Topic : ..... Date. : ..... Page No. : .....

Experiment 1

Aim

To implement & analyse the time complexity of the following algorithms - Bubble sort, Insertion Sort, Quicksort

Theory

BUBBLE SORT

Algorithm :

```
Bubblesort (a, n)
    for i=0 to n-1
        swap = false
        for j=i+1 to n
            if a[j-1] > a[j]
                swap (a[j], a[j+1])
                swap = true
        if (not swap) then
            break
```

Time Complexity Analysis : The algorithm has $n-1$ comparisons in the 1st pass
$n-2$ in the 2nd pass & so on

Total comparisons = $(n-1)+(n-2)+(n-3)+\cdots+3+2+1$

$= n(n-1)/2$

$= O(n^2)$

Case I - Best Case

When the array is already sorted

There are only n comparisons $\Rightarrow O(n)$

Case II - Worst Case

Reverse sorted array

$\Rightarrow O(n^2)$

Topic : ..... Date. : ..... Page No. : .....

Case III - Average Case
$\Rightarrow O(n^2)$

INSERTION SORT

Algorithm: Insertion Sort (A, n)

```
(I)   for j = 2 to n
(II)      key = A[j]
(III)     // Insert A[j] into sorted sequence A[1 .... j-1]
(IV)      i = j - 1
(V)       while i > 0 and A[i] > key
(VI)          A[i+1] = A[i]
(VII)         i = i - 1
(VIII)    A[i+1] = key
```

Time Complexity Analysis:

Line:		Cost:	*	Time:
I		c_1		$n-2$
II		c_2		$n-1$
III		c_4		1
(V)		c_5		$\sum_{j=2}^{n} t_j$
VI		c_6		$\sum_{j=2}^{n}(t_j-1)$
VII		c_7		$\sum_{j=2}^{n}(t_j-1)$
VIII		c_8		$n-1$

Case I - Best Case

$T(n) = c_1(n-2) + c_2(n-1) + c_4 + c_5(n-1) + c_8(n-1)$
$= an + b$
$\Rightarrow$ Linear function of n
$\Rightarrow O(n)$

Case II - Worst Case

Topic : ..... Date. : ..... Page No. : .....

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n(n+1)/2) - 1) + c_6(n(n-1)/2) + c_7(n(n-1)/2) + c_8(n-1)$$

$$= an^2 + bn + c$$

$$= O(n^2)$$

Case II: Average Case

$T(n) \Rightarrow O(n^2)$

SELECTION SORT

Algorithm: SelectionSort (A, n)

(I) for i = 1 to n
(II) min = i
(III) for j = i+1 to n
(IV) if A[j] < A[min]
(V) min = j
(VI) if index min != i
(VII) swap A[min] and A[i]

Time Complexity Analysis:

$$T(n) = c_1 + c_2 n + c_3(n-1) + c_4 \sum_{j=1}^{n-1} (n-j+1) + c_5 \left(\sum_{j=1}^{n-1} (n-j)\right) + c_6(n-1)$$

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

$$= O(n^2)$$

Best case = Worst Case = Average case

$= O(n^2)$

CLASSTIME

Date :
Page No. :

Technique Used

We used incremental technique or loop & conditionals to solve the problem

① Bubble Sort - Compares & swaps adjacent elements repeatedly in every pass. In i^{th} pass, $(i-1)$ elements are already sorted.

② Selection Sort - Selects the i^{th} smallest element & places at i^{th} position. This algo divides the array into two parts : sorted (left) & unsorted (right). It repeatedly selects the next smallest element.

③ Insertion Sort - Insert an element into its proper position. In i^{th} iteration, previous $(i-1)$ elements are already sorted. & the i^{th} element, $A[i]$ is inserted into its proper position

Result

Bubble Sort, Insertion Sort, Selection Sort were implemented using array as data structure & time complexities were analysed

Insertion Sort

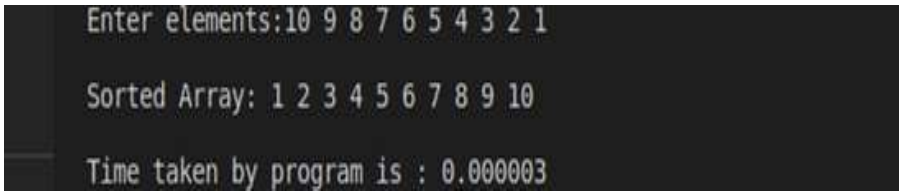
Source Code:

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
	int arr[100], i, j, len, key;
	clock_t start,end;
	cout<<"Enter array length:"; cin>>len;
	cout<<"Enter elements:";
	for(i=0;i<len;i++ )
	{
		cin>>arr[i];
	}
	start = clock();
	for(j=1;j<len;j++)
	{
		key = arr[j];
		i = j-1;
		while(i>=0 && arr[i]>key)
		{
			arr[i+1] = arr[i];
			i=i-1;
		}
		arr[i+1] = key;
	}
	end = clock();
	cout<<"\nSorted Array: ";
	for(i=0;i<len;i++)
	{
```

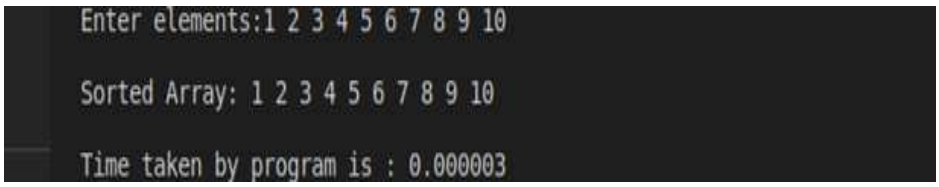
```
        cout<<arr[i]<<" ";
    }
    cout<<"\n\n";
    double time_taken = double(end - start)/double(CLOCKS_PER_SEC);
    cout << "Time taken by program is : " << fixed
        << time_taken << setprecision(10)<<"\n";
    return 0;
}
```

Output:

Worst Case



Best Case



Selection Sort

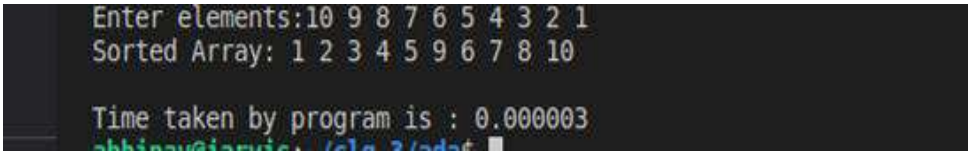
Source Code:

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int arr[20], i, j, len, lowest, count=0, pos;
    clock_t start, end;
    cout<<"Enter array length:"; cin>>len;
    cout<<"Enter elements:";
    for(i=0;i<len;i++ )
    {
        cin>>arr[i];
```

```
	}
	start = clock();
	for(i=0;i<len-1;i++)
	{
		lowest = arr[i];
		for(j=i+1; j<len;j++)
		{
			if(lowest>arr[j])
			{
				lowest = arr[j];
				count++;
			pos = j;
			}
		}
		if(count!=0)
		{
			int temp = arr[i];
			arr[i] = arr[pos];
			arr[pos] = temp;
		}
	}
	end = clock();
	cout<<"Sorted Array: ";
		for (i=0;i<len;i++)
			cout<<arr[i]<<" ";
	cout<<"\n\n";
	double time_taken = double(end - start)/double(CLOCKS_PER_SEC);
	cout << "Time taken by program is : " << fixed
		<< time_taken << setprecision(10)<<"\n";
	return 0;
}
```

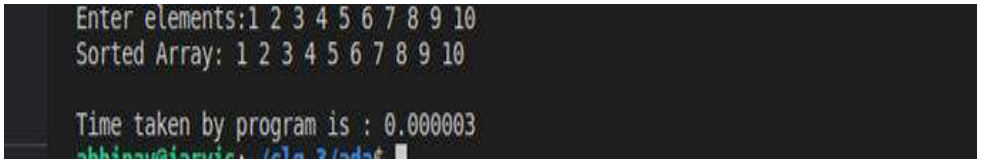
Output:

Worst Case



\

Best Case



Bubble Sort

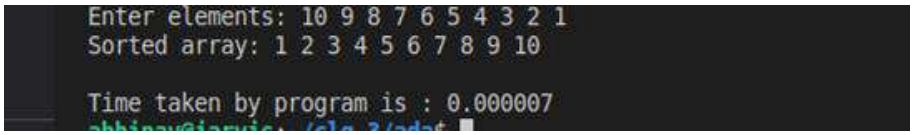
Source Code:

