LAB MANUAL OF

ANALYSIS OF ALGORITHM

LIST OF PRACTICALS

1. To Analyze time complexity of Bubble, Selection and Insertion sort.
2. To Analyze time complexity of Linear and Binary search.
3. To Analyze time complexity of Quick and Randomized Quick sort.
4. To Analyze time complexity of Merge sort.
5. To Implement Minmax Algorithm using Divide & Conquer.
6. To Implement Fractional Knapsack.
7. To Implement Dijkstra's Algorithm.
8. To Implement Prim's Algorithm.
9. To Implement 0/1 Knapsack.
10. To Implement Bellman Ford's Algorithm.
11. To Implement Floyd Warshall's Algorithm.
12. To Implement Optimal Binary Search Tree.
13. To Implement N-Queen.
14. To Implement Sum of Subsets.
15. To Implement Graph Coloring.
16. To Implement Longest Common Subsequence.

.

Best, Worst, and Average Case defintion

The ***worst case complexity*** of the algorithm is the function defined as the maximum number of steps (operations) done on any instance of size n to give desired output.

The ***best case complexity*** of the algorithm is the function defined by the minimum number of steps taken on any instance of size n.

The ***average-case complexity*** of the algorithm is the function defined by an average number of steps taken on any instance of size n.

Each of these complexities defines a numerical function -
time vs. no of

Experiment no: 1

Aim: *To Analyze time complexity of Bubble, Selection and Insertion sort.*

CALCULATION OF COMPLEXITY :

1.a) Bubble Sort

Bubble sort, sometimes referred to as **sinking sort**, is a simple sorting algorithm, that repeatedly steps through the list to be sorted, compares each pair of adjacent items and them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm, which is a comparison sort, is named for the way smaller elements "bubble" to the top of the list. Although the algorithm is simple, it is too slow and impractical for most problems even when compared to insertion sort . It can be practical if the input is usually in sort order but may occasionally have some out-of-order elements nearly in position.

Analysis of Bubble Sort:

Code:

```
for(pass=1;pass<=n-1;pas++)
{
        for(i=0;i<n-pass;i++)
{
        if(a[i]>a[i+1])
        {
                swap a[i] & a[i+1]
                }
        }
}
```

- For all cases , the complexity of Bubble sort is same .
- To derive the complexity

f(n) = $\sum_{pass=1}^{n-1} \sum_{i=0}^{n-pass-1} 1$

= $\sum_{pass=1}^{n-1} (n - pass - 1 - 0 + 1)$

= $\sum_{pass=1}^{n-1} (n - pass)$

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

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

$= \frac{(n-1)\,(n)}{2}$

$= \frac{n^2}{2} - \frac{n}{2}$

∴ By ignoring the lower order terms and the constant co-efficient of higher order terms, we get

f(n) = O(n^2)

Thus , the Complexity of Bubble sort is quadratic.

1.b) Modified Bubble Sort

The standard bubble sort can be made more efficient, this improved sorting technique is called the modified bubble sort. If we make a complete pass of the array elements and do not need to swap any items, then we know the array must be in order and we can stop the sort. Also, at the end of each pass ,one more item is in the correct position (starting at the end), so the list to be sorted can be shortened by one.

To accomplish this, we must use a flag variable that indicates if any swaps were done. The flag will be set on at the beginning of a pass and when a swap is performed, it will be set off. If at the end of a pass, the flag is still on, then we know no swaps took place and we can terminate the sort.

We will let our flag variable be called 'exchanged'. If we make it Boolean then it can only take the values 'true' or 'false'.

Analysis of Modified Bubble Sort:

Code:

```
int exchanged=1;

for(pass=1;pass<=n-1 && exchanged==1; pass++)

{
        exchanged=0;
for(i=0 ; i<n-pass ; i++)
{
                        if(a[i] > a[i+1])
{
        exchanged=1;
                        swap a[i] & a[i+1]
```

```
                    }
        }

}
```

- ***Best Case :***

It happens when the array is already sorted.

e.g.

1	2	3	4	5

There will be 1 pass but there are 4 comparisons i.e. in 1 pass (n-1) comparisons

∴ By ignoring the lower order terms and the constant co-efficient of higher order terms, we get

$f(n)=\Omega(n)$

Thus , the Best Case complexity is linear.

- ***Worst Case :***

In happens when the array is in reverse order.

e.g.

5	4	3	2	1

In this case , Modified Bubble sort works as Bubble sort.

∴ (n-1) passes takes place.

Therefore,

Passes	**Comparisons**
1	(n-1)
2	(n-2)
.	.
.	.
.	.
(n-1)	1

f(n) = 1 + . . . + (n-2) + (n-1)

$= \frac{(n-1)\,(n)}{2}$

$= \frac{n^2}{2} - \frac{n}{2}$

∴ By ignoring the lower order terms and the constant co-efficient of higher order terms, we get

f(n) = O(n^2)

Thus , the Worst Case complexity is quadratic.

<u>Average Case</u> :

Here, we have to consider all the input cases.

Let c(i) denote Array gets sorted after pass i.

