UNIT-4

SEARCHING AND SORTING

INTRODUCTION TO SORTING

Sorting means arranging the elements of an array so that they are placed in some relevant order which may be either ascending or descending. That is, if A is an array, then the elements of A are arranged in a sorted order (ascending order) in such a way that A[0] < A[1] < A[2] < ..... < A[N].

For example, if we have an array that is declared and initialized as

int A[] = {21, 34, 11, 9, 1, 0, 22};

Then the sorted array (ascending order) can be given as:

A[] = {0, 1, 9, 11, 21, 22, 34;

A sorting algorithm is defined as an algorithm that puts the elements of a list in a certain order, which can be numerical order, lexicographical order, or any user-defined order. Efficient sorting algorithms are widely used to optimize the use of other algorithms like search and merge algorithms which require sorted lists to work correctly.

There are two types of sorting:

Internal sorting which deals with sorting the data stored in the computer's memory

External sorting which deals with sorting the data stored in files. External sorting is applied when there is voluminous data that cannot be stored in the memory.

Efficiency of sorting Algorithms:-

Efficiency of an algorithm depends on two parameters:

1. Time Complexity
2. Space Complexity

Time Complexity: Time Complexity is defined as the number of times a particular instruction set is executed rather than the total time is taken. It is because the total time taken also depends on some external factors like the compiler used, processor's speed, etc.

Space Complexity: Space Complexity is the total memory space required by the program for its execution.

Both are calculated as the function of input size(n).

One important thing here is that in spite of these parameters the efficiency of an algorithm also depends upon the **nature** and **size of** the **input.**

1. BUBBLE SORT

Bubble sort is a very simple method that sorts the array elements by repeatedly moving the largest element to the highest index position of the array segment (in case of arranging elements in ascending order). In *bubble sorting*, consecutive adjacent pairs of elements in the array are compared with each other. If the element at the lower index is greater than the element at the higher index, the two elements are interchanged so that the element is placed before the bigger one. This process will continue till the list of unsorted elements exhausts.

This procedure of sorting is called bubble sorting because elements 'bubble' to the top of the list. Finally at the end of the first pass, the largest element in the list will be placed at its proper position (i.e., at the end of the list).

Technique:-

The basic methodology of the working of bubble sort is given as follows:

(a) In Pass 1, A[0] and A[1] are compared, then A[1] is compared with A[2], A[2] is compared with A[3], and so on. Finally, A[N–2] is compared with A[N–1]. Pass 1 involves n–1 comparisons and places the biggest element at the highest index of the array.

(b) In Pass 2, A[0] and A[1] are compared, then A[1] is compared with A[2], A[2] is compared with A[3], and so on. Finally, A[N–3] is compared with A[N–2]. Pass 2 involves n–2 comparisons and places the second biggest element at the second highest index of the array.

(c) In Pass 3, A[0] and A[1] are compared, then A[1] is compared with A[2], A[2] is compared with A[3], and so on. Finally, A[N–4] is compared with A[N–3]. Pass 3 involves n–3 comparisons and places the third biggest element at the third highest index of the array.

(d) In Pass n–1, A[0] and A[1] are compared so that A[0]<A[1]. After this step, all the elements of the array are arranged in ascending order.

Algorithm for bubble sort:-

BUBBLE_SORT(A, N)

Step 1: Repeat Step 2 For 1 = to N-1

Step 2: Repeat For J = to N - I

Step 3: IF A[J] > A[J + 1]

SWAP A[J] and A[J+1]

[END OF INNER LOOP]

[END OF OUTER LOOP]

Step 4: EXIT

In this algorithm, the outer loop is for the total number of passes which is N–1. The inner loop will be executed for every pass. However, the frequency of the inner loop will decrease with every pass because after every pass, one element will be in its correct position. Therefore, for every pass, the inner loop will be executed N–I times, where N is the number of elements in the array and I is the count of the pass.

Complexity of Bubble Sort:-

The complexity of any sorting algorithm depends upon the number of comparisons. In bubble sort, we have seen that there are N–1 passes in total. In the first pass, N–1 comparisons are made to place the highest element in its correct position. Then, in Pass 2, there are N–2 comparisons and the second highest element is placed in its position. Therefore, to compute the complexity of bubble sort, we need to calculate the total number of comparisons. It can be given as:

f(n) = (n – 1) + (n – 2) + (n – 3) + ... + 3 + 2 + 1

f(n) = n (n – 1)/2

f(n) = n2/2 + O(n) = O(n2)

Therefore, the complexity of bubble sort algorithm is O(n2). It means the time required to execute bubble sort is proportional to n2, where n is the total number of elements in the array.

Example of Bubble sort:-

Let us consider an array A[] that has the following elements:

A[] = {30, 52, 29, 87, 63, 27, 19, 54}

Pass 1:

(a) Compare 30 and 52. Since 30 < 52, no swapping is done.

(b) Compare 52 and 29. Since 52 > 29, swapping is done.

30, **29, 52**, 87, 63, 27, 19, 54
(c) Compare 52 and 87. Since 52 < 87, no swapping is done.
(d) Compare 87 and 63. Since 87 > 63, swapping is done.
30, 29, 52, **63, 87**, 27, 19, 54
(e) Compare 87 and 27. Since 87 > 27, swapping is done.
30, 29, 52, 63, **27, 87**, 19, 54
(f) Compare 87 and 19. Since 87 > 19, swapping is done.
30, 29, 52, 63, 27, **19**, **87**, 54
(g) Compare 87 and 54. Since 87 > 54, swapping is done.
30, 29, 52, 63, 27, 19, **54, 87**
Observe that after the end of the first pass, the largest element is placed at the highest index of the array. All the other elements are still unsorted.

Pass 2:

(a) Compare 30 and 29. Since 30 > 29, swapping is done.
29, 30, 52, 63, 27, 19, 54, 87
(b) Compare 30 and 52. Since 30 < 52, no swapping is done.
(c) Compare 52 and 63. Since 52 < 63, no swapping is done.
(d) Compare 63 and 27. Since 63 > 27, swapping is done.
29, 30, 52, **27, 63**, 19, 54, 87
(e) Compare 63 and 19. Since 63 > 19, swapping is done.
29, 30, 52, 27, **19, 63**, 54, 87
(f) Compare 63 and 54. Since 63 > 54, swapping is done.
29, 30, 52, 27, 19, **54, 63**, 87
Observe that after the end of the second pass, the second largest element is placed at the second highest index of the array. All the other elements are still unsorted.

Pass 3:

(a) Compare 29 and 30. Since 29 < 30, no swapping is done.
(b) Compare 30 and 52. Since 30 < 52, no swapping is done.
(c) Compare 52 and 27. Since 52 > 27, swapping is done.
29, 30, **27, 52**, 19, 54, 63, 87
(d) Compare 52 and 19. Since 52 > 19, swapping is done.
29, 30, 27**, 19, 52**, 54, 63, 87
(e) Compare 52 and 54. Since 52 < 54, no swapping is done.
Observe that after the end of the third pass, the third largest element is placed at the third highest index of the array. All the other elements are still unsorted.

Pass 4:

(a) Compare 29 and 30. Since 29 < 30, no swapping is done.
(b) Compare 30 and 27. Since 30 > 27, swapping is done.
29, **27, 30**, 19, 52, 54, 63, 87
(c) Compare 30 and 19. Since 30 > 19, swapping is done.
29, 27, **19, 30**, 52, 54, 63, 87
(d) Compare 30 and 52. Since 30 < 52, no swapping is done.
Observe that after the end of the fourth pass, the fourth largest element is placed at the fourth highest index of the array. All the other elements are still unsorted.

Pass 5:
(a) Compare 29 and 27. Since 29 > 27, swapping is done.
27, 29, 19, 30, 52, 54, 63, 87
(b) Compare 29 and 19. Since 29 > 19, swapping is done.
27, **19, 29**, 30, 52, 54, 63, 87
(c) Compare 29 and 30. Since 29 < 30, no swapping is done.
Observe that after the end of the fifth pass, the fifth largest element is placed at the fifth highest index of the array. All the other elements are still unsorted.
Pass 6:
(a) Compare 27 and 19. Since 27 > 19, swapping is done.
19, 27, 29, 30, 52, 54, 63, 87
(b) Compare 27 and 29. Since 27 < 29, no swapping is done.
Observe that after the end of the sixth pass, the sixth largest element is placed at the sixth largest index of the array. All the other elements are still unsorted.
Pass 7:
(a) Compare 19 and 27. Since 19 < 27, no swapping is done.

2. INSERTION SORT

Insertion sort is a very simple sorting algorithm in which the sorted array (or list) is built one element at a time. We all are familiar with this technique of sorting, as we usually use it for ordering a deck of cards while playing bridge.

The main idea behind insertion sort is that it inserts each item into its proper place in the final list. To save memory, most implementations of the insertion sort algorithm work by moving the current data element past the already sorted values and repeatedly interchanging it with the preceding value until it is in its correct place. Insertion sort is less efficient as compared to other more advanced algorithms such as quick sort, heap sort, and merge sort.

Technique:-

Insertion sort works as follows:

- The array of values to be sorted is divided into two sets. One that stores sorted values and another that contains unsorted values.
- The sorting algorithm will proceed until there are elements in the unsorted set.
- Suppose there are n elements in the array. Initially, the element with index 0 (assuming LB = is in the sorted set. Rest of the elements is in the unsorted set.
- The first element of the unsorted partition has array index 1 (if LB = 0).
- During each iteration of the algorithm, the first element in the unsorted set is picked up and inserted into the correct position in the sorted set.

Algorithm of insertion sort:-

INSERTION-SORT (ARR, N)
Step 1: Repeat Steps 2 to 5 for K = 1 to N – 1
Step 2: SET TEMP = ARR[K]
Step 3: SET J = K - 1
Step 4: Repeat while TEMP <= ARR[J]
SET ARR[J + 1] = ARR[J]
SET J = J - 1

[END OF INNER LOOP]
Step 5: SET ARR[J + 1] = TEMP
[END OF LOOP]
Step 6: EXIT

To insert an element A[K] in a sorted list A[0], A[1], ..., A[K–1], we need to compare A[K] with A[K–1], then with A[K–2], A[K–3], and so on until we meet an element A[J] such that A[J] <= A[K]. In order to insert A[K] in its correct position, we need to move elements A[K–1], A[K–2], ..., A[J] by one position and then A[K] is inserted at the (J+1)th location.

In the algorithm, Step 1 executes a for loop which will be repeated for each element in the array. In Step 2, we store the value of the Kth element in TEMP. In Step 3, we set the Jth index in the array. In Step 4, a for loop is executed that will create space for the new element from the unsorted list to be stored in the list of sorted elements. Finally, in Step 5, the element is stored at the (J+1)th location.

Complexity of Insertion Sort:-

For insertion sort, the best case occurs when the array is already sorted. In this case, the running time of the algorithm has a linear running time (i.e., O(n)). This is because, during each iteration, the first element from the unsorted set is compared only with the last element of the sorted set of the array.

Similarly, the worst case of the insertion sort algorithm occurs when the array is sorted in the reverse order. In the worst case, the first element of the unsorted set has to be compared with almost every element in the sorted set. Furthermore, every iteration of the inner loop will have to shift the elements of the sorted set of the array before inserting the next element. Therefore, in the worst case, insertion sort has a quadratic running time (i.e., O(n2)). Even in the average case, the insertion sort algorithm will have to make at least (K–1)/2 comparisons. Thus, the average case also has a quadratic running time.