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
	int arr[10], temp, i, j, len;
	cout<<"Enter length: "; cin>>len;
	cout<<"Enter elements: ";
	for(i=0;i<len;i++)
		cin>>arr[i];
	clock_t start, end;
	start = clock();
	for(i=0;i<len;i++)
	{
		for(j=0;j<len-i-1;j++)
		{
			if(arr[j]>arr[j+1])
			{
				temp = arr[j];
				arr[j] = arr[j+1];
```

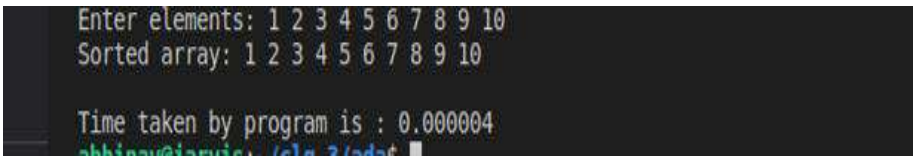
```
                    arr[j+1] = temp;
                }
            }
        }
        end = clock();
        cout<<"Sorted array: ";
        for(i=0;i<len;i++)
            cout<<arr[i]<<" ";
        cout<<"\n\n";
        double time_taken = double(end - start)/double(CLOCKS_PER_SEC);
        cout << "Time taken by program is : " << fixed
            << time_taken << setprecision(10)<<"\n";
        return 0;
}
```

Output:

Worst Case



Best Case



Topic : ..... Date. : ..... Page No. : .....

Experiment 2

Aim

To implement the following algorithms & analyse the time complexity -

(a) Merge Sort

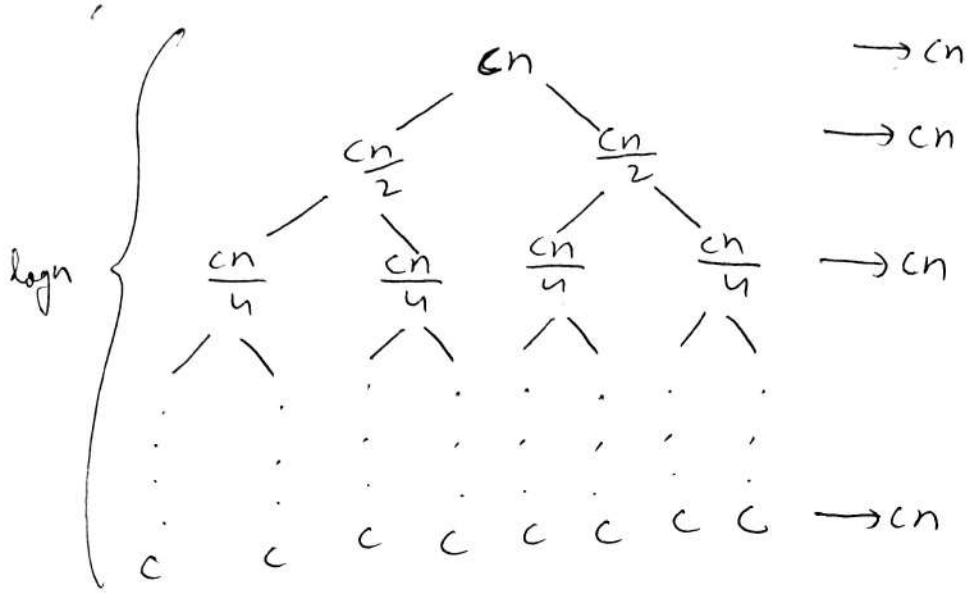
(b) Quick Sort

Theory

1. **Merge Sort** -

Algorithm:

```
Merge Sort (A, p, r)
{if p < r
    q = [(p+r)/2]
    Merge Sort (A, p, q)
    Merge Sort (A, q+1, r)
    Merge (A, p, q, r)
}

Merge (A, p, q, r)
{n1 = q-p+1
n2 = r-q
Let L[1 .. n1+1] & R[1 .. n2+1] be new arrays
for i = 1 to n1
    L[i] = A[p+i-1]
for j = 1 to n2
    R[j] = A[q+j]
L[n1+1] = ∞
R[n2+1] = ∞
i = 1    j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
```



$$T(n) = cn(\log n + 1)$$
$$= cn\log n + cn$$
$$= O(n\log n)$$

Merge Sort

Date :
Page No. :

```
    i = i + 1
    swap j = j + 1 A
    A[i] = A[j]
```

Time Complexity Analysis :

Case I - Best Case (Sorted array)
⇒ O(nlogn)

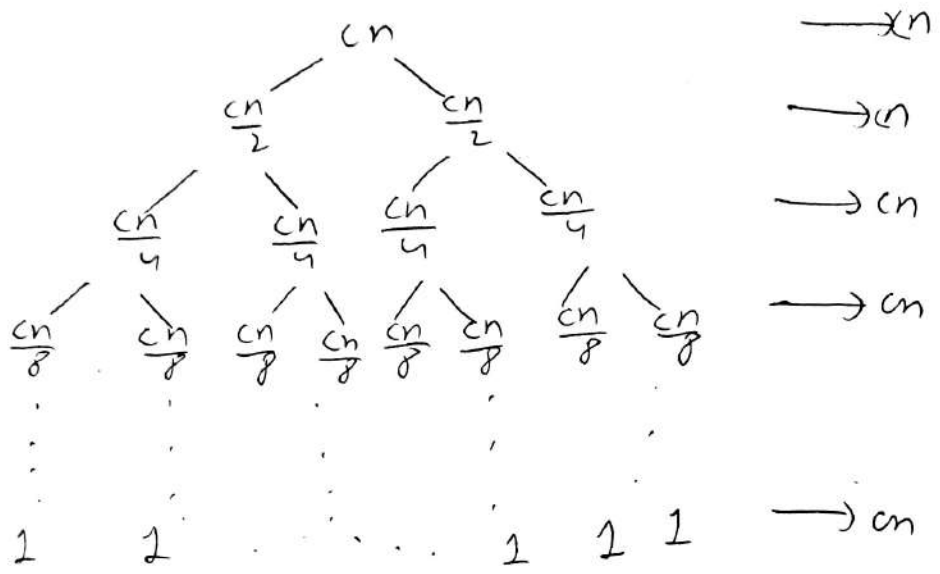
Case II - Worst Case (Reverse sorted)
⇒ O(nlogn)

Case III - Average Case
⇒ O(nlogn)

2. **Quick Sort**

Algorithm:

```
QuickSort (left, right)
    if right - left <= 0
        return
    else
        pivot = A[right]
        part = partition (left, right, pivot)
        QuickSort (left, part - 1)
        QuickSort (part + 1, right)
```



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

Quick Sort (Best Case)

Date :
Page No. :

```
partition (left, right, pivot)
    leftPtr = left
    rightPtr = right - 1
    while True do
        while A[++leftPtr] < pivot
        if leftPtr >= rightPtr
            break
        else
            swap leftPtr, rightPtr
        swap leftPtr, right
        return
```

Time Complexity Analysis :

Case I :- Best Case (Pivot is towards the middle)

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

Case II - Worst Case (Pivot is at extremes)

$T(n) = O(n^2)$

Technique Used

We use the Divide & Conquer technique in both quick & merge sort.

The problem is divided into smaller subproblems & each problem is solved independently.

Date :
Page No. :

when we keep on dividing the subproblems into even smaller subproblems, we may eventually reach a stage where no more division is possible. These smallest possible subproblems are sorted i.e. solved. The solutions are finally merged.

Result

MergeSort & QuickSort were implemented using array as data structure & the time complexity for different cases was analysed

Merge Sort

Source Code

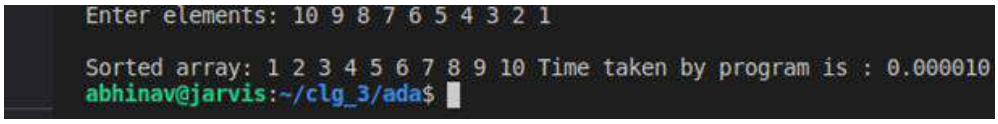
```
#include<bits/stdc++.h>
using namespace std;
void merge(int arr[], int p, int q, int r)
{
	int n1, n2, i, k, j;
	n1 = q-p+1;
	n2 = r-q;
	int left[n1], right[n2];
	for(i=0;i<n1;i++)
	{
		left[i] = arr[p+i];
	}
	for(i=0;i<n2;i++)
	{
		right[i] = arr[q+i+1];
	}
	i=0;j=0;k=p;
	while (i<n1 && j<n2)
	{
		if(left[i]<=right[j])
		{
			arr[k] = left[i];
			i++;
		}
		else
		{
```

