
        if( R1 != R2 && C1 != C2)
        {
                printf("Addition and subtraction are possible\n");
                exit(1);
        }
        else
        {
                printf("Enter the elements of matrix A\n");
                readmatA();
                printf("MATRIX A is\n");
                printmatA();
                printf("Enter the elements of matrix B\n");
                readmatB();
                printf("MATRIX B is\n");
                printmatB();
                sum();
                diff();
        }
        return 0;
}
void readmatA()
{
        for(i=0; i<R1; i++)
        {
                for(j=0; j<C1; j++)
                {
                        scanf("%d",&A[i][j]);
                }
        }
        return;
}
void readmatB()
{
        for(i=0; i<R2; i++)
        {
                for(j=0; j<C2; j++)
                {
                        scanf("%d",&B[i][j]);
                }
        }
}
void printmatA()
{
        for(i=0; i<R1; i++)
        {
                for(j=0; j<C1; j++)
                {
                        printf("%3d",A[i][j]);
                }
                printf("\n");
        }
}
void printmatB()
{
        for(i=0; i<R2; i++)
        {
                for(j=0; j<C2; j++)
                {
                        printf("%3d",B[i][j]);
                }
                printf("\n");
        }
}

void sum()
{
        for(i=0; i<R1; i++)
        {
                for(j=0; j<C2; j++)
                {
                        sumat[i][j] = A[i][j] + B[i][j];
                }
        }
        printf("Sum matrix is\n");
        for(i=0; i<R1; i++)
        {
                for(j=0; j<C2; j++)
                {
                        printf("%3d",sumat[i][j]) ;
                }
                printf("\n");
        }
        return;
}

void diff()
{
        for(i=0; i<R1; i++)
        {
                for(j=0; j<C2; j++)
                {
                        diffmat[i][j] = A[i][j] - B[i][j];
                }
        }
        printf("Difference matrix is\n");
        for(i=0; i<R1; i++)
        {
                for(j=0; j<C2; j++)
                {
                        printf("%3d",diffmat[i][j]);
                }
                printf("\n");
        }
        return;
}
```

53. Program to read a matrix A (MxN) and to find the following using functions:
a) Sum of the elements of each row
b) Sum of the elements of each column
c) Find the sum of all the elements of the matrix

Output the computed results with suitable headings

```
#include <stdio.h>
main()
{
        int arr[10][10];
        int i, j, row, col, rowsum, colsum,sumall=0;
        clrscr();

        printf("Enter the order of the matrix\n");
        scanf("%d %d", &row, &col);

        printf("Enter the elements of the matrix\n");
        for(i=0; i<row; i++)
        {
                for(j=0; j< col; j++)
                {
                        scanf("%d", &arr[i][j]);
                }
        }
        printf("Input matrix is\n");
        for(i=0; i<row; i++)
        {
                for(j=0;j<col;j++)
                {
                        printf("%3d", arr[i][j]);
                }
                printf("\n");
        }
                                /* computing row sum */
        for(i=0; i<row; i++)
        {
                rowsum = Addrow(arr,i,col);
                printf("Sum of row %d = %d\n", i+1, rowsum);
        }

        for(j=0; j<col; j++)
        {
                colsum = Addcol(arr,j,row);
                printf("Sum of column  %d = %d\n", j+1, colsum);
        }
                                /* computation of all elements */
        for(j=0; j< row; j++)
        {
                sumall = sumall + Addrow(arr,j,col);
        }
        printf("Sum of all elements of matrix = %d\n", sumall);
}

/* Function to add each row */

int Addrow(int A[10][10], int k, int c)
{
        int rsum=0, i;
        for(i=0; i< c; i++)
        {
                rsum = rsum + A[k][i];
        }
        return(rsum);
}

/* Function to add each column */

int Addcol(int A[10][10], int k, int r)
{
        int csum=0, j;
        for(j=0; j< r; j++)
        {
                csum = csum + A[j][k];
        }
         return(csum);
}
```

Output:

```
Enter the order of the matrix
3
3

Enter the elements of the matrix
1 2 3
4 5 6
7 8 9

Input matrix is
  1  2  3
  4  5  6
  7  8  9
Sum of row 1 = 6
Sum of row 2 = 15
Sum of row 3 = 24
Sum of column  1 = 12
Sum of column  2 = 15
Sum of column  3 = 18
Sum of all elements of matrix = 45
```

54. Program to find the sum of all elements of an array using pointers.

```
#include <stdio.h>

main()
{
        static int array[5]={ 200,400,600,800,1000 };
        int addnum(int *ptr);                           /* function prototype */

        int sum;
        sum = addnum(array);
        printf(" Sum of all array elements=%d\n", sum);

}

int  addnum(int *ptr)
{
        int total=0, index;
        for(index=0;index<5; index++)
                total+=*(ptr + index);
        return(total);
}
```

55. Program to swap particular elements in array using pointers.

```
#include <stdio.h>

main()
{
        float x[10];
        int i,n;
        float swap34(float *ptr1, float  *ptr2 );      /* Function Declaration */
        printf(" How many Elements...\n");
        scanf("%d", &n);
        printf(" Enter Elements one by one\n");
        for(i=0;i<n;i++)
                scanf("%f",x+i);
        swap34(x+2, x+3);                             /* Interchanging 3rd element by 4th */
        printf(" Resultant Array...\n");
        for(i=0;i<n;i++)
                printf("X[%d]=%f\n",i,x[i]);
}

float swap34(float *ptr1, float *ptr2 )               /* Function Definition */
{
        float temp;
        temp=*ptr1;
        *ptr1=*ptr2;
        *ptr2=temp;
}
```

56. Program to find the sum of two 1-D arrays using Dynamic Memory Allocation.

```
#include <stdio.h>
#include <malloc.h>

main()
{
        int i,n,sum;
        int *a,*b,*c;
        printf(" How many Elements in each array...\n");
        scanf("%d", &n);
        a=(int *) malloc(sizeof(n*(int));
        b=(int *) malloc(sizeof(n*(int));
        c=(int *) malloc(sizeof(n*(int));
        printf(" Enter Elements of First List\n");
        for(i=0;i<n;i++)
                scanf("%f",a+i);

        printf(" Enter Elements of Second List\n");
        for(i=0;i<n;i++)
                scanf("%f",b+i);

        for(i=0;i<n;i++)
        c+i=(a+i) +( b+i);
    printf(" Resultant List is\n");
    for(i=0;i<n;i++)
        printf("C[%d]=%f\n",i,c[i]);
}
```

Chapter 3

Storage Classes, Functions and User Defined Data Types

3.1. Storage Classes:

Storage classes are used to define the scope (visibility) and life-time of variables and/or functions. Every variable and function in C has two attributes: type and storage class. There are four storage classes:

- auto
- static
- extern or global and
- register

3.1.1 The auto storage class:

Variables declared within function bodies are automatic by default. If a compound statement starts with variable declarations, then these variables can be acted on within the scope of the enclosing compound statement. A compound statement with declarations is called a block to distinguish it from one that does not begin with declarations.

Declarations of variables within blocks are implicitly of storage class automatic. The key-word auto can be used to explicitly specify the storage class. An example is:

```
auto int a, b, c;
auto float f;
```

Because the storage class is auto by default, the key-word auto is seldom used.

When a block is entered, the system allocates memory for the automatic variables. With in that block, these variables are defined and are considered "local" to the block. When the block is exited, the system releases the memory that was set aside for the automatic variables. Thus, the values of these variables are lost. If the block is reentered, the system once again allocates memory, but previous values are unknown. The body of a function definition constitutes a block if it contains declarations. If it does, then each invocation of the function set up a new environment.

3.1.2. The static storage class:

It allows a local variable to retain its previous value when the block is reentered. This is in contrast to automatic variables, which lose their value upon block exit and must be reinitialized. The static variables hold their values throughout the execution of the program. As an example of the value-retention use of static, the outline of a function that behaves differently depending on how many times it has been called is:

```
        void fun (void)
        {
                static int cnt = 0
                ++ cnt;
                if (cnt % 2 == 0)
                        . . .                           /* do something */
                else
                        . . .                           /* do something different */
        }
```

The first time the function is invoked, the variable cnt is initialized to zero. On function exit, the value of cnt is preserved in memory. Whenever the function is invoked again, cnt is not reinitialized. Instead, it retains its previous value from the last time the function was called. The declaration of cnt as a *static int* inside of fun () keeps it private of fun (). If it was declared outside of the function, then other functions could access it, too.

Static variables are classified into two types.

1. internal static variables
2. external static variables

External static variables are declared outside of all functions including the main() function. They are global variables but are declared with the keyword static. The external static variables cannot be used across multi-file program.

A static variable is stored at a fixed memory location in the computer, and is created and initialised once when the program is first started. Such a variable maintains its value between calls to the block (a function, or compound statement) in which it is defined. When a static variable is not initialized, it takes a value of zero.

3.1.3. The storage class extern:

One method of transmitting information across blocks and functions is to use external variables. When a variable is declared outside a function, storage is permanently assigned to it, and its storage class is extern. Such a variable is considered to be global to all functions declared after it, and upon exit from the block or function, the external variable remains in existence. The following program illustrates this:

```
# include <stdio.h>

int a = 1, b = 2, c = 3;                        /* global variable*/

int fun (void);                                 /* function prototype*/

int main (void)
{
        printf (“%3d\n”, fun ());               /* 12 is printed */
        printf (“%3d%3d%3d\n”, a, b, c);        /* 4   2  3 is printed */
}

int fun (void)
{
        int b, c;
        a = b = c = 4;                          /* b and c are local */
        return (a + b + c);                    /* global b, c are masked*/
}
```

Note that we could have written:

```
        extern int a = 1, b = 2, c = 3;
```

The extern variables cannot be initialised in other functions whereas we can use for assignment.

This use of extern will cause some traditional C compilers to complain. In ANSI C, this use is allowed but not required. Variables defined outside a function have external storage class, even if the keyword extern is not used. Such variables cannot have auto or register storage class.

The keyword extern is used to tell the compiler to "look for it else where, either in this file or in some other file". Let us rewrite the last program to illustrate a typical use of the keyword extern:

In file file1.c

```
#include <stdio.h>
# include "file2.c"

int a=1, b=2, c=3;          /* external variable */

int fun (void)

int main (void)
{
        printf ("The values of a, b and c are: %3d%3d%3d\n", a, b, c);
        printf ("The sum of a + b + c is: %3d\n", fun ());
        printf ("The values of a, b and c are: %3d%3d%3d\n", a, b, c);
        return 0;
}
```

In file file2.c

```
int fun (void)
{
        extern int a;               /* look for it elsewhere */
        int b, c;
        a = b = c = 4;
        return (a + b + c);
}
```

Output:

```
The values of a, b and c are: 1   2   3
The sum of a + b + c is: 12
The values of a, b and c are: 4   2   3
```

The two files can be compiled separately. The use of extern in the second file tells the compiler that the variable a will be defined elsewhere, either in this file or in some other. The ability to compile files separately is important when writing large programs.

External variables never disappear. Because they exist throughout the execution life of the program, they can be used to transmit values across functions. They may, however, be hidden if the identifier is redefined.

Information can be passed into a function two ways; by use of external variable and by use of the parameter passing mechanism. The use of the parameter mechanism is the best preferred method. Don't overuse ‘extern’ variables. It is usually better to pass lots of arguments to functions rather than to rely on hidden variables being passed (since you end up with clearer code and reusuable functions).

3.1.4. The register storage class:

This is like `auto' except that it asks the compiler to store the variable in one of the CPU's fast internal registers. In practice, it is usually best not to use the `register' type since compilers are now so smart that they can do a better job of deciding which variables to place in fast storage than you can.

The use of storage class register is an attempt to improve execution speed. When speed is a concern, the programmer may choose a few variables that are most frequently accessed and declare them to be of storage class register. Common candidates for such treatment include loop variables and function parameters. The keyword register is used to declare these variables.

Examples:

1) register int x;
2) register int counter;

The declaration *register i*; is equivalent to *register int i;*

A complete summarized table to represent the lifetime and visibility (scope) of all the storage class specifier is as below:

Storage class	Life time	Visibility (Scope)
Auto	Local	Local (within function)
Extern	Global	Global (in all functions)
Static	Global	Local
Register	Local	Local

3.2. The type qualifiers, const and volatile:

The type qualifiers const and volatile restrict or qualify the way an identifier of a given type can be used.

3.2.1. Const Storage Class:

Variables can be qualified as ‘const’ to indicate that they are really constants, that can be initialised, but not altered. The declaration is:

```
        const int k = 5;
```

As the type of k has been qualified by const, we can initialise k, but thereafter k cannot be assigned to, incremented, decremented or otherwise modified.

3.2.2. Volatile Storage Class:

Variables can also be termed 'volatile' to indicate that their value may change unexpectedly during the execution of the program. The declaration is:

```
        extern const volatile int real_time_clock;
```

The extern means look for it elsewhere, either in this file or in some other file. The qualifier volatile indicates that the object may be acted on by the hardware. Because const is also a qualifier, the object may not be assigned to, incremented, or decremented within the program. The hardware can change the clock, but the code cannot.

3.3. FUNCTIONS:

Functions are a group of statements that have been given a name. This allows you to break your program down into manageable pieces and reuse your code.

The advantages of functions are:

1. Function makes the lengthy and complex program easy and in short forms. It means large program can be sub-divided into self-contained and convenient small modules having unique name.

2. The length of source program can be reduced by using function by using it at different places in the program according to the user's requirement.

3. By using function, memory space can be properly utilized. Also less memory is required to run program if function is used.

4. They also allow more than one person to work on one program.

5. Function increases the execution speed of the program and makes the programming simple.

6. By using the function, portability of the program is very easy.

7. It removes the redundancy (occurrence of duplication of programs) i.e. avoids the repetition and saves the time and space.

8. Debugging (removing error) becomes very easier and fast using the function sub-programming.

9. Functions are more flexible than library functions.

10. Testing (verification and validation) is very easy by using functions.

11. User can build a customized library of different functions used in daily routine having specific goal and link with the main program similar to the library functions.

The functions are classified into standard functions and user-defined functions.

The standard functions are also called library functions or built in functions. All standard functions, such as sqrt(), abs(), log(), sin(), pow() etc. are provided in the library of functions. These are selective.

Most of the applications need other functions than those available in the software. These functions must be written by the user (programmer), hence, the name user-defined functions.

You can write as many functions as you like in a program as long as there is only one main (). As long as these functions are saved in the same file, you do not need to include a header file. They will be found automatically by the compiler.

There are 3 parts for using functions:

- Calling them,
- Defining them and
- Declaring them (prototypes).

3.3.1. Calling functions:

When you wish to use a function, you can "call" it by name. Control is sent to the block of code with that name. Any values that the function needs to perform its job are listed in the parentheses that immediately follow the name are called arguments. The computer will retrieve the values held by the variables and hand them off to the function for its use. The number of arguments (or values) must match the variables in the definition by type and number. A semicolon must be present, when the function is called within the main() function.

The general form of a ANSI method of function call is as follows:

```
function_name (actual parameters);
```

3.3.2. Defining functions:

Defining a function means writing an actual code for the function which does a specific and identifiable task. Suppose you are defining a function which computes the square of a given number. Then you have to write a set of instructions to do this.

The general form of a ANSI method of function definition is as follows:

```
type_specifier function_name (formal parameters)
{
        variable declaration;          /* with in the function */
        body of function;
                *
                *
                *
        return (value_computed);
 }
```

type_specifier specifies the type of value that the function's return statement returns. If nothing is returned to the calling function, then data type is **void**.

function_name is a user-defined function name. It must be a valid C identifier.

formal parameters is the type declaration of the variables of parameter list.

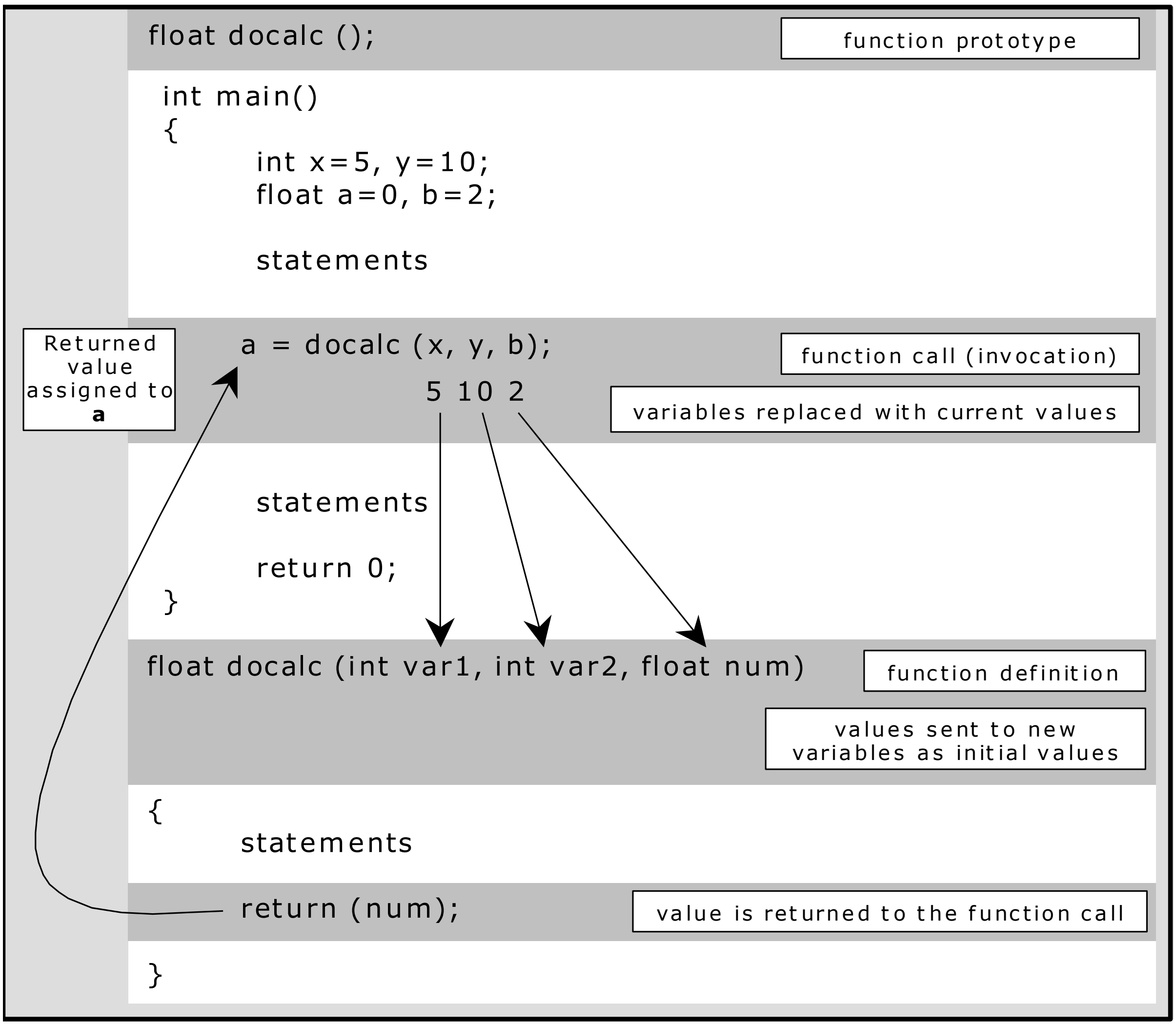
return is a keyword used to send the output of the function, back to the calling function. It is a means of communication from the called function to the calling function. There may be one or more return statements. When a return is encountered, the control is transferred to the calling function.

{ is the beginning of the function.

} is the end of function.

All the statements placed between the left brace and the corresponding right brace constitute the **body of a function**.

The following figure shows the flow through a function:



```
float docalc ();
 int main()
 {
        int x=5, y=10;
        float a=0, b=2;

        statements

      a = docalc (x, y, b);
                  5 10 2

        statements

        return 0;
 }
float docalc (int var1, int var2, float num)

{
      statements
      return (num);
}
```

function prototype
Returned value assigned to **a**
function call (invocation)
variables replaced with current values
function definition
values sent to new variables as initial values
value is returned to the function call

Any values that the function receives are called parameters. It is easy to confuse arguments and parameters. The values are called arguments in the statement that calls the function but become parameters in the header of the function definition. A semicolon does not follow the function header.

Example:

This program accepts two integers and computes their sum via function **addnums**().

```
# include <stdio.h>

int addnums(int, int);
main()
{
        int n1, n2, result;
        printf (“Enter two numbers:\n”);
        scanf (“%d %d”, &n1, &n2);
        result = addnums (n1, n2);                    /* Function call */
        printf (“The sum of %d and %d = %d\n”, n1, n2, result);
}

int addnums (int val1, int val2)            /* Function definition to add two numbers */
{
        int sum;
        sum=val1 + val2;
        return (sum);
}
```

Output:

```
Enter two numbers: 12 34
The sum of 12 and 34 = 46
```

In the above example, n1 and n2 are two input values. These are passed to the function addnums(). In function definition, the copies of n1 and n2 are obtained, namely val1 and val2. After, the required computations are made in the function, the control returns to the calling function along with the value computed. In the above example, the main program receives the value of 46. This is then assigned to the variable result. Finally, the value of the result is displayed.

A function may return int, float, char or double type data to the main (). However, there is a restriction is returning arrays. Whenever an array is to be returned, the return statement is not required. If the return value is not an int then explicitly the data type of the return value should be specified.

3.3.3. Function Declaration (Prototype):

Function declaration means specifying the function as a variable depending on the return value. It is declared in the declaration part of the main program. The default return value from a function is always an integer. If the function is returning a value other than an integer, it must be declared with the data type of the value it returns.

There is another reason for the function declaration. Functions may be written before or after the main(). If the functions are written before the main() then function declaration is not required. But, the functions which are written after the main() require function declaration. Function declaration informs the compiler, about the functions which are declared after the main().

A prototype may be created by copying the function header and pasting it above the main() definition and below the preprocessor directives (# include). A semicolon must follow a function prototype. The variable names do not need to be included in the prototype. If you include them, the compiler will ignore them.

The general form of function declaration in ANSI form is as follows:

Type_of_return_value function_name (parameter types);

Example:

1. float average ();
2. void message ();
3. int max_nums ();

3.4. Function Arguments/Parameters:

Arguments and parameters are the variables used in a program and a function. Variables used in the calling function are called arguments. These are written within the parentheses followed by the name of the function. They are also called *actual parameters*, as they are accepted in the main program (or calling function).

Variables used in the function definition (called function) are called parameters. They are also referred to as *formal parameters*, because they are not the accepted values. They receive values from the calling function. Parameters must be written within the parentheses followed by the name of the function, in the function definition.

Parameter passing mechanisms:

A method of information interchange between the calling function and called function is known as parameter passing. There are two methods of parameter passing:

1. Call by value
2. Call by reference

3.4.1. Call by Value:

The call-by-value method copies the *value* of an argument into the formal parameter of the function. Therefore, changes made by the function to the parameters have no effect on the variables used to call it.

Example:

```
# include <stdio.h>

void swap (int x, int y);                    /* function prototype */

main ()
{
        int x, y;
        x = 10;
        y = 20;
        swap (x, y);                         /* values passed */
}

void swap (int a, int b)                    /* function definition */
{
        int temp;

        temp = a;
        a = b;
        b = temp;
        printf ("%d %d\n", a, b);
 }
```

3.4.2. Call by Reference:

The call by reference method copies the *address* of an argument into the formal parameter. Inside the function, this address is used to access the argument used in the call. In this way, changes made to the parameter affect the variable used in the call to the function.

Call by reference is achieved by passing a pointer as the argument. Of course, in this case, the parameters must be declared as pointer types. The following program demonstrates this:

Example:

```
# include <stdio.h>

void swap (int *x, int *y);                 /* function prototype */

main ()
{
        int x, y;
        x = 10;
        y = 20;
        swap (&x, &y);                      /* addresses passed */
        printf ("%d %d\n", x, y);
}

void swap (int *a, int *b)
{
        int temp;

        temp = *a;
        *a = *b;
        *b = temp;
 }
```

The differences between call by value and call by reference:

Call by value	Call by reference
int a;	int a;
Formal parameter 'a' is a local variable.	Formal parameter is 'a' local reference.
It cannot change the actual parameter.	It can change the actual parameter.
Actual parameter may be a constant, a variable, or an expression.	Actual parameter must be a variable.

3.5. Category of functions:

We have some other type of functions where the arguments and return value may be present or absent. Such functions can be categorized into:

1. Functions with no arguments and no return value.
2. Functions with arguments and no return value.
3. Functions with no arguments but return value.
4. Functions with arguments and return value.

3.5.1. Functions with no arguments and no return value:

Here, the called functions does not receive any data from the calling function and, it does not return any data back to the calling function. Hence, there is no data transfer between the calling function and the called function.

Example:

This program illustrates the function with no arguments and no return value.