Therefore,

c(i)	**Comparisons**
c(1)	(n-1)
c(2)	(n-1) + (n-2)
.	.
.	.
.	.
(n-1)	(n-1) + (n-2)+ . . .+ 1

∴ c(i) = $\sum_{j=1}^{i} n - j$

= $\sum_{j=1}^{i} n$ - $\sum_{j=1}^{i} j$

$= n*i - \frac{(i+1)\,(i)}{2}$

Assuming that all classes are occurring equally likely the time taken for all operations is

$$= \frac{c(1)}{n-1} + \ldots + \frac{c(n-1)}{n-1}$$

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

$$= \frac{1}{n-1} \sum_{i=1}^{n-1} (\mathrm{n} * \mathrm{i} - \frac{(i+1)\,(i)}{2})$$

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

$$= \frac{n^2}{2} - \frac{1}{2\,(n-1)} \frac{(n-1)n\,(2n-1)}{6} - \frac{1}{2\,(n-1)} \frac{(n-1)n}{2}$$

$$= \frac{n^2}{2} - \frac{2n^2-n}{2} - \frac{n}{4}$$

∴ By ignoring the lower order terms and the constant co-efficient of higher order terms, we get

f(n) = $\Theta(n^2)$

Thus , the Average Case complexity is quadratic.

2.)Selection Sort :

In computer science, **selection sort** is a sorting algorithm, specifically an in-place algorithm comparison sort. Selection sort is noted for its simplicity, and it has performance advantages over more complicated algorithms in certain situations, particularly where auxiliary memory is limited.

The algorithm divides the input list into two parts: the sublist of items already sorted, which is built up from left to right at the front (left) of the list, and the sublist of items remaining to be sorted that occupy the rest of the list. Initially, the sorted sublist is empty and the unsorted sublist is the entire input list. The algorithm proceeds by finding the smallest (or largest, depending on sorting order) element in the unsorted sublist, exchanging it with the leftmost unsorted element (putting it in sorted order), and moving the sublist boundaries one element to the right.

Analysis of Selection Sort:

- For all cases , the complexity of Selection sort is same .
- To derive the complexity

f(n) = $\sum_{pass=1}^{n-1} \sum_{i=pass}^{n-1} 1$

= $\sum_{pass=1}^{n-1} (n - 1 - pass + 1)$

= $\sum_{pass=1}^{n-1} (n - pass)$

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

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

= $\frac{(n-1)\,(n)}{2}$

= $\frac{n^2}{2} - \frac{n}{2}$

∴ By ignoring the lower order terms and the constant co-efficient of higher order terms, we get

f(n) = O(n^2)
Thus , the Complexity of Selection sort is quadratic.

3.)Insertion Sort :

Insertion sort is a simple sorting algorithm that is relatively efficient for small lists and mostly -sorted lists, and often is used as part of more sophisticated algorithms. It works by taking elements from the list one by one and inserting them in their correct position into new sorted list. In arrays, the new list and the remaining elements can share the array's space, but insertion is expensive, requiring shifting all following elements over by one. The insertion sort works just like its name suggests -it inserts each item into its proper place in the final list. The simplest implementation of this requires two list structures -the source list and the list into which sorted items are inserted. To save memory, most implementations use an in-place sort that works by moving the current item past the already sorted items and repeatedly swapping it with the preceding item until it is in lace. Shell sort a variant of insertion sort that is more efficient for larger lists.

Analysis of Insertion Sort:

- ***Best Case :***

It happens when the array is already sorted.

e.g.

1	2	3	4	5

For every pass there will b 1 comparison.

Passes	**Comparisons**
1	1

2	1
.	.
.	.
.	.
(n-1)	1

f(n) = $\sum_{i=1}^{n-1} 1$

=(n-1)

∴ By ignoring the lower order terms, we get

f(n)=Ω(n)

Thus , the Best Case complexity is linear

- ***<u>Worst Case</u> :***

In happens when the array is in reverse order.

e.g.

5	4	3	2	1

Passes	**Comparisons**
1	1
2	2
.	.
.	.
.	.
(n-1)	(n-1)

f(n) = 1 + . . . + (n-2) + (n-1)

f(n) = $\sum_{i=1}^{n-1} i$

= $\frac{(n-1)\,(n)}{2}$

= $\frac{n^2}{2} - \frac{n}{2}$

∴ By ignoring the lower order terms and the constant co-efficient of higher order terms, we get

f(n) = O(n^2)

Thus , the Worst Case complexity is quadratic.

<u>Average Case</u> :

Here, we consider that each element is inserted half-way in order

∴ Complexity is given as

Passes	**Comparisons**
1	1
2	2
.	.
.	.
.	.
(n-1)	(n-1)

f(n) = $\frac{1}{2}$ [1 + . . . + (n-2) + (n-1)]

f(n) = $\frac{1}{2}$ [$\sum_{i=1}^{n-1} i$]

=$\frac{1}{2}$ [$\frac{(n-1)\,(n)}{2}$]

=$\frac{1}{2}$ [$\frac{n^2}{2}$ - $\frac{n}{2}$]

= $\frac{n^2}{4}$ - $\frac{n}{4}$

∴ By ignoring the lower order terms and the constant co-efficient of higher order terms, we get

f(n) = Θ(n^2)

Thus , the Average Case complexity is quadratic.

Experiment no: 2

Aim: To Analyze time complexity of Linear and Binary search

CALCULATION OF COMPLEXITY :

1.)Linear Search:

Linear search is the simplest search algorithm ; it is a special case of brute force search.

In computer science , **linear search** or **sequential search** is a method for finding a particular value in a list that checks each element in sequence until the desired element is found or the list is exhausted. The list need not be ordered.