```
                arr[k] = right[j];
                j++;
            }
            k++;
        }
        while (i<n1)
        {
            arr[k] = left[i];
            i++;
            k++;
        }
        while (j < n2)
        {
            arr[k] = right[j];
            j++;
            k++;
        }
}

void mergeSort(int arr[], int p, int r)
{
        int q;
        if(p<r)
        {
            q = (p+r)/2;
            mergeSort(arr, p,q);
            mergeSort(arr, q+1,r);
            merge(arr,p,q,r);
        }
```

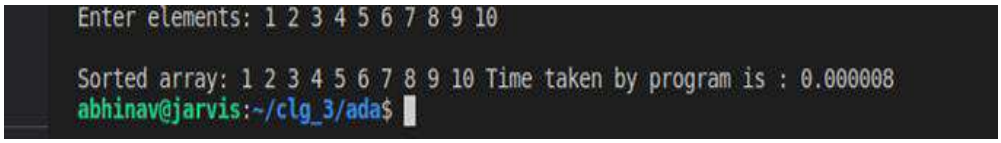
```
}

int main()
{
        int p=0, q, r, i, arr[20], len;
        clock_t start, end;
        cout<<"Enter no. of elements:"; cin>>len;
        cout<<"Enter elements: ";
        for(i=0;i<len;i++)
        {
                cin>>arr[i];
        }
        start = clock();
        mergeSort(arr, 0, len);
        end = clock();
        cout<<"\nSorted array: ";
        for(i=0;i<len;i++)
                cout<<arr[i]<<" ";
        double time_taken = double(end - start)/double(CLOCKS_PER_SEC);
        cout << "Time taken by program is : " << fixed
                << time_taken << setprecision(10)<<"\n";
        return 0;
}
```

Output

Worst Case



Best Case



Quick Sort

Source Code

```
#include<bits/stdc++.h>
using namespace std;
int Partition(int arr[], int p, int r)
{
	int x = arr[r];
	int i = p-1, temp;
	for(int j=p;j<r;j++)
	{
		if(arr[j]<=x)
		{
			i=i+1;
			temp = arr[i];
			arr[i] = arr[j];
			arr[j] = temp;
		}
	}
	temp = arr[i+1];
	arr[i+1] = arr[r];
	arr[r] = temp;
	return (i+1);
}
void QuickSort(int arr[], int p, int r)
```

```
{
	int q;
	if(p<r)
	{
		q = Partition(arr, p, r);
		QuickSort(arr, p, q-1);
		QuickSort(arr, q+1, r);
	}
}

int main()
{
	int arr[10], len;
	clock_t start, end;
	cout<<"Enter length: "; cin>>len;
	cout<<"Enter elements: ";
	for(int i=0;i<len;i++)
		cin>>arr[i];
	start = clock();
	QuickSort(arr, 0, len-1);
	end = clock();
	cout<<"Sorted array: ";
	for(int i=0;i<len;i++)
		cout<<arr[i]<<" ";
	cout<<"\n";
	double time_taken = double(end - start)/double(CLOCKS_PER_SEC);
	cout << "Time taken by program is : " << fixed
		<< time_taken << setprecision(5)<<"\n";
	return 0;
```

}

Output

Worst Case

```
Enter elements: 10 9 8 7 6 5 4 3 2 1
Sorted array: 1 2 3 4 5 6 7 8 9 10
Time taken by program is : 0.000006
abhinav@jarvis:~/clg_3/ada$
```

Best Case

```
Enter elements: 1 2 3 4 5 6 7 8 9 10
Sorted array: 1 2 3 4 5 6 7 8 9 10
Time taken by program is : 0.000005
abhinav@jarvis:~/clg_3/ada$
```

Date :
Page No. :

Experiment 3

Aim

To implement linear & binary search algorithms & analyse its time complexity

Theory

① Linear search

Algorithm: Linear Search (A, x)

```
i=1
if i>n
    return -1
for (i=1 to n)
if A[i] == x
    return position of x
end
```

Time Complexity Analysis :

Case I) Best Case

Element is present in beginning of list

$T = O(1)$

Case II) Worst Case

Elmnt is not present in list

Date :
Page No. :

$T = O(n)$

Case III) Average Case

Element may be anywhere in list

$T = O(n)$

② Binary Search

Algorithm : BinarySearch (A, n, x)

```
low = 0
high = n-1
while low ≤ high
    mid = (low + high)/2
    if A[mid] < x
        low = mid + 1
    else if A[mid] > x
        high = mid - 1
    else
        return mid
return -1
```

Time Complexity Analysis :

Case I) Best Case

Element present at mid

$T = O(1)$

Date :
Page No. :

Case II) Worst Case
Element not in array
$T = O(\log n)$

Case III) Average Case
$T = O(\log n)$

Technique used
We use an iterative approach in linear search & the divide & conquer approach in binary search.

Linear search requires 'n' comparisons on an average whereas binary search requires only $\log n$ comparisons in the average case

Result
Linear & binary search were implemented & their time complexities were analysed.

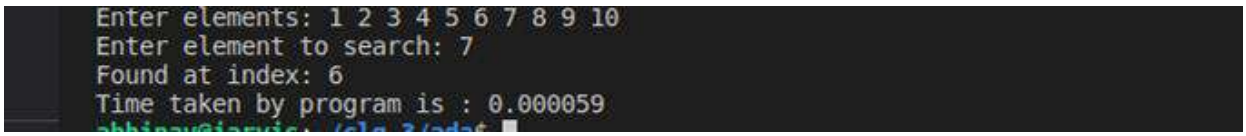
Linear Search

Source Code

```
#include<bits/stdc++.h>
using namespace std;

int main()
{
	int len, arr[20], x, flag=0, ind;
	clock_t start, end;
	cout<<"Enter length: ";cin>>len;
	cout<<"Enter elements: ";
	for(int i=0;i<len;i++)
		cin>>arr[i];
	cout<<"Enter element to search: "; cin>>x;
	start = clock();
	for(int i=0;i<len;i++)
	{
		if(arr[i]==x)
		{
			flag = 1;
			ind = i;
			break;
		}
	}
	if(flag ==1)
		cout<<"Found at index: "<<ind<<"¥n";
	else
		cout<<"not found"<<"¥n";
	end = clock();
```

```
	double time_taken = double(end - start)/double(CLOCKS_PER_SEC);
	cout << "Time taken by program is : " << fixed
		<< time_taken << setprecision(5)<<"¥n";
	return 0;
}
```

Output



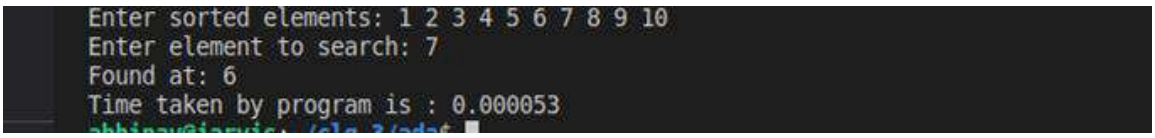
Binary Search

Source Code

```
#include<bits/stdc++.h>
using namespace std;
int binary(int arr[], int low, int high, int x)
{
	if(low<=high)
	{
		int mid = (low+high)/2;
		if(arr[mid]==x)
			return mid;
		if(arr[mid]>x)
			return binary(arr, low, mid-1,x);
		else
			return binary(arr, mid+1, high, x);
	}
```

```
	else
		return -1;
}

int main()
{
	int len, arr[20], x;
	clock_t start, end;
	cout<<"Enter length: ";cin>>len;
	cout<<"Enter sorted elements: ";
	for(int i=0;i<len;i++)
		cin>>arr[i];
	cout<<"Enter element to search: "; cin>>x;
	start = clock();
	int flag = binary(arr, 0, len,x);

	if(flag==-1)
		cout<<"Not found"<<"¥n";
	else
		cout<<"Found at: "<<flag<<"¥n";
	end = clock();
	double time_taken = double(end - start)/double(CLOCKS_PER_SEC);
	cout << "Time taken by program is : " << fixed
		<< time_taken << setprecision(5)<<"¥n";
	return 0;
}
```

Output


Enter sorted elements: 1 2 3 4 5 6 7 8 9 10
Enter element to search: 7
Found at: 6
Time taken by program is : 0.000053

Date :
Page No. :

Experiment 4

Aim

To implement matrix chain multiplication & analyse its time complexity

Theory

Algorithm: Matrix Chain Order (p)