```
# include <stdio.h>
void read_name ();

main ()
{
        read_name ();
}
void read_value ()                          /*no return value */
{
        char name [10];
        printf ("Enter your name: ");
        scanf ("%s", name);
        printf ("\nYour name is %s: ", name);
}
```

Output:

```
        Enter your name: Herbert
        Your name is: Herbert
```

3.5.2. Functions with arguments and no return value:

Here, the called function receives the data from the calling function. The arguments and parameters should match in number, data type and order. But, the called function does not return and value back to the calling function. Instead, it prints the data in its scope only. It is one-way data communication between the calling function and the called function.

Example:

This program illustrates the function with arguments but has no return value.

```
#include <stdio.h>
```

```
void maximum (x, y);

main()
{
        int x, y;
        printf ("Enter the values of x and y: ");
        scanf ("%d %d", &x, &y);
        maximum (x, y);
}

void maximum (int p, int q)                    /* function to compute the maximum */
{
        if (p > q)
                printf ("\n maximum = %d", p);
        else
                printf ("\n maximum = %d" q);
        return;
}
```

Output:

```
        Enter the values of x and y: 14 10
        maximum = 14
```

Function with no arguments but return value:

Here, the called function does not receive any data from the calling function. It manages with its local data to carry out the specified task. However, after the specified processing the called function returns the computed data to the calling function. It is also a one-way data communication between the calling function and the called function.

Example:

This program illustrates the function with no arguments but has a return value.

```
#include <stdio.h>

float total ();

main ()
{
        float sum;
        sum = total ();
        printf ("Sum = %f\n", sum);
}

float total ()
{
        float x, y;
        x = 20.0;
        y = 10.0;
        return (x + y);
}
```

Output:

```
Sum = 30.000000
```

3.5.4. Function with arguments and return value:

When a function has arguments it receives data from the calling function and does some process and then returns the result to the called function. In this way the main() function will have control over the function.

Example:

```
/* program to find the sum of first N natural numbers */

#include <stdio.h>
int sum (int x);                              /* function prototype*/

void main ()
{
        int n;
        printf (“Enter the limit: ”);
        scanf (“%d”, &n);
        printf (“sum of first %d natural numbers is: %d”, n, sum(n));
}

int sum(int x)
{
        int i, result = 0
        for (i=1; i <= x; i++)
                result += i;
        return (result);
}
```

Output:

```
Enter the limit: 5
Sum of first 5 natural numbers is: 15
```

The main() is the calling function which calls the function sum(). The function sum() receives a single argument. Note that the called function (i.e., sum ()) receives its data from the calling function (i.e., main()). The return statement is employed in this function to return the sum of the n natural numbers entered from the standard input device and the result is displayed from the main() function to the standard output device. Note that int is used before the function name sum() instead of void since it returns the value of type int to the called function.

3.6. Important points to be noted while calling a function:

- *Parenthesis are compulsory after the function name.*

- *The function name in the function call and the function definition must be same.*

- *The type, number, and sequence of actual and formal arguments must be same.*

- *A semicolon must be used at the end of the statement when a function is called*.

- *The number of arguments should be equal to the number of parameters.*

- *There must be one-to-one mapping between arguments and parameters. i.e. they should be in the same order and should have same data type.*

- *Same variables can be used as arguments and parameters.*
- *The data types of the parameters must match or be closely compatible. It is very risky to call a function using an actual parameter that is floating point data if the formal parameter was declared as an integer data type. You can pass a float to a double, but should not do the opposite. You can also pass a short int to an int, etc. But you should not pass any type of integer to any type of floating point data or do the opposite.*

3.7. Nested Functions:

We have seen programs using functions called only from the main() function. But there are situations, where functions other than main() can call any other function(s) used in the program. This process is referred as nested functions.

Example:

```
#include <stdio.h>

void func1();
void func2();

void main()
{
        printf (“\n Inside main function”);
        func1();
        printf (“\n Again inside main function”);
}

void func1()
{
        printf (“\n Inside function 1”);
        func2();
        printf (“\n Again inside function 1”);
}

void func2()
{
        printf (“\n Inside function 2”);
}
```

Output:

```
        Inside main function
        Inside function 1
        Inside function 2
        Again inside function 1
        Again inside main function
```

Uses two functions func1() and func2() other than the main() function. The main() function calls the function func1() and the function func1() calls the function func2().

3.8. Local and Global Variables:

Both the calling function and called function are using their own variables. The existence of these variables is restricted to the calling function or to the called functions. This is known as *scope of the variables*. Based on this scope the variables can be classified into:

1. Local variables
2. Global variables

Variables whose existence is known only to the main program or functions are called local variables. On the other hand, variables whose existence is known to both main() as well as other functions are called global variables. Local variables are declared within the main program or a function. But, the global variables are declared outside the main() and other functions.

The following program illustrates the concept of local and global variables:

Example:

```
#include <stdio.h>

int i = 10;                         /* global variable declaration */

main()
{
        int j;                      /* local variable declaration */
        printf ("i = %d\n", i);
        j = value (i);
        printf ("j = %d\n", j);
}
int value (int i)                   /* function to compute value */
{
        int k;                      /* local variable declaration */
        k = i +10
        return (k);
}
```

Output:

```
        i = 10
```

```
j = 20
```

Distinguishing between local and global variables:

Local variables	Global variables
These are declared within the body of the function.	These are declared outside the function.
These variables can be referred only within the function in which it is declared.	These variables can be referred from any part of the program.
The value of the variables disappear once the function finishes its execution.	The value of the variables disappear only after the entire execution of the program.

3.9. Calling Functions with Arrays:

Generally, the C convention is the call-by-value. An exception is where arrays are passed as arguments. We can pass either an entire array as an argument or its individual elements.

Arrays in functions:

An array name represents the address of the first element in that array. So, arrays are passed to functions as pointers (a pointer is a variable, which holds the address of other variables).

When an array is passed to a function, only the *address* is passed, not the entire array. When a function is called with just the array name, a pointer to the array is passed. In C, an array name without an index is a pointer to the first element. This means that the parameter declaration must be of a compatible pointer type. There are three ways to declare a parameter which, is to receive an array pointer:

Example:

```
void display (int num[10]);
```

in which case the compiler automatically converts it to a pointer.

As an unsized array:

```
void display (int num[]);
```

which is also converted to a pointer.

As a pointer:

```
void display (int *num);
```

which, is allowed because any pointer may be indexed with [] as if it were an array.

An array element used as an argument is treated like any other simple variable, as shown in the example below:

Example:

```
main ()
{
            . . .
        display (t [a]);
            . . .
}

void display (int num)
{
        printf ("%d", num);
 }
```

3.10. The return statement:

The return causes an immediate exit from a function to the point from where the function is called. It may also be used to return one value per call. All functions, except those of type void, return a value. If no return statement is present, most compilers will cause a 0 to be returned. The return statement can be any one of the types as shown below:

1. return;
2. return ();
3. return (constant);
4. return (variable);
5. return (expression);
6. return (conditional expression);
7. return (function);

The first and second return statements, does not return any value and are just equal to the closing brace the function. If the function reaches the end without using a return statement, the control is simply transferred back to the calling portion of the program without returning any value. The presence of empty return statement is recommended in such situations.

The third return statement returns a constant function. For example:

```
if (x <= 10)
        return (1);
```

The fourth return statement returns a variable to the calling function. For example:

```
if (x <= 10)
        return (x);
```

The fifth return statement returns a value depending upon the expression specified inside the parenthesis. For example:

```
return (a + b * c);
```

The sixth return statement returns a value depending upon the result of the conditional expression specified inside the parenthesis. For example:

```
return (a > b ? 1 : 2);
```

The last return statement calls the function specified inside the parenthesis and collects the result obtained from that function and returns it to the calling function. For example:

```
return (pow (4, 2));
```

The parenthesis used around the expression or value used in a return statement is optional.

3.11. User Defined Data Types:

C allows the programmer to create custom data types in five different ways. These are:

- The **structure**, which is a collection of variables under one name.
- Types created with **typedef**, which defines a new name for an existing type.
- The **union**, which enables the same piece of memory to be defined as two or more types of variables.
- The **enumeration**, which is a list of symbols.
- The **bit-field**, a variation of the structure, which allows easy access to the bits of a word.

3.11.1. Structures:

A structure is a user defined data type. Using a structure we have the ability to define a new type of data considerably more complex than the types we have been using.

A structure is a collection of variables referenced under one name, providing a convenient means of keeping related information together. The variables which make up a structure are called **structure elements**.

In the following, the C keyword struct tells the compiler that a structure template is being defined:

```
struct addr
{
        char name[30];
        char street[40];
        int postalcode;
};
```

In this example, addr is known as the **structure tag**. A structure definition is actually a statement hence it is terminated by a semicolon.

A structure variable declaration of type addr is:

```
    struct addr addr_info;
```

Variables may also be declared as the structure is defined. This can be done as follows:

```
    struct addr
    {
        char name[30];
        char street[40];
        int postalcode;
    } addr_info, binfo, cinfo;
```

If only one variable is needed, the structure tag is not needed, for example:

```
    struct
    {
        char name[30];
        char street[40];
        int postalcode;
    } addr_info;
```

Structure Assignments:

ANSI C compilers allow the information in one structure to be assigned to another structure as:

```
    binfo = addr_info;
```

Declaring Structure Pointers:

Assuming the previously defined structure addr, the following declares addr_pointer as a pointer to data of that type:

```
      struct addr *addr_pointer;
```

Referencing Structure Elements:

Individual elements of a structure are referenced using the dot operator. The pointer to a structure member operator -> is used to access a member of a structure using a pointer variable.

Accessing Structure members using dot operator:

Individual elements of a structure are referenced using the dot operator, for example:

```
    addr_info. postalcode = 1234;
    printf ("%d",addr_info. postalcode);
```

Example:

```
# include <stdio.h>
# include <string.h>

struct
{
        char name[15];                      /*  childs name  */
        int  age;                           /*  childs age  */
        int  grade;                         /*  childs grade in school */
} boy, girl;

int main()
{
        strcpy (boy.name, "Herbert");
        boy.age = 15;
        boy.grade = 75;
        girl.age = boy.age - 1;             /* she is one year younger */
        girl.grade = 82;
        strcpy (girl.name, "Fousett");
        printf ("%s is %d years old and got a grade of %d\n",
                                                girl.name, girl.age, girl.grade);
        printf ("%s is %d years old and got a grade of %d\n",
                                                boy.name, boy.age, boy.grade);
        return 0;
}
```

Output:

```
Fousett is 14 years old and got a grade of 82
Herbert is 15 years old and got a grade of 75
```

Accessing Structure Pointers using -> operator:

Structure pointers may be used to generate a call by reference to a function and to create linked lists and other dynamic data structures. Call by reference can be used for structures to avoid overheads occurred by the push and pop operations of all the structure elements of the stack. For example if:

```
        struct addr_info *ptr;
```

To access the elements of a structure using a pointer, the -> operator is used:

```
          ptr -> postalcode
```

where, ptr has been declared as a pointer to the type of structure and assigned the address of a variable of that type, and postalcode is a structure element within that variable.

This can also be expressed as:

```
        (*ptr). postalcode
```

The parenthesis is necessary because the structure member operator ". " takes higher precedence than does the indirection operator.

Example:

```
#include <stdio.h>

struct
```

```
{
        char initial;
        int  age;
        int  grade;
} kids[12], *point, extra;

int main()
{
        int index;

        for (index = 0 ; index < 12 ; index++)
        {
                point = kids + index;
                point->initial = 'A' + index;
                point->age = 16;
                point->grade = 84;
        }

        kids[3].age = kids[5].age = 17;
        kids[2].grade = kids[6].grade = 92;
        kids[4].grade = 57;

        for (index = 0 ; index < 12 ; index++)
        {
                point = kids + index;
                printf("%c is %d years old and got a grade of %d\n",
                                (*point).initial, kids[index].age, point->grade);
        }
         extra = kids[2];                       /* Structure assignment */
        extra = *point;                         /* Structure assignment */

        return 0;
}
```

Output:

```
A is 16 years old and got a grade of 84
B is 16 years old and got a grade of 84
C is 16 years old and got a grade of 92
D is 17 years old and got a grade of 84
E is 16 years old and got a grade of 57
F is 17 years old and got a grade of 84
G is 16 years old and got a grade of 92
H is 16 years old and got a grade of 84
I is 16 years old and got a grade of 84
J is 16 years old and got a grade of 84
K is 16 years old and got a grade of 84
L is 16 years old and got a grade of 84
```

Arrays of Structures:

The following example declares a 100-element array of type addr:

```
        struct addr addr_arr [100];
```

in which the postalcode element of structure 3 would be accessed by:

```
        printf ("%d", addr_info [2]. postalcode);
```

Example:

```
#include <stdio.h>

struct
{
        char initial;
        int  age;
        int  grade;
} kids[12];

int main()
{
        int index;
        for (index = 0 ; index < 12 ; index++)
        {
                kids[index].initial = 'A' + index;
                kids[index].age = 16;
                kids[index].grade = 84;
        }

        kids[3].age = kids[5].age = 17;
        kids[2].grade = kids[6].grade = 92;
        kids[4].grade = 57;

        kids[10] = kids[4];                    /* Structure assignment  */

        for (index = 0 ; index < 12 ; index++)
                printf("%c is %d years old and got a grade of %d\n",
                                kids[index].initial, kids[index].age, kids[index].grade);
        return 0;
}
```

Output:

```
A is 16 years old and got a grade of 84
B is 16 years old and got a grade of 84
C is 16 years old and got a grade of 92
D is 17 years old and got a grade of 84
E is 16 years old and got a grade of 57
F is 17 years old and got a grade of 84
G is 16 years old and got a grade of 92
H is 16 years old and got a grade of 84
I is 16 years old and got a grade of 84
J is 16 years old and got a grade of 84
E is 16 years old and got a grade of 57
L is 16 years old and got a grade of 84
```

Passing Structures to Functions:

When an element of a non-global structure is to be passed to a function, the value of that element is passed (unless that element is complex, such as an array of characters).

Example:

```
        struct fred
        {
                char x;
                int y;
                char s[10];
        } mike;
```

each element would be passed like this:

```
        funct (mike.x);             /* passes character value of x  */
        funct (mike.y);             /* passes integer value of y  */
        funct (mike.s);             /* passes address of string  */
        funct (mike.s[2]);          /* passes character val of s[2]  */
```

If the address of the element is required to be passed, the & operator is placed before the structure name. In the above example, this would apply except in funct (mike.s);.

Passing entire Structures to Functions:

When a structure is used as an argument to a function, the entire structure is passed using the standard call-by-value method. Of course, this means that changes made to the contents of the structure inside the function do not affect the structure used as the argument.

Arrays and Structures with in Structures:

Structure elements may be arrays and structures. When a structure is an element of another structure, it is called a nested structure.

Example:

```
#include <string.h>

struct person
{
        char name[25];
        int  age;
        char status;                /* M = married, S = single */
};

struct alldat
{
        int grade;
        struct person descrip;
        char lunch[25];
};

int main()
{
        struct alldat student[53];
        struct alldat teacher, sub;
        teacher.grade = 94;
        teacher.descrip.age = 34;
        teacher.descrip.status = 'M';
        strcpy(teacher.descrip.name, "Mary Smith");
        strcpy(teacher.lunch, "Baloney sandwich");

        sub.descrip.age = 87;
        sub.descrip.status = 'M';
        strcpy(sub.descrip.name, "Old Lady Brown");
        sub.grade = 73;
        strcpy(sub.lunch, "Yogurt and toast");

        student[1].descrip.age = 15;
        student[1].descrip.status = 'S';
        strcpy(student[1].descrip.name, "Billy Boston");
        strcpy(student[1].lunch, "Peanut Butter");
        student[1].grade = 77;

        student[7].descrip.age = 14;
        student[12].grade = 87;

        return 0;
}
```

Output:

This program does not generate any output

Differences between arrays and structures:

Arrays	Structures
A array is an single entity representing a collection of data items of same data types.	A structure is a single entity representing a collection of data items of different data types.
Individual entries in an array are called elements.	Individual entries in a structure are called members.
An array declaration reserves enough memory space for its elements.	The structure definition reserves enough memory space for its members.
There is no keyword to represent arrays but the square braces [] preceding the variable name tells us that we are dealing with arrays.	The keyword struct tells us that we can dealing with structures.
Initialization of elements can be done during array declaration.	Initialization of members can be done only during structure definition.
The elements of an array are stored in sequence of memory locations.	The members of a structure are not stored in sequence of memory locations.
The array elements are accessed by its followed by the square braces [] within which the index is placed.	The members of a structure are accessed by the dot operator.
Its general format is data type variable name [size];	Its general format is: struct <struct name> { data_type structure member 1; data_type structure member 2; . . .

	data_type structure member N;
	} structure variable;

Example:
int sum [100];
	Example:
	struct student
	{
	char studname [25];
	int rollno;
	} stud1;

3.11.2. typedef:

typedef allows a **new data type** to be explicitly defined. Note that it does not actually create a new data *class*, but simply defines a new name for an *existing* type. This process helps in making machine-dependent code more portable, since only the typedef statements need be changed. It can also aid in self-documenting a program by allowing the use of more descriptive names for the standard data types.

The general form of defining a new data type is:

t*ypedef data type identifier*;

Where, identifier refers to new name(s) given to the data type. User defined types obey the same scope rules as identifiers, so if one defined in a function, it is recognized only within that function. Few examples of type definitions are

```
typedef int age;
typedef float average;
typedef char string;
```

Where *age* symbolizes int, *average* symbolizes float and *string* symbolizes char.
They can be later used to declare variables as:

```
age child, adult;
average mark1, mark2;
string name[20];
```

Where *child* and *adult* are declared as integer variables, *mark1* and *mark2* are declared as floating point variables and name is declared as character array variable.

The typedef can also be used to define structures. The main advantage of type definition is that, you can create meaningful data type names for increasing the readability of the program. They also suggest the purpose of the data type names used in the program.

3.11.3. Unions:

A **union** is a memory location, which is shared by two or more different variables, generally of different types, at different times. The general form of a union definition is:

```
union tag
{
        type variable_name;
        type variable_name;
        type variable_name;
                .
                .
        } union_variables;
```

An example definition is as follows:

```
        union u_type
        {
                int i;
                char ch;
        };
```

Declaration options are as for structures, i.e, variable names may be placed at the end of the definition or a separate declaration statement may be used.

A union shares the memory space instead of wasting storage on variables that are not being used. The compiler allocates a piece of storage that is large enough to hold the largest type in the union.

Accessing Union Elements:

union elements are accessed using the same methods used for accessing structure elements.

The union aids in the production of machine-independent code because the compiler keeps track of the actual sizes of the variables, which make up the union.

unions are frequently used when type conversions are needed because the data held in a union can be referred to in different ways.

Example 1:

```
union exam
{
        int roll_no;
        char name[15];
        int mark1,mark2,mark3;
} u1;

struct exam1
{
        int roll_no;
        char name[15];
        int mark1, mark2, mark3;
} s1;

void main()
{
        printf("The size of union is %d\n", sizeof(u1));
        printf("The size of structure is %d", sizeof(s1));
}
```

Output:

```
        The size of the union is 15
        The size of the structure is 23
```

In the above example, union *exam* has 5 members. The first member is a character array name having 15 characters (i.e., 15 bytes). The second member is of type int that requires 2 bytes for their storage. All the other members mark1, mark2, mark3 are of type int which requires 2 bytes for their storage. In the union, all these 5 members are allocated in a common place of memory (i.e., all the members share the same memory location). As union shares the memory space instead of wasting storage on variables that are not being used, the compiler allocates a piece of storage that is large enough to hold the largest type in the union. In the above declaration, the member name requires, 15 characters, which is the maximum of all members, hence a total memory space of 15 bytes is allocated. In case of structures, the total memory space allocated will be 23 (i.e. 15+2+2+2+2) bytes.

Example 2:

```
#include<stdio.h>
void main()
{
        union dec
        {
                int x;
                char name[4];
        } dec1;
        int i;
        dec1.x = 300;
        printf("Size of union = %d\n", sizeof(dec1));
        printf("The value of x = %u\t", dec1.x);
        printf("\n%u %u %u %u", dec1.name[0], dec1.name[1], dec1.name[2],
                        dec1.name[3]);
}
```

Output:

```
Size of union = 4
The value of x = 300
44      1       65461        74
```

The binary value of 300 is 0000 0001 0010 1100. As per the internal storage representation first 0010 1100 = 44 is stored then 0000 0010 = 01 is stored.

Example 3:

```
#include<stdio.h>
#include<string.h>
#include <conio.h>

struct hos
{
        char name[10];
        char hostelname[20];
};
```

```
struct daysch
{
        int phonenumber;
        char name[10];
        char address[40];
        int rollno;
};

union  student
{
        struct hos hostler;
        struct daysch dayscholar ;
} stu_data;

void main()
{
        int n;
        clrscr();
        printf("\n   MENU ");
        printf("\n 1. Hostler \n 2. Day Scholar\n");
        printf("\n enter the choice: ");
        scanf("%d",&n);
        if(n==1)
        {
                strcpy(stu_data.hostler.name,"Herbert");
                strcpy(stu_data.hostler.hostelname,"ss2");
                printf("\n student name: %s",stu_data.hostler.name);
                printf("\n hostle name: %s",stu_data.hostler.hostelname);
                printf("\n Union Data size: %d",sizeof(stu_data) );
                printf("\n Hostler Data size: %d",sizeof(stu_data.hostler) );
        }
        else if(n==2)
        {
                strcpy(stu_data.dayscholar.name,"Schildt");
                strcpy(stu_data.dayscholar.address,"balaji colony");
                stu_data.dayscholar.phonenumber=5620;
                stu_data.dayscholar.rollno = 1290;
                printf("\n student name: %s", stu_data.dayscholar.name);
                printf("\n address: %s", stu_data.dayscholar.address);
                printf("\n phone number: %d", stu_data.dayscholar.phonenumber);
                printf("\n roll number: %d", stu_data.dayscholar.rollno);
                printf("\n Union Data size: %d", sizeof(stu_data) );
                printf("\n Day Scholar Data size: %d", sizeof(stu_data.dayscholar) );
        }
        else
                printf("\n it is wrong choice ");
        getch();
}
```

Output:

RUN 1:

```
        MENU
    1.  Hostler
    2.  Day Scholar
```

```
Enter Your Choice: 2

Student name: Schildt
Address: balaji colony
Phone number: 5620
Roll no: 1290
Union data size: 54
Day scholar data size: 54

RUN 2:

          MENU
     1.  Hostler
     2.  Day Scholar

Enter Your Choice: 1

Student name: Herbert
Hostel name: ss2
Union data size: 54
Hostler data size: 30
```

In the above example, the declaration structure name hos requires 30 bytes and structure daysch requires 54 bytes of memory. In union stu_data both structures are allocated in a common place of memory (i.e. both are sharing same memory location). In our case a student can be either an hostler or a data scholar. So, the same memory can be used for both type of students. So instead of allocating 84 bytes of memory the same can be achieved using 54 bytes of memory.

Differences between structures and unions:

Structures	Unions
Each member in a structure occupies and uses its own memory space	All the members of a union use the same memory space
The keyword struct tells us that we are dealing with structures	The keyword union tells us that we are dealing with unions
All the members of a structure can be initialized	Only the first member of an union can be initialized
More memory space is required since each member is stored in a separate memory locations	Less memory space is required since all members are stored int eh same memory locations

Its general format is:
struct <struct Name>
{
data_type structure Member1;
data_type structure Member2;
.
.
data_type structure Member N;
} structure variable;
	Its general formula is:
	union <union Name>
	{
	data_type structure Member1;
	data_type structure Member2;
	.
	.
	data_type structure Member N;
	} union variable;

Example:
struct student
{
char studname[25];
int rollno;
} stud1;
	Example:
	union student
	{
	char studname[25];
	int rollno;
	} stud1;

3.11.4. Enumerations:

The keyword **enum** is used to declare enumeration types. It provides a means of naming a finite set, and of declaring identifiers as elements of the set. Enumerations are defined much like structures. An example definition for user defined type day is as follows:

```
        enum day {sun, mon, tue, wed, thu, fri, sat};
```

An example declaration is:

```
        enum day day_of_week;
```

This creates the user defined type enum day. The keyword enum is followed by the tag name day. The enumerators are the identifiers sun, mon, . . ., sat. They are constants of type int. By default the first one 0 and each succeeding one has the next integer value.

Initialization:

The enumerators can be initialized. Also we can declare variables along with the template. The following example do so:

```
        enum suit {clubs = 1, diamonds, hearts, spades} a, b, c;
```

clubs has been initialized to 1, diamonds, hearts and spades have the values 2, 3, and 4 respectively. a, b, and c are variables of this type.

As another example of initialization:

```
        enum fruit {apple = 7, pear, orange = 3, lemon} frt;
```

Because the enumerator apple has been initialized to 7, pear has value 8. Similarly, because orange has a value 3, lemon has a value 4. Multiple values are allowed, but the identifiers themselves must be unique.

Example:

```
# include <stdio.h>
# include <conio.h>

enum day {sun, mon, tue, wed, thu, fri, sat};

main()
{
        int n;
        clrscr ();
        printf (“Enter a day number (0 for sunday and so on upto 6):”);
        scanf (“%d”,&n);

        switch (n)
        {
                case sun:
                        printf(“\nSunday“);
                        break;
                case mon:
                        printf("\nMonday");
                        break;
                case tue:
                        printf("\nTuesday");
                        break;
                case wed:
                        printf("\nWednesday");
                        break;
                case thu:
                        printf("\nThursday");
                        break;
                case fri:
                        printf("\nFriday");
                        break;
                case sat:
                        printf("\nSaturday");
                        break;
        }
        getch();
}
```

3.11.5. Bit-Fields:

C has a built-in method for accessing a single bit within a byte. This is useful because:

- If memory is tight, several Boolean (true/false) variables can be stored in one byte, saving memory.
- Certain devices transmit information encoded into bits within one byte.
- Certain encryption routines need to access individual bits within a byte.

Tasks involving the manipulation of bits may, of course, be performed using C's bitwise operators. However, the **bit-field** provides a means of adding more structure and efficiency to the coding.

Bit-Fields - A Type of Structure:

To access bits using bit-fields, C uses a method based on the structure. In fact, a bit-field is simply a special type of structure element, which defines how long in bits the field is to be.

The general form of a bit-field definition is:

```
struct tag
{
        type name1: length;
        type name2: length;
        type name3: length;
 } variable_list;
```

A bit-field must be declared as either int, unsigned, or signed. Bit-fields of length 1 should be declared as unsigned because a single bit cannot have a sign.

An Example Application:

Bit-fields are frequently used when analyzing input from a hardware device. For example, the status port of a serial communications device might return a status byte like this:

Bit	Meaning When Set
0	Change in clear-to-send line.
1	Change in data-set-ready.
2	Trailing edge detected.
3	Change in receive line.
4	Clear-to-send.
5	Data-set-ready.
6	Telephone ringing.
7	Received signal.

Defining the Bit-Field:

The foregoing can be represented in a status byte using the following bit-field definition/declaration:

```
struct status_type
{
        unsigned delta_cts: 1;
        unsigned delta_dsr: 1;
        unsigned tr_edge: 1;
        unsigned delta_rec: 1;
        unsigned cts: 1;
        unsigned dsr: 1;
        unsigned ring: 1;
        unsigned rec_line: 1;
} status;
```

Using this bit-field, the following routine would enable a program to determine whether it can send or receive data:

```
status = get_port_status;
if(status.cts)
        printf("Clear to send");
if(status.dsr)
        printf("Data set ready);
```

Referencing Bit-Field Elements:

Values are assigned to a bit-field using the usual method for assigning values to structure elements, e.g.:

```
status.ring = 0;
```

As with structures, if an element is referenced through a pointer, the -> operator is used in lieu of the dot operator.

Variations in Definition:

All bit-fields need not necessarily be named, which allows unused ones to be bypassed. For example, if, in the above example, access had only been required to cts and dsr, status_type could have been declared like this:

```
        struct status_type
        {
                unsigned: 4;
                unsigned cts: 1;
                unsigned dsr: 1;
        } status;
```

Example:

```
#include <stdio.h>
union
{
        int index;
        struct
        {
                unsigned int x : 1;
                unsigned int y : 2;
                unsigned int z : 2;
        } bits;
} number;

int main ()
{
        for (number.index = 0 ; number.index < 20 ; number.index++)
        {
                printf ("index = %3d, bits = %3d%3d%3d\n", number.index,
                                        number.bits.z, number.bits.y, number.bits.x);
        }

        return 0;
}
```

Output:

```
index =   0, bits =   0  0  0
index =   1, bits =   0  0  1
index =   2, bits =   0  1  0
index =   3, bits =   0  1  1
index =   4, bits =   0  2  0
index =   5, bits =   0  2  1
index =   6, bits =   0  3  0
index =   7, bits =   0  3  1
index =   8, bits =   1  0  0
index =   9, bits =   1  0  1
index =  10, bits =   1  1  0
index =  11, bits =   1  1  1
index =  12, bits =  1  2  0
index =  13, bits =  1  2  1
index =  14, bits =  1  3  0
index =  15, bits =  1  3  1
index =  16, bits =  2  0  0
index =  17, bits =  2  0  1
index =  18, bits =  2  1  0
index =  19, bits =  2  1  1
```

Mixing Normal and Bit-Field Structure Elements:

Normal and bit-field structure elements may be mixed, as in:

```
struct emp
{
        struct addr address;
        float pay;
        unsigned lay-off: 1;            /*  lay-off or active  */
        unsigned hourly:  1;            /*  hourly pay or wage  */
        unsigned deducts: 3;            /*  tax deductions  */
}
```

which demonstrates the use of only one byte to store information which would otherwise require three bytes.

Limitations of bit-fields:

Bit-field variables have certain limitations. For example:

- The address of a bit-field variable cannot be taken.
- Bit-field variables cannot be arrayed.
- Integer boundaries cannot be overlapped.
- There is no certainty, from machine to machine, as to whether the fields will run from right to left or from left to right. In other words, any code using bit-fields may have some machine dependencies.

Key terms and Glossary

automatic: declared when entering the block, lost upon leaving the block; the declarations must be the first thing after the opening brace of the block

static: the variable is kept through the execution of the program, but it can only be accessed by that block

extern: available to all functions in the file AFTER the declaration; use extern to make the variable accessible in other files

register: automatic variable which is kept in fast memory; actual rules are machine-dependent, and compilers can often be more efficient in choosing which variables to use as registers.

Top-Down Design is an analysis method in which a major task is sub-divided into smaller, more manageable, tasks called functions. Each sub-task is then treated as a completely new analysis project. These sub-tasks may be sufficiently large and complex that they also require sub-division, and so on. The document that analysts use to represent their thinking related to this activity is referred to as a structure diagram.

Functions (also called as **procedures**, **modules**, or **subroutines**) are sections or blocks of code that perform steps or sub-steps in a program. Every C program contains at least one function. Its name must be "main" (all lowercase) and program execution always begins with it. Any function can call (execute) other functions. A function must be declared in the source code ahead of where it is called.

Predefined Functions are functions that are written and stored for use in future programs. Sometimes groups of these functions are organized by the type of purpose they perform and stored in header files as function libraries. For this reason, predefined functions are often referred to as library functions.

The **scope of an identifier** defines the area in program code where it has meaning. Identifiers that are defined *within* a function are local to that function, meaning that they will not be recognized outside of it. Identifiers that are defined *outside of* all functions are global, meaning that they will be recognized within *all* functions.

Call: The action of one function activating or making use of another functions is referred to as calling. A function must be declared in the source code ahead of the statement that calls it.

***declaration*:** specifies the type of the identifier, so that it can subsequently be used.

***definition*:** reserves storage for the object

The syntax for **declaring a function** is:

```
    return-type function-name (argument declarations)
    {
        local variable declarations
        statements
    }
```

The function prototype is a declaration and is needed if the function is defined after its use in the program. The syntax is:

```
    return-type function-name (argument declarations);
```

where, the argument declarations must include the types of the arguments, but the argument names are optional. If the function is defined before its use, then a prototype is not necessary, since the definition also serves as a declaration.

If the *return-type* is omitted, int is assumed. If there are no *argument declarations*, use void, not empty parentheses.

A **function prototype** is a statement (rather than an entire function declaration) that is placed ahead of a calling function in the source code to define the child function's label before it is used. The entire declaration must still be written later in the code, but may appear at a more convenient position following the calling function. A function prototype is simply a reproduction of a function header (the first line of a function declaration) that is terminated with a semi-colon.

Parameters are values that are passed between functions. The C statement that calls a function must include the function label followed by a parenthesized list of parameters that are being passed into the function. These parameters are referred to as **actual parameters** because they are the *actual* values being provided by the calling function. Actual parameters are often also referred to as **arguments**.

In the C code where the function is declared, a function header is written to declare the function identifier and to declare the identifiers, known as **formal parameters**, that will be used inside of the function to represent any data being passed into it. Formal parameters do not have to be identical to the actual parameters used in the calling statement, but must match in quantity and be compatible in data type.

The **scope of an identifier** defines the area in program code where it has meaning. Identifiers that are defined *within* a function are local to that function, meaning that they will not be recognized outside of it. Identifiers that are defined *outside of* all functions are global, meaning that they will be recognized within *all* functions.

References (labels) and **referencing** (use of those labels) involve the manner in which we use labels to refer to stored data.

Direct referencing is the use of a storage label to refer directly to the contents of a storage location. Direct referencing is often used in formulas when we want to use a stored value in the formula. The statement Y = X + 1; refers directly to the contents of the storage location X. The value stored in X will be added to 1.

Indirect referencing is the use of a storage label in such a way that the contents of one storage location (see "pointer" below) is used to refer *indirectly* to the contents of a different storage location. (See the web page about Pointers and Indirection.) Indirect referencing is often used to pass data back from a function to its parent when more than one parameter is involved. The statement *YP = 0; refers *indirectly* to the contents of a storage location that is pointed to by an address stored in YP. The identifier YP must be declared as a "pointer" (see below). The use of the star as an indirection operator in front of the identifier makes it an indirect reference. This operator also is often called the dereferencing operator because it prevents *direct reference* to the variable.

Address referencing is the use of a storage label to refer to its memory address rather than its contents. The statement scanf ("%d", &N); passes the *address* of storage location N to the scanf function. Had we used the identifier N without the preceding &, that would have passed the *contents* of the N storage location to the function instead of its address. A function can return a value of any type, using the return statement, The syntax is:

```
return exp;
return (exp);
return;
```

The **return** statement can occur anywhere in the function, and will immediately end that function and return control to the function which called it. If there is no return statement, the function will continue execution until the closing of the function definition, and return with an undefined value.

A **structure** is a data type which puts a variety of pieces together into one object. The syntax is given below:

```
struct structure-tag-optional
{
        member-declarations
        structure-names-optional ;
}

struct structure-tag structure-name;

structure-name. member ptr-to-structure -> member
```

typedef: typedef defines a new type, which can simplify the code. Here is the syntax:

```
typedef data-type TYPE-NAME;
typedef struct structure-tag TYPE-NAME;

typedef struct
{
        member-declarations
} TYPE-NAME;
```

union: With union, different types of values can be stored in the same location at different times. Space is allocated to accomodate the largest member data type. They are syntactically identical to structures, The syntax is:

```
union union-tag-optional
{
        member-declarations
} union-names-optional;

union-name. member

ptr-to-union -> member
```

enum: The type enum lets one specify a limited set of integer values a variable can have. For example, flags are very common, and they can be either true or false. The syntax is:

```
enum enum-tag-optional {enum-tags} enum-variable-names-optional;

enum-name variable-name
```

The values of the enum variable are integers, but the program can be easier to read when using enum instead of integers. Other common enumerated types are weekdays and months.

Chapter 4

Pointers, Files, Command Line Arguments and Preprocessor

4.1. Pointers:

Pointer is a fundamental part of C. If you cannot use pointers properly then you have basically lost all the power and flexibility that C allows. The secret to C is in its use of pointers. C uses *pointers* a lot because:

- It is the only way to express some computations.
- It produces compact and efficient code.
- Pointers provided an easy way to represent multidimensional arrays.
- Pointers increase the execution speed.
- Pointers reduce the length and complexity of program.

C uses pointers explicitly with arrays, structures and functions.

A pointer is a variable which contains the address in memory of another variable. We can have a pointer to any variable type.

The ***unary*** operator **&** gives the "address of a variable''. The ***indirection*** or dereference operator __*__ gives the "contents of an object ***pointed to*** by a pointer''.

To declare a pointer to a integer variable do:

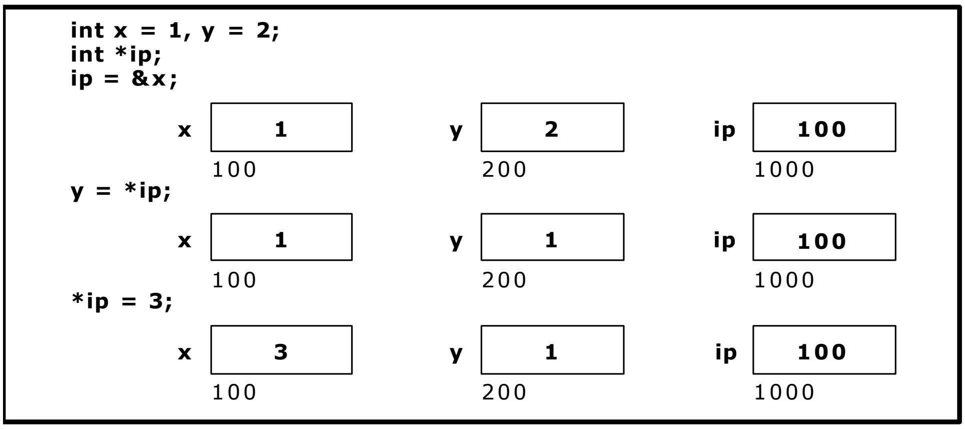
```
int *pointer;
```

We must associate a pointer to a particular type. We can't assign the address of a short int to a long int.

Consider the effect of the following code:

```
#include <stdio.h>
main()
{
        int x = 1, y = 2;
        int *ip;
        ip = &x;
        y = *ip;
        *ip = 3;
}
```

It is worth considering what is going on at the *machine level* in memory to fully understand how pointer works. Assume for the sake of this discussion that variable x resides at memory location 100, y at 200 and ip at 1000 shown in figure 4.1.



```
int x = 1, y = 2;
int *ip;
ip = &x;
```

```
y = *ip;
```

```
*ip = 3;
```

Fig. 4.1 Pointer, Variables and Memory

Now the assignments x = 1 and y = 2 obviously load these values into the variables. ip is declared to be a ***pointer to an integer*** and is assigned to the address of x (&x). So ip gets loaded with the value 100.

Next y gets assigned to the ***contents of*** ip. In this example ip currently ***points*** to memory location 100 -- the location of x. So y gets assigned to the values of x -- which is 1. Finally, we can assign a value 3 to the contents of a pointer (*ip).

IMPORTANT: When a pointer is declared it does not point anywhere. You must set it to point somewhere before you use it. So,

```
int *ip;
*ip = 100;
```

will generate an error (program crash!!). The correct usage is:

```
int *ip;
int x;
ip = &x;
*ip = 100;
++ip;
```

We can do integer arithmetic on a pointer, for example:

```
char m = ‘j’;
char *ch_ptr = &m;
float x = 20.55;
float *flp, *flq;
flp = &x;
*flp = *flp + 10;
++*flp;
(*flp)++;
flq = flp;
```

The reason we associate a pointer to a data type is that it knows how many bytes the data is stored in. When we increment a pointer we increase the pointer by one “block ‟ of memory.

So, for a character pointer ++ch_ptr adds 1 byte to the address. For an integer or float ++ip or ++flp adds 2 bytes and 4 bytes respectively to the address.

Here is another **example** showing some of the uses of pointers:

```
#include <stdio.h>

void main (void)
{
        int m = 0, n = 1, k = 2;
        int *p;
        char msg[] = "hello world";
        char *cp;
        p   = &m;                    /* p now points to m  */
        *p  = 1;                     /* m now equals 1 */
        k   = *p;                    /* k now equals 1 */
        cp  = msg;                   /* cp points to the first character of msg */
        *cp = 'H';                   /* change the case of the 'h' in msg       */
        cp  = &msg[6];               /* cp points to the 'w' */
        *cp = 'W';                   /* change its case  */
        printf ("m = %d, n = %d, k = %d\nmsg = \"%s\"\n", m, n, k, msg);
}
```

Output:

```
m = 1, n = 1, k = 1
msg = “Hello World”
```

Note the very important point that the name of an array (`msg' in the above example), if used without an index, is considered to be a pointer to the first element of the array.

In fact, an array name followed by an index is exactly equivalent to a pointer followed by an offset.

Example:

```
#include <stdio.h>

void main (void)
{
        char msg[] = "hello world";
        char *cp;
        cp = msg;
        cp[0]     = 'H';
        *(msg+6)  = 'W';
        printf ("%s\n", msg);
        printf ("%s\n", &msg[0]);
        printf ("%s\n", cp);
        printf ("%s\n", &cp[0]);
}
```

Output:

```
Hello World
Hello World
Hello World
Hello World
```

Note, however, that `cp' is a variable, and can be changed, whereas`msg' is a constant, and is not an lvalue.

4.1.1. Pointers and Arrays:

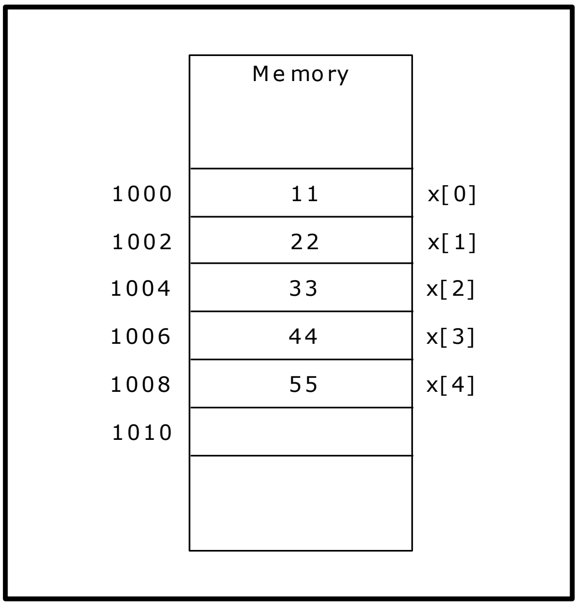
There is a close association between pointers and arrays. Let us consider the following statements:

```
int x[5] = {11, 22, 33, 44, 55};
int *p = x;
```

The array initialization statement is familiar to us. The second statement, array name x is the starting address of the array. Let we take a sample memory map as shown in figure 4.2.:

From the figure 4.2 we can see that the starting address of the array is 1000. When x is an array, it also represents an address and so there is no need to use the (&) symbol before x. We can write int *p = x in place of writing int *p = &x[0].

The content of p is 1000 (see the memory map given below). To access the value in x[0] by using pointers, the indirection operator * with its pointer variable p by the notation *p can be used.



	Memory	
1000	11	x[0]
1002	22	x[1]
1004	33	x[2]
1006	44	x[3]
1008	55	x[4]
1010		

Figure 4.2. Memory map - Arrays

The increment operator ++ helps you to increment the value of the pointer variable by the size of the data type it points to. Therefore, the expression p++ will increment p by 2 bytes (as p points to an integer) and new value in p will be 1000 + 2 = 1002, now *p will get you 22 which is x[1].

Consider the following expressions:

```
*p++;
*(p++);
(*p)++;
```

How would they be evaluated when the integers 10 & 20 are stored at addresses 1000 and 1002 respectively with p set to 1000.

p++ : The increment ++ operator has a higher priority than the indirection operator * . Therefore p is increment first. The new value in p is then 1002 and the content at this address is 20.

*(p++): is same as *p++.

(*p)++: *p which is content at address 1000 (i.e. 10) is incremented. Therefore (*p)++ is 11.

Note that, *p++ = content at incremented address.

Example:

```
#include <stdio.h>

main()
{
        int x[5] = {11, 22, 33, 44, 55};
        int *p = x, i;                          /*  p=&x[0] = address of the first element */
        for (i = 0; i < 5; i++)
        {
                printf ("\n x[%d] = %d", i, *p);            /* increment the address*/
                p++;
        }
}
```

Output:

```
x [0] = 11
x [1] = 22
x [2] = 33
x [3] = 44
x [4] = 55
```

The meanings of the expressions p, p+1, p+2, p+3, p+4 and the expressions *p, *(p+1), *(p+2), *(p+3), *(p+4) are as follows:

P = 1000 P+1 = 1000 + 1 x 2 = 1002 P+2 = 1000 + 2 x 2 = 1004 P+3 = 1000 + 3 x 2 = 1006 P+4 = 1000 + 4 x 2 = 1008	*p = content at address 1000 = x[0] *(p+1) = content at address 1002 = x[1] *(p+2) = content at address 1004 = x[2] *(p+3) = content at address 1006 = x[3] *(p+4) = content at address 1008 = x[4]

4.1.2. Pointers and strings:

A string is an array of characters. Thus pointer notation can be applied to the characters in strings. Consider the statements:

```
char tv[20] = "ONIDA";
char *p = tv;
```

For the first statement, the compiler allocates 20 bytes of memory and stores in the first six bytes the char values as shown below:

Variable	tv[0]	tv[1]	tv[2]	tv[3]	tv[4]	tv[5]
Value	‘O’	‘N’	‘I’	‘D’	‘A’	‘\0’
Address	1000	1001	1002	1003	1004	1005

The statement:

```
        char *p = tv;                               /* or  p = &tv[0] */
```

Assigns the address 1000 to p. Now, we will write a program to find the length of the string tv and print the string in reverse order using pointer notation.

Example:

```
#include <stdio.h>

main()
{
        int n, i;
        char tv[20] = “ONIDA”;                      /* p = 1000 */
        char *p = tv, *q;                           /* p = &tv[0], q is a pointer */
        q = p;
        while (*p != ‘\0’)               /* content at address of p is not null character */
                p++;
        n = p - q;                                  /* length of the string */
        --p;                     /* make p point to the last character A in the string */
        printf (“\nLength of the string is %d”, n);
        printf (“\nString in reverse order: \n”);
        for (i=0; i<n; i++)
        {
                putchar (*p);
                p--;
        }
}
```

Output:

```
Length of the string is 5
String in reverse order: ADINO
```

4.1.3. Pointers and Structures:

You have learnt so far that pointers can be set to point to an int, float, char, arrays and strings. Now, we will learn how to set pointers to structures. Consider the structure definition.

```
        struct student
        {
                int rollno;
                char name [20];
        };
```

and the declarations:

```
        struct student s;
        struct student *ps = &s;
```

in the last statement, we have declared a pointer variable ps and initialized it with the address of the structure s. We say that ps points to s. To access the structure members with the help of the pointer variable ps, we use the following syntax:

```
        ps → rollno     (or)    (*ps).rollno
        ps → name       (or)    (*ps).name
```

The symbol → is called *arrow operator* and is made up of a minus sign and a greater than sign. The parentheses around ps are necessary because the member operator (.) has a higher precedence than the indirection operator (*).

We will now write a program to illustrate the use of structure pointers.

```
# include <stdio.h>
# include <conio.h>

struct invent
{
        char name[20];
        int number;
        float price;
};

main()
{
        float temp;
        struct invent product[3], *ps;
        int size ;
        ps = &product[0];
        printf("input product details:");
        size = sizeof(product[0]);
        printf("\n sizeof(product[0]) = %d",size );
        printf("\n product = %u ",product);
        printf("\n &product[0] = %u ",&product[0]);
        printf("\n &product[1] = %u ",&product[1]);
        printf("\n &product[2] = %u ",&product[2]);
        printf("\nproduct + 3 = %u\n",(product+3) );
        printf("\n Name \t Number \t Price\n");
        for (ps=product; ps < product+3; ps++)
        {
                scanf ("%s %d %f", ps->name, &ps->number, &temp);
                ps->price = temp;
        }
        printf("\n Item Details...\n Name\tNumber\t Price\n");
        for (ps=product; ps < product+3; ps++)
                printf ("\n%s %d %f", ps->name, ps->number, ps->price);
        getch();
}
```

Output:

```
input Product details:
```

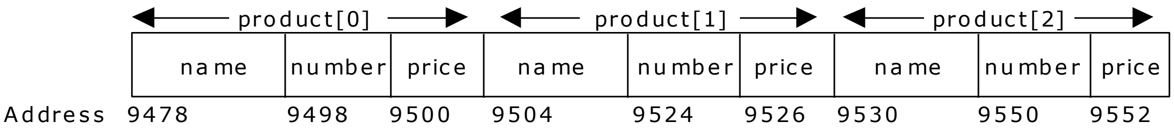
```
sizeof(product[0]) = 26
product = 9478
&product[0] = 9478
&product[1] = 9504
&product[2] = 9530
product + 3 = 9556

Name    Number   Price
Pen     101      10.45
Pencil  102      5.67
Book    103      12.63

Item Details…….

