DAA Lab

Design & Analysis of Algorithms

All 27 record programs · Algorithm, program, output and complexity

CONTENTS

01 Linear search 2
02 Binary search 4
03 Bubble sort 6
04 Selection sort 8
05 Insertion sort 10
06 Radix sort 12
07 Bucket sort 14
08 Quick sort 16
09 Merge sort 18
10 Fibonacci (recursive) 20
11 Fibonacci (DP) 22
12 Matrix chain multiplication 24
13 0/1 knapsack 26
14 Longest common subsequence 28
15 Fractional knapsack 31
16 Job sequencing with deadlines 34
17 Huffman coding 37
18 N-Queen 40
19 Travelling salesman (DP) 42
20 Breadth-first search 44
21 Depth-first search 46
22 Job assignment (branch & bound) 48
23 Dijkstra 52
24 Floyd–Warshall 56
25 Bellman–Ford 58
26 Prim's MST 61
27 Kruskal's MST 64

PROGRAM 01

Linear search

Aim: Write a program in c to search an element using linear search and find its execution time using the time function.

ALGORITHM

```
Algorithm: LinearSearch(A, n, x)

1. Start
2. For i <- 0 to (n-1) do
3.        if x == A[i] then
4.              print "Element Found"
5.              exit
6. If (i==n) then
7.         print "Element Not Found"
8. Stop
```

PROGRAM (C)

```
#include<stdio.h>
#include<time.h>

int main() {
    int n,i,x,count=0;
    printf("Enter the number of elements:");
    scanf("%d",&n);
    int a[n];
    printf("Enter the elements:");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);
    printf("Enter the element to search:");
    scanf("%d",&x);

    clock_t start=clock();
    for(i=0;i<n;i++) {
        count++;
        if(x==a[i])
            break;
    }
    clock_t end=clock();
    double time_taken=(double)(end-start)/CLOCKS_PER_SEC;

    if(i==n)
        printf("\n%d is not found",x);
    else
        printf("\n%d is found at position %d",x,i+1);
    printf("\nNumber of comparisons:%d",count);
    printf("\nTime taken:%f seconds",time_taken);
    printf("\nTime complexity: best case O(1), worst case O(n)");
    printf("\nSpace complexity: O(1)");
    printf("\n");
    return 0;
}
```

INPUT

```
6
14 27 8 35 19 42
35
```

OUTPUT

```
Enter the number of elements:6
Enter the elements:14 27 8 35 19 42
Enter the element to search:35

35 is found at position 4
Number of comparisons:4
Time taken:0.000003 seconds
Time complexity: best case O(1), worst case O(n)
Space complexity: O(1)
```

COMPLEXITY

```
Time complexity:  best case O(1), worst case O(n)
Space complexity: O(1)
Best: O(1)   Average: O(n)   Worst: O(n)   Paradigm: Brute force
```

PROGRAM 02

Binary search

Aim: Write a program in c to search an element using binary search and find its execution time using the time function.

ALGORITHM

```
Algorithm: BinarySearch(A, n, x)

1. Start
2. low <- 0, high <- (n-1)
3. While (low <= high) do
4.        mid <- (low+high)/2
5.        if x == A[mid] then
6.              print "Element Found"
7.              exit
8.        else if A[mid] < x then
9.              low <- mid+1
10.       else
11.             high <- mid-1
12. If (low > high) then
13.        print "Element Not Found"
14. Stop
```

PROGRAM (C)

```
#include<stdio.h>
#include<time.h>

int main() {
    int n,i,x,low,high,mid,count=0;
    printf("Enter the number of elements:");
    scanf("%d",&n);
    int a[n];
    printf("Enter the elements in ascending order:");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);
    printf("Enter the element to search:");
    scanf("%d",&x);

    clock_t start=clock();
    low=0;
    high=n-1;
    while(low<=high) {
        count++;
        mid=(low+high)/2;
        if(x==a[mid])
            break;
        else if(a[mid]<x)
            low=mid+1;
        else
            high=mid-1;
    }
    clock_t end=clock();
    double time_taken=(double)(end-start)/CLOCKS_PER_SEC;

    if(low>high)
        printf("\n%d is not found",x);
    else
        printf("\n%d is found at position %d",x,mid+1);
    printf("\nNumber of comparisons:%d",count);
    printf("\nTime taken:%f seconds",time_taken);
    printf("\nTime complexity: best case O(1), worst case O(log n)");
    printf("\nSpace complexity: O(1)");
```

```
    printf("\n");
    return 0;
}
```

INPUT

```
7
8 14 19 27 35 42 56
42
```

OUTPUT

```
Enter the number of elements:7
Enter the elements in ascending order:8 14 19 27 35 42 56
Enter the element to search:42

42 is found at position 6
Number of comparisons:2
Time taken:0.000005 seconds
Time complexity: best case O(1), worst case O(log n)
Space complexity: O(1)
```

COMPLEXITY

```
Time complexity:  best case O(1), worst case O(log n)
Space complexity: O(1)
Best: O(1)   Average: O(log n)   Worst: O(log n)   Paradigm: Divide & conquer
```

PROGRAM 03

Bubble sort

Aim: Write a program in c to sort the elements using bubble sort and find its execution time using the time function.