```
1.  n = p.length - 1
2.  Let m[1...n, 1...n] & s[1...n-1, 2...n]
      be new tables
3.  for i = 1 to n
4.      m[i, i] = 0
5.  for l = 2 to n
6.      for i = 1 to n-i+1
7.          j = i - l + 1
8.          m[i, j] = ∞
9.      for k = 1 to j - 1
10.         q = m[i, k] + m[k+1, j] + pi pk pj
11.         if q < m[i, j]
12.             m[i, j] = q
13.             s[i, j] = k
14. return m & s
```

Time Complexity Analysis:

1st level of M[1, n], k = 1 to n-1

⇒ (n-1)c

Date :
Page No. :

IInd level, $k = 1$ to $n-2$
we have 2 values $\Rightarrow 2c(n-2)$

IIIrd level $k = 1$ to $n-3$
we have 3 values $\Rightarrow 3c(n-3)$

For $(n-1)$ levels
$$\text{cost} = (n-1)(n-(n-1))$$

$$T(n) = c(n-1) + 2c(n-2) + 3c(n-3) + \cdots + (n-1)k(1)$$
$$= \frac{n^3 c}{6} = O(n^3)$$

Technique used

Dynamic programming is an algorithmic technique for solving an optimisation problem by breaking it down into simpler problems & using the fact that the optimal solution to the subproblems it may have been found earlier

There are two types of approach -

① Bottom up ② Top Down

Result

Matrix chain multiplication was implemented successfully.

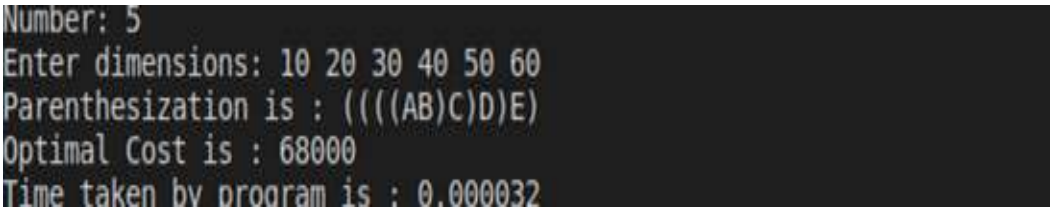
Matrix Chain Multiplication

Source Code

```
#include<bits/stdc++.h>
using namespace std;
void printParenthesis(int i, int j, int n,int *bracket, char &name)
{
    if (i == j)
    {
        cout << name++;
        return;
    }
    cout << "(";
    printParenthesis(i, *((bracket+i*n)+j), n,bracket, name);
    printParenthesis(*((bracket+i*n)+j) + 1, j,n, bracket, name);
    cout << ")";
}
void matrixChainOrder(int p[], int n)
{
    int m[n][n];
    int bracket[n][n];
    for (int i=1; i<n; i++)
        m[i][i] = 0;
    for (int L=2; L<n; L++)
    {
        for (int i=1; i<n-L+1; i++)
        {
            int j = i+L-1;
            m[i][j] = INT_MAX;
            for (int k=i; k<=j-1; k++)
            {
                int q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
```

```
            if (q < m[i][j])
            {
               m[i][j] = q;
               bracket[i][j] = k;
            }
         }
      }
   }
   char name = 'A';
   cout << "Parenthesization is : ";
   printParenthesis(1, n-1, n, (int *)bracket, name);
   cout << "\nOptimal Cost is : " << m[1][n-1];
}
int main()
{
   int m;
   clock_t start, end;
   cout<<"Number: ";
   cin>>m;
   int arr[m+1];
   cout<<"Enter dimensions: ";
   for(int i =0;i<m+1;i++)
      cin>> arr[i];
   int n = sizeof(arr)/sizeof(arr[0]);
   start = clock();
   matrixChainOrder(arr, n);
   end = clock();
   double time_taken = double(end - start)/double(CLOCKS_PER_SEC);
   cout << "\nTime taken by program is : " << fixed
      << time_taken << setprecision(10)<<"\n";
   return 0;
}
```

Output


Number: 5
Enter dimensions: 10 20 30 40 50 60
Parenthesization is : ((((AB)C)D)E)
Optimal Cost is : 68000
Time taken by program is : 0.000032

Date :
Page No. :

EXPERIMENT 5

Aim

WAP to find longest common subsequence & analyse its time complexity

Theory

For a set of given sequence, the LCS problem is to find the common subsequence of all sequences that is of maximum length.

If S_1 & S_2 are the given sequences, then Z is the common subsequence of S_1 & S_2 if Z is a subsequence of both S_1 & S_2. Furthermore, Z must be a strictly increasing sequence of indices of both S_1 & S_2. In a strictly increasing sequence, the indices of elements chosen must be in ascending order.

Algorithm:

```
LCS_length (x, y)
m = x length
n = y length
let b[1...m, 1...n] & c[0...m, 0...n] be new tables
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
```

Date :
Page No. :

```
for j = 1 to n
  if x_i == y_i
      c[i,j] = c[i-1, j-1] + 1
      b[i,j] = "↖"
  else if c[i-1,j] ≥ c[i,j-1]
      c[i,j] = c[i-1,j]
      b[i,j] = "↑"
  else c[i,j] = c[i,j-1]
      b[i,j] = "←"
return c & b

PRINT-LCS (b, X, i, j)
if i=0 or j==0
    return
if b[i,j] == "↖"
    PRINT-LCS(b, X, i-1, j-1)
    print x_i
else if b[i,j] == "↑"
    PRINT-LCS (b, X, i-1, j)
Else PRINT-LCS (b, X, i, j-1)
```

Complexity Analysis

time : For x[1 ... n], for y[1 ... n]

Since we are using two loops for both strings, therefore comp of finding LCS is O(m×n) where m & n are lengths

Since implementation also involves n rows & m columns,

Space complexity = O(n×m)

Source Code

```
#include<bits/stdc++.h>
using namespace std;

void lcsAlgo(char *S1, char *S2, int m, int n)
{
	int LCS_table[m + 1][n + 1];
	for (int i = 0; i <= m; i++)
	{
		for (int j = 0; j <= n; j++)
		{
			if (i == 0 || j == 0)
				LCS_table[i][j] = 0;
			else if (S1[i - 1] == S2[j - 1])
				LCS_table[i][j] = LCS_table[i - 1][j - 1] + 1;
			else
				LCS_table[i][j] = max(LCS_table[i - 1][j], LCS_table[i][j - 1]);
		}
	}

	int index = LCS_table[m][n];
	char lcsAlgo[index + 1];
	lcsAlgo[index] = '\0';

	int i = m, j = n;
	while (i > 0 && j > 0)
	{
		if (S1[i - 1] == S2[j - 1])
		{
			lcsAlgo[index - 1] = S1[i - 1];
			i--;
			j--;
			index--;
		}
		else if (LCS_table[i - 1][j] > LCS_table[i][j - 1])
			i--;
		else
			j--;
	}

	cout << "S1 : " << S1 << "\nS2 : " << S2 << "\nLCS: " << lcsAlgo <<"\
	nLength:"<<strlen(lcsAlgo)<<"\n";
}

int main()
{
	char S1[] = "ACADBA";
```

```
        char S2[] = "CBDAC";
        clock_t start, end;
        int m = strlen(S1);
        int n = strlen(S2);
        start = clock();
        lcsAlgo(S1, S2, m, n);
        end = clock();
        double time_taken = double(end - start)/double(CLOCKS_PER_SEC);
        cout << "Time taken by program is : " << fixed
                << time_taken << setprecision(10)<<"\n";
}
```

Output

Date :
Page No. :

EXPERIMENT 6

Aim

WAP to find optimal binary search tree & find its time complexity

Theory

A BST is a tree where the key values are stored in the internal nodes. External nodes are null nodes. Keys are ordered lexicographically i.e. for each internal node, all keys in left subtree are less than keys in right subtree.

A BST is converted to Optimal BST by placing the most frequently used data in the root & close to the root, while placing the least frequently used data near the leaves & in leaves. Dynamic programming is used.

Algorithm:

OPTIMAL-BST (p, q, n)

let e[1 ... n+1, 0 ... n], w[1 ... n+1, 0 ... n], & root[1 ... n, 1 ... n] be new tables

for i = 1 to n+1

e[i, i-1] = q_{i-1}

w[i, i-1] = q_{i-1}

for l = 1 to n

for i = 1 to n-l+1

j = i+l-1

Date :
Page No. :