Advantages of Insertion Sort:-

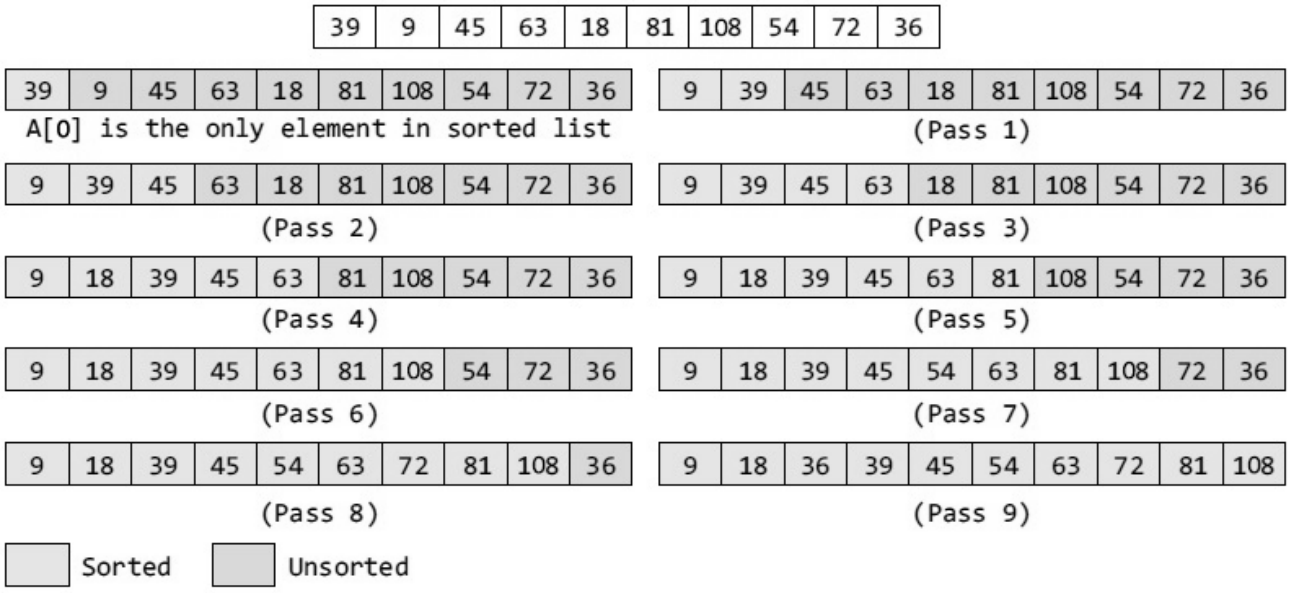
The advantages of this sorting algorithm are as follows:

- It is easy to implement and efficient to use on small sets of data.
- It can be efficiently implemented on data sets that are already substantially sorted.
- It performs better than algorithms like selection sort and bubble sort. Insertion sort algorithm is simpler than shell sort, with only a small trade-off in efficiency. It is over twice as fast as the bubble sort and almost 40 per cent faster than the selection sort.
- It requires less memory space (only O(1) of additional memory space).
- It is said to be online, as it can sort a list as and when it receives new elements.

Example:-

Consider an array of integers given below. We will sort the values in the array using insertion sort.

Initially, A[0] is the only element in the sorted set. In Pass 1, A[1] will be placed either before or after A[0], so that the array A is sorted. In Pass 2, A[2] will be placed either before A[0], in between A[0] and A[1], or after A[1]. In Pass 3, A[3] will be placed in its proper place. In Pass N–1, A[N–1] will be placed in its proper place to keep the array sorted.



3. SELECTION SORT

Selection sort is a sorting algorithm that has a quadratic running time complexity of O(n2), thereby making it inefficient to be used on large lists. Although selection sort performs worse than insertion sort algorithm, it is noted for its simplicity and also has performance advantages over more complicated algorithms in certain situations. Selection sort is generally used for sorting files with very large objects (records) and small keys.

Technique:-

Consider an array ARR with N elements. Selection sort works as follows:

First find the smallest value in the array and place it in the first position. Then, find the second smallest value in the array and place it in the second position. Repeat this procedure until the entire array is sorted. Therefore,

- In Pass 1, find the position POS of the smallest value in the array and then swap ARR[POS] and ARR[0]. Thus, ARR[0] is sorted.
- In Pass 2, find the position POS of the smallest value in sub-array of N–1 elements. Swap ARR[POS] with ARR[1]. Now, ARR[0] and ARR[1] are sorted.
- In Pass N–1, find the position POS of the smaller of the elements ARR[N–2] and ARR[N–1]. Swap ARR[POS] and ARR[N–2] so that ARR[0], ARR[1], ..., ARR[N–1] is sorted.

Algorithm for selection sort:-

SMALLEST (ARR, K, N, POS)
Step 1: [INITIALIZE] SET SMALL = ARR[K]
Step 2: [INITIALIZE] SET POS = K
Step 3: Repeat for J = K+1 to N
IF SMALL > ARR[J]
SET SMALL = ARR[J]
SET POS = J
[END OF IF]
[END OF LOOP]
Step 4: RETURN POS
SELECTION SORT(ARR, N)

Step 1: Repeat Steps 2 and 3 for K = 1
to N-1
Step 2: CALL SMALLEST(ARR, K, N, POS)
Step 3: SWAP A[K] with ARR[POS]
[END OF LOOP]
Step 4: EXIT

In the algorithm, during the Kth pass, we need to find the position POS of the smallest elements from ARR[K], ARR[K+1], ..., ARR[N]. To find the smallest element, we use a variable SMALL to hold the smallest value in the sub-array ranging from ARR[K] to ARR[N]. Then, swap ARR[K] with ARR[POS]. This procedure is repeated until all the elements in the array are sorted.

Complexity of Selection Sort:-

Selection sort is a sorting algorithm that is independent of the original order of elements in the array. In Pass 1, selecting the element with the smallest value calls for scanning all n elements; thus, n–1 comparisons are required in the first pass. Then, the smallest value is swapped with the element in the first position. In Pass 2, selecting the second smallest value requires scanning the remaining n – 1 element and so on. Therefore,

(n – 1) + (n – 2) + ... + 2 + 1= n(n – 1) / 2 = O(n2) comparisons.

Advantages of Selection Sort:-

- It is simple and easy to implement.
- It can be used for small data sets.
- It is 60 per cent more efficient than bubble sort.

However, in case of large data sets, the efficiency of selection sort drops as compared to insertion sort.

Example

Sort the array given below using selection sort.

39	9	81	45	90	27	72	18

PASS	POS	ARR[0]	ARR[1]	ARR[2]	ARR[3]	ARR[4]	ARR[5]	ARR[6]	ARR[7]
1	1	9	39	81	45	90	27	72	18
2	7	9	18	81	45	90	27	72	39
3	5	9	18	27	45	90	81	72	39
4	7	9	18	27	39	90	81	72	45
5	7	9	18	27	39	45	81	72	90
6	6	9	18	27	39	45	72	81	90
7	6	9	18	27	39	45	72	81	90

4. MERGE SORT

Merge sort is a sorting algorithm that uses the divide, conquer, and combine algorithmic paradigm.

Divide means partitioning the n-element array to be sorted into two sub-arrays of n/2 elements. If A is an array containing zero or one element, then it is already sorted. However, if there are more elements in the array, divide A into two sub-arrays, A1 and A2, each containing about half of the elements of A.

Conquer means sorting the two sub-arrays recursively using merge sort.

Combine means merging the two sorted sub-arrays of size n/2 to produce the sorted array of n elements.

Merge sort algorithm focuses on two main concepts to improve its performance (running time):

- A smaller list takes fewer steps and thus less time to sort than a large list.
- As number of steps is relatively less, thus less time is needed to create a sorted list from two sorted lists rather than creating it using two unsorted lists.

The basic steps of a merge sort algorithm are as follows:

- If the array is of length 0 or 1, then it is already sorted.
- Otherwise, divide the unsorted array into two sub-arrays of about half the size.
- Use merge sort algorithm recursively to sort each sub-array.
- Merge the two sub-arrays to form a single sorted list.

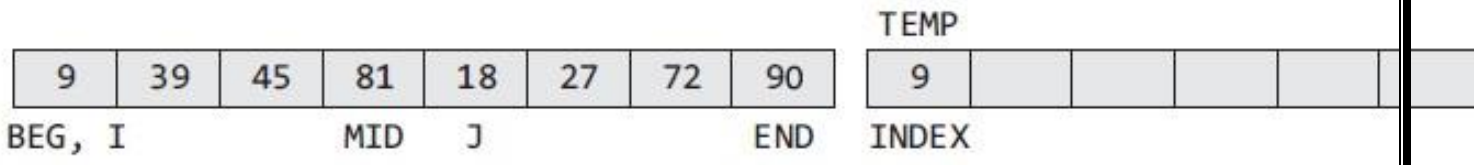
Algorithm:-

MERGE (ARR, BEG, MID, END)
Step 1: [INITIALIZE] SET I = BEG, J = MID + 1, INDEX =
Step 2: Repeat while (I <= MID) AND (J<=END)
IF ARR[I] < ARR[J]
SET TEMP[INDEX] = ARR[I]
SET I = I + 1
ELSE
SET TEMP[INDEX] = ARR[J]
SET J = J + 1
[END OF IF]
SET INDEX = INDEX + 1
[END OF LOOP]
Step 3: [Copy the remaining elements of right sub-array, if any]
IF I > MID
Repeat while J <= END
SET TEMP[INDEX] = ARR[J]
SET INDEX = INDEX + 1, SET J = J + 1
[END OF LOOP]
[Copy the remaining elements of left sub-array, if any]
ELSE
Repeat while I <= MID
SET TEMP[INDEX] = ARR[I]
SET INDEX = INDEX + 1, SET I = I + 1
[END OF LOOP]
[END OF IF]
Step 4: [Copy the contents of TEMP back to ARR] SET K=
Step 5: Repeat while K < INDEX
SET ARR[K] = TEMP[K]
SET K = K + 1
[END OF LOOP]
Step 6: END

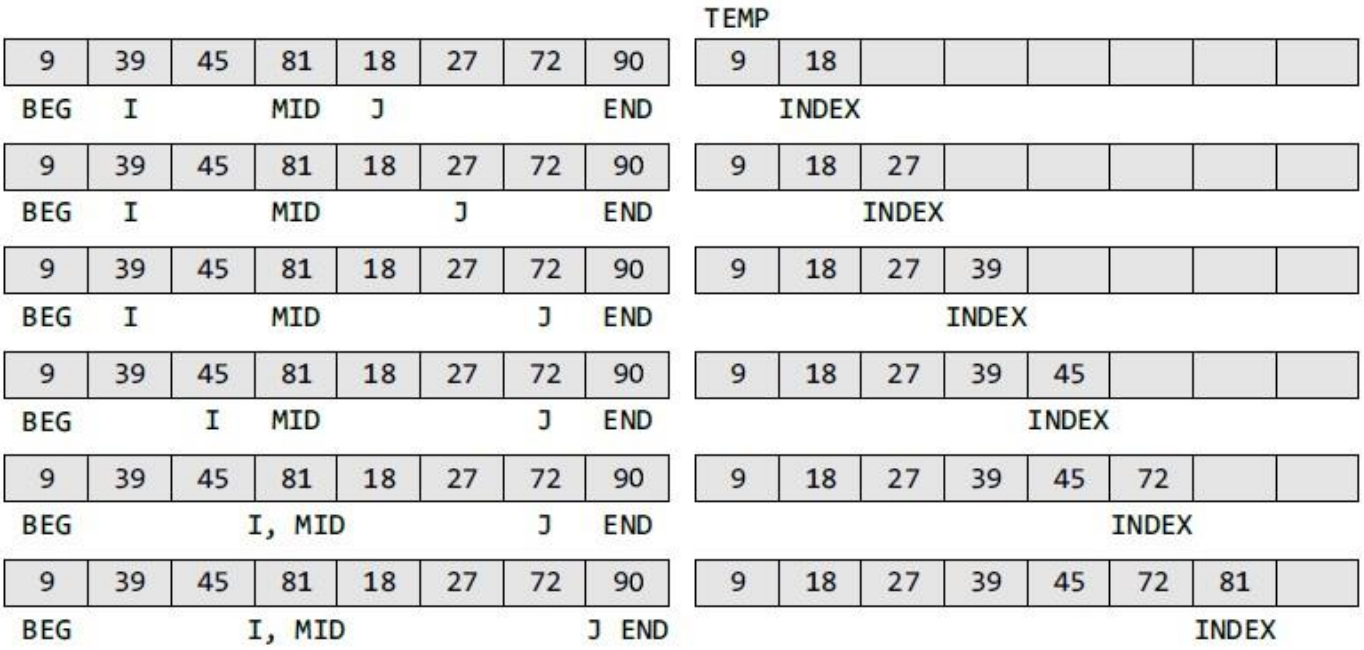
MERGE_SORT(ARR, BEG, END)
Step 1: IF BEG < END
SET MID = (BEG + END)/2
CALL MERGE_SORT (ARR, BEG, MID)
CALL MERGE_SORT (ARR, MID + 1, END)
MERGE (ARR, BEG, MID, END)

[END OF IF]
Step 2: END

The merge sort algorithm uses a function merge which combines the sub-arrays to form a sorted array. While the merge sort algorithm recursively divides the list into smaller lists, the merge algorithm conquers the list to sort the elements in individual lists. Finally, the smaller lists are merged to form one list. The same concept can be utilized to merge four sub-lists containing two elements, or eight sub-lists having one element each.