Assume that we have an array that stores numbers .
Assume there are n numbers.

Analysis of Linear Search:

- ***Best Case :***

It happens when the element to be searched is found at 1st position.

e.g.

11	26	37	46	54

∴ There will be only 1 comparison.

Hence ,

$f(n)=\Omega(1)$

Thus , the Best Case complexity is constant.

- ***Worst Case :***

It happens when the element to be searched is found at last position or not found at all.

e.g.

54	26	37	46	**11**

OR

54	26	37	46	23

∴ There will be (n-1) comparisons.

Hence ,

f(n)=O(n)

Thus , the Worst Case complexity is linear.

Average Case :

In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed (including the case of x not being present in array). So we sum all the cases and divide the sum by (n+1). ∴ complexity is given as

$$f(n) = \frac{\sum_{i=1}^{n+1} \Theta(i)}{(n+1)}$$

$$= \frac{\Theta((n+1)*(n+2))}{(n+1)}$$

$$= \Theta(n)$$

Thus , the Average Case complexity is linear.

2.)Binary Search:

In computer science, a **binary search** or **half-interval search** algorithm finds the position of a specified input value (the search "key") within an sorted array.
For binary search, the array should be arranged in ascending or descending order. In each step, the algorithm compares the search key value with the key value of the middle element of the array. If the keys match, then a matching element has been found and its index, or position, is returned. Otherwise, if the search key is less than the middle element's key, then the algorithm repeats its action on the sub-array to the left of the middle element or, if the search key is greater, on the sub-array to the right. If the remaining array to be searched is empty, then the key cannot be found in the array and a special "not found" indication is returned.

a.) Recursive Binary Search:

A straightforward implementation of binary search is recursion. The initial call uses the indices of the entire array to be searched. The procedure then calculates an index midway between the two indices, determines which of the two subarrays to search, and then does a recursive call to search that subarray.

Analysis of Recursive Binary Search :

Code:

```
int rec_binary_search(int a[],int low,int high,int x)
{
int mid;
if(low==high)
        {
                if(a[low]==x)
                return low;
                else
                return -1;
        }
        else
        {
                 mid=(low+high)/2;
                if(x==a[mid])
                {
                         return mid;
                }
                else if(x<a[mid])
                {
                        return (rec_binary_search (a,low,mid-1,x));
                }
                else if(x>a[mid])
                {
                        return (rec_binary_search (a,mid+1,high,x));
                }
  }
}
```

Let T(n) denote the time required to search an element in an array of size n.

∴ By Master Method,

T(n)=c . . . n=1

T(n)=T($\frac{n}{2}$) + 1

Here, a=1

b=2

f(n)=1

Now, we need to calculate,

$$n^{log_a b} = n^{log_2 1}$$

$$= n^0$$
$$= 1$$

Therefore,

$$n^{log_a b} = f(n)$$

T(n) = Θ($f(n)\ log_2 n$)

= Θ($log_2 n$)

Thus, the complexity of recursive Binary search is logarithmic.

a.) Non-Recursive Binary Search:

The binary search algorithm can also be expressed iteratively with two index limits that progressively narrow the search range

Analysis of Non-Recursive Binary Search :

Code:

```
int non_rec_binary_search(int a[],int low,int high,int x)
{
while(low<=high)
        {
                mid=(low+high)/2;
                if(x==a[mid])
                        break;
                if(x<a[mid])
                        high=mid-1;
                else
                        low=mid+1;
        }
        if(low<=high)
                printf("Element found at position:%d\n",mid+1);
        else
                printf("Element not found.");
}
```

The complexity of Non-Recursive binary search is

f(n)= O(1)

i.e. the complexity is constant.

Experiment no: 3

Aim: To Analyze time complexity of Quicksort and Randomized Quicksort.

CALCULATION OF COMPLEXITY :

1.)Quicksort:

Quicksort is a divide and conquer algorithm which relies on partition operation: to partition an array,an element, called a pivot is choosen,all smaller elements are moved before the pivot, and all greater elements are moved after it. This can be done efficiently in linear time and in-place. Then recursively sorting can be done for the lesser and greater sublists. Efficient implementations of quicksort (with in-place partitioning) are typically unstable sorts and somewhat complex, but are among the fastest sorting algorithms in practice. Together with its modest O(log n) space usage, this makes quicksort one of the most popular sorting algorithms, available in many standard libraries. The most complex issue in quicksort is choosing a good pivot element; consistently poor choices of pivots can result in drastically slower performance

Quicksort sorts by employing a divide and conquer strategy to divide a list into two sub-lists.Pick an element, called a pivot, from the list. Reorder the list so that all elements which are less than pivot come before the pivot and so that all elements greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation. Recursively sort the sub-list of lesser elements and the sub-list of greater elements

Analysis of Quick Sort:

Code :

```
int partition(int a[],int lower,int high)
{
   down = lower + 1
   up = high ;
   i = a[lower] ;
   while ( up>down )
   {
      while ( a[down] < i && down < high  )
 down++ ;
      while ( a[up] > i && up>lower )
 up-- ;
      if ( up > down )
     {
         swap a[down] & a[up]
      }
   }
```

```
    a[lower] = a[up] ;
    a[up] = i;
    return up ;
}

void quicksort(int a[],int low,int high)
{
        if(low<high)
        {
int pos=partition(a,low,high);                  quicksort(a,low,pos-1);
                quicksort(a,pos+1,high);
        }
}
```