```
e[i,j] = ∞
w[i,j] = w[i,j-1] + p_j + q_j
for r = i to j
  t = e[i,r-1] + e[r+1,j] + w[i,j]
  if t < e[i,j]
    e[i,j] = t
    root[i,j] = r
return e & root
```

Complexity Analysis:

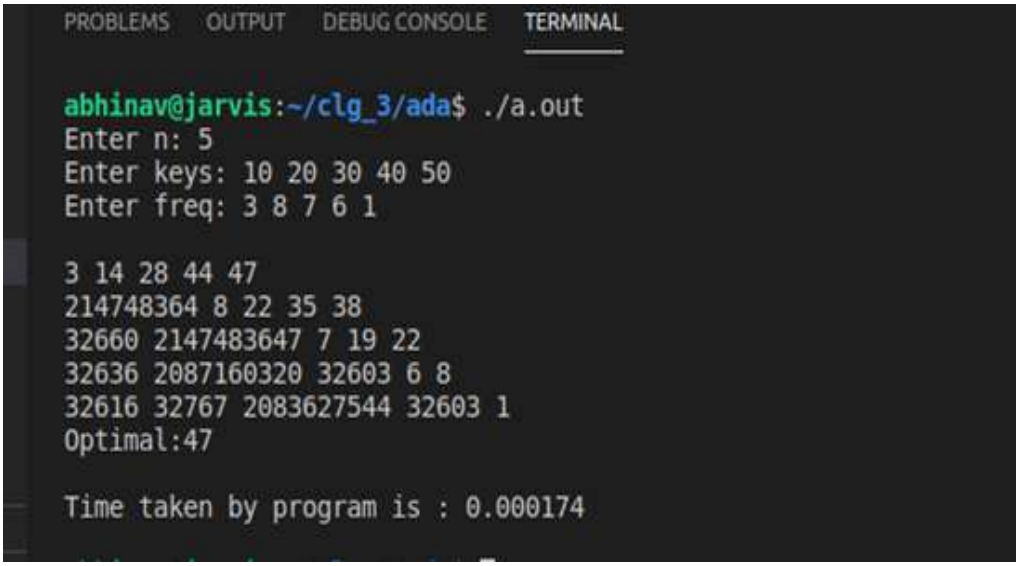
Requires $O(n^3)$ time, since three nested for loops are used. Each of these loops takes at most n values

Source Code

```
#include <bits/stdc++.h>
using namespace std;

int sum(int freq[], int low, int high)
{
	int sum = 0;
	for (int k = low; k <=high; k++)
	sum += freq[k];
	return sum;
}

void minCostBST(int keys[], int freq[], int n)
{
	int cost[n][n];
	for (int i = 0; i < n; i++)
		cost[i][i] = freq[i];

	for (int length=2; length<=n; length++)
	{
		for (int i=0; i<=n-length+1; i++)
		{
			int j = i+length-1;
			cost[i][j] = INT_MAX;

			for (int r=i; r<=j; r++)
			{
				int c=0, d=0;
				if(r>i)
					c=cost[i][r-1];
				else
					c=0;
				if(r<j)
					d=cost[r+1][j];
				else
					d=0;
				c=c+d+sum(freq,i,j);
				if (c < cost[i][j])
					cost[i][j] = c;
			}
		}
	}
	cout<<endl;

	for(int z=0;z<n;z++)
	{	for(int x=0;x<n;x++)
		cout<<cost[z][x]<<" ";
```

```
            cout<<endl;
        }
        cout<<"Optimal:"<<cost[0][n-1];
}

int main()
{
        int n, keys[10], freq[10];
        clock_t start, end;
        cout<<"Enter n: "; cin>>n;
        cout<<"Enter keys: ";
        for(int q=0;q<n;q++)
            cin>>keys[q];
        cout<<"Enter freq: ";
        for(int q=0;q<n;q++)
            cin>>freq[q];
        start = clock();
        minCostBST(keys, freq, n);
        end = clock();
        double time_taken = double(end - start)/double(CLOCKS_PER_SEC);
        cout << "\n\nTime taken by program is : " << fixed
            << time_taken << setprecision(10)<<"\n";
        cout<<endl;
}
```

Output



Date :
Page No. :

EXPERIMENT 7

Aim

To study & find time complexity of Huffmann coding

Theory

It is an effective method of compressing data without losing information. Huffmann codes compress data very effectively, saving 20% to 90% depending on the characteristics of data being compressed. Data is considered to be a sequence of characters. Huffman's greedy algorithm uses a table giving how often each character occurs to build up an optimal way of representing each character as a binary string.

It uses greedy approach to find the solution. A greedy algos always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution

algorithm:

```
HUFFMAN (c)
    n = |c|
    a = c
    for i = 1 to n-1
        allocate a new node z
```

Date :
Page No. :

2. left = x = EXTRACT_MIN(Q)
2. right = y = EXTRACT_MIN(Q)
2. freq = x.freq + y.freq
INSERT (Q, 2)
return EXTRACT_MIN(Q)

Complexity Analysis:
Time comp. is O(nlogn). Using a heap to store the weight of each tree each iteration requires O(nlogn) time
Space comp: O(n)

Source Code

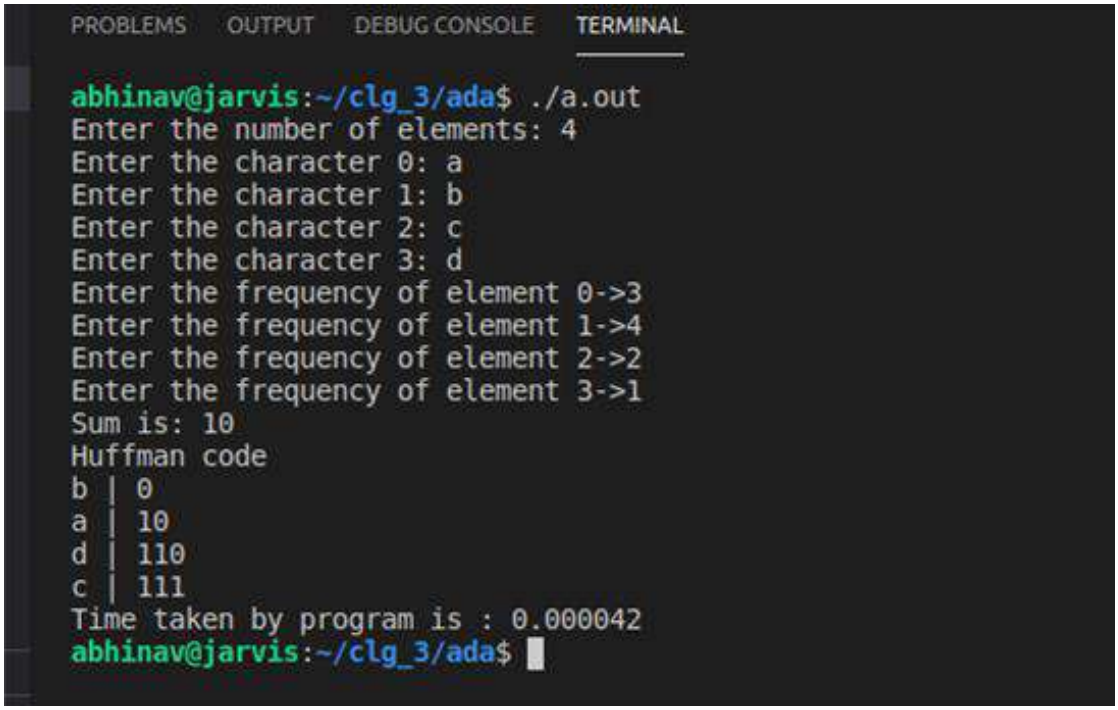
```
#include<iostream>
#include<bits/stdc++.h>
using namespace std;
#define MAX_TREE_HT 50

void printArray(int [], int );
struct MinHNode
{
	unsigned freq;
	char item;
	struct MinHNode *left, *right;
};

struct MinH
{
	unsigned size;
	unsigned capacity;
	struct MinHNode **array;
};

struct MinHNode *newNode(char item, unsigned freq)
{
	struct MinHNode *temp = (struct MinHNode *)malloc(sizeof(struct MinHNode));
	temp->left = temp->right = NULL;
	temp->item = item;
	temp->freq = freq;
	return temp;
}

struct MinH *createMinH(unsigned capacity)
{
	struct MinH *minHeap = (struct MinH *)malloc(sizeof(struct MinH));
	minHeap->size = 0;
	minHeap->capacity = capacity;
	minHeap->array = (struct MinHNode **)malloc(minHeap->capacity * sizeof(struct MinHNode
*));
	return minHeap;
}
void swapMinHNode(struct MinHNode **a, struct MinHNode **b)
{
	struct MinHNode *t = *a;
	*a = *b;
	*b = t;
}
void minHeapify(struct MinH *minHeap, int idx)
{
	int smallest = idx;
	int left = 2 * idx + 1;
	int right = 2 * idx + 2;
```