ALGORITHM

```
Algorithm: BubbleSort(A, n)

1. Start
2. For i <- 0 to (n-2) do
3.        For j <- 0 to (n-i-2) do
4.              if A[j] > A[j+1] then
5.                    temp <- A[j]
6.                    A[j] <- A[j+1]
7.                    A[j+1] <- temp
8. Print the sorted array A
9. Stop
```

PROGRAM (C)

```c
#include<stdio.h>
#include<time.h>

int main() {
    int n,i,j,temp,count=0;
    printf("Enter the number of elements:");
    scanf("%d",&n);
    int a[n];
    printf("Enter the elements:");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);

    clock_t start=clock();
    for(i=0;i<n-1;i++) {
        for(j=0;j<n-i-1;j++) {
            count++;
            if(a[j]>a[j+1]) {
                temp=a[j];
                a[j]=a[j+1];
                a[j+1]=temp;
            }
        }
    }
    clock_t end=clock();
    double time_taken=(double)(end-start)/CLOCKS_PER_SEC;

    printf("\nAfter bubble sorting:");
    for(i=0;i<n;i++)
        printf("\t%d",a[i]);
    printf("\nNumber of comparisons:%d",count);
    printf("\nTime taken:%f seconds",time_taken);
    printf("\nTime complexity: O(n^2)");
    printf("\nSpace complexity: O(1)");
    printf("\n");
    return 0;
}
```

INPUT

```
6
42 8 27 14 35 19
```

OUTPUT

```
Enter the number of elements:6
Enter the elements:42 8 27 14 35 19

After bubble sorting:   8       14      19      27      35      42
Number of comparisons:15
Time taken:0.000005 seconds
Time complexity: O(n^2)
Space complexity: O(1)
```

COMPLEXITY

```
Time complexity:  O(n^2)
Space complexity: O(1)
Best: O(n²)   Average: O(n²)   Worst: O(n²)   Paradigm: Brute force
```

PROGRAM 04

Selection sort

Aim: Write a program in c to sort the elements using selection sort and find its execution time using the time function.

ALGORITHM

```
Algorithm: SelectionSort(A, n)

1. Start
2. For i <- 0 to (n-2) do
3.        min <- i
4.        For j <- i+1 to (n-1) do
5.              if A[j] < A[min] then
6.                    min <- j
7.        temp <- A[i]
8.        A[i] <- A[min]
9.        A[min] <- temp
10. Print the sorted array A
11. Stop
```

PROGRAM (C)

```
#include<stdio.h>
#include<time.h>

int main() {
    int n,i,j,min,temp,count=0;
    printf("Enter the number of elements:");
    scanf("%d",&n);
    int a[n];
    printf("Enter the elements:");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);

    clock_t start=clock();
    for(i=0;i<n-1;i++) {
        min=i;
        for(j=i+1;j<n;j++) {
            count++;
            if(a[j]<a[min])
                min=j;
        }
        temp=a[i];
        a[i]=a[min];
        a[min]=temp;
    }
    clock_t end=clock();
    double time_taken=(double)(end-start)/CLOCKS_PER_SEC;

    printf("\nAfter selection sorting:");
    for(i=0;i<n;i++)
        printf("\t%d",a[i]);
    printf("\nNumber of comparisons:%d",count);
    printf("\nTime taken:%f seconds",time_taken);
    printf("\nTime complexity: O(n^2)");
    printf("\nSpace complexity: O(1)");
    printf("\n");
    return 0;
}
```

INPUT

```
6
42 8 27 14 35 19
```

OUTPUT

```
Enter the number of elements:6
Enter the elements:42 8 27 14 35 19

After selection sorting:        8       14      19      27      35      42
Number of comparisons:15
Time taken:0.000006 seconds
Time complexity: O(n^2)
Space complexity: O(1)
```

COMPLEXITY

```
Time complexity:  O(n^2)
Space complexity: O(1)
Best: O(n²)   Average: O(n²)   Worst: O(n²)   Paradigm: Brute force
```

PROGRAM 05

Insertion sort

Aim: Write a program in c to sort the elements using insertion sort and find its execution time using the time function.