Let T(n) denote the time required to sort an array of size n.
∴,
T(n)= 1 . . . n=1
T(n)= Time required to for Partition +
Time required to sort Left Sub-Array(LSA) +
Time required to sort Right Sub-Array(RSA)

= Time required to for Partition +T(i)+T(n-1-i)

Here, i = Size of LSA
n-1-i=size of RSA

Time for partitioning:
The inner part of the outer while loop will get executed for (n+0 OR 0+n)+c
∴ Complexity of inner code is O(n)
The outer loop is executed for constant no. of times
∴ Total complexity of partition method is
f(n)=(n*c)+c
∴ It is going to be a Linear time complexity.
∴ Final Recurrence Relation for Quicksort is
T(n)= 1 . . . n=1

T(n)= T(n) +T(i)+T(n-1-i) otherwise

- ***Best Case :***

It happens when the pivot is placed in the middle of array.

In every partition, we get 2-sub arrays of size $\frac{n}{2}$ approximately.

∴ The Recurrence Relation is given as

T(n)=2 T($\frac{n}{2}$) + n

∴ By Master Method,

Here, a=2

b=2

f(n)=n

Now, we need to calculate,

$$n^{log_a b} = n^{log_2 2} = n^1 = n$$

Therefore,

$$n^{log_a b} = f(n)$$

T(n) = Θ($f(n)\ log_2 n$)

= Θ($n\ log_2 n$)

i.e. T(n)=Ω($n\ log_2 n$)

Thus, the Best Case complexity of Quicksort is linear logarithmic.

- ***<u>Worst Case</u> :***

It happens when the given array is already sorted

e.g.

1	2	3	4	5

OR

5	4	3	2	1

∴ Considering given data is in ascending order

Pivot element (i.e. 1) is placed in same position

∴ LSA Size is 0 and RSA size is (n-1)

Now,Considering given data is in descending order

After placing pivot at proper position, we get

1	4	3	2	5

∴ LSA Size is (n-1) and RSA size is 0

Recurrence Relation for Quicksort is

T(n)= 1 . . . n=1

T(n)= T(n) +T(i)+T(n-1-i) otherwise

∴ T(n)= T(n-1)+n

By Iterative Method,

T(n)= T(n-1)+n

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

= T(n-3)+ (n-2)+ (n-1)+n

.

.

.

=T(n-k)+(n-k+1)+ (n-k+2)+. . . +n

But T(n-k)=T(1)

∴ n-k=1

∴ k=n-1

= T(n-(n-1))+(n-(n-1)+)+. . .+n

=T(1)+1+2+3+. . .+n

=1+ $\sum_{i=1}^{n} i$

= $1 + \frac{(n)\,(n+1)}{2}$

= $\frac{n^2}{2} + \frac{n}{2} + 1$

∴ By ignoring the lower order terms and the constant co-efficient of higher order terms, we get

T(n) = O(n^2)

Thus , the Worst Case complexity is quadratic.

<u>Average Case</u> :

After partitioning the size of LSA can be 0 or 1 or . . .(n-1)

The probability of occurring this size is $\frac{1}{n}$

The average time required to sort LSA is given as

$$\frac{1}{n}\sum_{i=0}^{n-1} T(i)$$

Here, i=Size of LSA

Similarly, the average time required to sort RSA is given as

$$\frac{1}{n}\sum_{i=0}^{n-1} T(i)$$

Here, i=Size of RSA

∴ Total time required to sot an array of size n is given as

T(n)= $\frac{2}{n}\sum_{i=0}^{n-1} T(i)$ + n

Multiplying both the sides by n , we get

n T(n)= 2 $\sum_{i=0}^{n-1} T(i) + n^2$ . . .(1)

Replacing n by (n-1) & rewriting the equation , we get

(n-1) T(n-1)= 2 $\sum_{i=0}^{n-2} T(i) + (n-1)^2$ . . .(2)

Subtracting equation (2) from equation (1),we get

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

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

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

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

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

Dividing both sides by n(n+1),we get

$\therefore \frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2}{n+1}$

$= \frac{T(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1}$

$= \frac{T(n-3)}{n-2} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1}$

.

.

.

$= \frac{T(n-k)}{[n-(k-1)]} + 2[\frac{1}{[n-(k-2)]} + \frac{1}{[n-(k-1)]} + \ldots + \frac{1}{n+1}]$

But T(n-k)=T(1)

∴ n-k=1

∴ k=n-1

$= \frac{T(1)}{2} + \frac{2}{3} + \ldots + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1}$

$= \frac{1}{2} + 2[\frac{1}{3} + \ldots + \frac{1}{n-1} + \frac{1}{n} + \frac{1}{n+1}]$

$= \frac{1}{2} + 2 \sum_{i=3}^{n+1} \frac{1}{i}$

$\frac{T(n)}{n+1} = \frac{1}{2} + 2 \ (\ log_2 n\ )$

∴ T(n) = 2 ($log_2 n$) (n+1)

∴ By ignoring the lower order terms and the constant co-efficient of higher order terms, we get

∴ T(n) = Θ($n log_2 n$)

Thus , the Average Case complexity is linear logarithmic.

2.)Randomized Quicksort:

Randomized Quick Sort is an extension of Quick Sort in which pivot element is chosen randomly.