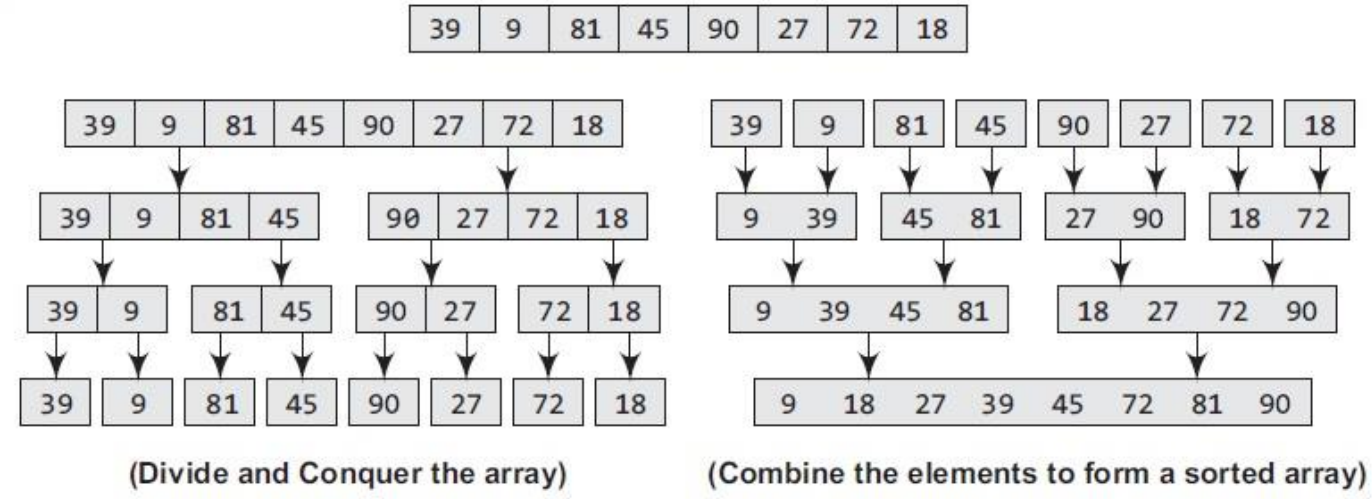
Compare ARR[I] and ARR[J], the smaller of the two is placed in TEMP at the location specified by INDEX and subsequently the value I or J is incremented.



When I is greater than MID, copy the remaining elements of the right sub-array in TEMP.

The running time of merge sort in the average case and the worst case can be given as O(n logn). Although merge sort has an optimal time complexity, it needs an additional space of O(n) for the temporary array TEMP.

Example: - Sort the array given below using merge sort.



5. QUICK SORT

Quick sort is a widely used sorting algorithm developed by C. A. R. Hoare that makes O(n log n) comparisons in the average case to sort an array of n elements. However, in the worst case, it has a quadratic running time given as O(n2). Basically, the quick sort algorithm is faster than other O(n log n) algorithms, because its efficient implementation can minimize the probability of requiring quadratic time. Quick sort is also known as partition exchange sort.

Like merge sort, this algorithm works by using a divide-and-conquer strategy to divide a single unsorted array into two smaller sub-arrays.

The quick sort algorithm works as follows:

1. Select an element pivot from the array elements.
2. Rearrange the elements in the array in such a way that all elements that are less than the pivot appear before the pivot and all elements greater than the pivot element come after it (equal values can go either way). After such a partitioning, the pivot is placed in its final position. This is called the *partition* operation.
3. Recursively sort the two sub-arrays thus obtained. (One with sub-list of values smaller than that of the pivot element and the other having higher value elements.)

Like merge sort, the *base case* of the recursion occurs when the array has zero or one element because in that case the array is already sorted. After each iteration, one element (pivot) is always in its final position. Hence, with every iteration, there is one less element to be sorted in the array. Thus, the main task is to find the pivot element, which will partition the array into two halves. To understand how we find the pivot element, follow the steps given below. (We take the first element in the array as pivot.)

Technique

Quick sort works as follows:

1. Set the index of the first element in the array to loc and left variables. Also, set the index of the last element of the array to the right variable.

That is, loc = 0, left = 0, and right = n–1 (where n in the number of elements in the array)

2. Start from the element pointed by right and scan the array from right to left, comparing each element on the way with the element pointed by the variable loc. That is, a[loc] should be less than a[right].

(a) If that is the case, then simply continue comparing until right becomes equal to loc. Once right = loc, it means the pivot has been placed in its correct position.

(b) However, if at any point, we have a[loc] > a[right], then interchange the two values and jump to Step 3.
(c) Set loc = right
3. Start from the element pointed by left and scan the array from left to right, comparing each element on the way with the element pointed by loc.
That is, a[loc] should be greater than a[left].
(a) If that is the case, then simply continue comparing until left becomes equal to loc. Once left = loc, it means the pivot has been placed in its correct position.
(b) However, if at any point, we have a[loc] < a[left], then interchange the two values and jump to Step 2.
(c) Set loc = left.

Algorithm:-

The quick sort algorithm makes use of a function Partition to divide the array into two sub-arrays.

PARTITION (ARR, BEG, END, LOC)
Step 1: [INITIALIZE] SET LEFT = BEG, RIGHT = END, LOC = BEG, FLAG =0
Step 2: Repeat Steps 3 to 6 while FLAG =0
Step 3: Repeat while ARR[LOC] <= ARR[RIGHT] AND LOC != RIGHT
SET RIGHT = RIGHT - 1
[END OF LOOP]
Step 4: IF LOC = RIGHT
SET FLAG = 1
ELSE IF ARR[LOC] > ARR[RIGHT]
SWAP ARR[LOC] with ARR[RIGHT]
SET LOC = RIGHT
[END OF IF]
Step 5: IF FLAG =0
Repeat while ARR[LOC] >= ARR[LEFT] AND LOC != LEFT
SET LEFT = LEFT + 1
[END OF LOOP]
Step 6: IF LOC = LEFT
SET FLAG = 1
ELSE IF ARR[LOC] < ARR[LEFT]
SWAP ARR[LOC] with ARR[LEFT]
SET LOC = LEFT
[END OF IF]
[END OF IF]
Step 7: [END OF LOOP]
Step 8: END

QUICK_SORT (ARR, BEG, END)
Step 1: IF (BEG < END)
CALL PARTITION (ARR, BEG, END, LOC)
CALL QUICKSORT(ARR, BEG, LOC - 1)
CALL QUICKSORT(ARR, LOC + 1, END)
[END OF IF]
Step 2: END

Complexity of Quick Sort

In the average case, the running time of quick sort can be given as O(n log n). The partitioning of the array which simply loops over the elements of the array once uses O(n) time.

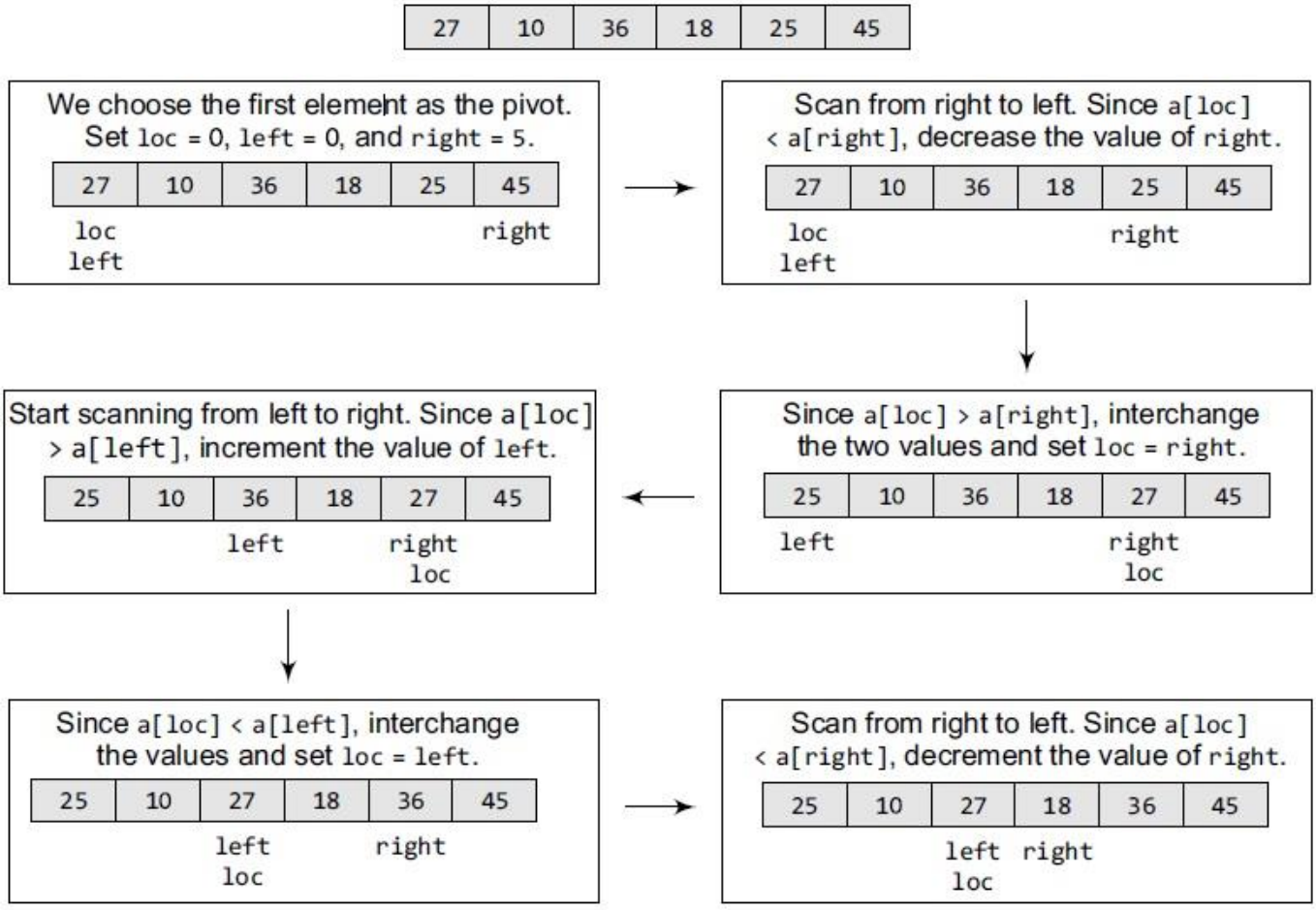
In the best case, every time we partition the array, we divide the list into two nearly equal pieces. That is, the recursive call processes the sub-array of half the size. At the most, only log n nested calls can be made before we reach a sub-array of size 1. It means the depth of the call tree is O(log n). And because at each level, there can only be O(n), the resultant time is given as O(n log n) time.

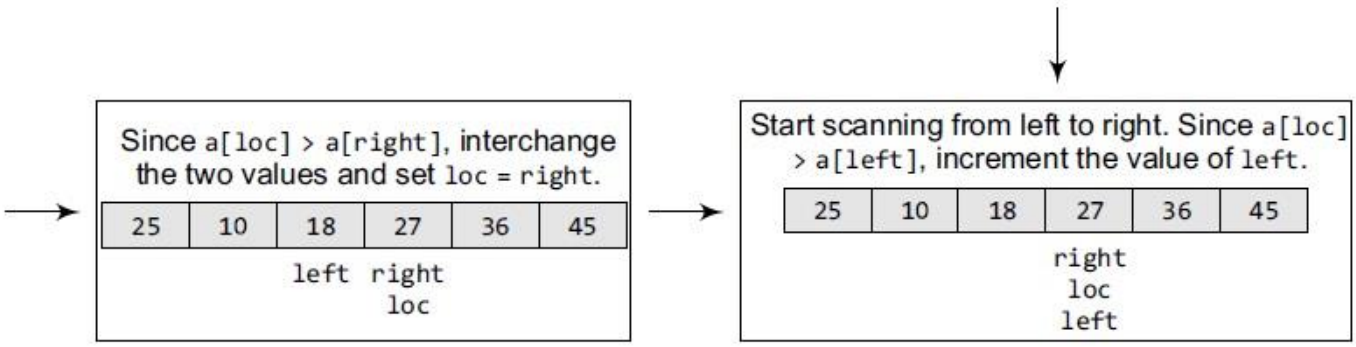
Practically, the efficiency of quick sort depends on the element which is chosen as the pivot. Its worst-case efficiency is given as $O(n^2)$. The worst case occurs when the array is already sorted (either in ascending or descending order) and the left-most element is chosen as the pivot. However, many implementations randomly choose the pivot element. The randomized version of the quick sort algorithm always has an algorithmic complexity of O(n log n).