ALGORITHM

```
Algorithm: InsertionSort(A, n)

1. Start
2. For i <- 1 to (n-1) do
3.        key <- A[i]
4.        j <- i-1
5.        While (j >= 0 and A[j] > key) do
6.              A[j+1] <- A[j]
7.              j <- j-1
8.        A[j+1] <- key
9. Print the sorted array A
10. Stop
```

PROGRAM (C)

```
#include<stdio.h>
#include<time.h>

int main() {
    int n,i,j,key,count=0;
    printf("Enter the number of elements:");
    scanf("%d",&n);
    int a[n];
    printf("Enter the elements:");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);

    clock_t start=clock();
    for(i=1;i<n;i++) {
        key=a[i];
        j=i-1;
        while(j>=0) {
            count++;
            if(a[j]>key) {
                a[j+1]=a[j];
                j--;
            }
            else
                break;
        }
        a[j+1]=key;
    }
    clock_t end=clock();
    double time_taken=(double)(end-start)/CLOCKS_PER_SEC;

    printf("\nAfter insertion sorting:");
    for(i=0;i<n;i++)
        printf("\t%d",a[i]);
    printf("\nNumber of comparisons:%d",count);
    printf("\nTime taken:%f seconds",time_taken);
    printf("\nTime complexity: best case O(n), worst case O(n^2)");
    printf("\nSpace complexity: O(1)");
    printf("\n");
    return 0;
}
```

INPUT

```
6
42 8 27 14 35 19
```

OUTPUT

```
Enter the number of elements:6
Enter the elements:42 8 27 14 35 19

After insertion sorting:        8       14      19      27      35      42
Number of comparisons:12
Time taken:0.000005 seconds
Time complexity: best case O(n), worst case O(n^2)
Space complexity: O(1)
```

COMPLEXITY

```
Time complexity:  best case O(n), worst case O(n^2)
Space complexity: O(1)
Best: O(n)   Average: O(n²)   Worst: O(n²)   Paradigm: Incremental
```

PROGRAM 06

Radix sort

Aim: Write a program in c to sort the elements using radix sort and find its execution time using the time function.

ALGORITHM

```
Algorithm: RadixSort(A, n)

1. Start
2. max <- largest element of A
3. exp <- 1
4. While (max/exp > 0) do
5.        CountSort(A, n, exp)
6.        exp <- exp*10
7. Print the sorted array A
8. Stop

Algorithm: CountSort(A, n, exp)

1. For i <- 0 to 9 do
2.        bucket[i] <- 0
3. For i <- 0 to (n-1) do
4.        d <- (A[i]/exp) mod 10
5.        bucket[d] <- bucket[d]+1
6. For i <- 1 to 9 do
7.        bucket[i] <- bucket[i]+bucket[i-1]
8. For i <- (n-1) down to 0 do
9.        d <- (A[i]/exp) mod 10
10.       bucket[d] <- bucket[d]-1
11.       output[bucket[d]] <- A[i]
12. For i <- 0 to (n-1) do
13.       A[i] <- output[i]
14. Return
```

PROGRAM (C)

```
#include<stdio.h>
#include<time.h>

int count=0;

void countSort(int a[],int n,int exp) {
    int output[n],bucket[10],i;

    for(i=0;i<10;i++)
        bucket[i]=0;
    for(i=0;i<n;i++) {
        count++;
        bucket[(a[i]/exp)%10]++;
    }
    for(i=1;i<10;i++)
        bucket[i]=bucket[i]+bucket[i-1];
    for(i=n-1;i>=0;i--) {
        count++;
        output[--bucket[(a[i]/exp)%10]]=a[i];
    }
    for(i=0;i<n;i++)
        a[i]=output[i];
}

int main() {
    int n,i,max,exp;
    printf("Enter the number of elements:");
```

```
    scanf("%d",&n);
    int a[n];
    printf("Enter the elements:");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);

    clock_t start=clock();
    max=a[0];
    for(i=1;i<n;i++)
        if(a[i]>max)
            max=a[i];
    for(exp=1;max/exp>0;exp=exp*10)
        countSort(a,n,exp);
    clock_t end=clock();
    double time_taken=(double)(end-start)/CLOCKS_PER_SEC;

    printf("\nAfter radix sorting:");
    for(i=0;i<n;i++)
        printf("\t%d",a[i]);
    printf("\nNumber of operations:%d",count);
    printf("\nTime taken:%f seconds",time_taken);
    printf("\nTime complexity: O(d*n)");
    printf("\nSpace complexity: O(n+k)");
    printf("\n");
    return 0;
}
```