Name    Number   Price
Pen     101      10.45
Pencil  102      5.67
Book    103      12.63
```

The compiler reserves the memory locations as shown below:



4.1.4. Pointer and Function:

Pointer has deep relationship with function. We can pass pointer to the function and also addresses can be passed to the function as pointer argument. This can be represented by two ways as defined below:

1. Pointer as function argument.
2. Pointer to function.

4.1.4.1. Pointer as function argument:

This is achieved by call by reference. The call by reference method copies the *address* of an argument into the formal parameter. Inside the function, this address is used to access the argument used in the call. In this way, changes made to the parameter affect the variable used in the call to the function.

Call by reference is achieved by passing a pointer as the argument. Of course, in this case, the parameters must be declared as pointer types. The following program demonstrates this:

Example:

```
# include <stdio.h>

void swap (int *x, int *y);                    /* function prototype */

main ()
{
        int x, y;
        x = 10;
        y = 20;
        swap (&x, &y);                          /* addresses passed */
        printf ("%d %d\n", x, y);
}
void swap (int *a, int *b)
{
        int temp;

        temp = *a;
        *a = *b;
        *b = temp;
 }
```

4.1.4.2. Pointer to Function:

A function works like a variable. It is also similar to data type declaration like the other variable. Pointer to a function variable occurs when you define a function as a pointer. As function has address location in memory the syntax used to declare pointer to function is as:

*Return _type (*function name) (list of arguments);*

or

*return_type (*fptr) (void);*

or

*void (*fptr) (float, int);*

Where, return_type is the data type which may be integer, float or character and *(fptr) (argument list) is pointer to function. The procedure to illustrate this concept is as follows:

```
        void add (int, int);
        void (*f1) (int, int);
        f1 = & add;
```

By using the above example, we say that add() is a simple function. But f1 is pointer variable which work as pointer to function and third statement the address of the add() function is assigned to f1 variable which is a memory variable. So, (*f1)(int, int) is a function pointer linked to the function add.

Example:

```
/* Program to add and substract two numbers by using the pointer to function. */

#include <stdio.h>

void add (int, int);
void sub (int, int);
void (*f1) (int, int);

main()
{
        f1 = & add;
        (*f1) (10, 15);
        f1 = & sub;
        (*f1) (11, 7);
}

void add (int a, int b)
{
        printf ("\n Sum = %d", a + b);
}

void sub (int a, int b)
{
        printf("\n sub = %d", a - b);
}
```

Output:

```
Sum = 25
Sub = 4
```

4.1.5. Array of Pointers:

We can have array of pointers since pointers are variables. ***Array of Pointers*** will handle variable length text lines efficiently and conveniently. How can we do this is:

Example:

```
#include <stdio.h>

void main(void)
{
          char *p[5];
          int x;
          p[0] = "DATA";
          p[1] = "LOGIC";
          p[2] = "MICROPROCESSOR";
          p[3] = "COMPUTER";
          p[4] = "DISKETTE";
          for (x = 0; x < 5; ++x)
                printf("%s\n", p[x]);
}
```

Output:

```
DATA
LOGIC
MICROPROCESSOR
COMPUTER
DISKETTE
```

4.1.6. Multidimensional arrays and pointers:

A 2D array is really a 1D array, each of whose elements is itself an array. Hence:

a[n][m] notation.

Array elements are stored row by row.

When we pass a 2D array to a function we must specify the number of columns -- the number of rows are irrelevant. The reason for this is pointers again. C needs to know how many columns in order that it can jump from row to row in memory.
Consider int a[5][35] to be passed in a function:

We can do:

```
fun (int a[][35])
{
        … statements...
}
```

or even:

```
fun (int (*a)[35])
{
        …statements...
}
```

We need parenthesis (*a) since [] have a higher precedence than *. So:

```
int (*a)[35];  declares a pointer to an array of 35 ints.
int *a[35];    declares an array of 35 pointers to ints.
```

Now lets look at the difference between pointers and arrays. Strings are a common application of this. Consider:

```
char *name[10];
char aname[10][20];
```

- **aname** is a <u>true</u> 200 elements 2D char array.
- **name** has 10 pointer elements.

The advantage of the latter is that each pointer can point to arrays of different length.
Let us consider (shown in figure 4.3):

```
char *name[10] = {"no month", "jan", "feb"};
char aname[10][20] = {"no month'', "jan", "feb"};
```

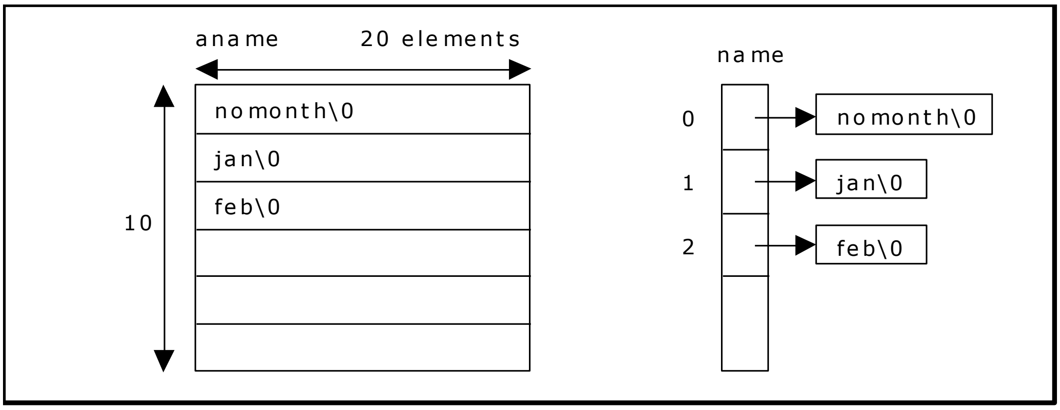


Fig. 4.3. 2D Arrays and Arrays of Pointers

4.1.7. Pointer to Pointer:

The concept of a pointer having the exclusive storage address of another pointer is called as pointer to pointer. Normally we do not use pointers to pointers very often but they can exist. The following program demonstrates this:

Example:

```
#include <stdio.h>

void main(void)
{
            int x, *p, **ptp;
             x = 454;
             p = &x;
             ptp = &p;
             printf("%d  %d\n", *p, **ptp);
}
```

Output:

```
454    454
```

**ptp is declared as a pointer to pointer of type int. This means that ptp handles the address of another pointer p. Variable x is assigned a value of 454, then its address is assigned to pointer p. Next, the address of pointer p is assigned to pointer to a pointer ptp. **ptp accesses the value of x.

4.1.8. Illegal indirection:

Suppose we have a function malloc() which tries to allocate memory dynamically (at run time) and returns a pointer to block of memory requested if successful or a NULL pointer otherwise.

(char *) malloc() -- a standard library function (see later).

Let us have a pointer: char *p;

For example, let us consider:

```
1.      *p = (char *) malloc(100);          /* request 100 bytes of memory */
2.      *p = `y';
```

There is a mistake above. That is, * before p in the statement 1. The correct code should be:

```
p = (char *) malloc(100);
```

If no memory is available for allocation p will be NULL. Therefore we can't do:

```
*p = `y';
```

A good C program should check for this:

```
p = (char *) malloc (100);
if ( p == NULL)
{
        printf("Error: Out of Memory\n");
        exit(1);
}
*p = 'y';
```

4.1.9. Dynamic Memory Allocation and Dynamic Structures:

Dynamic allocation is a unique feature to C among high level languages. It enables us to create data types and structures of any size and length to suit our program need. The process of allocating memory at run time is known as dynamic memory allocation.

The four important memory management functions for dynamic memory allocation and reallocation are:

1. malloc
2. calloc
3. free
4. realloc

The malloc function:

The function malloc is used to allocate a block of memory of specified size. It is defined by:

```
void *malloc (number_of_bytes)
```

The malloc function returns a pointer to the first byte of the allocated memory block.

Since a void * is returned the C standard states that this pointer can be converted to any type. For example,

```
char *cp;
cp = (char *) malloc (100);
```

attempts to get 100 bytes and assigns the starting address to cp.

We can also use the sizeof() function to specify the number of bytes. For example

```
int *ip;
```

```
ip = (int *) malloc (100*sizeof(int));
```

Some C compilers may require to cast the type of conversion. The (int *) means force to an integer pointer. Forcing to the correct pointer type is very important to ensure pointer arithmetic is performed correctly.

It is good practice to use sizeof() even if you know the actual size you want - it makes for device independent (portable) code.

The sizeof can be used to find the size of any data type, variable or structure. Simply supply one of these as an argument to the function. So:

```
int i;
struct complex
{
        int real;
        int imaginary;
};
typedef struct complex comp;
```

sizeof(int), sizeof(i),
sizeof(struct complex) and
sizeof(comp) are all ACCEPTABLE.

The free function:

It frees (releases) the memory space allocated for a block. The syntax is:

```
free (ptr);
```

This releases the block of memory allocated which is currently pointed to by the pointer variable ptr. The advantage is simply memory management when we no longer need a block.

The calloc and realloc functions:

There are two additional memory allocation functions, calloc() and realloc(). Their prototypes are given below:

```
void *calloc ( num_elements,  element_size};

void *realloc ( void *ptr,  new_size);
```

malloc does not initialise memory (to ***zero***) in any way. If you wish to initialise memory then use calloc. The calloc is slightly more computationally expensive but, occasionally, more convenient than malloc. The syntax difference between calloc and malloc is that calloc takes the number of desired elements, num_elements, and element_size, as two individual arguments.

Thus to assign 100 integer elements that are all initially zero you would do:

```
int *ip;
ip = (int *) calloc (100, sizeof(int));
```

The realloc is a function, which attempts to change the size of a previous allocated block of memory by malloc function. The new size can be larger or smaller. If the block is made larger than the older, it will extend only if it can find additional space in the same region. Otherwise, it may create entirely a new region (again, if space is available) and move the contents of the old block into the new block without any loss.

The realloc function to increase the size of block to 200 integers instead of 100, simply do:

```
        ip = (int *) calloc (ip, 200);
```

If the size is to be made smaller then the remaining contents are unchanged. The realloc function to reduce the size of block to 50 integers instead of 200, simply do:

```
        ip = (int *) calloc (ip, 50);
```

Example:

```
# include <stdio.h>
# include <alloc.h>
# include <string.h>

main()
{
        char *ptr, *qtr;
        ptr = (char *) malloc ( 12);
        strcpy(ptr, "hello world" );
        printf("\n Block now contains: %s", ptr);
        qtr =(char *) realloc(ptr,25);
        strcpy(qtr, "Hello beautiful world");
        printf("\n The Block contents after reallocation: %s", qtr);
        ptr =(char *) realloc(qtr, 5);
        printf("\n After reducing the size: %s", ptr);
}
```

Output:

```
Block now contains: hello world
The Block contents after reallocation: Hello beautiful world
After reducing the size: Hello
```

The differences between the address stored in a pointer and a value at the address:

The address stored in the pointer is the address of another variable. The value stored at that address is a stored in a different variable. The indirection operator (*) returns the value stored at the address, which itself is stored in a pointer.

The differences between the indirection operator and the address of operator:

The indirection operator (*) returns the value of the address stored in a pointer. The address of operator (&) returns the memory address of the variable.

4.2. Files:

File is a bunch of bytes stored in a particular area on some storage devices like floppy disk, hard disk, magnetic tape and cd-rom etc., which helps for the permanent storage.

There are 2 ways of accessing the files. These are:

- Sequential access: the data can be stored or read back sequentially.
- Random access: the data can be access randomly.

If a file can support random access (sometimes referred to as position requests), opening a file also initializes the file position indicator to the start of a file. This indicator is incremented as each character is read from, or written to, the file.

The close operation disassociates a file from a specific stream. If the file was opened for output, the close operation will write the contents (if any) of the stream to the device. This process is usually called flushing the stream.

All files are closed automatically when the program terminates, but not when it crashes. Each stream associated with a file has a file control structure of type ***FILE***.

4.2.1. Streams:

Even though different devices are involved (terminals, disk drives, etc), the buffered file system transforms each into a logical device called a **stream**. Because streams are device-independent, the same function can write to a disk file or to another device, such as a console. There are two types of streams:

Text Streams: A text stream is a sequence of characters. In a text stream, certain character translations may occur (for example, a newline may be converted to a carriage return/line-feed pair). This means that there may not be a one-to-one relationship between the characters written and those in the external device.

Binary Streams: A binary stream is a sequence of bytes with a one-to-one correspondence to those on the external device (i.e, no translations occur). The number of bytes written or read is the same as the number on the external device. However, an implementation-defined number of bytes may be appended to a binary stream (for example, to pad the information so that it fills a sector on a disk).

4.2.2. File Input and Output functions:

The ANSI file system comprises several interrelated functions. These are:

Function	Description
fopen()	Opens a file.
fclose()	Closes a file.
putc()	Writes a character.
fputc()	Writes a character.
getc()	Reads a character.
fgetc()	Reads a character.
fseek()	Seeks a specified byte in a file.

fprintf()	Is to a file what printf() is to the console.
fscanf()	Is to a file what scanf() is to a console.
feof()	Returns TRUE if end-of-file is reached.
ferror()	Returns TRUE if an error is detected.
rewind()	Resets file position to beginning of file.
remove()	Erases a file.
fflush()	Flushes a file.

Most of these functions begin with the letter "f". The header file stdio.h provides the prototypes for the I/O function and defines these three types:

```
typedef  unsigned long    size_t
typedef  unsigned long    fpos_t
typedef  struct _FILE      FILE
```

stdio.h also defines the following:

```
EOF              -1            /* value returned at end of file */
SEEK_SET          0            /* from beginning of file  */

SEEK_CUR          1            /* from current position  */
SEEK_END          2            /* from end of file  */
```

The latter three are used with fseek() function which performs random access on a file.

4.2.3. The File Pointer:

C treats a file just like a stream of characters and allows input and output as a stream of characters. To store data in file we have to create a buffer area. This buffer area allows information to be read or written on to a data file. The buffer area is automatically created as soon as the file pointer is declared. The general form of declaring a file is:

```
FILE *fp;
```

FILE is a defined data type, all files should be declared as type FILE before they are used. FILE should be compulsorily written in upper case. The pointer fp is referred to as the stream pointer. This pointer contains all the information about the file, which is subsequently used as a communication link between the system and the program.

A file pointer fp is a variable of type FILE that is defined in stdio.h.

4.2.4. Opening a File:

fopen() opens a stream for use, links a file with that stream and returns a pointer associated with that file. The prototype of fopen() function is as follows:

FILE *fopen (const char * *filename*, const char * *mode*);

Where, filename is a pointer to a string of characters that make a valid filename (and may include a path specification) and mode determines how the file will be opened. The legal values for mode are as follows:

Value	Description
r	Open a text file for reading.
w	Create a text file for writing.
a	Append to a text file.
rb	Open a binary file for reading.
wb	Create a binary file for writing.
ab	Append to a binary file.
r+	Open a text file for read/write.
w+	Create a text file for read/write.
a+	Append or create a text file for read/write.
r+b	Open a binary file for read/write.
w+b	Create a binary file for read/write.
a+b	Append a binary file for read/write.

A file may be opened in text or binary mode. In most implementations, in text mode, CR/LF sequences are translated to newline characters on input. On output, the reverse occurs. No such translation occurs on binary files.

The following opens a file named TEST for writing:

```
FILE *fp;
fp = fopen ("test", "w");
```

However, because fopen() returns a null pointer if an error occurs when a file is opened, this is better written as:

```
FILE *fp;
if ((fp = fopen ("test", "w")) == NULL)
{
        printf("cannot open file\n");
        exit(1);
}
```

4.2.5. Closing a File:

fclose() closes the stream, writes any data remaining in the disk buffer to the file, does a formal operating system level close on the file, and frees the associated file control block. fclose() has this prototype:

```
int fclose (FILE *fp);
```

A return value of zero signifies a successful operation. Generally, fclose() will fail only when a disk has been prematurely removed or a disk is full.

4.2.6. Writing a Character:

Characters are written using putc() or its equivalent fputc(). The prototype for putc() is:

```
int putc (int ch, FILE *fp);
```

where ch is the character to be output. For historical reasons, ch is defined as an int, but only the low order byte is used.

If the putc() operation is successful, it returns the character written, otherwise it returns EOF.

4.2.7. Reading a Character:

Characters are read using getc() or its equivalent fgetc(). The prototype for getc() is:

```
int getc(FILE *fp);
```

For historical reasons, getc() returns an integer, but the high order byte is zero. getc() returns an EOF when the end of file has been reached. The following code reads a text file to the end:

```
do
{
        ch = getc (fp);
} while(ch != EOF);
```

4.2.8. Using feof():

As previously stated, the buffered file system can also operate on binary data. When a file is opened for binary input, an integer value equal to the EOF mark may be read, causing the EOF condition. To solve this problem, C includes the function feof(), which determines when the end of the file is reached when reading binary data.

The prototype is:

```
int feof (FILE *fp);
```

The following code reads a binary file until end of file is encountered:

```
while (! feof (fp))
        ch = getc(fp);
```

Of course, this method can be applied to text files as well as binary files.

4.2.9. Working With Strings - fputs() and fgets():

In addition to getc() and putc(), C supports the related functions fputs() and fgets(), which read and write character strings. They have the following prototypes:

```
int fputs (const char *str, FILE *fp);

char *fgets (char *str, int length, FILE *fp);
```

The function fputs() works like puts() but writes the string to the specified stream. The fgets() function reads a string until either a newline character is read or length-1 characters have been read. If a newline is read, it will be part of the string (unlike gets()). The resultant string will be null-terminated.

4.2.10. rewind ():

rewind() resets the file position indicator to the beginning of the file. The syntax of rewind() is:

```
rewind(fptr);
```

where, fptr is a file pointer.

4.2.11. ferror ():

ferror() determines whether a file operation has produced an error. It returns TRUE if an error has occurred, otherwise it returns FALSE. ferror() should be called immediately after each file operation, otherwise an error may be lost.

4.2.12. Erasing Files:

remove () erases a specified file. It returns zero if successful.

4.2.13. Flushing a Stream:

fflush () flushes the contents of an output stream. fflush() writes the contents of any unbuffered data to the file associated with fp. It returns zero if successful.

4.2.14. fread () and fwrite ():

To read and write data types which are longer than one byte, the ANSI standard provides fread() and fwrite(). These functions allow the reading and writing of blocks of any type of data. The prototypes are:

size_t fread (void **buffer*, size_t *num_bytes*, size_t *count*, FILE **fp*);

size_t fwrite (const void **buffer*, size_t *num_bytes*, size_t *count*, FILE **fp*);

For fread(), *buffer* is a pointer to a region of memory which will receive the data from the file. For fwrite(), *buffer* is a pointer to the information which will be written. The buffer may be simply the memory used to hold the variable, for example, &l for a long integer.

The number of bytes to be read/written is specified by *num_bytes*. count determines how many items (each *num_bytes* in length) are read or written.

fread() returns the number of items read. This value may be less than *count* if the end of file is reached or an error occurs. fwrite() returns the number of items written.

One of the most useful applications of fread() and fwrite() involves reading and writing user-defined data types, especially structures. For example, given this structure:

```
struct struct_type
{
        float balance;
        char name[80];
} cust;
```

The following statement writes the contents of cust:

```
fwrite (&cust, sizeof(struct struct_type), 1, fp);
```

4.2.15. fseek() and Random Access I/O:

Random read and write operations may be performed with the help of fseek(), which sets the file position locator. The prototype is:

```
int fseek(FILE *fp, long numbytes, int origin);
```

in which *numbytes* is the number of bytes from the origin, which will become the new current position, and origin is one of the following macros defined in stdio.h:

Origin	Macro Name
Beginning of file	SEEK_SET
Current position	SEEK_CUR
End-of-file	SEEK_END

fseek() returns 0 when successful and a non-zero value if an error occurs. fseek() may be used to seek in multiples of any type of data by simply multiplying the size of the data by the number of the item to be reached, for example:

```
fseek (fp, 9*sizeof (struct list), SEEK_SET);
```

Which seeks the tenth address.

4.2.16. fprint() and fscanf():

fprint() and fscanf() behave exactly like print() and scanf() except that they operate with files. The prototypes are:

```
int fprintf (FILE *fp, const char *control_string, ...);
int fscanf (FILE *fp, const char *control_string, ...);
```

Although these functions are often the easiest way to read and write assorted data, they are not always the most efficient. Because formatted ASCII data is being written as it would appear on the screen (instead of in binary), extra overhead is incurred with each call. If speed or file size is of concern, use fread() and fwrite().

4.2.17. The Standard Streams:

Whenever a C program starts execution, three streams are opened automatically. These are:

- Standard input (stdin)
- Standard output (stdout)
- Standard error (stderr)

Normally, these streams refer to the console, but they may be redirected by the operating system to some other device or environment. Because the standard streams are file pointers, they may be used to provide buffered I/O operations on the console, for example:

```
putchar(char c)
{
      putc(c, stdout);
}
```

4.3. Command Line Arguments:

Some times it is very useful to pass information into a program when we run it from the command prompt. The general method to pass information into main() function is through the use of command line arguments.

A command line argument is the information that follows the program's name on the command prompt of the operating system.

For example: TC program_name

There are three special built_in_arguments to main(). They are:

- The first argument is argc (**arg**ument **c**ount) must be an integer value, which represents the number arguments in the command prompt. It will always be at least one because the name of the program qualifies as the first argument.

- The second argument argv (**arg**ument **v**ector) is a pointer to an array of strings.

- The third argument env (**env**ironment data) is used to access the DOS environmental parameters active at the time the program begins execution.

When an array is used as an argument to function, only the address of the array is passed, not a copy of the entire array. When you call a function with an array name, a pointer to the first element in the array is passed into a function. (In C, an array name without as index is a pointer to the first element in the array).
Each of the command line arguments must be separated by a space or a tab. If you need to pass a command line argument that contains space, you must place it between quotes as:

"this is one argument"

Declaration of argv must be done properly, A common method is:

```
char *argv[];
```

That is, as a array of undetermined length.

The env parameter is declared the same as the argv parameter, it is a pointer to an array of strings that contain environmental setting.

Example 4.3.1:

```
/*      The following program will print "hello tom".        */

# include <stdio.h>
# include <process.h>

main( int argc, char *argv[])
{
        if(argc!=2)
        {
                printf("You forgot to type your name\n");
                exit(0);
        }
        printf("Hello %s ", argv[1]);
}
```

if the name of this program is name.exe, at the command prompt you have to type as:

- name tom

Output:

```
hello tom
```

Example 4.3.2:

```
/*      This program prints current environment settings          */
# include <stdio.h>

main (int argc, char *argv[], char *env[])
{
        int i;
        for(i=0; env[i]; i++)
                printf("%s\n", env[i]);
}
```

We must declare both argc and argv parameters even if they are not used because the parameter declarations are position dependent.

4.4. Example Programs on File I/O and Command Line Arguments:

The following programs demonstrate the use of C's file I/O functions.

Example 4.4.1:

Program on fopen(), fclose(), getc(), putc(). Specify filename on command line. Input chars on keyboard until $ is entered. File is then closed. File is then re-opened and read.

```
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char *argv[])
{
        FILE *fp;                                       /* file pointer  */
        char ch;
        if(argc!=2)
        {
                printf("You forgot to enter the file name\n");
                exit(1);
        }

        if((fp=fopen(argv[1], "w"))==NULL)                /* open file  */
        {
                printf("Cannot open file\n");
                exit(1);
        }
        do                                            /* get keyboard chars until `$'  */
        {
                ch = getchar();
                putc(ch, fp);
        } while (ch! = '$');

        fclose(fp);                                       /* close file  */

        if((fp=fopen(argv[1], "r"))==NULL)                /* open file  */
        {
                printf("Cannot open file\n");
                exit(1);
        }
        ch = getc(fp);                                    /* read one char  */

        while(ch != EOF)
        {
                putchar(ch);                              /* print on screen  */
                ch = getc(fp);                            /* read another char */
        }

        fclose(fp);                                       /* close file  */
}
```

Example 4.4.2:

Program on feof() to check for EOF condition in Binary Files. Specify filenames for input and output at command prompt. The program copies the source file to the destination file. feof() checks for end of file condition. The feof() can also be used for text files.