Pros and Cons of Quick Sort

It is faster than other algorithms such as bubble sort, selection sort, and insertion sort. Quick sort can be used to sort arrays of small size, medium size, or large size. On the flip side, quick sort is complex and massively recursive.

Example: - Sort the elements given in the following array using quick sort algorithm.





Now left = loc, so the procedure terminates, as the pivot element (the first element of the array, that is, 27) is placed in its correct position. All the elements smaller than 27 are placed before it and those greater than 27 are placed after it.

The left sub-array containing 25, 10, 18 and the right sub-array containing 36 and 45 are sorted in the same manner.

6. RADIX SORT

Radix sort is a linear sorting algorithm for integers and uses the concept of sorting names in alphabetical order. When we have a list of sorted names, the *radix* is 26 (or 26 buckets) because there are 26 letters in the English alphabet. So radix sort is also known as bucket sort. Observe that words are first sorted according to the first letter of the name. That is, 26 classes are used to arrange the names, where the first class stores the names that begin with A, the second class contains the names with B, and so on.

During the second pass, names are grouped according to the second letter. After the second pass, names are sorted on the first two letters. This process is continued till the nth pass, where n is the length of the name with maximum number of letters.

After every pass, all the names are collected in order of buckets. That is, first pick up the names in the first bucket that contains the names beginning with A. In the second pass, collect the names from the second bucket, and so on.

When radix sort is used on integers, sorting is done on each of the digits in the number. The sorting procedure proceeds by sorting the least significant to the most significant digit. While sorting the numbers, we have ten buckets, each for one digit (0, 1, 2… 9) and the number of passes will depend on the length of the number having maximum number of digits.

Algorithm for radix sort:-

Algorithm for RadixSort (ARR, N)

Step 1: Find the largest number in ARR as LARGE
Step 2: [INITIALIZE] SET NOP = Number of digits in LARGE
Step 3: SET PASS =
Step 4: Repeat Step 5 while PASS <= NOP-1
Step 5: SET I = and INITIALIZE buckets
Step 6: Repeat Steps 7 to 9 while I<N-1
Step 7: SET DIGIT = digit at PASSth place in A[I]
Step 8: Add A[I] to the bucket numbered DIGIT
Step 9: INCEREMENT bucket count for bucket numbered DIGIT
[END OF LOOP]
Step 10: Collect the numbers in the bucket
[END OF LOOP]

Step 11: END

Complexity of Radix Sort

To calculate the complexity of radix sort algorithm, assume that there are n numbers that have to be sorted and k is the number of digits in the largest number. In this case, the radix sort algorithm is called a total of k times. The inner loop is executed n times. Hence, the entire radix sort algorithm takes O(kn) time to execute. When radix sort is applied on a data set of finite size (very small set of numbers), then the algorithm runs in O(n) asymptotic time.

Pros and Cons of Radix Sort

Radix sort is a very simple algorithm. When programmed properly, radix sort is one of the fastest sorting algorithms for numbers or strings of letters.

But there are certain trade-offs for radix sort that can make it less preferable as compared to other sorting algorithms. Radix sort takes more space than other sorting algorithms. Besides the array of numbers, we need 10 buckets to sort numbers, 26 buckets to sort strings containing only characters, and at least 40 buckets to sort a string containing alphanumeric characters.

Another drawback of radix sort is that the algorithm is dependent on digits or letters. This feature compromises with the flexibility to sort input of any data type. For every different data type, the algorithm has to be rewritten. Even if the sorting order changes, the algorithm has to be rewritten. Thus, radix sort takes more time to write and writing a general purpose radix sort algorithm that can handle all kinds of data is not a trivial task.

Radix sort is a good choice for many programs that need a fast sort, but there are faster sorting algorithms available. This is the main reason why radix sort is not as widely used as other sorting algorithms.

Example: - Sort the numbers given below using radix sort.

345, 654, 924, 123, 567, 472, 555, 808, 911

In the first pass, the numbers are sorted according to the digit at ones place. The buckets are pictured upside down as shown below.

Number	0	1	2	3	4	5	6	7	8	9
345						345				
654					654					
924					924					
123				123						
567								567		
472			472							
555						555				
808									808	
911		911								

After this pass, the numbers are collected bucket by bucket. The new list thus formed is used as an input for the next pass. In the second pass, the numbers are sorted according to the digit at the tens place. The buckets are pictured upside down.

Number	0	1	2	3	4	5	6	7	8	9
911		911								
472								472		
123			123							
654						654				
924			924							
345					345					
555						555				
567							567			
808	808									

In the third pass, the numbers are sorted according to the digit at the hundreds place. The buckets are pictured upside down.

Number	0	1	2	3	4	5	6	7	8	9
808									808	
911										911
123		123								
924										924
345				345						
654							654			
555						555				
567						567				
472					472					

The numbers are collected bucket by bucket. The new list thus formed is the final sorted result. After the third pass, the list can be given as 123, 345, 472, 555, 567, 654, 808, 911, and 924.

7. HEAP SORT (TOURNAMENT SORT)

Given an array ARR with n elements, the heap sort algorithm can be used to sort ARR in two phases:

- In phase 1, build a heap H using the elements of ARR.
- In phase 2, repeatedly delete the root element of the heap formed in phase 1.

In a max heap, we know that the largest value in H is always present at the root node. So in phase 2, when the root element is deleted, we are actually collecting the elements of ARR in decreasing order.

The algorithm of heap sort is shown in below

HEAPSORT(ARR, N)

Step 1: [Build Heap H]
Repeat for I = to N-1
CALL Insert_Heap(ARR, N, ARR[I])
[END OF LOOP]
Step 2: (Repeatedly delete the root element)
Repeat while N>0

CALL Delete_Heap(ARR, N, VAL)
SET N = N + 1
[END OF LOOP]
Step 3: END

Complexity of Heap Sort

Heap sort uses two heap operations: *insertion* and *root deletion*. Each element extracted from the root is placed in the last empty location of the array.

In phase 1, when we build a heap, the number of comparisons to find the right location of the new element in H cannot exceed the depth of H. Since H is a complete tree, its depth cannot exceed m, where m is the number of elements in heap H.

Thus, the total number of comparisons g(n) to insert n elements of ARR in H is bounded as:

g(n) <= n log n

Hence, the running time of the first phase of the heap sort algorithm is O(n log n).

In phase 2, we have H which is a complete tree with m elements having left and right sub-trees as heaps. Assuming L to be the root of the tree, ***reheaping*** the tree would need 4 comparisons to move L one step down the tree H. Since the depth of H cannot exceed O(log m), **reheaping** the tree will require a maximum of 4 log m comparisons to find the right location of L in H.

Since n elements will be deleted from heap H, **reheaping** will be done n times. Therefore, the number of comparisons to delete n elements is bounded as:

h(n) <= 4n log n

Hence, the running time of the second phase of the heap sort algorithm is O(n log n).

Each phase requires time proportional to O(n log n). Therefore, the running time to sort an array of n elements in the worst case is proportional to O(n log n).

Therefore, we can conclude that heap sort is a simple, fast, and stable sorting algorithm that can be used to sort large sets of data efficiently.

8. SHELL SORT

Shell sort, invented by Donald Shell in 1959, is a sorting algorithm that is a generalization of insertion sort. While discussing insertion sort, we have observed two things:

- First, insertion sort works well when the input data is 'almost sorted'.
- Second, insertion sort is quite inefficient to use as it moves the values just one position at a time.

Shell sort is considered an improvement over insertion sort as it compares elements separated by a gap of several positions. This enables the element to take bigger steps towards its expected position. In Shell sort, elements are sorted in multiple passes and in each pass, data are taken with smaller and smaller gap sizes. However, the last step of shell sort is a plain insertion sort. But by the time we reach the last step, the elements are already 'almost sorted', and hence it provides good performance.

If we take a scenario in which the smallest element is stored in the other end of the array, then sorting such an array with either bubble sort or insertion sort will execute in O(n2) time and take roughly n comparisons and exchanges to move this value all the way to its correct position. On the other hand, Shell sort first moves small values using giant step sizes, so a small value will move a long way towards its final position, with just a few comparisons and exchanges.

Technique

To visualize the way in which shell sort works, perform the following steps:

Step 1: Arrange the elements of the array in the form of a table and sort the columns (using insertion sort).

Step 2: Repeat Step 1, each time with smaller number of longer columns in such a way that at the end, there is only one column of data to be sorted.

Note that we are only visualizing the elements being arranged in a table, the algorithm does its sorting in-place.

Algorithm for shell sort

The algorithm to sort an array of elements using shell sort is shown in below. In the algorithm, we sort the elements of the array Arr in multiple passes. In each pass, we reduce the gap_size (visualize it as the number of columns) by a factor of half as done in Step 4. In each iteration of the for loop in Step 5, we compare the values of the array and interchange them if we have a larger value preceding the smaller one.

Shell_Sort(Arr, n)

Step 1: SET FLAG = 1, GAP_SIZE = N
Step 2: Repeat Steps 3 to 6 while FLAG = 1 OR GAP_SIZE > 1
Step 3: SET FLAG =0
Step 4: SET GAP_SIZE = (GAP_SIZE + 1) / 2
Step 5: Repeat Step 6 for I =0 to I < (N - GAP_SIZE)
Step 6: IF Arr[I + GAP_SIZE] > Arr[I]
SWAP Arr[I + GAP_SIZE], Arr[I]
SET FLAG =0
Step 7: END

Example Sort the elements given below using shell sort.

63, 19, 7, 90, 81, 36, 54, 45, 72, 27, 22, 9, 41, 59, 33

Solution

Arrange the elements of the array in the form of a table and sort the columns.

63	19	7	90	81	36	54	45
72	27	22	9	41	59	33	

Result:

63	19	7	9	41	36	33	45
72	27	22	90	81	59	54	

The elements of the array can be given as:

63, 19, 7, 9, 41, 36, 33, 45, 72, 27, 22, 90, 81, 59, 54

Repeat Step 1 with smaller number of long columns.

63	19	7	9	41
36	33	45	72	27
22	90	81	59	54

Result:

22	19	7	9	27
36	33	45	59	41
63	90	81	72	54

The elements of the array can be given as:

22, 19, 7, 9, 27, 36, 33, 45, 59, 41, 63, 90, 81, 72, 54

Repeat Step 1 with smaller number of long columns.

			Result:		
22	19	7	9	19	7
9	27	36	22	27	36
33	45	59	33	45	54
41	63	90	41	63	59
81	72	54	81	72	90

The elements of the array can be given as:

9, 19, 7, 22, 27, 36, 33, 45, 54, 41, 63, 59, 81, 72, 90

Finally, arrange the elements of the array in a single column and sort the column.

	Result:
9	7
19	9
7	19
22	22
27	27
36	33
33	36
45	41
54	45
41	54
63	59
59	63
81	72
72	81
90	90

Finally, the elements of the array can be given as:

7, 9, 19, 22, 27, 33, 36, 41, 45, 54, 59, 63, 72, 81, 90

INTRODUCTION TO SEARCHING

Searching means to find whether a particular value is present in an array or not. If the value is present in the array, then searching is said to be successful and the searching process gives the location of that value in the array. However, if the value is not present in the array, the searching process displays an appropriate message and in this case searching is said to be unsuccessful.

There are two popular methods for searching the array elements: *linear search* and *binary search*. The algorithm that should be used depends entirely on how the values are organized in the array. For example, if the elements of the array are arranged in ascending order, then binary search should be used, as it is more efficient for sorted lists in terms of complexity. We will discuss all these methods in detail in this section.

1. LINEAR SEARCH (OR) SEQEUENTIAL SEARCH

Linear search, also called as *sequential search*, is a very simple method used for searching an array for a particular value. It works by comparing the value to be searched with every element of the array one by one in a sequence until a match is found. Linear search is mostly used to search an unordered list of elements (array in which data elements are not sorted). For example, if an array A[] is declared and initialized as,

int A[] = {10, 8, 2, 7, 3, 4, 9, 1, 6, 5}; and the value to be searched is VAL = 7, then searching means to find whether the value '7' is present in the array or not.
If yes, then it returns the position of its occurrence. Here, POS = 3 (index starting from 0).

Algorithm for linear search:-

LINEAR_SEARCH(A, N, VAL)
Step 1: [INITIALIZE] SET POS = -1
Step 2: [INITIALIZE] SET I = 1
Step 3: Repeat Step 4 while I<=N
Step 4: IF A[I] = VAL
SET POS = I
PRINT POS
Go to Step 6
[END OF IF]
[END OF LOOP]
Step 6: EXIT
SET I = I + 1
Step 5: IF POS = –1
PRINT VALUE IS NOT PRESENT
IN THE ARRAY
[END OF IF]

In Steps 1 and 2 of the algorithm, we initialize the value of POS and I. In Step 3, a while loop is executed that would be executed till I is less than N (total number of elements in the array). In Step 4, a check is made to see if a match is found between the current array element and VAL. If a match is found, then the position of the array element is printed, else the value of I is incremented to match the next element with VAL. However, if all the array elements have been compared with VAL and no match is found, then it means that VAL is not present in the array.