```
	if (left < minHeap->size && minHeap->array[left]->freq < minHeap-
	>array[smallest]->freq)
		smallest = left;
	if (right < minHeap->size && minHeap->array[right]->freq < minHeap-	>array[smallest]->freq)
		smallest = right;
	if (smallest != idx)
	{
		swapMinHNode(&minHeap->array[smallest], &minHeap->array[idx]);
		minHeapify(minHeap, smallest);
	}
}

int checkSizeOne(struct MinH *minHeap)
{
	return (minHeap->size == 1);
}
struct MinHNode *extractMin(struct MinH *minHeap)
{
	struct MinHNode *temp = minHeap->array[0];
	minHeap->array[0] = minHeap->array[minHeap->size - 1];
	--minHeap->size;
	minHeapify(minHeap, 0);
	return temp;
}
void insertMinHeap(struct MinH *minHeap, struct MinHNode *minHeapNode)
{
	++minHeap->size;
	int i = minHeap->size - 1;
	while (i && minHeapNode->freq < minHeap->array[(i - 1) / 2]->freq)
	{
		minHeap->array[i] = minHeap->array[(i - 1) / 2];
		i = (i - 1) / 2;
	}
	minHeap->array[i] = minHeapNode;
}

void buildMinHeap(struct MinH *minHeap)
{
	int n = minHeap->size - 1;
	int i;
	for (i = (n - 1) / 2; i >= 0; --i)
		minHeapify(minHeap, i);
}

int isLeaf(struct MinHNode *root)
{
	return !(root->left) && !(root->right);
}


struct MinH *createAndBuildMinHeap(char item[], int freq[], int size)
```

```
{
	struct MinH *minHeap = createMinH(size);
	for (int i = 0; i < size; ++i)
		minHeap->array[i] = newNode(item[i], freq[i]);
	minHeap->size = size;
	buildMinHeap(minHeap);
	return minHeap;
}

struct MinHNode *buildHfTree(char item[], int freq[], int size)
{
	struct MinHNode *left, *right, *top;
	struct MinH *minHeap = createAndBuildMinHeap(item, freq, size);
	while (!checkSizeOne(minHeap))
	{
		left = extractMin(minHeap);
		right = extractMin(minHeap);
		top = newNode('$', left->freq + right->freq);
		top->left = left;
		top->right = right;
		insertMinHeap(minHeap, top);
	}
	return extractMin(minHeap);
}

void printHCodes(struct MinHNode *root, int arr[], int top)
{
	if (root->left)
	{
		arr[top] = 0;
		printHCodes(root->left, arr, top + 1);
	}
	if (root->right)
	{
		arr[top] = 1;
		printHCodes(root->right, arr, top + 1);
	}
	if (isLeaf(root))
	{
		cout << root->item << " | ";
		printArray(arr, top);
	}
}

void HuffmanCodes(char item[], int freq[], int size)
{
	struct MinHNode *root = buildHfTree(item, freq, size);
	int arr[MAX_TREE_HT], top = 0;
	printHCodes(root, arr, top);
}
```

```
void printArray(int arr[], int n)
{
	int i;
	for (i = 0; i < n; ++i)
		cout << arr[i];
	cout << "\n";
}

int main()
{
	int n;
	clock_t start, end;
	int sum=0;
	cout<<"Enter the number of elements: ";
	cin>>n;
	char arr[n];
	int freq[n];
	for(int i=0;i<n;i++)
	{
		cout<<"Enter the character "<< i << ": ";
		cin>>arr[i];
	}
	for(int i=0;i<n;i++)
	{
		cout<<"Enter the frequency of element "<< i <<"->";
		cin>>freq[i];
	}
	for(int i=0;i<n;i++)
	sum+=freq[i];

	cout<<"Sum is: "<<sum <<"\n";
	start = clock();
	if(sum == 10 || sum == 100)
	{
		cout << "Huffman code \n";
		HuffmanCodes(arr, freq, n);
	}
	else
	{
		if(sum <10)
			cout<<"Sum of frequencies is less than 10!";
		else if(sum > 100)
			cout<<"Sum of frequencies is greater than 100!";
		else if(sum > 10 && sum < 100)
			cout<<"Sum of frequencies is greater than 10 or less than 100!";

	}
	end = clock();
	double time_taken = double(end - start) / double(CLOCKS_PER_SEC);
	cout << "Time taken by program is : " << fixed
		<< time_taken << setprecision(5)<<endl;
```

```
    return 0;
}
```


PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
abhinav@jarvis:~/clg_3/ada$ ./a.out
Enter the number of elements: 4
Enter the character 0: a
Enter the character 1: b
Enter the character 2: c
Enter the character 3: d
Enter the frequency of element 0->3
Enter the frequency of element 1->4
Enter the frequency of element 2->2
Enter the frequency of element 3->1
Sum is: 10
Huffman code
b | 0
a | 10
d | 110
c | 111
Time taken by program is : 0.000042
abhinav@jarvis:~/clg_3/ada$

Date :
Page No. :

EXPERIMENT 8

Aim

To implement Dijikstra's algorithm & analyse the time complexity

Theory

Dijikstra's algorithm finds the shortest path from a single source node by dividing a set of nodes that have min. distances from the source.
It uses the greedy strategy

Algorithm:

```
DIJIKSTRA (G,w,s)
  INITIALISE-SINGLE-SOURCE (G,s)
    S = ϕ
    Q = G.V
    while Q ≠ ϕ
    u = EXTRACT_MIN(Q)
    S = S U {u}
    for each vertex v ∈ G.
      RELAX (u,v,w)

INITIALISE-SINGLE_SOURCE (G,s)
  for each vertex v ∈ G.V
      v.d = ∞
      v.π = NIL
  s.d = 0
```

Date :
Page No. :

```
RELAX(u,v,w)
    if v.d > u.d + w(u,v)
        v.d = u.d + w(u,v)
        v.π = u
```

Complexity Analysis:

Building the priority queue takes $O(V)$ time. Once the queue is constructed, the while loop is executed once for every vertex since vertex are all added to the beginning & only removed after that within that loop

Worst case complexity = $O(E + V \log V)$

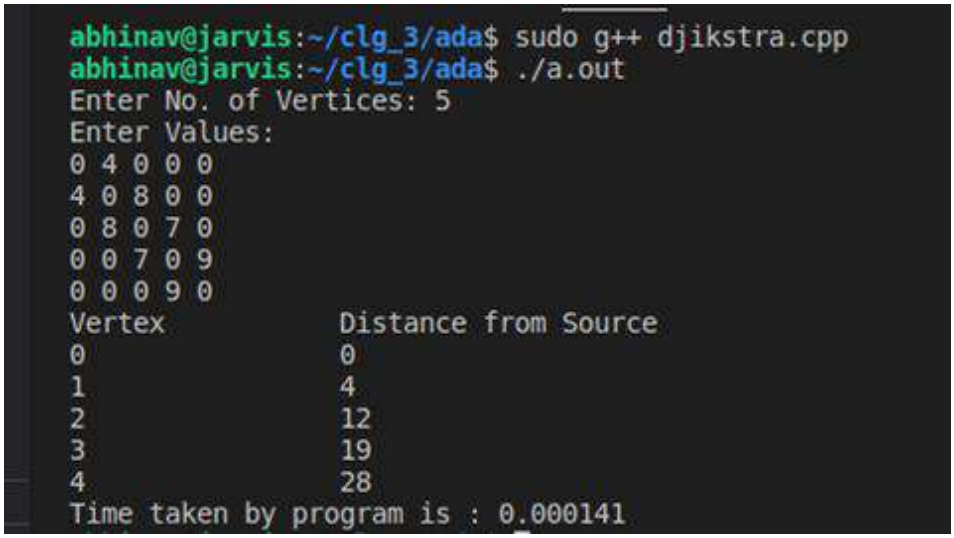
Average case complexity = $O(E + V \log V)$

Best case complexity = $O(E + V \log V)$

Source Code

```
#include <bits/stdc++.h>
using namespace std;

int V =9;

int minDistance(int dist[], bool sptSet[])
{
	int min = INT_MAX, min_index;
	for (int v = 0; v < V; v++)
		if (sptSet[v] == false && dist[v] <= min)
			min = dist[v], min_index = v;
			return min_index;
}

void printSolution(int dist[])
{
	printf("Vertex \t\t Distance from Source\n");
	for (int i = 0; i < V; i++)
		printf("%d \t\t %d\n", i, dist[i]);
}
void dijkstra(int graph[10][10], int src)
{
	int dist[V];
	bool sptSet[V];
	for (int i = 0; i < V; i++)
		dist[i] = INT_MAX, sptSet[i] = false;
	dist[src] = 0;

	for (int count = 0; count < V - 1; count++)
	{
		int u = minDistance(dist, sptSet);
		sptSet[u] = true;
		for (int v = 0; v < V; v++)
			if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX
					&& dist[u] + graph[u][v] < dist[v])
				dist[v] = dist[u] + graph[u][v];
	}
	printSolution(dist);
}

int main()
{
	int graph[10][10];
	clock_t start, end;
	cout<<"Enter No. of Vertices: ";
	cin>>V;
```