Randomized quicksort has the desirable property that, for any input, it requires only $O(nlog_2 n)$ expected value time (averaged over all choices of pivots)

Experiment no: 4

Aim: To Analyze time complexity of Merge sort

CALCULATION OF COMPLEXITY :

Merge Sort:

Merge sort takes advantage of the ease of merging already sorted lists into a new sorted list. It starts by comparing every two elements (i.e. 1 with 2, then 3 with 4...) and swapping them if the first should come after the second. It when merges each of the resulting lists of two into lists of four, then merges those lists of four, and so on; until at last two lists are merged into the final sorted list. Of the algorithms described here, this is the first that scales well to very large lists.

Merge sort works as follows:
1. Divide the unsorted list into two sub lists of about half the size
2. Sort each of the two sub lists
3. Merge the two sorted sub lists back into one sorted list.

Analysis of Merge Sort:

Pseudocode :

```
mergesort(m)

var list left, right
if length(m) ≤ 1
return m
else
middle = length(m) / 2
for each x in m up to middle
add x to left
for each x in m after middle
add x to right
left = mergesort(left)
right = mergesort(right)
result = merge(left, right)
return result
```

There are several variants for the merge() function, the simplest variant could look like this:

```
merge(left,right)

var list result
```

while length(left) > 0 and length(right) > 0
if first(left) ≤ first(right)
append first(left) to result
left = rest(left)
else
append first(right) to result
right = rest(right)
if length(left) > 0
append left to result
if length(right) > 0
append right to result
return result

The Recurrence Realtion for mergesort is

T(n) = 1 . . . if(n=1)

T(n) = 2 $\frac{T(n)}{2}$ **+ n otherwise**

∴ By Master method,
Here, a=2,
b=2
f(n)=n

Now, we need to calculate,

$$n^{log_a b}$$
$$= n^{log_2 2}$$
$$= n^1$$
$$= n$$

Therefore,

$$n^{log_a b} = f(n)$$

T(n) = Θ($f(n)\ log_2 n$)

= Θ($n\ log_2 n$)

i.e. T(n)=Ω($n\ log_2 n$)

Thus, the complexity of Merge Sort is linear logarithmic.

Experiment no: 5

Aim : To Implement Minmax Algorithm using Divide and Conquer approach.

Theory or the Logic :

This problem is solved using Divide and Conquer

Let a be an array of n-elements
Split the array into 2 parts such that half elements in 1 sub array and half elements in another sub array
i.e. a[0] . . . a[(n-1)/2] and a[((n-1)/2)+1] . . . a[n-1]
Now, find largest & smallest of both the sub-arrays separately
However ,the sub-array can be further spilt until we get a sub array whose largest and smallest can be found directly without further splitting
i.e. if the array size is 2 or 1 , we can directly get the ans.

Algorithm:

- *Start*
- *Declare variables maxl, maxr, minl, minr and mid*
- ***If*** *low=high*

then *min ← a[low]*
max ← a[low]

- ***else if*** *low+1=high*
  then if *a[low] > a[high])*
  then *max ← a[low]*
  min ← a[high]
  else
  max ← a[high]
  min ← a[low]
- ***else***
  mid ← (low+high)/2
  minmax(a,low,mid,&minl,&maxl)
  minmax(a,mid+1,high,&minr,&maxr)
  if *minl>minr*
  then *min ← minr*
  else
  min ← minl
  if *maxl>maxr*
  then *max ← maxl*
  else
  max ← maxr
- *End*

Complexity: The Complexity of Minmax problem is Θ(n).

Experiment no: 6

Aim: To Implement Fractional Knapsack

Theory or the Logic :

This problem is solved using Greedy Approach.

A bag or knapsack of Capacity m is given
There are n items associated with profits $\boldsymbol{p_i} \ldots \boldsymbol{p_n}$ and weights $\boldsymbol{w_1} \ldots \boldsymbol{w_n}$
We have to fill knapsack using given items such that profit is maximum
Let x_i be the portion of the of the i^{th} element selected
$\therefore x_i$=1 implies i^{th} element is selected fully
and x_i=0 implies i^{th} element is not selected
and 0< x_i < 1 implies i^{th} element is partly selected
Problem is expressed mathematically as

Maximize :

$$\sum_{i=1}^{n} \boldsymbol{x_i}\ \boldsymbol{p_i} \ldots\ldots (\mathbf{1})$$

Subject to :

$$\sum_{i=1}^{n} \boldsymbol{x_i}\ \boldsymbol{w_i} \leq \boldsymbol{M} \ldots\ldots (\mathbf{2})$$

Any solution that satisfies equation (2) is a feasible solution and a solution which satisfies equation (1) is the optimal solution.

Algorithm:

- Start
- Declare n, M, p[ ], w[ ], u, tp
- Take input from user
- Sort the items in descending order of Profit/Weight ratio
- *Initialize u to M, tp to 0.0 and i to 0*
- ***while*** *u > 0*

if *w[i] < = u*
then *x[i] ← 1*
u ← u-w[i]
i++
else
x[i] ← u/w[i]
u ← 0

- ***end while***
- ***for*** *i = 0 to n-1*

*tp ← tp+x[i] * p[i]*
i++

- ***end for***
- *Return tp*
- Display Solution vector x [ ]
- *End*

Complexity:
The complexity of the Fractional Knapsack depends on the sorting technique used i.e. If bubble sort is used for sorting then the complexity is O(n^2).

Experiment no: 7

Aim: To Implement Dijkstra's Algorithm.