Complexity of Linear Search Algorithm

Linear search executes in O(n) time where n is the number of elements in the array. Obviously, the best case of linear search is when VAL is equal to the first element of the array. In this case, only one comparison will be made. Likewise, the worst case will happen when either VAL is not present in the array or it is equal to the last element of the array. In both the cases, n comparisons will have to be made. However, the performance of the linear search algorithm can be improved by using a sorted array.

EXAMPLE:-

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define size 20 // Added so the size of the array can be altered more easily
int main(int argc, char *argv[]) {
```

```
int arr[size], num, i, n, found = 0, pos = -1;
printf("\n Enter the number of elements in the array : ");
scanf("%d", &n);
printf("\n Enter the elements: ");
for(i=0;i<n;i++)
{
scanf("%d", &arr[i]);
}
printf("\n Enter the number that has to be searched : ");
scanf("%d", &num);
for(i=0;i<n;i++)
{
if(arr[i] == num)
{
found =1;
pos=i;
printf("\n %d is found in the array at position= %d", num,i+1);
break;
}
}
if (found == 0)
printf("\n %d does not exist in the array", num);
return 0;
}
```

2. BINARY SEARCH

Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.

Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the sub array reduces to zero.

Now, let us consider how this mechanism is applied to search for a value in a sorted array. Consider an array A[] that is declared and initialized as int A[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; and the value to be searched is VAL = 9. The algorithm will proceed in the following manner.

BEG = 0, END = 10, MID = (0 + 10)/2 = 5

Now, VAL = 9 and A[MID] = A[5] = 5

A[5] is less than VAL, therefore, we now search for the value in the second half of the array. So, we change the values of BEG and MID.

Now, BEG = MID + 1 = 6, END = 10, MID = (6 + 10)/2 =16/2 = 8

VAL = 9 and A[MID] = A[8] = 8

A[8] is less than VAL, therefore, we now search for the value in the second half of the segment. So, again we change the values of BEG and MID.

Now, BEG = MID + 1 = 9, END = 10, MID = (9 + 10)/2 = 9

Now, VAL = 9 and A[MID] = 9.

Algorithm for binary search:-

BINARY_SEARCH (A, lower_bound, upper_bound, VAL)

Step 1: [INITIALIZE] SET BEG = lower_bound
END = upper_bound, POS = - 1
Step 2: Repeat Steps 3 and 4 while BEG <= END
Step 3: SET MID = (BEG + END)/2
Step 4: IF A[MID] = VAL
SET POS = MID
PRINT POS
Go to Step 6
ELSE IF A[MID] > VAL
SET END = MID - 1
ELSE
SET BEG = MID + 1
[END OF IF]
[END OF LOOP]
Step 5: IF POS = -1
PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
[END OF IF]
Step 6: EXIT

In this algorithm, we see that BEG and END is the beginning and ending positions of the segment that we are looking to search for the element. MID is calculated as (BEG + END)/2. Initially, BEG = lower_bound and END = upper_bound. The algorithm will terminate when A[MID] = VAL. When the algorithm ends, we will set POS = MID. POS is the position at which the value is present in the array.

However, if VAL is not equal to A[MID], then the values of BEG, END, and MID will be changed depending on whether VAL is smaller or greater than A[MID].

(a) If VAL < A[MID], then VAL will be present in the left segment of the array. So, the value of END will be changed as END = MID – 1.

(b) If VAL > A[MID], then VAL will be present in the right segment of the array. So, the value of BEG will be changed as BEG = MID + 1.

Finally, if VAL is not present in the array, then eventually, END will be less than BEG. When this happens, the algorithm will terminate and the search will be unsuccessful.

In Step 1, we initialize the value of variables, BEG, END, and POS. In Step 2, a while loop is executed until BEG is less than or equal to END. In Step 3, the value of MID is calculated. In Step 4, we check if the array value at MID is equal to VAL (item to be searched in the array). If a match is found, then the value of POS is printed and the algorithm exits. However, if a match is not found, and if the value of A[MID] is greater than VAL, the value of END is modified, otherwise if A[MID] is greater than VAL, then the value of BEG is altered. In Step 5, if the value of POS = –1, then VAL is not present in the array and an appropriate message is printed on the screen before the algorithm exits.

Complexity of Binary Search Algorithm

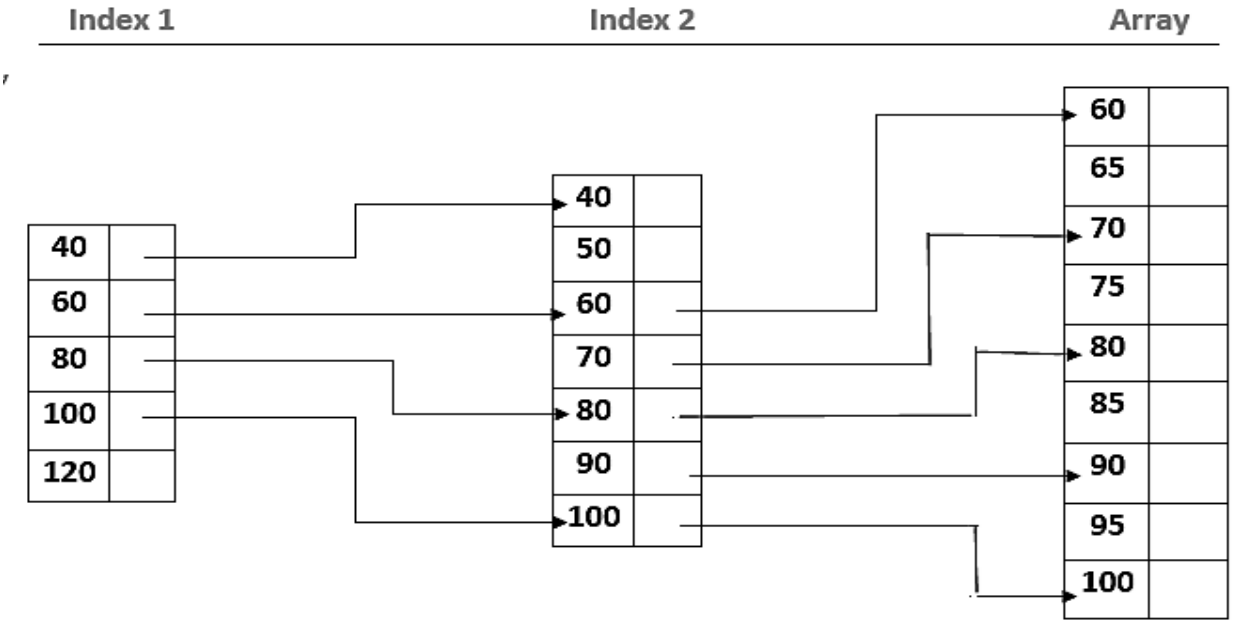
The complexity of the binary search algorithm can be expressed as f(n), where n is the number of elements in the array. The complexity of the algorithm is calculated depending on the number of comparisons that are made. In the binary search algorithm, we see that with each comparison, the size of the segment where search has to be made is reduced to half. Thus, we can say that, in order to locate a particular value in the array, the total number of comparisons that will be made is given as 2f(n) > n or f(n) = log2n.

3. Indexed Sequential Search

In this searching method, first of all, an index file is created, that contains some specific group or division of required record when the index is obtained, then the partial indexing takes less time because it is located in a specified group. When the user makes a request for specific records it will find that index group first where that specific record is recorded.

Characteristics of Indexed Sequential Search:

- In Indexed Sequential Search a sorted index is set aside in addition to the array.
- Each element in the index points to a block of elements in the array or another expanded index.
- The index is searched 1st then the array and guides the search in the array.



Example:-

```
#include <stdio.h>
#include <stdlib.h>
void indexedSequentialSearch(int arr[], int n, int k)
{
        int elements[20], indices[20], temp, i, set = 0;
        int j = 0, ind = 0, start, end;
        for (i = 0; i < n; i += 3) {
                // Storing element
                elements[ind] = arr[i];
                // Storing the index
```