```
	cout<<"Enter Values: ";
	for(int i=0;i<V;i++)
		for(int j=0;j<V;j++)
			cin>>graph[i][j];
	start = clock();
	dijkstra(graph, 0);
	end = clock();
	double time_taken = double(end - start)/double(CLOCKS_PER_SEC);
	cout << "Time taken by program is : " << fixed
		<< time_taken << setprecision(10)<<"\n";


	return 0;
}
```



Date :
Page No. :

EXPERIMENT 9

Aim

To implement Bellmann Ford Algorithm & analyse the time complexity

Theory

It solves the single shortest path problem in which edge weight may be negative but no negative edge cycle exists.

The algorithm works correctly when some of the edges of the direct graph G may have negative weight. When there are no cycles of negative weight, then we can find out the shortest path b/w source & destination.

Algorithm:

```
BELLMAN-FORD (G,W,S)
INITIALIZE-SINGLE-SOURCE (G,S)
 for i=1 to |G.V|-1
     for each edge (u,v) in G.E
         RELAX(u,v,w)
     for each edge (u,v) ∈ G.E
         if v.d > u.d + w(u,v)
             return FALSE
 return TRUE
```

Date :
Page No. :

INITIALIZE-SINGLE-SOURCE (G, s)

for each vertex $v \in G.V$

$v.d = \infty$

$v.\pi = NIL$

$s.d = 0$

RELAX (u, v, w)

if $v.d > u.d + w(u,v)$

$v.d = u.d + w(u,v)$

$v.\pi = u$

Complexity Analysis:

Bellman-Ford makes $|E|$ relations for every iteration & therefore are $|V|-1$ iteration. Therefore, the worst case scenarios is that the algo runs in $O(|V| \cdot |E|)$ time. However, in some scenario, the no. of iterations can be much lower

Best case: $O(|E|)$

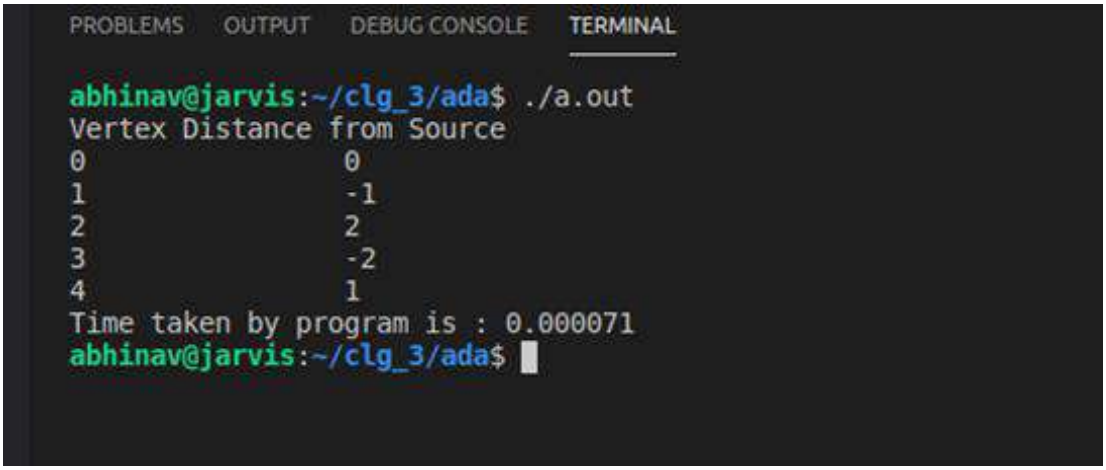
Thus, complexity of Bellman Ford is $O(|V||E|)$

Source Code

```
#include <limits.h>
#include <bits/stdc++.h>
#include<chrono>
using namespace std;
using namespace std::chrono;
struct Edge
{
	int src, dest, weight;
};

struct Graph
{
	int V, E;
	struct Edge* edge;
};

struct Graph* createGraph(int V, int E)
{
	struct Graph* graph = new Graph;
	graph->V = V;
	graph->E = E;
	graph->edge = new Edge[E];
	return graph;
}

void printArr(int dist[], int n)
{
	printf("Vertex Distance from Source\n");
	for (int i = 0; i < n; ++i)
		printf("%d \t\t %d\n", i, dist[i]);
}

void BellmanFord(struct Graph* graph, int src)
{
	int V = graph->V;
	int E = graph->E;
	int dist[V];
	for (int i = 0; i < V; i++)
		dist[i] = INT_MAX;
	dist[src] = 0;
	for (int i = 1; i <= V - 1; i++)
	{
		for (int j = 0; j < E; j++)
		{
			int u = graph->edge[j].src;
			int v = graph->edge[j].dest;
```

```
                int weight = graph->edge[j].weight;
                if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
                    dist[v] = dist[u] + weight;
            }
        }
        for (int i = 0; i < E; i++)
        {
            int u = graph->edge[i].src;
            int v = graph->edge[i].dest;
            int weight = graph->edge[i].weight;
            if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
            {
                printf("Graph contains negative weight cycle");
                return;
            }
        }
        printArr(dist, V);
        return;
}

int main()
{
        int V = 5;
        int E = 8;
        clock_t start, end;
        struct Graph* graph = createGraph(V, E);
        graph->edge[0].src = 0;
        graph->edge[0].dest = 1;
        graph->edge[0].weight = -1;
        graph->edge[1].src = 0;
        graph->edge[1].dest = 2;
        graph->edge[1].weight = 4;
        graph->edge[2].src = 1;
        graph->edge[2].dest = 2;
        graph->edge[2].weight = 3;
        graph->edge[3].src = 1;
        graph->edge[3].dest = 3;
        graph->edge[3].weight = 2;
        graph->edge[4].src = 1;
        graph->edge[4].dest = 4;
        graph->edge[4].weight = 2;
        graph->edge[5].src = 3;
        graph->edge[5].dest = 2;
        graph->edge[5].weight = 5;
        graph->edge[6].src = 3;
        graph->edge[6].dest = 1;
        graph->edge[6].weight = 1;
        graph->edge[7].src = 4;
```

```
    graph->edge[7].dest = 3;
    graph->edge[7].weight = -3;
    start = clock();
    BellmanFord(graph, 0);
    end = clock();
    double time_taken = double(end - start) / double(CLOCKS_PER_SEC);
    cout << "Time taken by program is : " << fixed
         << time_taken << setprecision(5)<<endl;
    return 0;
}
```



Date :
Page No. :

Content beyond Syllabus !

Aim

WAP to multiply two matrices using Strassen's matrix multiplication

Theory

Algorithm:

Strassen (n, a, b, c)

If n = threshold then compute

c = a * b is a conventional matrix

else

Partition a into four submatrices $a_{11}, a_{12}, a_{21}, a_{22}$

Partition b into four submatrices $b_{11}, b_{12}, b_{21}, b_{22}$

Strassen $(n/2, a_{11}+a_{22}, b_{11}+b_{22}, d_1)$

Strassen $(n/2, a_{21}+a_{22}, b_{11}, d_2)$

Strassen $(n/2, a_{11}, b_{12}-b_{22}, d_3)$

Strassen $(n/2, a_{22}, b_{21}-b_{11}, d_4)$

Strassen $(n/2, a_{11}+a_{12}, b_{22}, d_5)$

Strassen $(n/2, a_{21}-a_{11}, b_{11}+b_{22}, d_6)$

Strassen $(n/2, a_{12}-a_{22}, b_{21}+b_{22}, d_7)$

$c_{11} = d_1 + d_4 - d_5 + d_7$ $\quad$ $c_{12} = d_3 + d_5$

$c_{21} = d_2 + d_4$ $\quad$ $c_{22} = d_1 + d_3 - d_2 - d_6$

return c

Date :
Page No. :

Complexity Analysis:

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ 7T(n/2) + O(n^2) & \text{if } n > 1 \end{cases}$$

Using Master's theorem

$$T = O(n^{\log 7})$$
$$= O(n^{2.807})$$

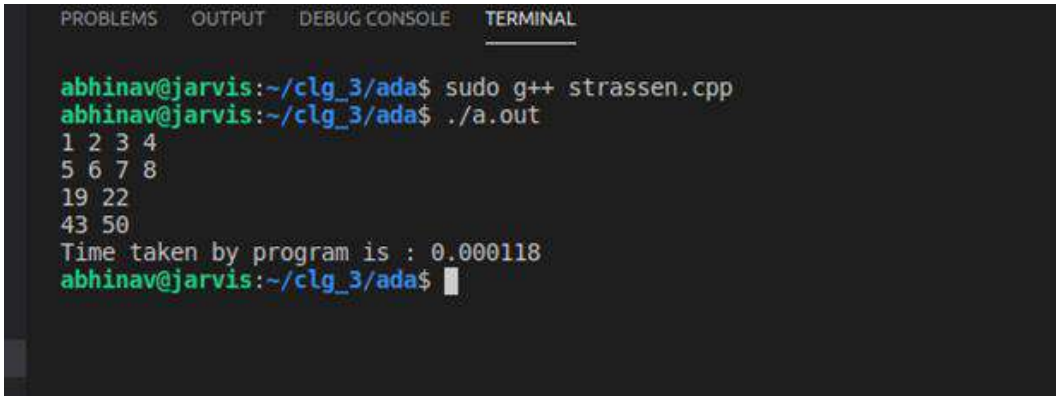
Technique used:

Uses divide & conquer technique. Problem is rapidly divided into subproblems until the smallest subproblem is reached.

Source Code

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
	int a[2][2],b[2][2];
	clock_t start, end;

	for(int i = 0; i < 2; i++)
		for(int j = 0; j < 2; j++)
			cin>>a[i][j];

	for(int i = 0; i < 2; i++)
		for(int j = 0; j < 2; j++)
			cin>>b[i][j];
	start = clock();
	int p1 = (a[0][0]+a[1][1])*(b[0][0]+b[1][1]);
	int p2 = (a[1][0] + a[1][1])*b[0][0];
	int p3 = a[0][0]*(b[0][1]-b[1][1]);
	int p4 = a[1][1]*(b[1][0] - b[0][0]);
	int p5 = (a[0][0] + a[0][1])*b[1][1];
	int p6 = (a[1][0] - a[0][0])*(b[0][0] + b[0][1]);
	int p7 = (a[0][1] - a[1][1])*(b[1][0] + b[1][1]);


	cout<<p1+p4-p5+p7<<" "<<p3+p5<<"\n";
	cout<<p2+p4<<" "<<p1+p3-p2+p6<<"\n";
	end = clock();
	double time_taken = double(end - start) / double(CLOCKS_PER_SEC);
	cout << "Time taken by program is : " << fixed
		<< time_taken << setprecision(5)<<endl;

}
```