Theory or the Logic :

This algorithm is an example of Greedy Approach.

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks.
For a given source node in the graph, the algorithm finds the shortest path between that node and every other.

Algorithm:

- Start
- Declare g[ ] [ ], u, s[ ], q[ ], adj[ ] [ ], v
- *Take input from User*
- *Initialize u as source node*
- ***for*** *every vertex*

dist [i] = cost [u] [i]
pred[i] = u
end for

- *Initialize* pred[u] as Nil
- ***while*** *q is not empty*
- Select vertex v from q with minimum distance
- Update s and q
- Select the adjacent node to v belonging to q
- Find the minimum distance for that adjacent node
- Accordingly update distance
- Display dist array
- **end while**
- End

Complexity: The Complexity of Dijkstra's Algorithm is O(n^2)

Experiment no: 8

Aim: To Implement Prim's Algorithm

Theory or the Logic :

This algorithm is an example of Greedy Approach.

In computer science, Prim's algorithm is a greedy algorithm that finds a minimum spanning tree for a connected graph weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized.

Algorithm:

- Start
- Declare g[ ] [ ],cost [ ] [ ], u, s[ ], q[ ], adj[ ] [ ], pred[ ] [ ], v
- *Take input from User*
- *Initialize u as source node*
- ***for*** *every vertex*

dist [i] ← cost [u] [i]
pred[i] ← u

end for

- *Initialize* pred[u] as Nil
- ***while*** *q is not empty*
- Select vertex v from q with minimum distance
- Update s and q and pred
- Add the cost of the dist [v] to cost
- Select the adjacent node to v belonging to q
- Find the minimum distance for that adjacent node
- Accordingly update distance and pred arrays
- **end while**
- Display dist array
- Display pred array
- End

Complexity:

The Complexity of Prim's Algorithm is O($| v |^2$)
i.e. O(n^2)

Experiment no: 9

Aim: To Implement 0/1 Knapsack.

Theory or the Logic :

This problem is solved using dynamic programming.

A bag or knapsack of Capacity m is given
There are n items associated with profits $p_i \ldots p_n$ and weights $w_1 \ldots w_n$
We have to fill knapsack using given items such that profit is maximum
Let x_i be the portion of the of the i^{th} element selected
$\therefore x_i$=1 implies i^{th} element is selected fully
and x_i=0 implies i^{th} element is not selected

Problem is expressed mathematically as
Maximize :

$$\sum_{i=1}^{n} x_i \; p_i \ldots\ldots (1)$$

Subject to :

$$\sum_{i=1}^{n} x_i \; w_i \leq M \ldots\ldots (2)$$

Any solution that satisfies equation (2) is a feasible solution and a solution which satisfies equation (1) is the optimal solution.
Let V[i] [j] denotes the profit gained
Here, i denotes the no. of items and j denotes the capacity
we have to use one more array Keep [i] [j] to check if i^{th} goes in the bag of capacity of j or not/
It will be a Boolean array (i.e. either 0 or 1)
For w [i] > j
V[i] [j] = V[i - 1] [j]
Keep [i] [j] = 0
For w [i] < j
Select the max of

- V[i-1] [j]
  if selected make Keep [i] [j] = 0
- p [i] + V[i - 1] [j - w[i]]
  if selected make Keep [i] [j] = 1

Algorithm:

- Start
- Declare V[ ] [ ],Keep[ ] [ ], n, M, p[ ], w[ ], r

- Take input from user
- Initialize V[n+1] [n+1] to 0
- Initialize Keep[n+1] [M+1] to 0
- **for** i = 1 to n

for j = 1 to M

if w[i] > j

then V[i] [j] ← V[i - 1] [j]

Keep [i] [j] ← 0

else

if V[i-1] [j] > p [i] + V[i - 1] [j - w[i]]

then V[i] [j] ← V[i - 1] [j]

Keep [i] [j] ← 0

else

V[i] [j] ← p [i] + V[i - 1] [j - w[i]]

Keep [i] [j] ← 1

end for

end for

- Initialize r with M
- **for** i = n to 1

if Keep [i] [r] ← 1

then x[i] ← 1

r ← r – w[i]

end for

- Display Solution vector x [ ]
- End

Complexity:

The Complexity of 0/1 Knapsack is O(nM)

Experiment no: 10

Aim: To Implement Bellman Ford's Algorithm

Theory or the Logic :

This algorithm is an example of dynamic programming.

The Bellman–Ford algorithm is an algorithm that computes shortest from a single source vertex to all of the other vertices in a weighted digraph. It is slower than Dijkstra's algorithm for the same problem, but more versatile, as it is capable of handling graphs in which some of the edge weights are negative numbers. Negative edge weights are found in various applications of graphs, hence the usefulness of this algorithm. If a graph contains a "negative cycle" (i.e. a cycle whose edges sum to a negative value) that is reachable from the source, then there is no *cheapest* path: any path can be made cheaper by one more walk around the negative cycle. In such a case, the Bellman–Ford algorithm can detect negative cycles and report their existence

Algorithm:

```
Start
Declare dist [ ], pred[ ], u, v, w[ ], cost[ ], s (source node)
Take Input from the User
Initialize dist [ s ] to 0
Initialize pred [ s ] to Nil
for every vertex
        dist [ i ] ← cost [ s ] [ i ]
        pred [ i ] ← s
end for
for i = 1 to v-1
        for every edge
                Relax(u,v,w)
        end for
end for
End
```

Relax Method:

```
Start
for each edge
        if d [ v ] > d [ u ] + w
                return false
end for
Return true
End
```

Complexity: The Complexity of Bellman Ford Algorithm is O(n^3)

Experiment no: 11

Aim: To Implement Floyd Warshall's Algorithm.