```
            indices[ind] = i;
            ind++;
        }
        if (k < elements[0]) {
            printf("Not found");
            exit(0);
        }
        else {
            for (i = 1; i <= ind; i++)
                if (k <= elements[i]) {
                    start = indices[i - 1];
                    end = indices[i];
                    set = 1;
                    break;
                }
        }
        if (set == 0) {
            start = indices[i - 1];
            end = n;
        }
        for (i = start; i <= end; i++) {
            if (k == arr[i]) {
                j = 1;
                break;
            }
        }
        if (j == 1)
            printf("Found at index %d", i);
        else
            printf("Not found");
}
void main()
{
        int arr[] = { 6, 7, 8, 9, 10 };
        int n = sizeof(arr) / sizeof(arr[0]);
        // Element to search
        int k = 8;
        indexedSequentialSearch(arr, n, k);}
```

GRAPHS

1. INTRODUCTION TO GRAPHS

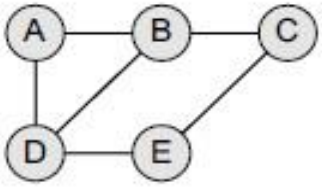
A graph is an abstract data structure that is used to implement the mathematical concept of graphs. It is basically a collection of vertices (also called nodes) and edges that connect these vertices. A graph is often viewed as a generalization of the tree structure, where instead of having a purely parent-to-child relationship between tree nodes, any kind of complex relationship can exist.

Graphs are widely used to model any situation where entities or things are related to each other in pairs. For example, the following information can be represented by graphs:

- *Family trees* in which the member nodes have an edge from parent to each of their children.
- *Transportation networks* in which nodes are airports, intersections, ports, etc. The edges can be airline flights, one-way roads, shipping routes, etc.

Definition of Graphs:-

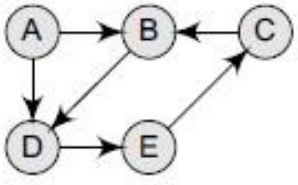
A graph G is defined as an ordered set (V, E), where V(G) represents the set of vertices and E(G) represents the edges that connect these vertices.



Undirected graph

The above Figure shows a graph with V(G) = {A, B, C, D and E} and E(G) = {(A, B), (B, C), (A, D), (B, D), (D, E), (C, E)}. Note that there are five vertices or nodes and six edges in the graph.

A graph can be directed or undirected. In an undirected graph, edges do not have any direction associated with them. That is, if an edge is drawn between nodes A and B, then the nodes can be traversed from A to B as well as from B to A. Figure shows an **undirected graph** because it does not give any information about the direction of the edges.



Directed graph

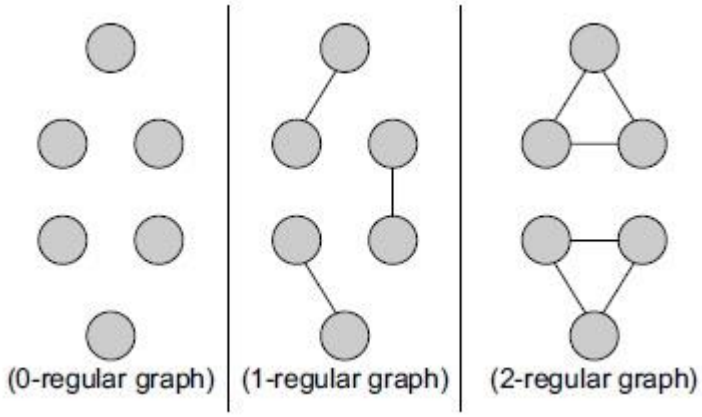
Look at Fig. which shows a directed graph. In a directed graph, edges form an ordered pair. If there is an edge from A to B, then there is a path from A to B but not from B to A. The edge (A, B) is said to initiate from node A (also known as initial node) and terminate at node B (terminal node).

2. Graph Terminology:-

Adjacent nodes or neighbors For every edge, e = (u, v) that connects nodes u and v, the nodes u and v are the end-points and are said to be the adjacent nodes or neighbors.

Degree of a node Degree of a node u, deg(u), is the total number of edges containing the node u. If deg(u) = 0, it means that u does not belong to any edge and such a node is known as an isolated node.

Regular graph It is a graph where each vertex has the same number of neighbors. That is, every node has the same degree. A regular graph with vertices of degree k is called a k–regular graph or a regular graph of degree k.Figure shows regular graphs.



Path A path P written as P = {v0, v1, v2, ..., vn), of length n from a node u to v is defined as a sequence of (n+1) nodes. Here, u = v0, v = vn and vi–1 is adjacent to vi for i = 1, 2, 3, ..., n.

Closed path A path P is known as a closed path if the edge has the same end-points. That is, if v0 = vn.

Simple path A path P is known as a simple path if all the nodes in the path are distinct with an exception that v0 may be equal to vn. If v0 = vn, then the path is called a closed simple path.

Cycle A path in which the first and the last vertices are same. A *simple cycle* has no repeated edges or vertices (except the first and last vertices).

Connected graph A graph is said to be connected if for any two vertices (u, v) in V there is a path from u to v. That is to say that there are no isolated nodes in a connected graph. A connected graph that does not have any cycle is called a tree. Therefore, a tree is treated as a special graph.

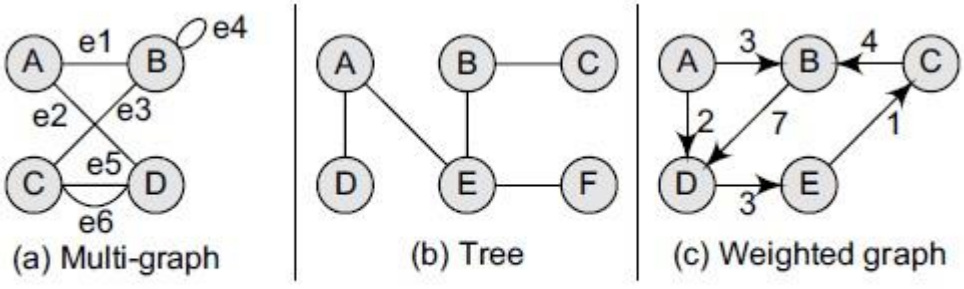


Figure Multi-graph, tree, and weighted graph

Complete graph A graph G is said to be complete if all its nodes are fully connected. That is, there is a path from one node to every other node in the graph. A complete graph has n(n–1)/2 edges, where n is the number of nodes in G.

Clique In an undirected graph G = (V, E), clique is a subset of the vertex set $C \subseteq V$, such that for every two vertices in C, there is an edge that connects two vertices.

Labelled graph or weighted graph A graph is said to be labelled if every edge in the graph is assigned some data. In a weighted graph, the edges of the graph are assigned some weight or length. The weight of an edge denoted by w(e) is a positive value which indicates the cost of traversing the edge. Figure (c) shows a weighted graph.

Multiple edges Distinct edges which connect the same end-points are called multiple edges. That is, e = (u, v) and e' = (u, v) are known as multiple edges of G.

Loop An edge that has identical end-points is called a loop. That is, e = (u, u).

Multi-graph A graph with multiple edges and/or loops is called a multi-graph. Figure (a) shows a multi-graph.

Size of a graph The size of a graph is the total number of edges in it.

3. Graph Representation

By Graph representation, we simply mean the technique which is to be used in order to store some graph into the computer's memory.

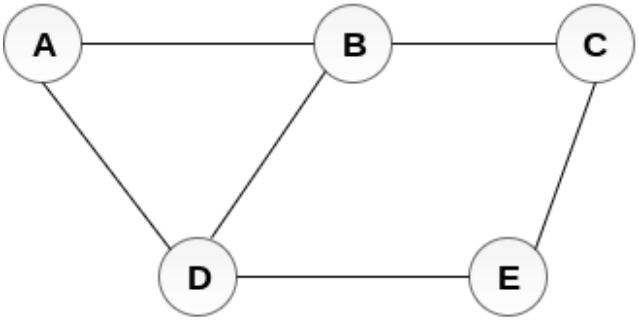
There are two ways to store Graph into the computer's memory. In this part of this tutorial, we discuss each one of them in detail.

1. Sequential Representation of graph:-

In sequential representation, we use adjacency matrix to store the mapping represented by vertices and edges. In adjacency matrix, the rows and columns are represented by the graph vertices. A graph having n vertices will have a dimension **n x n**.

An entry M_{ij} in the adjacency matrix representation of an undirected graph G will be 1 if there exists an edge between V_i and V_j.

An undirected graph and its adjacency matrix representation is shown in the following figure.



Undirected Graph

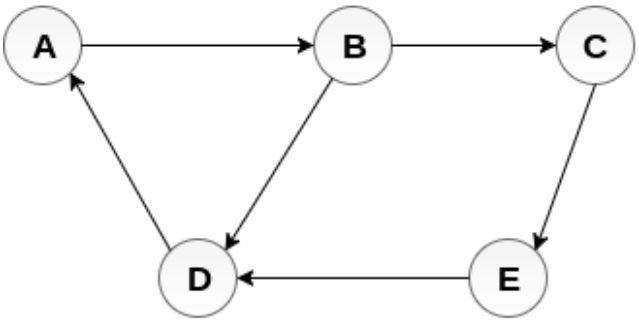
	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	1	0
C	0	1	0	0	1
D	1	1	0	0	1
E	0	0	1	1	0

Adjacency Matrix

In the above figure, we can see the mapping among the vertices (A, B, C, D, E) is represented by using the adjacency matrix which is also shown in the figure.

There exist different adjacency matrices for the directed and undirected graph. In directed graph, an entry A_{ij} will be 1 only when there is an edge directed from V_i to V_j.

A directed graph and its adjacency matrix representation is shown in the following figure.



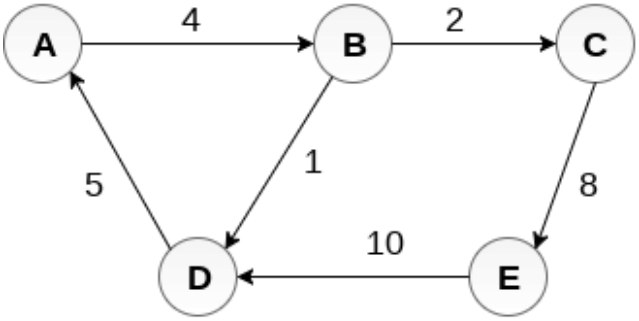
Directed Graph

	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	1	0
C	0	0	0	0	1
D	1	0	0	0	0
E	0	0	0	1	0

Adjacency Matrix

Representation of weighted directed graph is different. Instead of filling the entry by 1, the Non-zero entries of the adjacency matrix are represented by the weight of respective edges.

The weighted directed graph along with the adjacency matrix representation is shown in the following figure.



Weighted Directed Graph

	A	B	C	D	E
A	0	4	0	0	0
B	0	0	2	1	0
C	0	0	0	0	8
D	5	0	0	0	0
E	0	0	0	10	0

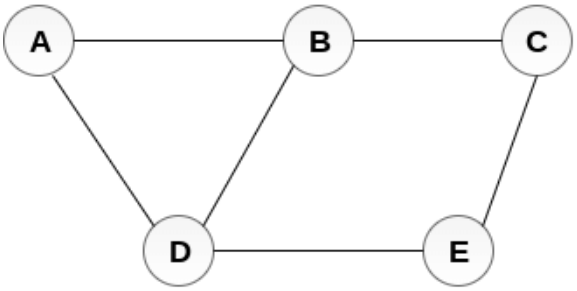
Adjacency Matrix

2. Linked Representation of graphs:-

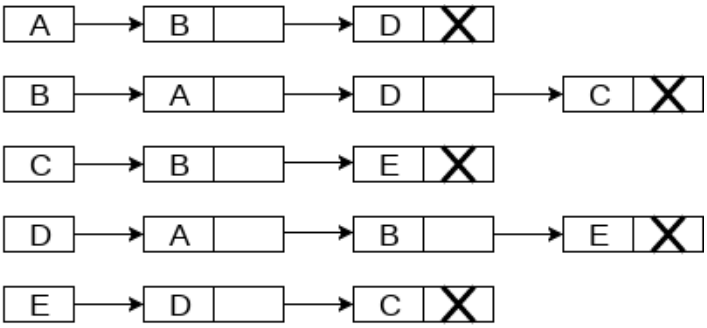
In the linked representation, an adjacency list is used to store the Graph into the computer's memory.

Consider the undirected graph shown in the following figure and check the adjacency list representation.

An adjacency list is maintained for each node present in the graph which stores the node value and a pointer to the next adjacent node to the respective node. If all the adjacent nodes are traversed then store the NULL in the pointer field of last node of the list. The sum of the lengths of adjacency lists is equal to the twice of the number of edges present in an undirected graph.

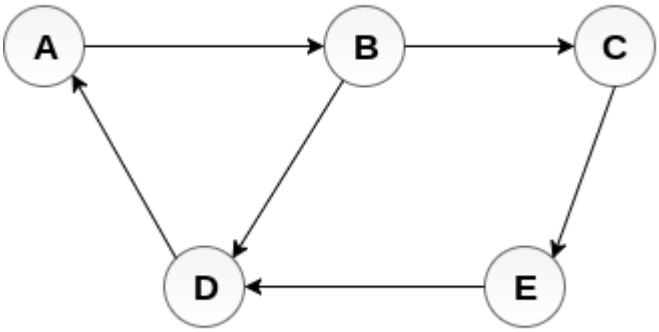


Undirected Graph

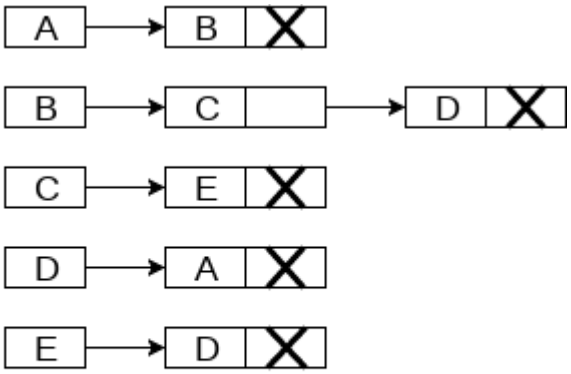


Adjacency List

Consider the directed graph shown in the following figure and check the adjacency list representation of the graph.



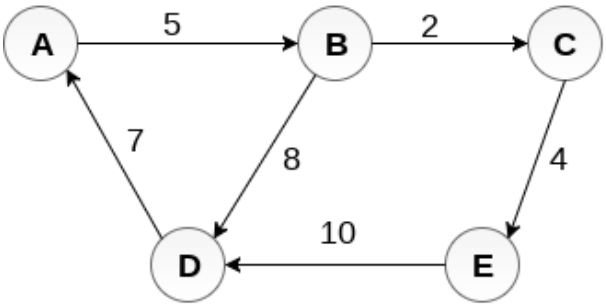
Directed Graph



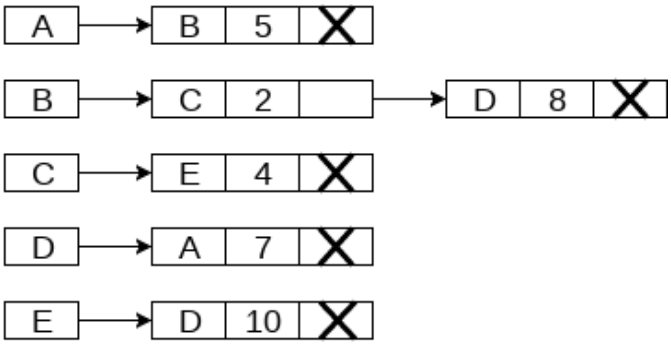
Adjacency List

In a directed graph, the sum of lengths of all the adjacency lists is equal to the number of edges present in the graph.

In the case of weighted directed graph, each node contains an extra field that is called the weight of the node. The adjacency list representation of a directed graph is shown in the following figure.



Weighted Directed Graph



Adjacency List

4. GRAPH TRAVERSAL ALGORITHMS

By traversing a graph, we mean the method of examining the nodes and edges of the graph. There are two standard methods of graph traversal which we will discuss in this section. These two methods are:

1. Breadth-first search
2. Depth-first search

While breadth-first search uses a queue as an auxiliary data structure to store nodes for further processing, the depth-first search scheme uses a stack. But both these algorithms make use of a variable STATUS. During the execution of the algorithm, every node in the graph will have the variable STATUS set to 1 or 2, depending on its current state. Table shows the value of STATUS and its significance.

Value of status and its significance

Status	State of the node	Description
1	Ready	The initial state of the node N
2	Waiting	Node N is placed on the queue or stack and waiting to be processed
3	Processed	Node N has been completely processed

1. BREADTH-FIRST SEARCH ALGORITHM(BFS)

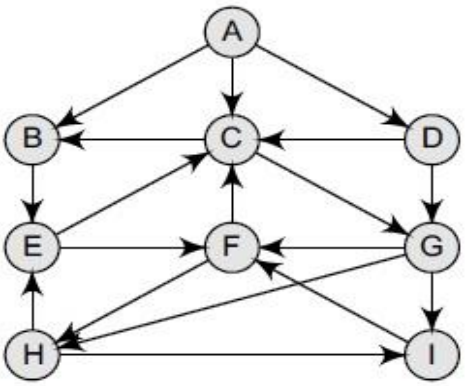
Breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all the neighboring nodes. Then for each of those nearest nodes, the algorithm explores their unexplored neighbor nodes, and so on, until it finds the goal.

Step 1: SET STATUS = 1 (ready state) for each node in G
Step 2: Enqueue the starting node A and set its STATUS = 2 (waiting state)
Step 3: Repeat Steps 4 and 5 until QUEUE is empty
Step 4: Dequeue a node N. Process it and set its STATUS = 3 (processed state).
Step 5: Enqueue all the neighbors of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)
[END OF LOOP]
Step 6: EXIT

That is, we start examining the node A and then all the neighbors of A are examined. In the next step, we examine the neighbors of neighbors of A, so on and so forth. This means that we need to track the neighbors of the node and guarantee that every node in the graph is processed and no node is processed more than once. This is accomplished by using a queue that will hold the nodes that are waiting for further processing and a variable STATUS to represent the current state of the node.

Example:-

Consider the graph G given in Fig. The adjacency list of G is also given. Assume that G represents the daily flights between different cities and we want to fly from city A to I with minimum stops. That is, find the minimum path P from A to I given that every edge has a length of 1.



Adjacency lists

A: B, C, D
B: E
C: B, G
D: C, G
E: C, F
F: C, H
G: F, H, I
H: E, I
I: F

Graph G and its adjacency list

The minimum path P can be found by applying the breadth-first search algorithm that begins at city 'A' and ends when 'I' is encountered. During the execution of the algorithm, we use two arrays: QUEUE and ORIG.

While QUEUE is used to hold the nodes that have to be processed, ORIG is used to keep track of the origin of each edge. Initially, FRONT = REAR = –1.

The algorithm for this is as follows:

(a) Add A to QUEUE and add NULL to ORIG.

FRONT = 0	QUEUE = A
REAR = 0	ORIG = \0

(b) Dequeue a node by setting FRONT = FRONT + 1 (remove the FRONT element of QUEUE) and enqueue the neighbors of A. Also, add A as the ORIG of its neighbors.

FRONT = 1	QUEUE = A	B	C	D
REAR = 3	ORIG = \0	A	A	A

(c) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbors of B. Also, add B as the ORIG of its neighbors.

FRONT = 2	QUEUE = A	B	C	D	E
REAR = 4	ORIG = \0	A	A	A	B

(d) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbors of C. Also, add C as the ORIG of its neighbors. Note that C has two neighbors B and G. Since B has already been added to the queue and it is not in the Ready state, we will not add B and only add G.

FRONT = 3	QUEUE = A	B	C	D	E	G
REAR = 5	ORIG = \0	A	A	A	B	C

(e) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbors of D. Also, add D as the ORIG of its neighbors. Note that D has two neighbors C and G. Since both of them have already been added to the queue and they are not in the Ready state, we will not add them again.

FRONT = 4	QUEUE = A	B	C	D	E	G
REAR = 5	ORIG = \0	A	A	A	B	C

(f) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbors of E. Also, add E as the ORIG of its neighbors. Note that E has two neighbors C and F. Since C has already been added to the queue and it is not in the Ready state, we will not add C and add only F.

FRONT = 5	QUEUE = A	B	C	D	E	G	F
REAR = 6	ORIG = \0	A	A	A	B	C	E

(g) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbors of G. Also, add G as the ORIG of its neighbors. Note that G has three neighbors F, H, and I.

FRONT = 6	QUEUE = A	B	C	D	E	G	F	H	I
REAR = 9	ORIG = \0	A	A	A	B	C	E	G	G

Since F has already been added to the queue, we will only add H and I. As I is our final destination, we stop the execution of this algorithm as soon as it is encountered and added to the QUEUE. Now, backtrack from I using ORIG to find the minimum path P. Thus, we have P as A -> C -> G -> I.

Features of Breadth-First Search Algorithm

Space complexity:-In the breadth-first search algorithm, all the nodes at a particular level must be saved until their child nodes in the next level have been generated. The space complexity is therefore proportional to the number of nodes at the deepest level of the graph. Given a graph with branching factor b (number of children at each node) and depth d, the asymptotic space complexity is the number of nodes at the deepest level $O(b^d)$.

If the number of vertices and edges in the graph are known ahead of time, the space complexity can also be expressed as O (| E | + | V |), where | E | is the total number of edges in G and | V | is the number of nodes or vertices.

Time complexity: - In the worst case, breadth-first search has to traverse through all paths to all possible nodes, thus the time complexity of this algorithm asymptotically approaches O(bd). However, the time complexity can also be expressed as O(| E | + | V |), since every vertex and every edge will be explored in the worst case.