```
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char *argv[])
{
        FILE *in, *out;                                   /* file pointers  */
        char ch;

        if(argc != 3)
        {
                printf("You forgot to enter the filenames\n");
                exit(1);
        }
        if((in=fopen(argv[1], "rb"))==NULL)                /* open source file  */
        {                                                  /* for read binary  */
                printf("Cannot open source file\n");
                exit(1);
        }

        if((out=fopen(argv[2], "wb"))==NULL)               /* open dest file  */
        {                                                  /* for write binary  */
                printf("Cannot open destination file\n");
                exit(1);
        }
        while(! feof(in))                                  /* here it is  */
        {
                ch = getc(in);

                if(! feof(in))                             /* and again  */
                        putc(ch, out);
        }

        fclose(in);                                        /* close files  */
        fclose(out);
}
```

Example 4.4.3:

Program on fputs(), fgets and rewind(). Strings are entered from keyboard until blank line is entered. Strings are then written to a file called 'testfile'. Since gets() does not store the newline character, '\n' is added before the string is written so that the file can be read more easily. The file is then rewind, input and displayed.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void main(void)
{
        FILE *fp;                                          /* file pointer  */
        char str[80];

        if((fp=fopen("testfile", "w+"))==NULL)      /* open file for text read & write  */
        {
                printf("Cannot open file\n");
                exit(1);
        }

        do                                  /* get strings until CR and write to file  */
        {
                printf("Enter a string (CR to quit):\n");
                gets(str);
                strcat(str, "\n");
                fputs(str, fp);
        } while(*str != '\n');

        rewind(fp);                                        /* rewind  */
        while(! feof(fp))                                       /* read and display file  */
        {
                fgets(str, 79, fp);
                printf(str);
        }

        fclose(fp);                                             /* close file  */
}
```

Example 4.4.4:

Program on fread() and fwrite() (for Data Types Longer than Byte) which writes, then reads back, a double, an int and a long. Notice how sizeof () is used to determine the length of each data type. These functions are useful for reading and writing user-defined data types, especially structures.

```
#include <stdio.h>
#include <stdlib.h>
void main(void)
{

        FILE *fp;
        double d=12.23;
        int i=101;
        long l=123023;
        if((fp=fopen("testfile", "wb+"))==NULL)         /* open for binary read & write */
        {
                printf("Cannot open file\n");
                exit(1);
        }

/* parameters are: *buffer, number of bytes, count of items, file pointer  */

        fwrite(&d, sizeof(double), 1, fp);
        fwrite(&i, sizeof(int), 1, fp);
        fwrite(&l, sizeof(long), 1, fp);

        rewind(fp);

        fread(&d, sizeof(double), 1, fp);
        fread(&i, sizeof(int), 1, fp);
        fread(&l, sizeof(long), 1, fp);

        printf("%2.3f   %d   %ld\n",d,i,l);

        fclose(fp);
}
```

Example 4.4.5:

Program on fprintf() and fscanf(). Reads a string and an integer from the keyboard, writes them to a file, then reads the file and displays the data. These functions are easy to write mixed data to files but are very slow compared with fread() and fwrite().

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <exec/io.h>

void main (void)
{

        FILE *fp;
        char s[ 80];
        int t;
        if ((fp=fopen("testfile", "w"))==NULL)              /* open for text write  */
        {
                        printf ("Cannot open file\n");
                        exit (1);
        }

        printf ("Enter a string and a number: ");

                        /* parameters are: FILE *fp, const char *control_string, ...  */

        fscanf (stdin, "%s%d", s, &t);
        fprintf (fp, "%s %d", s, t);
        fclose (fp);

        if ((fp=fopen("testfile", "r"))==NULL)              /* open for text read */
        {
                printf ("Cannot open file\n");
                exit (1);
        }

        fscanf (fp,"%s%d",s,&t);
        fprintf (stdout, "%s %d\n", s, t);

        fclose (fp)
}
```

4.5. The C Preprocessor:

As defined by the ANSI standard, the C preprocessor contains the following directives:

```
#if         #ifdef      #ifndef     #else       #elif       #include
#define     #undef      #line       #error      #pragma     #define
```

#define:

Defines an identifier (the macro name) and a string (the macro substitution), which will be substituted for the identifier each time the identifier is encountered in the source file.

Once a macro name has been defined, it may be used as part of the definition of other macro names.

If the string is longer than one line, it may be continued by placing a backslash on the end of the first line.

By convention, C programmers use uppercase for defined identifiers. Example macro #defines are:

```
#define TRUE 1
#define FALSE 0
```

The macro name may have arguments, in which case every time the macro name is encountered, the arguments associated with it are replaced by the actual arguments found in the program, as in:

```
#define ABS(a) (a)<0 ? -(a) : (a)
        ...
printf("abs of -1 and 1: %d %d", ABS(-1), ABS(1));
```

Such macro substitutions in place of real functions increase the speed of the code at the price of increased program size.

#error

#error forces the compiler to stop compilation. It is used primarily for debugging. The general form is:

```
#error  error_message
```

When the directive is encountered, the error message is displayed, possibly along with other information (depending on the compiler).

#include

#include instructs the compiler to read another source file, which must be included between double quotes or angle brackets. Examples are:

```
#include "stdio.h"
#include <stdio.h>
```

Both instruct the compiler to read and compile the named header file.

If a file name is enclosed in angle brackets, the file is searched for the included file in a special known \include directory or directories. If the name is enclosed in double quotes, the file is searched in the current directory. If the file is not found, the search is repeated as if the name had been enclosed in angle brackets.

4.5.1. Conditional Compilation:

Several directives control the selective compilation of portions of the program code. They are:

```
#if          #else          #elif          #endif
```

The general form of **#if** is:

```
#if constant_expression
        statement sequence
#endif
```

#else works much like the C keyword else. #elif means "else if" and establishes an if-else-if compilation chain.

Amongst other things, #if provides an alternative method of "commenting out" code. For example, in

```
        #if 0
                printf("#d", total);
        #endif
```

the compiler will ignore printf("#d", total);.

#ifdef and #ifndef

#ifdef means "if defined", and is terminated by an #endif. #indef means "if not defined".

#undef

#undef removes a previously defined definition.

#line

#line changes the contents of __LINE__ (which contains the line number of the currently compiled code) and __FILE__ (which is a string which contains the name of the source file being compiled), both of which are predefined identifiers in the compiler.

#pragma

The #pragma directive is an implementation-defined directive which allows various instructions to be given to the compiler.

Example 4.5.1.1:

```
/*      program  to demonstrate #ifdef,  #ifndef , #else  and #endif     */

# include <stdio.h>
# include <conio.h>
# define  a 10

main()
{
        #ifdef a
                printf("\n Hello a is defined..");
        #endif

        #ifndef a
                printf("\n Not defined ");
        #else
                printf("\n defined ");
        #endif

        getch();
}
```

Output:

```
Hello a is defined..
```

defined

4.5.2. The # and ## Preprocessor Operators:

The # and ## preprocessor operators are used when using a macro #define.

The # operator turns the argument it precedes into a quoted string. For example, given:

```
#define mkstr(s) # s
```

the preprocessor turns the line:

```
printf (mkstr (I like C);
```

into:

```
printf ("I like C");
```

The ## operator concatenates two tokens. For example, given:

```
#define concat(a, b) a ## b
int xy=10;
printf("%d", concat (x, y));
```

the preprocessor turns the last line into:

```
printf("%d", xy);
```

4.5.3. Mathematics: <math.h>

Mathematics is relatively straightforward library to use again. You **must** include #include <math.h>. Below we list some common math functions.

double acos(double x);	Compute arc cosine of x.
double asin(double x);	Compute arc sine of x.
double atan(double x);	Compute arc tangent of x.
double atan2(double y, double x);	Compute arc tangent of y/x.
double ceil(double x);	Get smallest integral value that exceeds x.
double cos(double x);	Compute cosine of angle in radians.
double cosh(double x);	Compute the hyperbolic cosine of x.
div_t div(int number, int denom);	Divide one integer by another.
double exp(double x);	Compute exponential of x
double fabs (double x);	Compute absolute value of x.
double floor(double x);	Get largest integral value less than x.
double fmod(double x, double y);	Divide x by y with integral quotient and return remainder.
double frexp(double x, int *expptr);	Breaks down x into mantissa and exponent of no.
labs(long n);	Find absolute value of long integer n.
double ldexp(double x, int exp);	Reconstructs x out of mantissa and exponent of two.
ldiv_t ldiv(long number, long denom);	Divide one long integer by another.
double log(double x);	Compute log(x).

double log10 (double x);	Compute log to the base 10 of x.
double modf(double x, double *intptr);	Breaks x into fractional and integer parts.
double pow (double x, double y);	Compute x raised to the power y.
double sin(double x);	Compute sine of angle in radians.
double sinh(double x);	Compute the hyperbolic sine of x.
double sqrt(double x);	Compute the square root of x.
void srand(unsigned seed);	Set a new seed for the random number generator (rand).
double tan(double x);	Compute tangent of angle in radians.
double tanh(double x);	Compute the hyperbolic tangent of x.

Math Constants:

The math.h library defines many (often neglected) constants. It is always advisable to use these definitions:

HUGE	The maximum value of a single-precision floating-point number.
M_E	The base of natural logarithms (e).
M_LOG2E	The base-2 logarithm of e.
M_LOG10E	The base-10 logarithm of e.
M_LN2	The natural logarithm of 2.
M_LN10	**The natural logarithm of 10.**
M_PI	п.
M_PI_2	п/2.
M_PI_4	п/4.
M_1_PI	1/ п.
M_2_PI	2/ п.
M_2_SQRTPI	2 / $\sqrt{\pi}$.
M_SQRT2	The positive square root of 2.
M_SQRT1_2	The positive square root of 1/2.
MAXFLOAT	The maximum value of a non-infinite single-precision floating point number.
HUGE_VAL	positive infinity.

There are also a number a machine dependent values defined in #include <value.h>.

4.5.4. Character conversions and testing: ctype.h

The library #include <ctype.h> which contains many useful functions to convert and test ***single*** characters. The common functions are prototypes are as follows:

Character testing:

int isalnum(int c);	True if c is alphanumeric.
int isalpha(int c);	True if c is a letter.
int isascii(int c);	True if c is ASCII.
int iscntrl(int c);	True if c is a control character.
int isdigit(int c);	True if c is a decimal digit
int isgraph(int c);	True if c is a graphical character.
int islower(int c);	True if c is a lowercase letter
int isprint(int c);	True if c is a printable character
int ispunct (int c);	True if c is a punctuation character.
int isspace(int c);	True if c is a space character.
int isupper(int c);	True if c is an uppercase letter.
int isxdigit(int c);	True if c is a hexadecimal digit

Character Conversion:

int toascii(int c);	Convert c to ASCII.
Int tolower(int c);	Convert c to lowercase.
int toupper(int c);	Convert c to uppercase.

The use of these functions is straightforward.

4.5.5. Memory Operations:

The library #include <memory.h> contains the basic memory operations. Although not strictly string functions the functions are prototyped in #include <string.h>:

void *memchr (void *s, int c, size_t n);	Search for a character in a buffer.
int memcmp (void *s1, void *s2, size_t n);	Compare two buffers.
void *memcpy (void *dest, void *src, size_t n);	Copy one buffer into another.
void *memmove (void *dest, void *src, size_t n);	Move a number of bytes from one buffer to another.
void *memset (void *s, int c, size_t n);	Set all bytes of a buffer to a given character.

Note that in all case to **bytes** of memory are copied. The sizeof() function comes in handy again here, for example:

```
        char src[SIZE], dest[SIZE];
        int  isrc[SIZE], idest[SIZE];
```

memcpy(dest, src, SIZE);	Copy chars (bytes)
memcpy(idest, isrc, SIZE*sizeof(int));	Copy arrays of ints
memmove() behaves in exactly the same way as memcpy() except that the source and destination locations may overlap.	

memcmp() is similar to strcmp() except here ***unsigned bytes*** are compared and returns less than zero if s1 is less than s2 ***etc.***

4.6. String Searching:

The library also provides several string searching functions:

char *strchr (const char *string, int c);	Find first occurrence of character c in string.
char *strrchr (const char *string, int c);	Find last occurrence of character c in string.
char *strstr (const char *s1, const char *s2);	locates the first occurrence of the string s2 in string s1.
char *strpbrk (const char *s1, const char *s2);	returns a pointer to the first occurrence in string s1 of any character from string s2, or a null pointer if no character from s2 exists in s1
size_t strspn (const char *s1, const char *s2);	returns the number of characters at the begining of s1 that match s2.
size_t strcspn (const char *s1, const char *s2);	returns the number of characters at the begining of s1 that ***do not*** match s2
char *strtok (char *s1, const char *s2);	break the string pointed to by s1 into a sequence of tokens, each of which is delimited by one or more characters from the string pointed to by s2.
char *strtok_r (char *s1, const char *s2, char **lasts);	has the same functionality as strtok () except that a pointer to a string placeholder lasts must be supplied by the caller.
strchr () and strrchr () are the simplest to use.	

Example 4.6.1:

```
#include <string.h>
#include <stdio.h>

main()
{
        char *str1 = "Hello";
        char *ans;
        ans = strchr (str1,'l');
        printf("%s\n", ans);
}
```

Output: llo

After this execution, ans points to the location str1 + 2

strpbrk () is a more general function that searches for the first occurrence of any of a group of characters, for example:

```
#include <string.h>
#include <stdio.h>

main()
{
        char *str1 = "Hello";
        char *ans;
        ans = strpbrk (str1,"aeiou");
        printf("%s\n",ans);
}
```

Output: ello

Here, ans points to the location str1 + 1, the location of the first e.

strstr () returns a pointer to the specified search string or a null pointer if the string is not found. If s2 points to a string with zero length (that is, the string ""), the function returns s1.

Example 4.6.2:

```
#include <string.h>
#include <stdio.h>

main()
{
        char *str1 = "Hello";
        char *ans;
        ans = strstr (str1, "lo");
        printf("%s\n", ans);
 }
```

```
Output: lo              /*      will yield ans = str + 3.       */
```

strtok () is a little more complicated in operation. If the first argument is not NULL then the function finds the position of any of the second argument characters. However, the position is remembered and any subsequent calls to strtok() will start from this position if on these subsequent calls the first argument is NULL. For example, If we wish to break up the string str1 at each space and print each token on a new line we could do:

```
#include <string.h>
#include <stdio.h>

main()
{
        char *str1 = "Hello Big Boy";
        char *t1;
        for(t1 = strtok (str1," "); t1 != NULL; t1 = strtok (NULL," "))
                printf("%s\n", t1);
 }
```

Output:

```
Hello
Big
Boy
```

The initialisation calls strtok () loads the function with the string str1.We terminate when t1 is NULL. We keep assigning tokens of str1 to t1 until termination by calling strtok () with a NULL first argument.

SAMPLE PROGRAMS ON POINTERS

Example 1:

```
#include <stdio.h>
void main(void)
{
        int i, *p;
        i = 43;
        p = & i;
        printf ("%d  %d\n", i, *p);
}
```

Output:

```
43 43
```

Example 2:

```
#include <stdio.h>
void main(void)
{
```

```
                int i, *p;
                i = 43;
                p = &i;
                *p = 16;
                printf("%d\n", i);
        }
```

Output:

```
        16
```

Example 3:

```
        #include <stdio.h>

        void change(int *i);

        void main(void)
        {
                int x;
                x = 146;
                change(&x);
                printf("%d\n", x);
        }

        void change(int *x)
        {
                *x = 351;
        }
```

Output:

```
        351
```

Example 4:

```
        #include <stdio.h>
        void swap(int *i, int *ii);
        void main(void)
        {
                int x, y;
                x = 10;
                y = 127;

                swap(&x, &y);
                printf("%d  %d\n", x, y);
        }

        void swap(int *a, int *b)
        {
                int i;
                i = *a;
                *a = *b;
                *b = i;
        }
```

Output:

```
127  10
```

Example 5:

```
#include <stdio.h>
void main(void)
{
        void *v;
        char *c;
        c = "Testing void pointer";
        v = c;
        printf("%s\n", v);
}
```

Output:

```
Testing void pointer
```

Example 6:

```
#include <stdio.h>
#include <string.h>

void main(void)
{
        char a[40], *p;
        strcpy(a, "DATA");
        p = a;
        printf("%s\n", p);
}
```

Output:

```
DATA
```

Example 7:

```
#include <stdio.h>
void main(void)
{
        char *a;
        int x;
        a = "DATA";
        for (x = 0; x <= 4; ++x)
                printf("%c\n", *(a + x));
}
```

Output:

```
D
A
T
A
```

Example 8:

```
#include <stdio.h>
void main(void)
{
          char *a;
          a = "DATA";
          while (*a != '\0')
        printf("%c\n", *a++);
}
```

Output:

```
D
A
T
A
```

Example 9:

```
#include <stdio.h>

void main(void)
{
          char *a;
          a = "DATA";
          while (*a)
                   printf("%c\n", *a++);
}
```

Output:

```
D
A
T
A
```

Example 10:

```
#include <stdio.h>
void main(void)
{
          char *a;
          a = "DATA";
          printf("%u\n", a);
          while (*a)
                   printf("%c\n", *a++);
          printf("%u\n", a);
}
```

Output:

```
118
D
A
T
A
122
```

Example 11:

```
#include <stdio.h>
void main(void)
{
        char *p[5];
        int x;
        p[0] = "DATA";
        p[1] = "LOGIC";
        p[2] = "MICROPROCESSOR";
        p[3] = "COMPUTER";
        p[4] = "DISKETTE";
        for (x = 0; x < 5; ++x)
                printf("%s\n", p[x]);
}
```

Output:

```
DATA
LOGIC
MICROPROCESSOR
COMPUTER
DISKETTE
```

Example 12:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <conio.h>

void main(void)
{
 char *a, b[256];
 printf("enter a string:");
 gets(b);
 a = (char*) malloc(strlen(b));
 if (a == NULL)
 {
        printf("Out of memory\n");
        exit(0);
 }
 else
        strcpy(a, b);
 puts(a);
 getch();
}
```

Output:

```
Enter a string: aabbc
aabbc
```

Example 13:

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

void main(void)
{
        char *a, b[256];
        printf("enter a string:");
       gets(b);
        if ((a = (char*) malloc(strlen(b))) == NULL)
        {
                printf("Out of memory\n");
                exit(0);
        }
        else
                strcpy(a, b);
puts(a);
 getch();
}
```

Output:

```
Enter a string: aabbc
aabbc
```

Example 14:

```
#include <stdio.h>
void main(void)
{
        int x[40];
        *(x + 0) = 65;
        printf("%c\n", *(x + 0));
}
```

Output:

```
A
```

Example 15:

```
#include <stdio.h>

int square(int i, *ii);

void main(void)
{
        int x, y;
        x = 8;
        y = 11;
        x = square(x, &y);
        printf("%d  %d\n", x, y);
}

int square(int a, int *b)
{
        *b = *b * *b;                /* rewrite memory */
        return(a * a);               /* return square */
}
```

Output:

```
64  121
```

Example 16:

```
#include <stdio.h>
void square(int *i, int *j);

void main(void)
{
        int x, y;
      x = 8;
        y = 11;
        square(&x, &y);
        printf("%d  %d\n", x, y);
}

void square(int *a, int *b)
{
        *a *= *a;
        *b *= *b;
}
```

Output:

```
64  121
```

Example 17:

```
#include <stdio.h>
#include <string.h>

char *combine(char *, char *);

void main(void)
{
        char a[10], b[10], *p;
        strcpy(a, "house");
        strcpy(b, "fly");
        p = combine(a, b);
        printf("%s\n", p);
}

char *combine(char *s, char *t)
{
        int x, y;
        char r[100];
        strcpy(r, s);
        y = strlen(r);
        for (x = y; *t != '\0'; ++x)
                r[x] = *t++;
        r[x] = '\0';
        return(r);
}
```

Output:

```
housefly
```

Example 18:

```
struct test
{
        int a;
        char b;
        char name[20];
};

#include <stdio.h>
#include <string.h>

void load(struct test *);

void main(void)
{
        struct test r;
        load(&r);
        printf("%d  %c  %s\n", r.a, r.b, r.name);
}

void load(struct test *s)
{
        s->a = 14;
        s->b = 'A';
        strcpy(s->name, "Group");
}
```

Output:

```
14  A  Group
```

Example 19:

```
struct e_record
{
        char name[15];
        int id;
        int years;
        double sal;
};

#include <stdio.h>
#include <string.h>

void main(void)
{
        struct e_record e, *emp;
        emp = &e;
```

```
            strcpy(emp->name, "Herbert Schildt");
            emp->id = 14;
            emp->years = 22;
            emp->sal = 5305.00;
            printf("Employee Name: %s\n", emp->name);
            printf("Employee Number: %d\n", emp->id);
            printf("Years Employed: %d\n", emp->years);
            printf("Employee Salary: Rs.%-.2lf\n", emp->sal);
    }
```

Output:

```
    Employee Name: Herbert Schildt
    Employee Number: 14
    Years Employed: 22
    Employee Salary: Rs. 5305.00
```

Example 20:

```
    struct e_record
    {
            char name[15];
            int id;
            int years;
            double sal;
    };

    #include <stdio.h>
    #include <string.h>
    #include <stdlib.h>

    void main(void)
    {
            struct e_record *emp;
            if ((emp = (e_record*) malloc(sizeof(struct e_record))) == NULL)
                    exit(0);
            strcpy(emp->name, "Herbert Schildt");
            emp->id = 14;
            emp->years = 22;
            emp->sal = 535.00;
            printf("Employee Name: %s\n", emp->name);
            printf("Employee Number: %d\n", emp->id);
            printf("Years Employed: %d\n", emp->years);
            printf("Employee Salary: Rs.%-.2lf\n", emp->sal);
    }
```

Output:

```
    Employee Name: Herbert Schildt
    Employee Number: 14
    Years Employed: 22
    Employee Salary: Rs. 5305.00
```

Example 21:

```
#include <stdlib.h>
void main(void)
{
        int *x;
        float *y;
        double *z;
    if ((z = malloc(sizeof(double))) == NULL)
        exit(0);
        y = (float *) z;
        x = (int *) z;
                        /* x, y, and z all contain the same address */
                        /* memory allocation is for a double type */
        *x = 19;
        *y = 131.334;
        *z = 9.9879;
}
```

Output: This program does not generate any output.

Example 22:

```
#include <stdlib.h>
union test
{
        int x;
        float y;
        double z;
};
#include <stdio.h>
#include <stdlib.h>
void main(void)
{
        union test *ptr;
        if ((ptr = malloc(sizeof(union test))) == NULL)
                exit(0);
        printf("%d\n", sizeof(union test));
        ptr->x = 19;
        ptr->y = 131.334;
        ptr->z = 9.9879;
}
```

Output: 8

Example 23:

```
struct e_record
{
        char name[15];
        int id;
        int years;
        double sal;
};

struct a_record
{
        char name[15];
        char address[40];
};

union com
{
          struct e_record a;
          struct a_record b;
};

#include <stdio.h>
#include <string.h>
void main(void)
{
          union com v;
          strcpy(v.a.name, "Bill Collins");
          v.a.id = 44;
          v.a.years = 12;
          v.a.sal = 351.22;
          printf("Name: %s\n", v.a.name);
          printf("Identification: %d\n", v.a.id);
          printf("Tenure: %d years\n", v.a.years);
          printf("Salary: $%-.2lf\n\n", v.a.sal);
          strcpy(v.b.name, "Bill Collins");
          strcpy(v.b.address, "523 Short St.; Front Royal, VA  22630");
          printf("Name: %s\n", v.b.name);
          printf("Address: %s\n", v.b.address);
}
```

Example 24:

```
#include <stdio.h>
void main(void)
{
          int x, *p, **ptp;
          x = 454;
          p = &x;
          ptp = &p;
          printf("%d  %d\n", *p, **ptp);
}
```

Output:

```
454    454
```

Example 25:

```
#include <stdio.h>
#include <string.h>
void main(void)
{
          char a[20], *b, **c;
          strcpy(a, "Capacitor");
          b = a;
          c = &b;
          printf("%s  %s\n", b, *c);
}
```

Output:

```
Capacitor       Capacitor
```

Example 26:

```
#include <stdio.h>

void main(void)
{
        int x, *p, **ptp, ***ptptp, ****ptptptp;
        x = 274;
        p = &x;
        ptp = &p;
        ptptp = &ptp;
        ptptptp = &ptptp;
        printf("%d\n", ****ptptptp);
}
```

Output:

```
274
```

IMPORTANT REVIEW QUESTIONS

1. Distinguish between getchar() and gets() functions.

getchar()	**gets()**
Used to receive a single character.	Used to receive a single string with white spaces.
Does not require any argument.	It requires a single argument.

2. Distinguish between scanf() and gets() functions.

scanf()	**gets()**
Strings with spaces cannot be accessed.	Strings with any number of spaces can be accessed.
All data types can be accessed.	Only character array data can be accessed.
Spaces and tabs are not acceptable as a part of the input string.	Spaces and tabs are perfectly acceptable of the input string as a part.
Any number of characters, integers, float etc. can be read.	Only one string can be read at a time.

3. Distinguish between printf() and puts() functions.

puts()	**printf()**
It can display only one string at a time.	It can display any number of characters, integers or strings at a time.
No such conversion specifications. Every thing is treated as string.	Each data type is considered separately depending upon the conversion specifications.

4. What is the difference between a pre increment and a post increment operation?

   A pre-increment operation such as ++a, increments the value of a by 1, before a is used for computation, while a post increment operation such as a++, uses the current value of a in the calculation and then increments the value of a by 1.

5. Distinguish between break and continue statement.

Break	**Continue**
Used to terminate the loops or to exit from loop or switch.	Used to transfer the control to the start of loop.
The break statement when executed causes immediate termination of loop.	The continue statement when executed cause immediate termination of the current iteration of the loop.

6. Distinguish between while and do-while loops.

While loop	**Do-while loop**
The while loop tests the condition before each iteration.	The do-while loop tests the condition after the first iteration.
If the condition fails initially the loop is skipped entirely even in the first iteration.	Even if the condition fails initially the loop is executed once.

7. Distinguish between local and global variables

Local variables	Global variables
These are declared within the body of the function.	These are declared outside the function.
These variables can be referred only within the function in which it is declared.	These variables can be referred from any part of the program.
The value of the variables disappear once the function finishes its execution.	The value of the variables disappear only after the entire execution of the program.

8. State the differences between the function prototype the function definition

Function prototype	Function Definition
It declares the function.	It defines the function.
It ends with a semicolon.	It doesn't ends with a semicolon.
The declaration need not include parameters.	It should include names for the parameters.

9. Compare and contrast recursion and iteration

Both involve repetition.
Both involve a termination test.
Both can occur infinitely.

Iteration	Recursion
Iteration explicitly user a repetition structure.	Recursion achieves repetition through repeated function calls.
Iteration terminates when the loop continuation condition fails.	Recursion terminates when a base case is recognized.
Iteration keeps modifying the counter until the loop continuation condition fails.	Recursion keeps producing simple versions of the original problem until the base case is reached.
Iteration normally occurs within a loop so, the extra memory assigned is omitted.	Recursion causes another copy of the function and hence a considerable memory space is occupied.
It reduces the processor's operating time.	It increases the processor's operating time.

10. State the uses of pointers.

- Pointers provided an easy way to represent multidimensional arrays.
- Pointers increase the execution speed.
- Pointers reduce the length and complexity of program.

11. What is the difference between the address stored in a pointer and a value at the address?

The address stored in the pointer is the address of another variable. The value stored at that address is a stored in a different variable. The indirection operator (*) returns the value stored at the address.

12. What is the difference between the indirection operator and the address of operator?

The indirection operator (*) returns the value of the address stored in a pointer. The address of operator (&) returns the memory address of the variable.

13. State the difference between call by value and call by reference.

Call by value	Call by reference
Formal parameter is a local variable.	Formal parameter is a reference variable.
It cannot change the actual parameter.	It can change the actual parameter.
Actual parameter may be a constant, a variable, or an expression.	Actual parameter must be a variable.

14. State the advantages of using bit-fields.

- To store Boolean variables (true/false) in one bit.
- To encrypt certain routines to access the bits within a byte.
- To transmit status information of devices encoded into one or more bits within a byte.

15. State the difference between arrays and structures.

Arrays	Structures
An array is an single entity representing a collection of data items of same data types.	A structure is a single entity representing a collection of data items of different data types.
Individual entries in an array are called elements.	Individual entries in a structure are called members.
An array declaration reserves enough memory space for its elements.	The structure definition reserves enough memory space for its members.
There is no keyword to represent arrays but the square braces [] preceding the variable name tells us that we are dealing with arrays.	The keyword struct tells us that we are dealing with structures.
Initialization of elements can be done during array declaration.	Initialization of members can be done only during structure definition.
The elements of an array are stored in sequence of memory locations.	The members of a structure are stored in sequence of memory locations.
The array elements are accessed by the square braces [] within which the index is placed.	The members of a structure are accessed by the dot operator.
Its general format: Data type array name [size];	Its general format is: struct structure name {

	data_type structure member 1;
	data_type structure member 2;
	.
	.
	.
	data_type structure member N;
	} structure variable;

Example:
int sum [100];
	Example:
	struct student
	{
	char studname [25];
	int rollno;
	} stud1;

16. State the difference between structures and unions.

Structures	**Unions**
Each member in a structure occupies and uses its own memory space.	All the members of a union use the same memory space.
The keyword struct tells us that we are dealing with structures.	The keyword union tells us that we are dealing with unions.
All the members of a structure can be initialized.	Only one member of a union can be initialized.
More memory space is required since each member is stored in a separate memory locations.	Less memory space is required since all members are stored in the same memory locations.

Its general format is:
struct structure name
{
data_type structure Member1;
data_type structure Member2;
.
.
data_type structure Member N;
} structure variable;
	Its general formula is:
	union union name
	{
	data_type structure Member1;
	data_type structure Member2;
	.
	.
	data_type structure Member N;
	} union variable;

Example:
struct student
{
char studname[25];
int rollno;
} stud1;
	Example:
	union student
	{
	char studname[25];
	int rollno;
	} stud1;

17. Advantages of functions:

12. Function makes the lengthy and complex program easy and in short forms. It means large program can be sub-divided into self-contained and convenient small modules having unique name.

13. The length of source program can be reduced by using function by using it at different places in the program according to the user’s requirement.

14. By using function, memory space can be properly utilized. Also less memory is required to run program if function is used.

15. A function can be used by many programs.

16. By using the function, portability of the program is very easy.

17. It removes the redundancy (occurrence of duplication of programs) i.e. avoids the repetition and saves the time and space.

18. Debugging (removing error) becomes very easier and fast using the function sub-programming.

19. Functions are more flexible than library functions.

20. Testing (verification and validation) is very easy by using functions.

21. User can build a customized library of different functions used in daily routine having specific goal and link with the main program similar to the library functions.

18. A complete summarized table to represent the lifetime and visibility (scope) of all the storage class specifier is as below:

Storage class	Life time	Visibility (Scope)
Auto	Local	Local (within function)
Extern	Global	Global (in all functions)
Static	Global	Local
Register	Local	Local

19. Tokens:

Tokens are the smallest individual unit in a program. The different types of tokens used in C are as follows:

- Keywords.
- Identifiers.
- Constants.
- Operators.
- Strings.

Annexure - I

SAMPLE C – PROGRAMS

1. Write a C programme to accept a list of data items & find the II largest & II smallest in it & take average of both & search for that value. Display appropriate message on successful search.

```
main ()
{
        int i,j,a,n,counter,ave,number[30];
        printf ("Enter the value of N\n");
        scanf ("%d", &n);
printf ("Enter the numbers \n");
        for (i=0; i<n; ++i)
        scanf ("%d",&number[i]);
        for (i=0; i<n; ++i)
        {
                for (j=i+1; j<n; ++j)
                {
if (number[i] < number[j])
                        {
a = number[i];
                                number[i] = number[j];
                                number[j] = a;
                        }
                }
        }
        printf ("The numbers arranged in ascending order are given below\n");
        for (i=0; i<n; ++i)
                printf ("%10d\n",number[i]);
        printf ("The 2nd largest number is  = %d\n", number[1]);
        printf ("The 2nd smallest number is = %d\n", number[n-2]);
        ave = (number[1] +number[n-2])/2;
        counter = 0;
        for (i=0; i<n; ++i)
        {
                if (ave==number[i])
                ++counter;
        }
        if (counter==0)
                printf("The average of 2nd largest & 2nd smallest is not in the array\n");
        else
printf("The average of 2nd largest & 2nd smallest in array is %d in numbers\n",
counter);
  }
```

2. Write a C programme to arrange the given numbers in ascending order.

```
main ()
{
        int i,j,a,n,number[30];
        printf ("Enter the value of N\n");
        scanf ("%d", &n);
        printf ("Enter the numbers \n");
        for (i=0; i<n; ++i)
                scanf ("%d",&number[i]);
        for (i=0; i<n; ++i)
        {
                        for (j=i+1; j<n; ++j)
                        {
                                if (number[i] > number[j])
                                {
                                        a= number[i];
                                        number[i] = number[j];
                                        number[j] = a;
                                }
                        }
        }
        printf("Number in Asscending order:\n");
        for(i=0;i<n;i++)
                printf("\t%d\n",number[i]);
}
```

3. Program to convert the given binary number into decimal.

```
# include <stdio.h>
main()
{
        int   num, bnum, dec = 0, base = 1, rem ;
        printf("Enter the binary number(1s and 0s)\n");
        scanf("%d", &num);           /*maximum five digits */
        bnum = num;
        while( num > 0)
        {
                        rem = num % 10;
                        if(rem>1)
                        {
                                printf("\nError in input");
                                break;
                        }
                        dec = dec + rem * base;
                        num = num / 10 ;
                        base = base * 2;
        }
        if(num==0)
        {
                printf("The Binary number is = %d\n", bnum);
                printf("Its decimal equivalent is =%d\n", dec);
        }
}
```

4. Write to accept a 1-dimensional array of N elements & split into 2 halves & sort 1st half in ascending order & 2nd into descending order.

```
#include<stdio.h>

main ()
{
        int i,j,a,n,b,number[30];
        printf ("Enter the value of N\n");
        scanf ("%d", &n);
        b = n/2;
        printf ("Enter the numbers \n");
        for (i=0; i<n; ++i)
                scanf ("%d",&number[i]);
        for (i=0; i<b; ++i)
        {
                for (j=i+1; j<b; ++j)
                {
                        if (number[i] > number[j])
                        {
                                a = number[i];
                                number[i] = number[j];
                                number[j] = a;
                        }
                }
        }
        for (i=b; i<n; ++i)
        {
                        for (j=i+1; j<n; ++j)
                        {
                                if (number[i] < number[j])
                                {
                                        a = number[i];
                                        number[i] = number[j];
                                        number[j] = a;
                                }
                        }
         }
        printf (" The 1st half numbers\n");
        printf (" arranged in asc\n");
        for (i=0; i<b; ++i)
                printf ("%d ",number[i]);
        printf("\nThe 2nd half Numbers\n");
        printf("order arranged in desc.order\n");
        for(i=b;i<n;i++)
                printf("%d ",number[i]);
}
```

5. Program to delete the desired element from the list.

```
# include <stdio.h>

main()
{
        int  vectx[10];
        int  i, n, found = 0, pos, element;

        printf("Enter how many elements\n");
        scanf("%d", &n);

        printf("Enter the elements\n");
        for(i=0; i<n; i++)
        {
                scanf("%d", &vectx[i]);
        }
        printf("Input array elements are\n");
        for(i=0; i<n; i++)
        {
                printf("%d\n", vectx[i]);
        }

         printf("Enter the element to be deleted\n");
         scanf("%d",&element);

        for(i=0; i<n; i++)
        {
                if ( vectx[i] == element)
                {
                        found = 1;
                        pos = i;
                        break;
                }
        }
        if (found == 1)
        {
                for(i=pos; i< n-1; i++)
                {
                        vectx[i] = vectx[i+1];
                }
                printf("The resultant vector is \n");
                for(i=0; i<n-1; i++)
                {
                        printf("%d\n",vectx[i]);
                }
        }
        else
                printf("Element %d is not found in the vector\n", element);

}
```

6. Write a "C" program to Interchange the main diagonal elements with the scondary diagonal elements.

```
#include<stdio.h>
main ()
{
        int i,j,m,n,a;
        static int ma[10][10];
        printf ("Enetr the order of the matix \n");
        scanf ("%dx%d",&m,&n);

if (m==n)
        {
                printf ("Enter the co-efficients of the matrix\n");
                for (i=0;i<m;++i)
                {
                        for (j=0;j<n;++j)
                        {
                                scanf ("%dx%d",&ma[i][j]);
                        }
                }
                printf ("The given matrix is \n");
                for (i=0;i<m;++i)
                {
                        for (j=0;j<n;++j)
                        {
                                printf (" %d",ma[i][j]);
                        }
                        printf ("\n");
                }
                for (i=0;i<m;++i)
                {
                        a = ma[i][i];
                        ma[i][i]   = ma[i][m-i-1];
                        ma[i][m-i-1] = a;
                }
                printf ("THe matrix after changing the \n");
                printf ("main diagonal & secondary diagonal\n");
                for (i=0;i<m;++i)
                {
                        for (j=0;j<n;++j)
                        {
                                printf (" %d",ma[i][j]);
                        }
                        printf ("\n");
                }
}
else
printf ("The given order is not square matrix\n");
}
```

7. Program to insert an element at an appropriate position in an array.

```
#include <stdio.h>
#include <conio.h>

main()
{
        int  x[10];
        int  i, j, n, m, temp, key, pos;

        clrscr();
        printf("Enter how many elements\n");
        scanf("%d", &n);

        printf("Enter the elements\n");
        for(i=0; i<n; i++)
        {
                scanf("%d", &x[i]);
        }
        printf("Input array elements are\n");
        for(i=0; i<n; i++)
        {
                printf("%d\n", x[i]);
        }

        for(i=0; i< n; i++)
        {
                for(j=i+1; j<n; j++)
                {
                        if (x[i] > x[j])
                        {
                                temp = x[i];
                                x[i] = x[j];
                                x[j] = temp;
                        }
                }
        }
        printf("Sorted list is:\n");
        for(i=0; i<n; i++)
        {
                printf("%d\n", x[i]);
        }

        printf("Enter the element to be inserted\n");
        scanf("%d",&key);

        for(i=0; i<n; i++)
        {
                if ( key < x[i] )
                {
                        pos = i;
                        break;
                }
        }
        m = n - pos + 1 ;
        for(i=0; i<= m ; i++)
        {
                x[n-i+2] = x[n-i+1] ;
        }
        x[pos] = key;

        printf("Final list is:\n");
        for(i=0; i<n+1; i++)
        {
                printf("%d\n", x[i]);
        }
}
```

8.   Program to generate the fibonacci sequence.

```
#include <stdio.h>
main()
{
        int   fib1=0, fib2=1, fib3, limit, count=0;

        printf("Enter the limit to generate the fibonacci sequence\n");
        scanf("%d", &limit);

        printf("Fibonacci sequence is ...\n");
        printf("%d\n",fib1);
        printf("%d\n",fib2);
        count = 2;                      /* fib1 and fib2 are already used */

        while( count < limit)
        {
                fib3 = fib1 + fib2;
                count ++;
                printf("%d\n",fib3);
                fib1 = fib2;
                fib2 = fib3;
        }
}
```

9.   Program to compute mean, varience and standard deviation.

```
main()
{
                float x[10];
                int  i, n;
                float avrg, var, SD, sum=0, sum1=0;

                printf("Enter how many elements\n");
                scanf("%d", &n);
                printf("Enter %d numbers:",n);
                for(i=0; i<n; i++)
                {
                        scanf("%f", &x[i]);
                }
/* Compute the sum of all elements */
                for(i=0; i<n; i++)
                {
                        sum = sum + x[i];
                }
                avrg = sum /(float) n;
                                        /* Compute varaience and standard deviation  */
                for(i=0; i<n; i++)
                {
                        sum1 = sum1 + pow((x[i] - avrg),2);
                }
                var = sum1 / (float) n;
                SD = sqrt(var);

                printf("Average of all elements =%.2f\n", avrg);
                printf("Varience of all elements =%.2f\n", avrg);
                printf("Standard Deviation of all elements =%.2f\n", avrg);
}
```

10. Program to sort names in alphabetical order using structures.

```
#include<stdio.h>
#include<string.h>
#include<conio.h>

struct tag
{
        char name[10];
        int rno;
};

typedef struct tag node;

node  s[5];

sort(int no)
{
        int i,j;
        node temp;
        for(i=0;i<no-1;i++)
                for(j=i+1;j<no;j++)
                        if(strcmp(s[i].name,s[j].name)>0)
                        {
                                temp=s[i];
                                s[i]=s[j];
                                s[j]=temp;
                        }
}


main()
{
        int no,i;
        clrscr();
        fflush(stdin);

        printf("Enter The Number Of Students:");
        scanf("%d",&no);

        for(i=0;i<no;i++)
        {
                printf("Enter The Name:");
                fflush(stdin);
                gets(s[i].name);
                printf("Enter the Roll:");
                scanf("%d",&s[i].rno);
        }

        sort(no);
        for(i=0;i<no;i++)
        {
                printf("%s\t",s[i].name);
                printf("%d\n",s[i].rno);
        }
        getche();
}
```

11. Program to reverse the given integer (palindrome).

```
#include <stdio.h>
main()
{
        int   num, rev = 0, found = 0, temp, digit;

        printf("Enter the number\n");
        scanf("%d", &num);

        temp = num;
        while(num > 0)
        {
                digit = num % 10;
                rev = rev * 10 + digit;
                num /= 10;
        }
        printf("Given number =%d\n", temp);
        printf("Its reverse is =%d\n", rev);

        if(temp == rev )
                printf("Number is a palindrome\n");
        else
                printf("Number is not a palindrome\n");
}
```

12. Program to find the frequency of odd numbers & even numbers in the input of a matrix.

```
#include<stdio.h>
main ()
{
        int i,j,m,n,even=0,odd=0;
        static int ma[10][10];
        printf ("Enter the order ofthe matrix \n");
        scanf ("%d %d",&m,&n);
        printf ("Enter the coefficients if matrix \n");
        for (i=0;i<m;++i)
        {
                for (j=0;j<n;++j)
                {
                        scanf ("%d", &ma[i][j]);
                        if ((ma[i][j]%2) == 0)
                        {
                                ++even;
                        }
                        else
                                ++odd;
                }
        }
printf ("The given matrix is\n");
for (i=0;i<m;++i)
{
                for (j=0;j<n;++j)
                        printf (" %d",ma[i][j]);
                printf ("\n");
}
printf ("The frequency of odd number occurance = %d\n",odd);
printf ("The frequency of even number occurance = %d\n",even);
}
```

13. Program to determine the given number is odd.

```
#include <stdio.h>

main()
{
int numb;
printf(" Enter the number\n");
scanf("%d", &numb);

if((numb%2)!=0)
                printf(" %d , is an odd number\n", numb);

}
```

14. Program to check if the two numbers are equal.

```
#include <stdio.h>

main()
{
int m,n;
printf(" Enter the values for  M and N\n");
scanf("%d%d", &m,&n);
if(m == n )
                printf("M and  N are equal\n");
else
                printf("M and N are not equal\n");
}
```

15. Program to find the largest among three numbers.

```
#include <stdio.h>

main()
{
int a,b,c;
printf(" Enter  the values for A,B,C\n");
scanf("%d %d %d", &a,&b,&c);
if( a > b )
{
        if ( a > c)
            printf(" A is the Largest\n");
        else
            printf("C is the largest\n");
}
else if ( b > c)
        printf(" B is the Largest\n");
    else
        printf("C is the Largest\n");
}
```

16. Program to determine the quadrant of a point in a cartesian coordinate system.

```
#include <stdio.h>
main()
{
int x,y;
printf("Enter the values for X and Y\n");
scanf("%d %d",&x,&y);

if( x > 0 && y > 0)
        printf(" point (%d,%d) lies in the First quandrant\n",x,y);
else if( x < 0 && y > 0)
        printf(" point (%d,%d) lies in the Second quandrant\n",x,y);
else if( x < 0 && y < 0)
        printf(" point (%d, %d) lies in the Third quandrant\n",x,y);
else if( x > 0 && y < 0)
        printf(" point (%d,%d) lies in the Fourth quandrant\n",x,y);
else if( x == 0 && y == 0)
        printf(" point (%d,%d) lies at the origin\n",x,y);
}
```

17. Program to find the areas of different geometrical figures using switch statement.

```
#include <stdio.h>
main()
{
int  fig_code;
float  side,base,length,bredth,height,area,radius;
printf("-------------------------\n");
printf(" 1 --> Circle\n");
printf(" 2 --> Rectangle\n");
printf(" 3 --> Triangle\n");
printf(" 4 --> Square\n");
printf("-------------------------\n");

printf("Enter the Figure code\n");
scanf("%d",&fig_code);

switch(fig_code)
{
                case 1:
printf(" Enter the radius\n");
                        scanf("%f",&radius);
                        area=3.142*radius*radius;
                        printf("Area of a circle=%f\n", area);
                        break;
                case 2:
printf(" Enter the bredth and length\n");
                        scanf("%f %f",&bredth, &length);
                        area=bredth *length;
                        printf("Area of a Reactangle=%f\n", area);
                        break;
                case 3:
printf(" Enter the base and height\n");
                        scanf("%f %f",&base,&height);
                        area=0.5 *base*height;
                        printf("Area of a Triangle=%f\n", area);
                        break;
                case 4:
printf(" Enter the side\n");
                        scanf("%f",&side);
                        area=side * side;
                        printf("Area of a Square=%f\n", area);
                        break;
                default:
printf(" Error in figure code\n");
                        break;
}                                       /* End of switch */

}                /* End of main() */
```

18.   Program to grade the student.

```
#include <stdio.h>
#include <string.h>

main()
{
char remark[15], grade;

printf("Enter the grade\n");
scanf("%c",&grade);

grade=toupper(grade);              /* lower case letter to upper case */

switch(grade)
{
                case 'S':
strcpy(remark," SUPER");
```

```
                        break;
                case 'A':
strcpy(remark," VERY GOOD");
                        break;
                case 'B':
strcpy(remark," FAIR");
                        break;
                case 'Y':
strcpy(remark," ABSENT");
                        break;
                case 'F':
strcpy(remark," FAILS");
                        break;
                default :
strcpy(remark, "ERROR IN GRADE\n");
                        break;
}                                       /* End of switch */
printf("RESULT: %s\n",remark);
}
```

19.   Program to find the sum of first 50 natural numbers.

```
#include <stdio.h>
main()
{
int  num,sum=0;
for(num=1;num<=50; num++)
                sum += num;
printf("Sum =%4d\n", sum);
}
```

20.   Program to illustrate compound statement with for loop.

```
#include <stdio.h>
main()
{
int  index=0, value=0,number;
printf("Enter The Number\n");
scanf("%d", &number);
for(index=1;index<=number;index++)
{
                value++;
                printf("%d\n",value);
}
printf("Goodbye\n");
}
```

21.   Program to find the factorial of a number.

```
#include <stdio.h>
main()
{
int  i,fact=1,num;
printf("Enter the number\n");
scanf("%d",&num);
if( num <0)
        printf("Factorial is not there for –ve numbers");
        else if(num==0 || num==1)
                fact=1;
else
{
                for(i=1;i<=num; i++)
                        fact *= i;
}
printf(" Factorial of %d =%5d\n", num,fact);
}
```

22. Program to illustrate for loop without initial and increment/decrement expressions.

```
#include <stdio.h>
main()
{
int  i=0,limit=5;
printf(" Values of I\n");
for(  ; i<limit;   )
{
                i++;
                printf("%d\n",i);
}
}
```

23. Program to accept a string and find the sum of all digits in the string.

```
#include <stdio.h>
main()
{
char string[80];
int count,nc=0,sum=0;
printf("Enter the string containing both digits and alphabet\n");
scanf("%s",string);
for(count=0;string[count]!='\0'; count++)
{
                if((string[count]>='0') && (string[count]<='9'))
                {
                        nc += 1;
                        sum += (string[count] - '0');
                }
}
printf("NO. of Digits in the string=%d\n",nc);
printf("Sum of all digits=%d\n",sum);

}
```

24. Program to find the sum of the sine series.

```
#include <stdio.h>
#include <math.h>
#define pi 3.142
```

```
main()
{
int  i,n,k,sign;
float sum=0,num,den,xdeg,xrad,xsqr,term;
printf("Enter the angle( in degree)\n");
scanf("%f",&xdeg);
printf("Enter the no. of terms\n");
scanf("%d",&n);
xrad=xdeg * (pi/180.0);    /* Degrees to radians*/
xsqr= xrad*xrad;
sign=1; k=2; num=xrad; den=1;

for(i=1;i<=n; i++)
{
                term=(num/den)* sign;
                sum += term;
                sign *= -1;
                num *= xsqr;
                den *= k*(k+1);
                k += 2;
}
printf("Sum of sine series of %d terms =%8.3f\n",n,sum);
}
```

25.   Program to find the sum of cos(x) series.