Theory or the Logic :

This algorithm is an example of dynamic programming.

In computer science, the Floyd–Warshall algorithm (also known as Floyd's algorithm, Roy–Warshall algorithm, Roy–Floyd algorithm, or the WFI algorithm) is a graph analysis algorithm for finding shortest path in a weighted graph with positive or negative edge weights (but with no negative cycles) and also for finding transitive closure of a relation R. A single execution of the algorithm will find the lengths (summed weights) of the shortest paths between *all* pairs of vertices, though it does not return details of the paths themselves.
The Floyd–Warshall algorithm compares all possible paths through the graph between each pair of vertices

Algorithm:

```
Start
Declare cost[ ], n, D[ ] [ ]
for i = 1 to n
        for j = 1 to n
                D[ i ] [ j ] ←cost [ i ] [ j ]
        end for
end for

for k = 1 to n
        for i = 1 to n
                for j = 1 to n
                        D[ i ] [ j ] ← min { D[ i ] [ j ] , D[ i ] [ k ] + D [ k ] [ j ] }
                end for
        end for
end for
End
```

Complexity:

The Complexity of Floyd Warshall's Algorithm is O(n^3)

Experiment no: 12

Aim: To Implement Optimal Binary Search Tree.

Theory or the Logic :

This problem is solved using dynamic programming.

In computer science, , an optimal binary search tree (BST), sometimes called a weight-balanced binary tree, is a Binary Search Tree which provides the smallest possible search time or expected search time) for a given sequence of accesses (or access probabilities).
N distinct keys(item to be searched for) are given
Probability of searching each key is also given
We want to create a BST using these trees such that average no. of comparisons needed to search a key should be minimum
Such a tree is called Optimal Binary Search Tree.
Let p_i be the probability of searching key k_i and let q_i be the probability of unsuccessful search
$\therefore \sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$
If a successful search terminates at internal node k_i at level l then the cost of searching that internal node is $(l+1)* p_i$
If an unsuccessful search terminates at external node e_j at level l then cost of unsuccessful search will be $(l)* q_j$
∴The expected cost of BST is
$\sum_{i=1}^{n} p_i * [\, level\ of\ (k_i) + 1] + \sum_{j=0}^{n} q_j * [level\ of\ (e_j)]$
The BST for the identifier set { $k_1, k_2, \ldots . k_n$ } which gives the minimum cost is called as OBST

Algorithm:

- Start
- Declare w[ ] [ ], cost[ ] [ ], root[ ] [ ], min, mindex, n
- Initialize weight, cost and root array to 0
- Take Input from the User
- **for** i = 0 to n

for j = 0 to n
  if i < j
    then weight[i][j]←p[j]+q[j]+weight[i][j – 1] **else**
if i = j
    then weight[i][j] ←q[j]
  else
    weight[i][j] ←0
end for
end for

- Display weight table

- **for** i = n to 0

for j = 0 to n

if i = j

then cost[i][j] ←0

root[i][j] ←0

else if i < j

Initialize min to 99

Initialize mindex to 8

for k = i+1 to j

if weight[i][j]+(cost[i][k-1]+cost[k][j]) < min

then min←weight[i][j]+(cost[i][k-1]+cost[k][j])

mindex←k

end for

cost[i][j] ←min

root[i][j] ←mindex;

else

cost[i][j] ←0

root[i][j] ←0

end for

end for

- Display cost table
- Display root table
- End

Complexity:

The Complexity of Optimal Binary Search Tree is O(n^2).

Experiment no: 13

Aim : To Implement N-Queen.

Theory or the Logic :

This problem is solved using Backtracking.

The n-queens puzzle is the problem of placing n chess queens on an n×n chessboard so that no two queens threaten each other. Thus, a solution requires that no two queens share the same row, column, or diagonal. The eight queens puzzle is an example of the more general *n*-queens problem of placing *n* queens on an *n*×*n* chessboard, where solutions exist for all natural numbers *n* with the exception of *n*=2 and *n*=3
To make problem simple assume queens are numbered 1 to n and the rows are also numbered 1 to n (i.e. queen no. represents the row no.)
So we have to find the column no. for each queen
The solution vector will contain values 1 to n and it signifies the column position of queens numbered from 1 to n

Algorithm:

- Start
- Declare col[ ], q, c and n
- **for** c = 1 to n

if place (q , c)
then col[q] ← c
if q = n **then** display solution vector col
else
nqueen (q+1)
end for

- End

place Method :

- Start
- **for** k = 1 to q-1
  if c = col[k] **OR** abs(q – k) = abs(c – col[k])

then return 0
end for

- **else**

return 1

- End

Complexity:

The complexity of n-queen problem for row attack is O(n^n).
The complexity of n-queen problem for row attack and column attack is O(n!).
The complexity of n-queen problem for row attack,column and diagonal attack reduces further.

Experiment no: 14

Aim: To Implement Sum of Subsets.

Theory or the Logic :

This problem is solved using Backtracking.