Completeness:-Breadth-first search is said to be a complete algorithm because if there is a solution, breadth-first search will find it regardless of the kind of graph. But in case of an infinite graph where there is no possible solution, it will diverge.

Optimality: - Breadth-first search is optimal for a graph that has edges of equal length, since it always returns the result with the fewest edges between the start node and the goal node. But generally, in real-world applications, we have weighted graphs that have costs associated with each edge, so the goal next to the start does not have to be the cheapest goal available.

Applications of Breadth-First Search Algorithm

Breadth-first search can be used to solve many problems such as:

- Finding all connected components in a graph G.
- Finding all nodes within an individual connected component.
- Finding the shortest path between two nodes, u and v, of an unweighted graph.
- Finding the shortest path between two nodes, u and v, of a weighted graph.

2. Depth-first Search Algorithm(DFS)

The depth-first search algorithm progresses by expanding the starting node of G and then going deeper and deeper until the goal node is found, or until a node that has no children is encountered. When a dead-end is reached, the algorithm backtracks, returning to the most recent node that has not been completely explored.

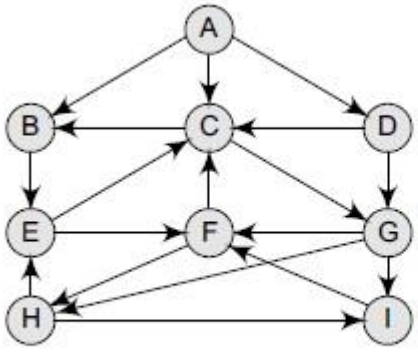
```
Step 1: SET STATUS = 1 (ready state) for each node in G
Step 2: Push the starting node A on the stack and set
its STATUS = 2 (waiting state)
Step 3: Repeat Steps 4 and 5 until STACK is empty
Step 4: Pop the top node N. Process it and set its
STATUS = 3 (processed state)
Step 5: Push on the stack all the neighbours of N that
are in the ready state (whose STATUS = 1) and
set their STATUS = 2 (waiting state)
[END OF LOOP]
Step 6: EXIT
```

In other words, depth-first search begins at a starting node A which becomes the current node. Then, it examines each node N along a path P which begins at A. That is, we process a neighbor of A, then a neighbor of neighbor of A, and so on. During the execution of the algorithm, if we reach a path that has a node N that has already been processed, then we backtrack to the current node. Otherwise, the unvisited (unprocessed) node becomes the current node. The algorithm proceeds like this until we reach a dead-end (end of path P). On reaching the dead-end, we backtrack to find another path P¢. The algorithm terminates when backtracking leads back to the starting node A. In this algorithm, edges that lead to a new vertex are called *discovery edges* and edges that lead to an already visited vertex are called *back edges*.

Observe that this algorithm is similar to the in-order traversal of a binary tree. Its implementation is similar to that of the breadth-first search algorithm but here we use a stack instead of a queue. Again, we use a variable STATUS to represent the current state of the node.

Example:-

Consider the graph G given in Fig. The adjacency list of G is also given.



Adjacency lists

A: B, C, D
B: E
C: B, G
D: C, G
E: C, F
F: C, H
G: F, H, I
H: E, I
I: F

Graph G and its adjacency list

Suppose we want to print all the nodes that can be reached from the node H (including H itself). One alternative is to use a depth-first search of G starting at node H. The procedure can be explained here.

(a) Push H onto the stack.

STACK: H

(b) Pop and print the top element of the STACK, that is, H. Push all the neighbors of H onto the stack that are in the ready state. The STACK now becomes

PRINT: H STACK: E, I

(c) Pop and print the top element of the STACK, that is, I. Push all the neighbors of I onto the stack that are in the ready state. The STACK now becomes

PRINT: I STACK: E, F

(d) Pop and print the top element of the STACK, that is, F. Push all the neighbors of F onto the stack that are in the ready state. (Note F has two neighbors, C and H. But only C will be added, as H is not in the ready state.) The STACK now becomes

PRINT: F STACK: E, C

(e) Pop and print the top element of the STACK, that is, C. Push all the neighbors of C onto the stack that are in the ready state. The STACK now becomes

PRINT: C STACK: E, B, G

(f) Pop and print the top element of the STACK, that is, G. Push all the neighbors of G onto the stack that are in the ready state. Since there are no neighbors of G that are in the ready state, no push operation is performed. The STACK now becomes

PRINT: G STACK: E, B

(g) Pop and print the top element of the STACK, that is, B. Push all the neighbors of B onto the stack that are in the ready state. Since there are no neighbors of B that are in the ready state, no push operation is performed. The STACK now becomes

PRINT: B STACK: E

(h) Pop and print the top element of the STACK, that is, E. push all the neighbors of E onto the stack that are in the ready state. Since there are no neighbors of E that are in the ready state, no push operation is performed. The STACK now becomes empty.

PRINT: E STACK:

Since the STACK is now empty, the depth-first search of G starting at node H is complete and the nodes which were printed are:

H, I, F, C, G, B, E

These are the nodes which are reachable from the node H.

Features of Depth-First Search Algorithm

Space complexity The space complexity of a depth-first search is lower than that of a breadth first search.

Time complexity The time complexity of a depth-first search is proportional to the number of vertices plus the number of edges in the graphs that are traversed. The time complexity can be given as (O(|V| + |E|)).

Completeness Depth-first search is said to be a complete algorithm. If there is a solution, depth first search will find it regardless of the kind of graph. But in case of an infinite graph, where there is no possible solution, it will diverge.

Applications of Depth-First Search Algorithm

Depth-first search is useful for:

- Finding a path between two specified nodes, u and v, of an unweighted graph.
- Finding a path between two specified nodes, u and v, of a weighted graph.
- Finding whether a graph is connected or not.
- Computing the spanning tree of a connected graph.

5. SHORTEST PATH ALGORITHMS

There are three different algorithms to calculate the shortest path between the vertices of a graph G. These algorithms include:

1. Minimum spanning tree
2. Dijkstra's algorithm
3. Warshall's algorithm

While the first two use an adjacency list to find the shortest path, Warshall's algorithm uses an adjacency matrix to do the same.

1. Minimum Spanning Trees (MST)

A spanning tree of a connected, undirected graph G is a sub-graph of G which is a tree that connects all the vertices together. A graph G can have many different spanning trees. We can assign *weights* to each edge (which is a number that represents how unfavorable the edge is), and use it to assign a weight to a spanning tree by calculating the sum of the weights of the edges in that spanning tree.

A ***minimum spanning tree*** **(MST)** is defined as a spanning tree with weight less than or equal to the weight of every other spanning tree. In other words, a minimum spanning tree is a spanning tree that has weights associated with its edges, and the total weight of the tree (the sum of the weights of its edges) is at a minimum.

Properties of minimum spanning tree:-

Possible multiplicity There can be multiple minimum spanning trees of the same weight. Particularly, if all the weights are the same, then every spanning tree will be minimum.

Uniqueness When each edge in the graph is assigned a different weight, then there will be only one unique minimum spanning tree.

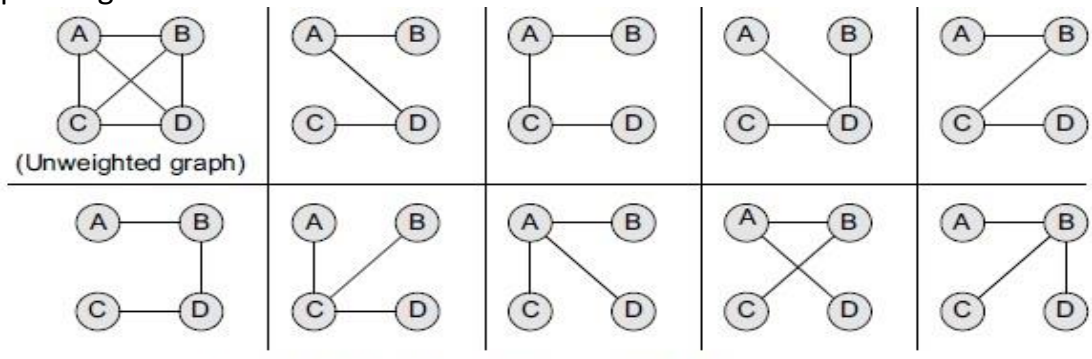
Minimum-cost sub graph: - If the edges of a graph are assigned non-negative weights, then a minimum spanning tree is in fact the minimum-cost sub graph or a tree that connects all vertices.

Cycle property If there exists a cycle C in the graph G that has a weight larger than that of other edges of C, then this edge cannot belong to an MST.

Usefulness Minimum spanning trees can be computed quickly and easily to provide optimal solutions. These trees create a sparse sub graph that reflects a lot about the original graph.

Simplicity The minimum spanning tree of a weighted graph is nothing but a spanning tree of the graph which comprises of n–1 edges of minimum total weight. Note that for an unweighted graph, any spanning tree is a minimum spanning tree.

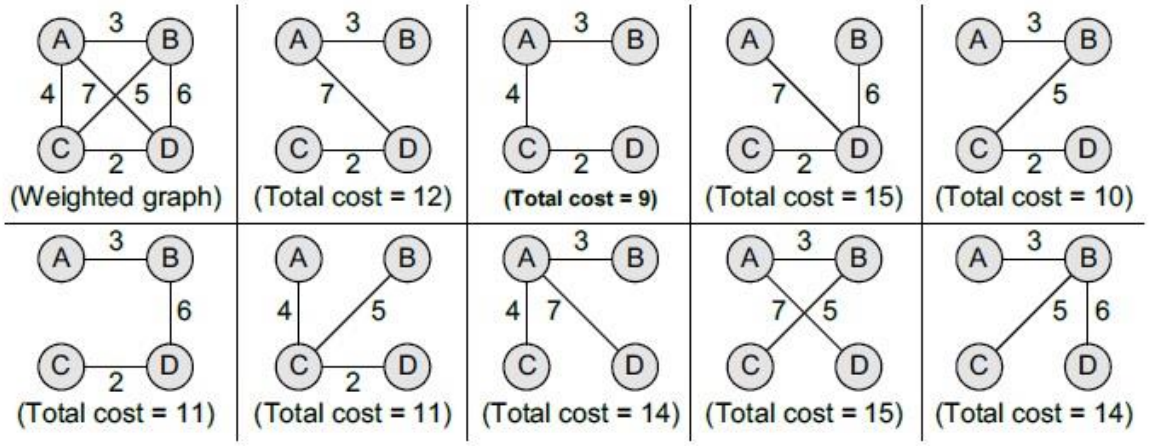
Example1: - Consider an unweighted graph G given below. From G, we can draw many distinct spanning trees. Eight of them are given here. For an unweighted graph, every spanning tree is a minimum spanning tree.



Unweighted graph and its spanning trees

Example2:-

Consider a weighted graph G shown in Fig.



Weighted graph and its spanning trees

From G, we can draw three distinct spanning trees. But only a single minimum spanning tree can be obtained, that is, the one that has the minimum weight (cost) associated with it. Here the minimum cost is 9 that is called the minimum spanning tree, as it has the lowest cost associated with it.

Applications of Minimum Spanning Trees

1. MSTs are widely used for designing networks. For instance, people separated by varying distances wish to be connected together through a telephone network. A minimum spanning tree is used to determine the least costly paths with no cycles in this network, thereby providing a connection that has the minimum cost involved.
2. MSTs are used to find airline routes. While the vertices in the graph denote cities, edges represent the routes between these cities. No doubt, more the distance between the cities, higher will be the amount charged. Therefore, MSTs are used to optimize airline routes by finding the least costly path with no cycles.
3. MSTs are also used to find the cheapest way to connect terminals, such as cities, electronic components or computers via roads, airlines, railways, wires or telephone lines.
4. MSTs are applied in routing algorithms for finding the most efficient path.

2. Dijkstra's Algorithm

Dijkstra's algorithm, given by a Dutch scientist Edsger Dijkstra in 1959, is used to find the shortest path tree. This algorithm is widely used in network routing protocols, most notably IS-IS and OSPF (Open Shortest Path First).

Given a graph G and a source node A, the algorithm is used to find the shortest path (one having the lowest cost) between A (source node) and every other node. Moreover, Dijkstra's algorithm is also used for finding the costs of the shortest paths from a source node to a destination node.

For example, if we draw a graph in which nodes represent the cities and weighted edges represent the driving distances between pairs of cities connected by a direct road, then Dijkstra's algorithm when applied gives the shortest route between one city and all other cities.

Algorithm:-

Dijkstra's algorithm is used to find the length of an optimal path between two nodes in a graph. The term optimal can mean anything, shortest, cheapest, or fastest. If we start the algorithm with an initial node, then the distance of a node Y can be given as the distance from the initial node to that node.

1. Select the source node also called the initial node
2. Define an empty set N that will be used to hold nodes to which a shortest path has been found.
3. Label the initial node with 0, and insert it into N.
4. Repeat Steps 5 to 7 until the destination node is in N or there are no more labeled nodes in N.
5. Consider each node that is not in N and is connected by an edge from the newly inserted node.
6. (a) If the node that is not in N has no label then SET the label of the node = the label of the newly inserted node + the length of the edge.

(b) Else if the node that is not in N was already labeled, then SET its new label = minimum (label of newly inserted vertex + length of edge, old label)

7. Pick a node not in N that has the smallest label assigned to it and add it to N.

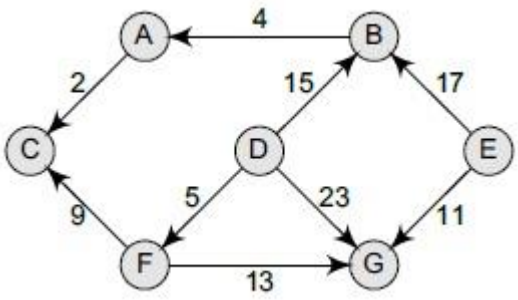
Dijkstra's algorithm labels every node in the graph where the labels represent the distance (cost) from the source node to that node. There are two kinds of labels: *temporary* and *permanent*. Temporary labels are assigned to nodes that have not been reached, while permanent labels are given to nodes that have been reached and their distance (cost) to the source node is known. A node must be a permanent label or a temporary label, but not both.

The execution of this algorithm will produce either of the following two results:

1. If the destination node is labeled, then the label will in turn represent the distance from the source node to the destination node.
2. If the destination node is not labeled, then there is no path from the source to the destination node.

Example

Consider the graph G given in Fig.



Taking D as the initial node, execute the Dijkstra's algorithm on it.

Step 1: Set the label of D = 0 and N = {D}.

Step 2: Label of D = 0, B = 15, G = 23, and F = 5. Therefore, N = {D, F}.

Step 3: Label of D = 0, B = 15, G has been re-labeled 18 because minimum (5 + 13, 23) = 18, C has been re-labeled 14 (5 + 9). Therefore, N = {D, F, C}.

Step 4: Label of D = 0, B = 15, G = 18. Therefore, N = {D, F, C, B}.

Step 5: Label of D = 0, B = 15, G = 18 and A = 19 (15 + 4). Therefore, N = {D, F, C, B, G}.

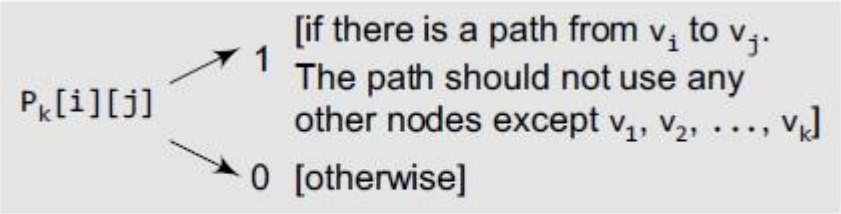
Step 6: Label of D = 0 and A = 19. Therefore, N = {D, F, C, B, G, A}

Note that we have no labels for node E; this means that E is not reachable from D. Only the nodes that are in N are reachable from B.

The running time of Dijkstra's algorithm can be given as O(|V|2+|E|)=O(|V|2) where V is the set of vertices and E in the graph.

3. Warshall's Algorithm

If a graph G is given as G=(V, E), where V is the set of vertices and E is the set of edges, the path matrix of G can be found as, $P = A + A^2 + A^3 + ... + A^n$. This is a lengthy process, so Warshall has given a very efficient algorithm to calculate the path matrix. Warshall's algorithm defines matrices $P_0, P_1, P_2 \ldots\ldots P_n$ as given in below.



Path matrix entry

This means that if $P_0[i][j] = 1$, then there exists an edge from node vi to vj.

If $P_1[i][j] = 1$, then there exists an edge from v_i to v_j that does not use any other vertex except v_1.

If $P_2[i][j] = 1$, then there exists an edge from v_i to v_j that does not use any other vertex except v_1 and v_2.

Note that P_0 is equal to the adjacency matrix of G. If G contains n nodes, then $P_n = P$ which is the path matrix of the graph G.

From the above discussion, we can conclude that $P_k[i][j]$ is equal to 1 only when either of the two following cases occur:

There is a path from v_i to v_j that does not use any other node except $v_1, v_2, ..., v_{k-1}$. Therefore, **$P_{k-1}[i][j] = 1$.**

There is a path from v_i to v_k and a path from v_k to v_j where all the nodes use $v_1, v_2, ..., v_{k-1}$. Therefore,

$P_{k-1}[i][k] = 1$ AND $P_{k-1}[k][j] = 1$

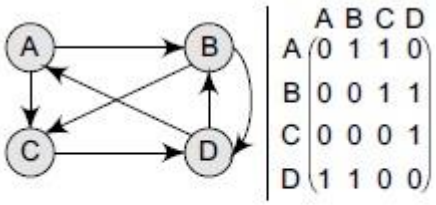
Hence, the path matrix P_n can be calculated with the formula given as:

$P_k[i][j] = P_{k-1}[i][j] \cup (P_{k-1}[i][k] \cap P_{k-1}[k][j])$ where U indicates logical OR operation and ∩ indicates logical AND operation.

Below shows the Warshall's algorithm to find the path matrix P using the adjacency matrix A.