```
#include<stdio.h>
#include<math.h>
main()
{
        float x,sign,cosx,fact;
        int n,x1,i,j;
        printf("Enter the number of the terms in a series\n");
        scanf("%d",&n);
        printf("Enter the value of x(in degrees)\n");
        scanf("%f",&x);
        x1=x;
        x=x*(3.142/180.0); /* Degrees to radians*/
        cosx=1;
        sign=-1;
        for(i=2;i<=n;i=i+2)
        {
                fact=1;
                for(j=1;j<=i;j++)
                {
                        fact=fact*j;
                }
                cosx=cosx+(pow(x,i)/fact)*sign;
                sign=sign*(-1);
        }
        printf("Sum of the cosine series=%f\n",cosx);
        printf("The value of cos(%d) using library function=%f\n",x1,cos(x));
}
```

26. Program to find the sum of all elements of an array using pointers.

```
#include <stdio.h>

main()
{
static int array[5]={ 200,400,600,800,1000 };
int addnum(int *ptr);     /* function prototype */

int sum;
sum = addnum(array);
printf(" Sum of all array elements=%d\n", sum);

}

int  addnum(int *ptr)
{
int total=0, index;
for(index=0;index<5; index++)
                total+=*(ptr+index);
return(total);
}
```

27. Program to swap particular elements in array using pointers.

```
#include <stdio.h>

main()
{
float x[10];
int i,n;
float swap34(float *ptr1, float  *ptr2 );      /* Function Declaration */
printf(" How many Elements...\n");
scanf("%d", &n);
printf(" Enter Elements one by one\n");
for(i=0;i<n;i++)
                scanf("%f",x+i);

swap34(x+2, x+3);                           /* Interchanging 3rd element by 4th */
printf(" Resultant Array...\n");
for(i=0;i<n;i++)
                printf("X[%d]=%f\n",i,x[i]);
}

float swap34(float *ptr1, float *ptr2 )                /* Function Definition */
{
        float temp;
        temp=*ptr1;
        *ptr1=*ptr2;
        *ptr2=temp;
}
```

28. Program to find the sum of two 1-D arrays using Dynamic Memory Allocation.

```
#include <stdio.h>
#include <malloc.h>

main()
{
int i,n,sum;
int *a,*b,*c;
printf(" How many Elements in each array...\n");
scanf("%d", &n);
a=(int *) malloc(sizeof(n*(int));
b=(int *) malloc(sizeof(n*(int));
c=(int *) malloc(sizeof(n*(int));
printf(" Enter Elements of First List\n");
for(i=0;i<n;i++)
                scanf("%f",a+i);

printf(" Enter Elements of Second List\n");
for(i=0;i<n;i++)
                scanf("%f",b+i);

for(i=0;i<n;i++)
                c+i=(a+i) +( b+i);
printf(" Resultant List is\n");
for(i=0;i<n;i++)
                printf("C[%d]=%f\n",i,c[i]);
}
```

29. Program to reverse the given integer.

```
#include <stdio.h>

main()
{
        int   num, rev = 0, found = 0, temp, digit;

        printf("Enter the number\n");
        scanf("%d", &num);

        temp = num;
        while(num > 0)
        {
                digit = num % 10;
                rev = rev * 10 + digit;
                num /= 10;
        }
        printf("Given number =%d\n", temp);
        printf("Its reverse is =%d\n", rev);
}
```

30. Program to sort all rows of matrix in ascending order & all columns in descending order.

```
main ()
{
        int i,j,k,a,m,n;
        static int ma[10][10],mb[10][10];
        printf ("Enter the order of the matrix \n");
        scanf ("%d %d", &m,&n);
        printf ("Enter co-efficients of the matrix \n");
        for (i=0;i<m;++i)
        {
                for (j=0;j<n;++j)
                {
                        scanf ("%d",&ma[i][j]);
                        mb[i][j] = ma[i][j];
                }
        }
        printf ("The given matrix is \n");
        for (i=0;i<m;++i)
        {
                for (j=0;j<n;++j)
                {
                        printf (" %d",ma[i][j]);
                }
                printf ("\n");
        }
        printf ("After arranging rows in ascending order\n");
        for (i=0;i<m;++i)
        {
                for (j=0;j<n;++j)
                {
                        for (k=(j+1);k<n;++k)
                        {
                                 if (ma[i][j] > ma[i][k])
                                {
                                        a = ma[i][j];
                                        ma[i][j] = ma[i][k];
                                        ma[i][k] = a;
                                }
                        }
                }
        }
        for (i=0;i<m;++i)
        {
                for (j=0;j<n;++j)
                {
                        printf (" %d",ma[i][j]);
                }
                printf ("\n");
        }
printf ("After arranging the columns in descending order \n");
for (j=0;j<n;++j)
{
                for (i=0;i<m;++i)
                {
                        for (k=i+1;k<m;++k)
```

```
            {
                if (mb[i][j] < mb[k][j])
                {
                    a = mb[i][j];
                    mb[i][j] = mb[k][j];
                    mb[k][j] = a;
                }
            }
}
}
for (i=0;i<m;++i)
{
        for (j=0;j<n;++j)
        {
             printf (" %d",mb[i][j]);
        }
        printf ("\n");
}
}
```

31. Program to convert lower case letters to upper case vice-versa.

```
#include <stdio.h>
#include <ctype.h>
#include <conio.h>

main()
{
    char sentence[100];
    int count, ch, i;
    clrscr();

    printf("Enter a sentence\n");
    for(i=0; (sentence[i] = getchar())!='\n'; i++);
    count = i;
    sentence[count]='\0';
    printf("Input sentence is : %s",sentence);
    printf("\nResultant sentence is\n");
    for(i=0; i < count; i++)
    {
            ch = islower(sentence[i]) ? toupper(sentence[i]) :
tolower(sentence[i]);
            putchar(ch);
        }
}
```

32. Program to find the sum of the rows & columns of a matrix.

```
main ()
{
    int i,j,m,n,sum=0;
    static int m1[10][10];
    printf ("Enter the order of the matrix\n");
    scanf ("%d%d", &m,&n);
    printf ("Enter the co-efficients of the matrix\n");
    for (i=0;i<m;++i)
```

```
        {
                for (j=0;j<n;++j)
                {
                        scanf ("%d",&m1[i][j]);
                }
        }
        for (i=0;i<m;++i)
        {
                for (j=0;j<n;++j)
                {
                        sum = sum + m1[i][j] ;
                }
                printf ("   Sum of the %d row is = %d\n",i,sum);
                sum = 0;
        }
        sum=0;
        for (j=0;j<n;++j)
        {
                for (i=0;i<m;++i)
                {
                        sum = sum+m1[i][j];
                }
                 printf ("Sum of the %d column is = %d\n", j,sum);
                sum = 0;
        }
 }
```

33.   Program to find the transpose of a matrix.

```
#include<stdio.h>
main ()
{
        int i,j,m,n;
        static int ma[10][10];
        printf ("Enter the order of the matrix \n");
        scanf ("%d %d",&m,&n);
printf ("Enter the coefiicients of the matrix\n");
        for (i=0;i<m;++i)
        {
                for (j=0;j<n;++j)
                {
                        scanf ("%d",&ma[i][j]);
                }
        }
        printf ("The given matrix is \n");
        for (i=0;i<m;++i)
        {
                for (j=0;j<n;++j)
                {
                        printf (" %d",ma[i][j]);
                }
                printf ("\n");
        }
        printf ("Transpose of matrix is \n");
        for (j=0;j<n;++j)
        {
```

```
		for (i=0;i<m;++i)
		{
			printf (" %d",ma[i][j]);
		}
		printf ("\n");
	}
 }
```

34. Program to illustrate the unions.

```
#include <stdio.h>
#include <conio.h>

main()
{
	union number
	{
		int  n1;
		float n2;
	};
	union number x;
	clrscr() ;
	printf("Enter the value of n1: ");
	scanf("%d", &x.n1);
	printf("Value of n1 =%d", x.n1);
	printf("\nEnter the value of n2: ");
	scanf("%d", &x.n2);
	printf("Value of n2 = %d\n",x.n2);
}
```

35. Program to accepts two strings and compare them. Finally print whether, both are equal, or first string is greater than the second or the first string is less than the second string without using string library.

```
#include<stdio.h>
#include<conio.h>

void main()
{
	int count1=0,count2=0,flag=0,i;
	char str1[10],str2[10];

	clrscr();
	puts("Enter a string:");
	gets(str1);

	puts("Enter another string:");
	gets(str2);
					/*Count the number of characters in str1*/
	while (str1[count1]!='\0')
		count1++;
					/*Count the number of characters in str2*/
	while (str2[count2]!='\0')
		count2++;
	i=0;
/*The string comparison starts with the first character in each string and continues with
subsequent characters until the corresponding characters differ or until the end of the
strings is reached.*/

        while ( (i < count1) && (i < count2))
        {
                if (str1[i] == str2[i])
                {
                        i++;
                        continue;
                }
                if (str1[i]<str2[i])
                {
                        flag = -1;
                        break;
                }
                if (str1[i] > str2[i])
                {
                        flag = 1;
                        break;
                }
        }

        if (flag==0)
           printf("Both strings are equal\n");
        if (flag==1)
           printf("String1 is greater than string2\n", str1, str2);
        if (flag == -1)
           printf("String1 is less than string2\n", str1, str2);
        getch();
}
```

Output:

```
Enter a string: happy
Enter another string: HAPPY
String1 is greater than string2

RUN2:
Enter a string: Hello
Enter another string: Hello
Both strings are equal

RUN3:
Enter a string: gold
Enter another string: silver
String1 is less than string2
```

36. Program to accept a decimal number and convert it binary and count the number of 1's in the binary number.

```
#include <stdio.h>

main()
{
        long   num, dnum, bin = 0, base = 1;
        int rem, no_of_1s = 0 ;

        printf("Enter a decimal integer:\n");
        scanf("%ld", &num);                              /*maximum five digits */
        dnum = num;

        while( num > 0)
        {
                rem = num % 2;
                if (rem==1)                          /*To count no.of 1s*/
                {
                        no_of_1s++;
                }
                bin = bin + rem * base;
                num = num / 2 ;
                base = base * 10;
        }
        printf("Input number is = %ld\n", dnum);
        printf("Its Binary equivalent is =%ld\n", bin);
        printf("No. of 1's in the binary number is = %d\n", no_of_1s);
}
```

Output:

```
Enter a decimal integer: 75
Input number is = 75
Its Binary equivalent is =1001011
No. of 1's in the binary number is = 4

RUN2
Enter a decimal integer: 128
Input number is = 128
Its Binary equivalent is=10000000
No. of 1's in the binary number is = 1
```

37. Program to calculate the salary of an employee.

```
#include<stdio.h>
#include<conio.h>

void main()
{
        struct employee
        {
                long int code;
                char name[20];
                char vertical[15];
                float salary;
        };
        struct employee emp;
        clrscr();
        printf("employee code:");
        scanf("%ld",&emp.code);
        fflush(stdin);
        printf("employee name:");
        scanf("%[^\n]",emp.name);
        fflush(stdin);
        printf("vertical:");
        scanf("%[^\n]",emp.vertical);
        fflush(stdin);
        printf("salary:");
        scanf("%f",&emp.salary);
        clrscr();
        printf("        EMPLOYEE INFORMATION SYSTEM\n");
        printf("------------------------------------------------\n");
        printf("CODE |      NAME      |    VERTICAL    |SALARY\n");
        printf("%ld  | %-15s| %-15.10s| 7.2f\n", emp.code, emp.name, emp.vertical,
emp.salary);
        printf("------------------------------------------------\n");
        getch();
}
```

38. Program to accept N integer number and store them in an array AR. The odd elements in the AR are copied into OAR and other elements are copied into EAR. Display the contents of OAR and EAR.

```
 #include <stdio.h>

main()
 {
        long int ARR[10], OAR[10], EAR[10];
        int i,j=0,k=0,n;

        printf("Enter the size of array AR:\n");
        scanf("%d",&n);

        printf("Enter the elements of the array:\n");
        for(i=0;i<n;i++)
        {
                scanf("%ld",&ARR[i]);
                fflush(stdin);
        }
                        /*Copy odd and even elemets into their respective arrays*/
        for(i=0;i<n;i++)
        {
                if (ARR[i]%2 == 0)
                {
                        EAR[j] = ARR[i];
                        j++;
                }
                else
                {
                        OAR[k] = ARR[i];
                        k++;
                }
        }

        printf("The elements of OAR are:\n");
        for(i=0;i<j;i++)
        {
                printf("%ld\n",OAR[i]);
        }

        printf("The elements of EAR are:\n");
        for(i=0;i<k;i++)
        {
                printf("%ld\n", EAR[i]);
        }
}
```

Output:

```
 Enter the size of array AR: 6
 Enter the elements of the array:
 12
 345
 678
 899
 900
 111

The elements of OAR are:
 345
 899
 111

The elements of EAR are:
 12
 678
 900
```

39.

```
#include<stdio.h>
#include<conio.h>

void main()
{       struct date
        {
                int day;
                int month;
                int year;
        };
        struct details
        {
                char name[20];
                int price;
                int code;
                int qty;
                struct date mfg;
        };

        struct details item[50];
        int n,i;

        clrscr();
        printf("Enter number of items:");
        scanf("%d",&n);
        fflush(stdin);
        for(i=0;i<n;i++)
        {
                fflush(stdin);
                printf("Item name:");
                scanf("%[^\n]",item[i].name);

                fflush(stdin);
                printf("Item code:");
                scanf("%d",&item[i].code);
                fflush(stdin);

                printf("Quantity:");
                scanf("%d",&item[i].qty);
                fflush(stdin);

                printf("price:");
                scanf("%d",&item[i].price);
                fflush(stdin);

                printf("Manufacturing date(dd-mm-yyyy):");
                scanf("%d-%d-%d",&item[i].mfg.day,&item[i].mfg.month,&item[i].mfg.year);
        }
        printf("            *****  INVENTORY *****\n");
        printf("---------------------------------------------------------------\n");
        printf("S.N.|    NAME          |   CODE   |  QUANTITY |  PRICE  |MFG.DATE\n");
        printf("---------------------------------------------------------------\n");
        for(i=0;i<n;i++)
printf("%d     %-15s       %-d          %-5d     %-5d
%d/%d/%d\n",i+1,item[i].name,item[i].code,item[i].qty,item[i].price,item[i].mfg.day,it
em[i].mfg.month,item[i].mfg.year);
        printf("---------------------------------------------------------------\n");
        getch();
}
```

40. Program to find whether a given year is leap year or not.

```
 # include <stdio.h>

 main()
 {
        int year;

        printf("Enter a year:\n");
        scanf("%d",&year);

        if ( (year % 4) == 0)
                printf("%d is a leap year",year);
        else
                printf("%d is not a leap year\n",year);
 }
```

Output:

```
Enter a year: 2000
```

```
2000 is a leap year

RUN2:

Enter a year: 1999
1999 is not a leap year
```

41. Program to multiply given number by 4 using bitwise operators.

```
 # include <stdio.h>

 main()
 {
        long number, tempnum;
        printf("Enter an integer:\n");
        scanf("%ld",&number);
        tempnum = number;
        number = number << 2;   /*left shift by two bits*/

        printf("%ld x 4 = %ld\n", tempnum,number);
}
```

Output:

```
Enter an integer: 15
15 x 4 = 60

RUN2:
Enter an integer: 262
262 x 4 = 1048
```

42.

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int count1=0,count2=0,i,j,flag;
        char str[80],search[10];
        clrscr();
        puts("Enter a string:");
        gets(str);
        puts("Enter search substring:");
        gets(search);
        while (str[count1]!='\0')
                count1++;
        while (search[count2]!='\0')
                count2++;
        for(i=0;i<=count1-count2;i++)
        {
                for(j=i;j<i+count2;j++)
                {
                        flag=1;
                        if (str[j]!=search[j-i])
                        {
                                flag=0;
                                break;
                        }
                }
                if (flag==1)
                        break;
        }
        if (flag==1)
                puts("SEARCH SUCCESSFUL!");
        else
                puts("SEARCH UNSUCCESSFUL!");
        getch();
}
```

43. Program to compute the value of X ^ N given X and N as inputs.

```
 #include <stdio.h>
 #include <math.h>

 void main()
 {
        long int x,n,xpown;
        long int power(int x, int n);

        printf("Enter the values of X and N: \n");
        scanf("%ld %ld",&x,&n);

        xpown = power (x,n);

        printf("X to the power N = %ld\n");
 }

 /*Recursive function to computer the X to power N*/

 long int power(int x, int n)
 {
        if (n==1)
                return(x);
         else if ( n%2 == 0)
                return (pow(power(x,n/2),2));  /*if n is even*/
        else
                return (x*power(x, n-1));     /* if n is odd*/
}
```

Output:

```
Enter the values of X and N: 2 5
X to the power N = 32

RUN2:
Enter the values of X and N: 4 4
X to the power N ==256

RUN3:
Enter the values of X and N: 5 2
X to the power N = 25
```

```
RUN4:
Enter the values of X and N: 10 5
X to the power N = 100000
```

44.   Program to accept a string and find the number of times the word 'the' appears in it.

```
#include<stdio.h>
#include<conio.h>

void main()
{
        int count=0,i,times=0,t,h,e,space;
        char str[100];

        clrscr();
        puts("Enter a string:");
        gets(str);
/*Traverse the string to count the number of characters*/
        while (str[count]!='\0')
        {
                count++;
        }
                                /*Finding the frequency of the word 'the'*/
        for(i=0;i<=count-3;i++)
        {
                t=(str[i]=='t'||str[i]=='T');
                h=(str[i+1]=='h'||str[i+1]=='H');
                e=(str[i+2]=='e'||str[i+2]=='E');
                space=(str[i+3]==' '||str[i+3]=='\0');
                if ((t&&h&&e&&space)==1)
                  times++;
        }
        printf("Frequency of the word \'the\' is %d\n",times);
        getch();
}
```

Output:

```
Enter a string: The Teacher's day is the birth day of Dr.S.Radhakrishnan
Frequency of the word 'the' is 2
```

45.   Program to find the number of characters, words and lines.

```
#include<conio.h>
#include<string.h>
#include<stdio.h>

void main()
{
        int count=0,chars,words=0,lines,i;
        char text[1000];

        clrscr();
        puts("Enter text:");
        gets(text);

        while (text[count]!='\0')
                count++;
        chars=count;

        for (i=0;i<=count;i++)
        {
                if ((text[i]==' '&&text[i+1]!=' ')||text[i]=='\0')
                   words++;
        }

        lines=chars/80+1;
        printf("no. of characters:%d\n",chars);
        printf("no. of words:%d\n",words);
        printf("no. of lines:%d\n",lines);
        getch();
}
```

46. Program to swap the contents of two numbers using bitwise XOR operation. Don't use either the temporary variable or arithmetic operators.

```
 # include <stdio.h>

 main()
 {
        long i,k;
        printf("Enter two integers: \n");
        scanf("%ld %ld",&i,&k);
        printf("\nBefore swapping i= %ld and k = %ld",i,k);
        i = i^k;
        k = i^k;
        i = i^k;
        printf("\nAfter swapping i= %ld and k = %ld",i,k);

 }
```

Output:

```
Enter two integers: 23 34
Before swapping i= 23 and k = 34
After swapping i= 34 and k = 23
```

47. Program to find the length of a string without using the built-in function, also check whether it is a palindrome or not.

```
 #include <stdio.h>
 #include <string.h>
 #include <conio.h>

 main()
 {
        char string[25], revString[25]={'\0'};
        int  i, length = 0, flag = 1;
        clrscr();
        fflush(stdin);

        printf("Enter a string\n");
        gets(string);

        for (i=0; string[i] != '\0'; i++)       /*keep going through each */
        {                                        /*character of the string */
                length++;                         /*till its end */
        }
        printf("The length of the string: \'%s\' = %d\n", string, length);

        for (i=length-1; i >= 0 ; i--)
        {
                revString[length-i-1] = string[i];
        }
                /*Compare the input string and its reverse. If both are equal
         then the input string is palindrome. Otherwise it is not a palindrome */

for (i=0; i < length ; i++)
        {
                if (revString[i] != string[i])
                        flag = 0;
         }

        if (flag == 1)
                printf ("%s is a palindrome\n", string);
        else
                printf("%s is not a palindrome\n", string);
 }
```

Output:

```
Enter a string: madam
The length of the string 'madam' = 5
madam is a palindrome

RUN2:
Enter a string: good
The length of the string 'good' = 4
good is not a palindrome
```

48. Program to accept two strings and concatenate them i.e. the second string is appended to the end of the first string.

```
 #include <stdio.h>
 #include <string.h>

main()
 {
    char string1[20], string2[20];
    int i,j,pos;

    strset(string1, '\0');              /*set all occurrences in two strings to NULL*/
    strset(string2,'\0');
    printf("Enter the first string:");
    gets(string1);
    fflush(stdin);
    printf("Enter the second string:");
    gets(string2);

    printf("First string  = %s\n", string1);
    printf("Second string = %s\n", string2);

                                 /*To concate the second stribg to the end of the string
     travserse the first to its end and attach the second string*/
     for (i=0; string1[i] != '\0'; i++)
     {
        ;                        /*null statement: simply trvsering the string1*/
     }

     pos = i;
     for (i=pos,j=0; string2[j]!='\0'; i++)
     {
          string1[i] = string2[j++];
     }
     string1[i]='\0';             /*set the last character of string1 to NULL*/

     printf("Concatenated string = %s\n", string1);
}
```

Output:

```
Enter the first string: CD-
Enter the second string: ROM
First string  = CD-
Second string = ROM
Concatenated string = CD-ROM
```

49.

```
#include <stdio.h>
 int A[10][10], B[10][10], sumat[10][10], diffmat[10][10];
 int i, j, R1, C1, R2, C2;
main()
{
        void readmatA();
        void printmatA();
        void readmatB();
        void printmatB();
        void sum();
        void diff();

        printf("Enter the order of the matrix A\n");
        scanf("%d %d", &R1, &C1);

        printf("Enter the order of the matrix B\n");
        scanf("%d %d", &R2,&C2);
if( R1 != R2 && C1 != C2)
        {
                printf("Addition and subtraction are possible\n");
                exit(1);
        }
        else
        {
                printf("Enter the elements of matrix A\n");
                readmatA();
                printf("MATRIX A is\n");
                printmatA();

                printf("Enter the elements of matrix B\n");
                readmatB();
                printf("MATRIX B is\n");
                printmatB();

                sum();
                diff();
        }
return 0;
}

void readmatA()
{
        for(i=0; i<R1; i++)
        {
                for(j=0; j<C1; j++)
                {
                        scanf("%d",&A[i][j]);
                }
        }
        return;
}

void readmatB()
{
        for(i=0; i<R2; i++)
        {
                for(j=0; j<C2; j++)
                {
                        scanf("%d",&B[i][j]);
                }
        }
}

void printmatA()
{
        for(i=0; i<R1; i++)
        {
                for(j=0; j<C1; j++)
                {
                        printf("%3d",A[i][j]);
                }
                printf("\n");
        }
}
```

```
void printmatB()
{
	for(i=0; i<R2; i++)
	{
		for(j=0; j<C2; j++)
		{
			printf("%3d",B[i][j]);
		}
		printf("\n");
	}
}

void sum()
{
	for(i=0; i<R1; i++)
	{
		for(j=0; j<C2; j++)
		{
			sumat[i][j] = A[i][j] + B[i][j];
		}
	}
	printf("Sum matrix is\n");
	for(i=0; i<R1; i++)
	{
		for(j=0; j<C2; j++)
		{
			printf("%3d",sumat[i][j]) ;
		}
		printf("\n");
	}
	return;
}

void diff()
{
	for(i=0; i<R1; i++)
	{
		for(j=0; j<C2; j++)
		{
			diffmat[i][j] = A[i][j] - B[i][j];
		}
	}
	printf("Difference matrix is\n");
	for(i=0; i<R1; i++)
	{
		for(j=0; j<C2; j++)
		{
			printf("%3d",diffmat[i][j]);
		}
		printf("\n");
	}
	return;
}
```

50.

```
#include <stdio.h>
#include <conio.h>

main()
{
        int A[10][10], B[10][10];
        int i, j, R1, C1, R2, C2, flag =1;

        printf("Enter the order of the matrix A\n");
        scanf("%d %d", &R1, &C1);

        printf("Enter the order of the matrix B\n");
        scanf("%d %d", &R2,&C2);
        printf("Enter the elements of matrix A\n");
        for(i=0; i<R1; i++)
        {
                for(j=0; j<C1; j++)
                {
                        scanf("%d",&A[i][j]);
                }
        }
        printf("Enter the elements of matrix B\n");
        for(i=0; i<R2; i++)
        {
                for(j=0; j<C2; j++)
                {
                        scanf("%d",&B[i][j]);
                }
        }
        printf("MATRIX A is\n");
        for(i=0; i<R1; i++)
        {
                for(j=0; j<C1; j++)
                {
                        printf("%3d",A[i][j]);
                }
                printf("\n");
        }
        printf("MATRIX B is\n");
        for(i=0; i<R2; i++)
        {
                for(j=0; j<C2; j++)
                {
                        printf("%3d",B[i][j]);
                }
                printf("\n");
        }
/* Comparing two matrices for equality */

if(R1 == R2 && C1 == C2)
{
                printf("Matrices can be compared\n");
                for(i=0; i<R1; i++)
                {
                        for(j=0; j<C2; j++)
                        {
                                if(A[i][j] != B[i][j])
                                {
                                        flag = 0;
                                        break;
                                }
                        }
                }
}
        else
        {
printf(" Cannot be compared\n");
                exit(1);
        }

        if(flag == 1 )
                printf("Two matrices are equal\n");
        else
                printf("But,two matrices are not equal\n");
}
```