Output



Date:
Page No.:

CONTENT BEYOND SYLLABUS - 2

Aim

Find the k^{th} largest element in an unsorted array & analyse its time complexity.

Theory

Algorithm

(I) Using Merge / Heap Sort

A simple solution is to sort the given array using a sorting algorithm like Merge sort, heap sort etc. & return the element at index k-1 in the sorted array

MergeSort(arr[], l, r)
→ Find the middle point to divide the array into two halves. m = (l+r)/2
→ Call merge sort for first half.
MergeSort(arr, l, m)
→ Call sort for second half
MergeSort(arr, m+1, r)
→ Merge the two halves sorted earlier
Merge(arr, l, m, r)
→ Return the $(k-1)^{th}$ element

Time Complexity

MergeSort : $2T(n/2) + O(n)$
$= O(n \log n)$ (Master's Theorem)

Date :
Page No. :

Space Complexity
$O(n)$

(II) Using Max/Min Heap
→ Build a min heap MH of the first k elements (arr[0] to arr[k-1]) of the given array - $O(k)$
→ For each element, after the k^{th} element, compare with root of MH
- If element is greater than root, make it root & call heapify for MH

→ Finally, MH has k largest elements & root of the MH is the k^{th} largest element
→ Put the value $k=1$ in the main function

Time Complexity
1st step = $O(k)$
IInd step = $O((n-k)*\log k)$
Total Time = $O(k + (n-k)*\log k)$

CODE:

1. Approach 1 Using Sorting:

```
#include<bits/stdc++.h>
#include <chrono>

using namespace std;
using namespace std::chrono;

int k_largest(int arr[], int n, int k){
 sort(arr,arr+n);
 return arr[n-k];
}
int main(){
 int n, k;
 cout << "Enter Number of elements: ";
 cin >> n;
 cout << "Enter elements: ";
 int arr[n];
 for(int i=0;i<n;i++){
 cin >> arr[i];
 }
 cout << "Enter value of k: ";
 cin >> k;
 auto start = steady_clock::now();
 int ans = k_largest(arr,n,k);
 cout << "Kth Largest element is: " << ans;
 auto stop = steady_clock::now();
 auto duration = duration_cast<nanoseconds>(stop - start);
 cout << "\nTime taken by function: "<< duration.count() << " nanoseconds" << endl;
 return 0;
}
```

2. A) Approach 2 Using Min Heap:

```
#include<bits/stdc++.h>
#include <chrono>

using namespace std;
using namespace std::chrono;

class MinHeap{
 int *harr;
```

```
 int heap_size;
 int capacity;
 public:
 MinHeap(int a[], int size);
 void minHeapify(int i);

 int parent(int i){
 return (i-1)/2;
 }
 int left(int i){return 2*i+1;}
 int right(int i){return 2*i+2;}
 int getMin(){return harr[0];}
 void replaceMin(int x){
 harr[0] = x;
 minHeapify(0);
 }
};
MinHeap::MinHeap(int a[], int size){
 heap_size = size;
 harr = a;
 int i = (heap_size-1)/2;
 while(i>=0){
 minHeapify(i);
 i--;
 }
}
void MinHeap::minHeapify(int i){
 int l = left(i);
 int r = right(i);
 int smallest = i;
 if(l<heap_size && harr[l] < harr[i]){
 smallest = l;
 }
 if(r<heap_size && harr[r] < harr[smallest]){
 smallest = r;
 }
 if(smallest !=i){
 swap(harr[i],harr[smallest]);
 minHeapify(smallest);
 }
}
int KthLargest(int arr[], int n, int k){
 MinHeap mh(arr,k);
 for(int i=k;i<n;i++){
 if(arr[i]>mh.getMin()){
 mh.replaceMin(arr[i]);
 }
```

```
 }
 return mh.getMin();
}
int main(){
 int n, k;
 cout << "Enter Number of elements: ";
 cin >> n;
 cout << "Enter elements: ";
 int arr[n];
 for(int i=0;i<n;i++){
 cin >> arr[i];
 }
 cout << "Enter value of k: ";
 cin >> k;
 auto start = steady_clock::now();
 int ans = KthLargest(arr,n,k);
 cout << "Kth Largest element is: " << ans;
 auto stop = steady_clock::now();
 auto duration = duration_cast<nanoseconds>(stop - start);
 cout << "\nTime taken by function: "<< duration.count() << " nanoseconds" << endl;
 return 0;
}
```

B) Using Max Heap :

```
#include<bits/stdc++.h>
#include <chrono>

using namespace std;
using namespace std::chrono;

class MaxHeap{
int *harr;
int heap_size;
int capacity;
public:
MaxHeap(int a[], int size);
void maxHeapify(int i);
int parent(int i){
return (i-1)/2;
}
int left(int i){return 2*i+1;}
int right(int i){return 2*i+2;}
int extract_max();
int getMax(){return harr[0];}
};
MaxHeap::MaxHeap(int a[], int size){
heap_size = size;
harr = a;
```

```
int i = (heap_size-1)/2;
while(i>=0){
maxHeapify(i);
i--;
}
}
void MaxHeap::maxHeapify(int i){
int l = left(i);
int r = right(i);
int largest = i;
if(l<heap_size && harr[l] > harr[i]){
largest = l;
}
if(r<heap_size && harr[r] > harr[largest]){
largest = r;
}
if(largest !=i){
swap(harr[i],harr[largest]);
maxHeapify(largest);
}
}
int MaxHeap::extract_max(){
if(heap_size == 0){
return INT_MAX;
}
int root = harr[0];
if(heap_size > 1){
harr[0] = harr[heap_size-1];
maxHeapify(0);
}
heap_size--;
return root;
}
int KthLargest(int arr[], int n, int k){
MaxHeap mh(arr, n);
for(int i=0;i<k-1;i++){
mh.extract_max();
}
return mh.getMax();
}
int main(){
cout << "Enter no. of array elements: ";
int n;
cin >> n;
cout << "Enter elements: ";
int arr[n];
for(int i=0;i<n;i++){
```

```
cin >> arr[i];
}
int k;
cout << "Enter k to find the kth largest element: ";
cin >> k;
auto start = steady_clock::now();
int ans = KthLargest(arr,n,k);
cout << "Kth Largest element is array is: " << ans;
auto stop = steady_clock::now();
auto duration = duration_cast<nanoseconds>(stop - start);
cout << "\nTime taken by function: "<< duration.count() << " nanoseconds" << endl;
return 0;
}
```

OUTPUT:

1. Approach 1 Using Sorting:

```
Enter Number of elements: 5
Enter elements: 2 1 6 4 5
Enter value of k: 2
Kth Largest element is: 5
Time taken by function: 1148000 nanoseconds
```

2. A) Approach 2 Using Min Heap:

```
Enter Number of elements: 5
Enter elements: 2 1 6 4 5
Enter value of k: 2
Kth Largest element is: 5
Time taken by function: 1158100 nanoseconds
```

B) Using Max Heap:

```
Enter no. of array elements: 5
Enter elements: 2 1 6 4 5
Enter k to find the kth largest element: 2
Kth Largest element is array is: 5
Time taken by function: 1264400 nanoseconds
```