```
Step 1: [ the Path Matrix] Repeat Step 2 for I = to n-1, where n is the
number of nodes in the graph
Step 2: Repeat Step 3 for J = to n-1
Step 3: IF A[I][J] = , then SET P[I][J] =
ELSE P[I][J] = 1
[END OF LOOP]
[END OF LOOP]
Step 4: [Calculate the path matrix P] Repeat Step 5 for K = to n-1
Step 5: Repeat Step 6 for I = to n-1
Step 6: Repeat Step 7 for J= to n-1
Step 7: SET Pk[i][j] = Pk-1[i][j] U (Pk-1[i][k] ∩ Pk-1[k][j])
Step 8: EXIT
```

Example Consider the graph in Fig. and its adjacency matrix A



	A	B	C	D
A	0	1	1	0
B	0	0	1	1
C	0	0	0	1
D	1	1	0	0

Graph G and its path matrix P

We can straightaway calculate the path matrix P using the Warshall's algorithm.
The path matrix P can be given in a single step as:

$$P = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \end{array}$$

Thus, we see that calculating A, A2, A3, A4, ..., A5 to calculate P is a very slow and inefficient technique as compared to the Warshall's technique.

6. APPLICATIONS OF GRAPHS

Graphs are constructed for various types of applications such as:

- In circuit networks where points of connection are drawn as vertices and component wires become the edges of the graph.
- In transport networks where stations are drawn as vertices and routes become the edges of the graph.
- In maps that draw cities/states/regions as vertices and adjacency relations as edges.
- In program flow analysis where procedures or modules are treated as vertices and calls to these procedures are drawn as edges of the graph.
- Once we have a graph of a particular concept, they can be easily used for finding shortest paths, project planning, etc.
- In flowcharts or control-flow graphs, the statements and conditions in a program are represented as nodes and the flow of control is represented by the edges.
- In state transition diagrams, the nodes are used to represent states and the edges represent legal moves from one state to the other.
- Graphs are also used to draw activity network diagrams. These diagrams are extensively used as a project management tool to represent the interdependent relationships between groups, steps, and tasks that have a significant impact on the project.

**********THE END**********