In computer science, the subset sum problem is an important problem in computational theory and cryptography. The problem is this: given a set (or multiset) of integers, is there a non-empty subset whose sum is zero? For example, given the set {−7, −3, −2, 5, 8}, the answer is *yes* because the subset {−3, −2, 5} sums to zero.

An equivalent problem is this: given a set of integers and an integer *s*, does any non-empty subset sum to *s*? Subset sum can also be thought of as a special case of the knapsack problem. One interesting special case of subset sum is the partition problem, in which *s* is half of the sum of all elements in the set.

N distinct positive numbers are given ,they are called weights($w_1$to w_n)

Variable S is given, we have to find subsets of $w_1 \ldots w_n$ such that their sum is equal to S

Solution is represented by a Boolean array x[ ] of size n

x_i=1 implies i^{th} weight is selected
and x_i=0 implies i^{th} weight is not selected

WeightSoFar (wsf):
wsf denotes partial solution represented by node at level i (i.e. sum of weights that we have obtained by considering level 0 to i)

TotalProfitLeft (tpl):
tpl denotes the weight that can be obtained by considering the level (i+1) to n

There are 2 conditions to be checked to check whether a node will give solution or not in future
1.) wsf + w [i + 1] ≤ S
2.) wsf + tpl ≥ S

Algorithm:

- Start
- Declare S and sol[ ] as global
- Initialize sol[ ] to 0
- Declare wsf, l, tpl
- Take Input from user
- **if** wsf = S

then display solution vector sol
else if promising(l , wsf , tpl)
then x[l + 1] ←1
sumofsubsets(l+1,wsf+w[l+1],tpl- w[l+1])
x[l+1] ←0
sumofsubsets(l+1,wsf,tpl- w[l+1])

- End

promising method:

- Start
- **if** wsf + w [i + 1] ≤ S && wsf + tpl ≥ S

then return 1
else return 0

- End

Complexity:

The Complexity of Sum of Subsets is O (Sn)

Experiment no: 15

Aim: To Implement Graph Coloring.

Theory or the Logic :

This problem is solved using Backtracking.

In graph theory, graph coloring is a special case of graph labeling; it is an assignment of labels traditionally called "colors" to elements of a graph subject to certain constraints. In its simplest form, it is a way of coloring the vertices of a graph such that no two adjacent vertex share the same color; this is called a vertex coloring.
Graph coloring enjoys many practical applications as well as theoretical challenges. Beside the classical types of problems, different limitations can also be set on the graph, or on the way a color is assigned, or even on the color itself. It has even reached popularity with the general public in the form of the popular number puzzle Sudoku. Graph coloring is still a very active field of research.
Given a graph G(V,E)
We need to color the graph using m colors but no 2 adjacent nodes having same color
This problem is also called as m-colorability decision problem
The smallest number by which we can color the nodes of the graph is called chromatic number (m)

Algorithm:

- Start
- Declare adj[ ] [ ], sol[ ], m, node, n and c
- **for** c = 1 to m

if cancolor (node , c)
then sol[node] ← c
if node = n
then display solution vector sol
else
mcolor (node+1)
end for

- End

cancolor Method :

- Start
- **for** k = 1 to node - 1
if c = sol[k] **AND** adj[j] [node] = 1
then return 0
end for
- **else**

return 1

- End

Complexity: The Complexity of Graph Coloring is O(n^2)

Experiment no: 16

Aim: To Implement Longest Common Subsequence

Theory or the Logic :

This problem is solved using Dynamic Programming.

The longest common subsequence (LCS) problem is the problem of finding the longest subsequence common to all sequences in a set of sequences (often just two sequences). It differs from problems of finding common substring : unlike substrings, subsequences are not required to occupy consecutive positions within the original sequences. The longest common subsequence problem is a classic computer science problem, the basis of Data comparison programs such as the diff utility, and has applications in Bioinformatics. It is also widely used by revision control such as Git for reconciling multiple changes made to a revision-controlled collection of files.
In this problem, we are given 2 sequences X and Y where
X=$X_1X_2X_3 \ldots X_m$ and Y=$Y_1Y_2Y_3 \ldots Y_n$
We can say that sequence Z is common subsequence of X and Y if Z is the subsequence of both

Algorithm:

- Start
- Declare x[ ], y[ ], c[ ][ ], b[ ][ ], m, n, sub[ ], len
- Take Input from the User
- Initialize m as the length of char array x
- Initialize n as the length of char array y
- Initialize len to 0
- **for** i = 1 to m

c[i] [0] ←0
end for

- **for** j = 1 to n

c[0] [j] ←0
end for

- **for** i = 1 to m

for j = 1 to n
if x[i] = y[j]
then c[i] [j] ←c[i - 1] [j - 1] + 1
b[i] [j] ← ” ↖ ”
else if c[i - 1] [j] > c[i] [j - 1]
then c[i] [j] ←c[i - 1] [j]
b[i] [j] ← ” ↑ ”
else
c[i] [j] ←c[i] [j - 1]
b[i] [j] ← ” ← ”
end for

end for

- Display c and b arrays
- **while 1**

if b[m][n] = 0
then break
else if b[m][n] **=** ” ↖ ”
then sub[len] ←x[m-1]
m←m-1
n←n-1
len++
else if b[m][n]**=** ” ↑ ”
then m←m-1
else if b[m][n]**=** ” ← ”
then n←n-1
end while

- Display sub array
- End

Complexity:

The Complexity of Longest Common Subsequence is O(mn).