51. Program to illustrate how user authentication is made before allowing the user to access the secured resources. It asks for the user name and then the password. The password that you enter will not be displayed, instead that character is replaced by '*'.

```
#include <stdio.h>

void main()
{
        char pasword[10],usrname[10], ch;
        int i;
        clrscr();

        printf("Enter User name: ");
        gets(usrname);
        printf("Enter the password <any 8 characters>: ");

        for(i=0;i<8;i++)
        {
                ch = getch();
                pasword[i] = ch;
                ch = '*' ;
                printf("%c",ch);
        }
        pasword[i] = '\0';
        printf("\n\n\nYour password is :");

        for(i=0;i<8;i++)
        {
                 printf("%c",pasword[i]);
        }
}
```

52. Program to find the length of a string without using the built-in function.

```
 # include <stdio.h>

 main()
 {
        char string[50];
        int  i, length = 0;

        printf("Enter a string\n");
        gets(string);

        for (i=0; string[i] != '\0'; i++) /*keep going through each */
        {                               /*character of the string */
           length++;                       /*till its end */
        }
        printf("The length of a string is the number of characters in it\n");
        printf("So, the length of %s =%d\n", string, length);
 }
```

Output:

```
Enter a string
hello
The length of a string is the number of characters in it
So, the length of hello = 5

RUN2
Enter a string
E-Commerce is hot now
The length of a string is the number of characters in it
So, the length of E-Commerce is hot now =21
```

53. Program to read a matrix A (MxN) and to find the following using functions:
a) Sum of the elements of each row
b) Sum of the elements of each column
c) Find the sum of all the elements of the matrix

Output the computed results with suitable headings

```
#include <stdio.h>

main()
{
        int arr[10][10];
        int i, j, row, col, rowsum, colsum,sumall=0;
        clrscr();

        printf("Enter the order of the matrix\n");
        scanf("%d %d", &row, &col);

        printf("Enter the elements of the matrix\n");
        for(i=0; i<row; i++)
        {
                for(j=0; j< col; j++)
                {
                        scanf("%d", &arr[i][j]);
                }
        }
        printf("Input matrix is\n");
        for(i=0; i<row; i++)
        {
                for(j=0;j<col;j++)
                {
                        printf("%3d", arr[i][j]);
                }
                printf("\n");
        }
/* computing row sum */
        for(i=0; i<row; i++)
        {
                rowsum = Addrow(arr,i,col);
                printf("Sum of row %d = %d\n", i+1, rowsum);
        }

        for(j=0; j<col; j++)
        {
                colsum = Addcol(arr,j,row);
                printf("Sum of column  %d = %d\n", j+1, colsum);
        }
                                /* computation of all elements */
        for(j=0; j< row; j++)
        {
                sumall = sumall + Addrow(arr,j,col);
        }
        printf("Sum of all elements of matrix = %d\n", sumall);
}

/* Function to add each row */

int Addrow(int A[10][10], int k, int c)
{
        int rsum=0, i;
        for(i=0; i< c; i++)
        {
                rsum = rsum + A[k][i];
        }
        return(rsum);
}

/* Function to add each column */

int Addcol(int A[10][10], int k, int r)
{
        int csum=0, j;
        for(j=0; j< r; j++)
        {
                csum = csum + A[j][k];
        }
         return(csum);
}
```

Output:

```
Enter the order of the matrix
3
3

Enter the elements of the matrix
1 2 3
4 5 6
7 8 9

Input matrix is
  1  2  3
  4  5  6
  7  8  9

Sum of row 1 = 6
Sum of row 2 = 15
Sum of row 3 = 24
Sum of column  1 = 12
Sum of column  2 = 15
Sum of column  3 = 18
Sum of all elements of matrix = 45
```

54. Program to check whether a given integer number is positive or negative.

```
#include <stdio.h>

main()
{
        int number;
        clrscr();

        printf("Enter a number\n");
        scanf ("%d", &number);

        if (number > 0)
                printf ("%d, is a positive number\n", number);
        else
                printf ("%d, is a negative number\n", number);

}
```

Output:

```
Enter a number
-5
-5, is a negative number

RUN2
Enter a number
89
89, is a positive number
```

55. Program to find and output all the roots of a quadratic equation, for non-zero coefficients. In case of errors your program should report suitable error message.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

main()
{
        float A, B, C, root1, root2;
        float realp, imagp, disc;
        clrscr();

        printf("Enter the values of A, B and C\n");
        scanf("%f %f %f", &A,&B,&C);

        if( A==0 || B==0 || C==0)
        {
                printf("Error: Roots cannot be determined\n");
                exit(1);
        }
        else
        {
                disc = B*B - 4.0*A*C;
                if(disc < 0)
                {
                        printf("Imaginary Roots\n");
                        realp = -B/(2.0*A) ;
                        imagp = sqrt(abs(disc))/(2.0*A);
                        printf("Root1 = %f  +i %f\n",realp, imagp);
                        printf("Root2 = %f  -i %f\n",realp, imagp);
                }
                else if(disc == 0)
                {
                        printf("Roots are real and equal\n");
                        root1 = -B/(2.0*A);
                        root2 = root1;
                        printf("Root1 = %f  \n",root1);
                        printf("Root2 = %f  \n",root2);
                }
                else if(disc > 0 )
                {
                        printf("Roots are real and distinct\n");
                        root1 =(-B+sqrt(disc))/(2.0*A);
                        root2 =(-B-sqrt(disc))/(2.0*A);
                        printf("Root1 = %f  \n",root1);
                        printf("Root2 = %f  \n",root2);
                }
        }

 }
```

Output:

```
 RUN 1
 Enter the values of A, B and C: 3 2 1
 Imaginary Roots
 Root1 = -0.333333  +i 0.471405
 Root2 = -0.333333  -i 0.471405

 RUN 2
 Enter the values of A, B and C: 1 2 1
 Roots are real and equal
 Root1 = -1.000000
 Root2 = -1.000000

 RUN 3
 Enter the values of A, B and C: 3 5 2
 Roots are real and distinct
 Root1 = -0.666667
 Root2 = -1.000000
```

56. Program to simulate a simple calculator to perform arithmetic operations like addition, subtraction, multiplication and division only on integers. Error message should be reported if any attempt is made to divide by zero.

```
#include <stdio.h>

main()
{
        char operator;
        float n1, n2;
        float sum, diff, prod, quot, result;

        clrscr();

        printf ("Simulation of Simple Calculator\n\n");

        printf("Enter two numbers: \n");
        scanf ("%f %f", &n1, &n2);

        fflush (stdin);

        printf("Enter the operator [+,-,*,/]: \n");
        scanf ("%c", &operator);


        switch (operator)
        {
                case '+':
result = n1 + n2;
                        break;
                case '-':
result = n1 - n2;
                        break;
                case '*':
result = n1 * n2;
                        break;
                case '/':
result = n1 / n2;
                        break;
                default :
printf ("Error in operation\n");
```

```
                        break;
            }
            printf ("\n%5.2f %c %5.2f= %5.2f\n", n1,operator, n2, result);

}
```

Output:

```
Simulation of Simple Calculator

Enter two numbers: 3 5
Enter the operator [+,-,*,/]: +

 3.00 + 5.00 = 8.00

RUN2
Simulation of Simple Calculator

Enter two numbers: 12.75 8.45
Enter the operator [+,-,*,/]: -

12.75 - 8.45 = 4.30

RUN3
Simulation of Simple Calculator

Enter two numbers: 12 12
Enter the operator [+,-,*,/]: *

12.00 * 12.00 = 144.00

RUN4
Simulation of Simple Calculator

Enter two numbers: 5 9
Enter the operator [+,-,*,/]: /

 5.00 / 9.00 = 0.56
```

57. Program to find the GCD and LCM of two integers output the results along with the given integers. Use Euclids' algorithm.

```
#include <stdio.h>

main()
{
            int  num1, num2, gcd, lcm, remainder, numerator, denominator;
            clrscr();

            printf("Enter two numbers: \n");
            scanf("%d %d", &num1,&num2);

            if (num1 > num2)
            {
                        numerator = num1;
```

```
                denominator = num2;
        }
        else
        {
                numerator = num2;
                denominator = num1;
        }
        remainder = num1 % num2;
        while(remainder !=0)
        {
                numerator   = denominator;
                denominator = remainder;
                remainder   = numerator % denominator;
        }
        gcd = denominator;
        lcm = num1 * num2 / gcd;
        printf("GCD of %d and %d = %d \n", num1,num2,gcd);
        printf("LCM of %d and %d = %d \n", num1,num2,lcm);
}
```

Output:

```
RUN 1
Enter two numbers: 5 15
GCD of 5 and 15 = 5
LCM of 5 and 15 = 15
```

58. Program to find the sum of odd numbers and sum of even numbers from 1 to N. Output the computed sums on two different lines with suitable headings.

```
#include <stdio.h>

main()
{
        int i, N, oddSum = 0, evenSum = 0;
        clrscr();

        printf("Enter the value of N: \n");
        scanf ("%d", &N);

        for (i=1; i <=N; i++)
        {
                if (i % 2 == 0)
                evenSum = evenSum + i;
        }
else
                oddSum = oddSum + i;
        printf ("Sum of all odd numbers  = %d\n", oddSum);
        printf ("Sum of all even numbers = %d\n", evenSum);
}
```

Output:

```
RUN1
Enter the value of N: 10
Sum of all odd numbers  = 25
Sum of all even numbers = 30

RUN2
Enter the value of N: 50
Sum of all odd numbers  = 625
Sum of all even numbers = 650
```

59.	Program to check whether a given number is prime or not and output the given number with suitable message.

```
#include <stdio.h>
#include <stdlib.h>

main()
{
	int num, j, flag;
	clrscr();

	printf("Enter a number: \n");
	scanf("%d", &num);
	if ( num <= 1)
	{
		printf("%d is not a prime numbers\n", num);
		exit(1);
	}

	flag = 0;
	for ( j=2; j<= num/2; j++)
	{
		if( ( num % j ) == 0)
		{
			flag = 1;
			break;
		}
	}
	if(flag == 0)
		printf("%d is a prime number\n",num);
	else
		printf("%d is not a prime number\n", num);
}
```

Output:

```
RUN 1
Enter a number: 34
34 is not a prime number

RUN 2
Enter a number: 29
29 is a prime number
```

60.	Program to generate and print prime numbers in a given range. Also print the number of prime numbers.

```
#include <stdio.h>
#include <math.h>

main()
{
	int M, N, i, j, flag, temp, count = 0;

	clrscr();

	printf("Enter the value of M and N: \n");
	scanf("%d %d", &M,&N);

	if(N < 2)
	{
		printf("There are no primes upto %d\n", N);
		exit(0);
	}
	printf("Prime numbers are\n");
	temp = M;

	if ( M % 2 == 0)
	{
		M++;
	}
	for (i=M; i<=N; i=i+2)
	{
		flag = 0;

		for (j=2; j<=i/2; j++)
		{
			if( (i%j) == 0)
			{
				flag = 1;
				break;
			}
		}
		if(flag == 0)
		{
			printf("%d\n",i);
			count++;
		}
	}
	printf("Number of primes between %d  and %d = %d\n",temp,N,count);
}
```

Output:

```
Enter the value of M and N: 15 45

Prime numbers are
17
19
23
```

```
29
31
37
41
43
Number of primes between 15 and 45 = 8
```

61. Program to read N integers (zero, +ve and -ve) into an array A and to

a) Find the sum of negative numbers
b) Find the sum of positive numbers and
c) Find the average of all input numbers

Output the various results computed with proper headings.

```
#include <stdio.h>
#define MAXSIZE 10

main()
{
        int array[MAXSIZE];
        int i, N, negsum=0, posum=0, count1=0, count2=0;
        float total=0.0, averg;

        clrscr();
        printf ("Enter the value of N\n");
        scanf("%d", &N);

        printf("Enter %d numbers (-ve, +ve and zero)\n", N);
        for(i=0; i< N ; i++)
        {
                scanf("%d",&array[i]);
                fflush(stdin);
        }
        printf("Input array elements\n");
        for(i=0; i< N ; i++)
        {
                printf("%+3d\n",array[i]);
        }

                        /* Summing  begins */
        for(i=0; i< N ; i++)
        {

                if(array[i] < 0)
                {
                        negsum = negsum +  array[i];
                }
                else if(array[i] > 0)
                {
                        posum = posum + array[i];
                }
                else if( array[i] == 0)
                {
                         ;
                }
                total = total + array[i] ;
        }

        averg = total / N;
        printf("\nSum of all negative numbers    = %d\n",negsum);
        printf("Sum of all positive numbers    = %d\n", posum);
        printf("\nAverage of all input numbers   = %.2f\n", averg);

}
```

Output:

```
Enter the value of N: 5
Enter 5 numbers (-ve, +ve and zero)
5
-3
0
-7
6

Input array elements
 +5
 -3
 +0
 -7
 +6

Sum of all negative numbers    = -10
Sum of all positive numbers    = 11

Average of all input numbers   = 0.20
```

62. Program to sort N numbers in ascending order using Bubble sort and print both the given and the sorted array with suitable headings.

```
#include <stdio.h>
#define MAXSIZE 10

main()
{
        int array[MAXSIZE];
        int i, j, N, temp;

        clrscr();

        printf("Enter the value of N: \n");
        scanf("%d",&N);

        printf("Enter the elements one by one\n");
        for(i=0; i<N ; i++)
        {
                scanf("%d",&array[i]);
        }
        printf("Input array is\n");
        for(i=0; i<N ; i++)
        {
```

```
        printf("%d\n",array[i]);
    }
                /* Bubble sorting begins */
    for(i=0; i< N ; i++)
    {
        for(j=0; j< (N-i-1) ; j++)
        {
            if(array[j] > array[j+1])
            {
                temp      = array[j];
                array[j]  = array[j+1];
                array[j+1] = temp;
            }
        }
    }
    printf("Sorted array is...\n");
    for(i=0; i<N ; i++)
    {
        printf("%d\n",array[i]);
    }
}
```

Output:

```
Enter the value of N: 5
Enter the elements one by one
390
234
111
876
345

Input array is
390
234
111
876
345

Sorted array is...
111
234
345
390
876
```

63. Program to accept N numbers sorted in ascending order and to search for a given number using binary search. Report success or failure in the form of suitable messages.

```
#include <stdio.h>

main()
{
    int array[10];
    int i, j, N, temp, keynum, ascending = 0;
    int low,mid,high;
    clrscr();

    printf("Enter the value of N: \n");
    scanf("%d",&N);

    printf("Enter the elements one by one\n");
    for(i=0; i<N ; i++)
    {
        scanf("%d",&array[i]);
    }
    printf("Input array elements\n");
    for(i=0; i<N ; i++)
    {
        printf("%d\n",array[i]);
    }
                    /* Bubble sorting begins */
    for(i=0; i< N ; i++)
    {
        for(j=0; j< (N-i-1) ; j++)
        {
            if(array[j] > array[j+1])
            {
                temp = array[j];
                array[j] = array[j+1];
                array[j+1] = temp;
            }
        }
    }
    printf("Sorted array is...\n");
    for(i=0; i<N ; i++)
    {
        printf("%d\n",array[i]);
    }

    printf("Enter the element to be searched\n");
    scanf("%d", &keynum);

            /* Binary searching begins */
    low=1;
    high=N;
    do
    {
        mid= (low + high) / 2;
        if ( keynum < array[mid] )
            high = mid - 1;
        else if ( keynum > array[mid])
            low = mid + 1;
    } while( keynum!=array[mid] && low <= high);        /* End of do- while */

    if( keynum == array[mid] )
    {
        printf("SUCCESSFUL SEARCH\n");
    }
    else
        printf("Search is FAILED\n");
}
```

Output:

```
Enter the value of N: 4

Enter the elements one by one
3
1
4
2

Input array elements
3
1
4
2

Sorted array is...
1
2
3
4

Enter the element to be searched
4

SUCCESSFUL SEARCH
```

64. Program read a sentence and count the number of number of vowels and consonants in the given sentence. Output the results on two lines with suitable headings.

```
#include <stdio.h>

main()
{
        char sentence[80];
        int i, vowels=0, consonants=0, special = 0;
        clrscr();
        printf("Enter a sentence\n");
        gets(sentence);

        for(i=0; sentence[i] != '\0'; i++)
        {
                if((sentence[i] == 'a'||sentence[i] == 'e'||sentence[i] == 'i'||
                sentence[i] == 'o'||sentence[i] == 'u') ||(sentence[i] == 'A'||
                sentence[i] == 'E'||sentence[i] == 'I'|| sentence[i] == 'O'||
                sentence[i] == 'U'))
                {
                        vowels = vowels + 1;
                }
                else
                {
                        consonants = consonants + 1;
                }
                 if (sentence[i] =='\t' ||sentence[i] =='\0' || sentence[i] ==' ')
                {
                        special = special + 1;
                }
        }
        consonants = consonants - special;
        printf("No. of vowels in %s = %d\n", sentence, vowels);
        printf("No. of consonants in %s = %d\n", sentence, consonants);
}
```

Output:

```
Enter a sentence

Good Morning

No. of vowels in Good Morning = 4

No. of consonants in Good Morning = 7
```

IMPORTANT REVIEW QUESTIONS

1. Distinguish between getchar() and gets() functions.

getchar()	gets()
Used to receive a single character.	Used to receive a single string, white spaces and blanks.
Does not require any argument.	It requires a single argument.

2. Distinguish between scanf() and gets() functions.

scanf()	gets()
Strings with spaces cannot be accessed until ENTER key is pressed.	Strings with any number of spaces can be accessed.
All data types can be accessed.	Only character data type can be accessed.
Spaces and tabs are not acceptable as a part of the input string.	Spaces and tabs are perfectly acceptable of the input string as a part.
Any number of characters, integers.	Only one string can be received at a time. Strings, floats can be received at a time.

3. Distinguish between printf() and puts() functions.

puts()	printf()
They can display only one string at a time.	They can display any number of characters, integers or strings a time.
All data types of considered as characters.	Each data type is considered separately depending upon the conversion specifications.

4. What is the difference between a pre increment and a post increment operation?

A pre-increment operation such as ++a, increments the value of a by 1, before a is used for computation, while a post increment operation such as a++, uses the current value of a in the calculation and then increments the value of a by 1.

5. Distinguish between break and continue statement.

Break	Continue
Used to terminate the loops or to exist loop from a switch.	Used to transfer the control to the start of loop.
The break statement when executed causes immediate termination of loop containing it.	The continue statement when executed cause immediate termination of the current iteration of the loop.

6. Distinguish between while and do-while loops.

While loop	Do-while loop
The while loop tests the condition before each iteration.	The do-while loop tests the condition after the first iteration.
If the condition fails initially the loop is skipped entirely even in the first iteration.	Even if the condition fails initially the loop is executed once.

7. Distinguish between local and global variables

Local variables	Global variables
These are declared within the body of the function.	These are declared outside the function.
These variables can be referred only within the function in which it is declared.	These variables can be referred from any part of the program.
The value of the variables disappear once the function finishes its execution.	The value of the variables disappear only after the entire execution of the program.

8. State the differences between the function prototype the function definition

Function prototype	Function Definition
It declares the function.	It defines the function.
It ends with a semicolon.	It doesn't ends with a semicolon.
The declaration needn't include parameters.	It should include names for the parameters.

9. Compare and contrast recursion and iteration

Both involve repetition.
Both involve a termination test.
Both can occur infinitely.

Iteration	Recursion
Iteration explicitly user a repetition structure.	Recursion achieves repetition through repeated function calls.
Iteration terminates when the loop continuation.	Recursion terminates when a base case is recognized.
Iteration keeps modifying the counter until the loop continuation condition fails.	Recursion keeps producing simple versions of the original problem until the base case is reached.
Iteration normally occurs within a loop so the extra memory assigned is omitted.	Recursion causes another copy of the function and hence a considerable memory space's occupied.

It reduces the processor’s operating time.	It increases the processor’s operating time.

10. State the uses of pointers.

- Pointers provided an easy way to represent multidimensional arrays.
- Pointers increase the execution speed.
- Pointers reduce the length and complexity of program.

11. What is the difference between the address stored in a pointer and a value at the address?

The address stored in the pointer is the address of another variable. The value stored at that address is a stored in a different variable. The indirection operator (*) returns the value stored at the address, which itself is stored in a pointer.

12. What is the difference between the indirection operator and the address of operator?

The indirection operator (*) returns the value of the address stored in a pointer. The address of operator (&) returns the memory address of the variable.

13. State the difference between call by value and call by reference.

Call by value	Call by reference
int a;	int a;
Formal parameter ‘a’ is a local variable.	Formal parameter is ‘a’ local reference.
It cannot change the actual parameter.	It can change the actual parameter.
Actual parameter may be a constant, a variable, or an expression.	Actual parameter must be a variable.

14. State the advantages of using bit-fields

- To store Boolean variables (true/false) in one byte
- To encrypt certain routines to access the bits within a byte
- To transmit status information of devices encoded into one or more bits within a byte.

15. State the difference between arrays and structures.

Arrays	Structures
A array is an single entity representing a collection of data items of same data types.	A structure is a single entity representing a collection of data items of different data types.

Individual entries in an array are called elements.	Individual entries in a structure are called members.
An array declaration reserves enough memory space for its elements.	The structure definition reserves enough memory space for its members.
There is no keyword to represent arrays but the square braces [] preceding the variable name tells us that we are dealing with arrays.	The keyword struct tells us that we can dealing with structures.
Initialization of elements can be done during array declaration.	Initialization of members can be done only during structure definition.
The elements of an array are stored in sequence of memory locations.	The members of a structure are not stored in sequence of memory locations.
The array elements are accessed by its followed by the square braces [] within which the index is placed.	The members of a structure are accessed by the dot operator.

Its general format is data type variable name [size];	Its general format is:
	struct <struct name>
	{
	data_type structure member 1;
	data_type structure member 2;
	.
	.
	.
	data_type structure member N;
	} structure variable;

Example:
int sum [100];
	Example:
	struct student
	{
	char studname [25];
	int rollno;
	} stud1;

16. State the difference between structures and unions

Structures	Unions
Each member in a structure occupies and uses its own memory space	All the members of a union use the same memory space
The keyword struct tells us that we are dealing with structures	The keyword union tells us that we are dealing with unions
All the members of a structure can be initialized	Only the first member of an union can be initialized
More memory space is required since each member is stored in a separate memory locations	Less memory space is required since all members are stored int eh same memory locations

Its general format is:
struct <struct Name>
{
data_type structure Member1;
data_type structure Member2;
.
.
data_type structure Member N;
} structure variable;
	Its general formula is:
	union <union Name>
	{
	data_type structure Member1;
	data_type structure Member2;
	.
	.
	data_type structure Member N;
	} union variable;

Example:	Example:

struct student
{
char studname[25];
int rollno;
} stud1;
	union student
	{
	char studname[25];
	int rollno;
	} stud1;

17. Advantages of functions:

22. Function makes the lengthy and complex program easy and in short forms. It means large program can be sub-divided into self-contained and convenient small modules having unique name.

23. The length of source program can be reduced by using function by using it at different places in the program according to the user’s requirement.

24. By using function, memory space can be properly utilized. Also less memory is required to run program if function is used.

25. A function can be used by many programs.

26. Function increases the execution speed of the program and makes the programming simple.

27. By using the function, portability of the program is very easy.

28. It removes the redundancy (occurrence of duplication of programs) i.e. avoids the repetition and saves the time and space.

29. Debugging (removing error) becomes very easier and fast using the function sub-programming.

30. Functions are more flexible than library functions.

31. Testing (verification and validation) is very easy by using functions.

32. User can build a customized library of different functions used in daily routine having specific goal and link with the main program similar to the library functions.

18. A complete summarized table to represent the lifetime and visibility (scope) of all the storage class specifier is as below:

Storage class	Life time	Visibility (Scope)
Auto	Local	Local (within function)
Extern	Global	Global (in all functions)
Static	Global	Local
Register	Local	Local