INPUT

```
6
42 8 27 14 35 19
```

OUTPUT

```
Enter the number of elements:6
Enter the elements:42 8 27 14 35 19

After radix sorting:    8       14      19      27      35      42
Number of operations:24
Time taken:0.000009 seconds
Time complexity: O(d*n)
Space complexity: O(n+k)
```

COMPLEXITY

```
Time complexity:  O(d*n)
Space complexity: O(n+k)
Best: O(d·n)   Average: O(d·n)   Worst: O(d·n)   Paradigm: Non-comparison
```

PROGRAM 07

Bucket sort

Aim: Write a program in c to sort the elements using bucket sort and find its execution time using the time function.

ALGORITHM

```
Algorithm: BucketSort(A, n)

1. Start
2. max <- largest element of A
3. For i <- 0 to (n-1) do
4.        size[i] <- 0
5. For i <- 0 to (n-1) do
6.        index <- (A[i]*n)/(max+1)
7.        bucket[index][size[index]] <- A[i]
8.        size[index] <- size[index]+1
9. For i <- 0 to (n-1) do
10.       sort bucket[i] using insertion sort
11. k <- 0
12. For i <- 0 to (n-1) do
13.       For j <- 0 to (size[i]-1) do
14.             A[k] <- bucket[i][j]
15.             k <- k+1
16. Print the sorted array A
17. Stop
```

PROGRAM (C)

```
#include<stdio.h>
#include<time.h>

int main() {
    int n,i,j,k,key,index,max,count=0;
    printf("Enter the number of elements:");
    scanf("%d",&n);
    int a[n];
    printf("Enter the elements:");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);

    clock_t start=clock();
    max=a[0];
    for(i=1;i<n;i++)
        if(a[i]>max)
            max=a[i];

    int bucket[n][n],size[n];
    for(i=0;i<n;i++)
        size[i]=0;

    for(i=0;i<n;i++) {
        index=(a[i]*n)/(max+1);
        bucket[index][size[index]++]=a[i];
    }

    for(i=0;i<n;i++) {
        for(j=1;j<size[i];j++) {
            key=bucket[i][j];
            k=j-1;
            while(k>=0) {
                count++;
                if(bucket[i][k]>key) {
                    bucket[i][k+1]=bucket[i][k];
```

```
                    k--;
                }
                else
                    break;
            }
            bucket[i][k+1]=key;
        }
    }

    k=0;
    for(i=0;i<n;i++)
        for(j=0;j<size[i];j++)
            a[k++]=bucket[i][j];
    clock_t end=clock();
    double time_taken=(double)(end-start)/CLOCKS_PER_SEC;

    printf("\nAfter bucket sorting:");
    for(i=0;i<n;i++)
        printf("\t%d",a[i]);
    printf("\nNumber of comparisons:%d",count);
    printf("\nTime taken:%f seconds",time_taken);
    printf("\nTime complexity: average O(n+k), worst O(n^2)");
    printf("\nSpace complexity: O(n^2)");
    printf("\n");
    return 0;
}
```

INPUT

```
6
42 8 27 14 35 19
```

OUTPUT

```
Enter the number of elements:6
Enter the elements:42 8 27 14 35 19

After bucket sorting:   8       14      19      27      35      42
Number of comparisons:1
Time taken:0.000002 seconds
Time complexity: average O(n+k), worst O(n^2)
Space complexity: O(n^2)
```

COMPLEXITY

```
Time complexity:  average O(n+k), worst O(n^2)
Space complexity: O(n^2)
Best: O(n+k)   Average: O(n+k)   Worst: O(n²)   Paradigm: Distribution
```

PROGRAM 08

Quick sort

Aim: Write a program in c to sort the elements using quick sort and find its execution time using the time function.

ALGORITHM

```
Algorithm: QuickSort(A, low, high)

1. Start
2. If (low < high) then
3.        p <- Partition(A, low, high)
4.        QuickSort(A, low, p-1)
5.        QuickSort(A, p+1, high)
6. Stop

Algorithm: Partition(A, low, high)

1. pivot <- A[high]
2. i <- low-1
3. For j <- low to (high-1) do
4.        if A[j] <= pivot then
5.              i <- i+1
6.              swap A[i] and A[j]
7. swap A[i+1] and A[high]
8. Return i+1
```

PROGRAM (C)

```
#include<stdio.h>
#include<time.h>

int count=0;

int partition(int a[],int low,int high) {
    int pivot=a[high];
    int i=low-1,j,temp;

    for(j=low;j<high;j++) {
        count++;
        if(a[j]<=pivot) {
            i++;
            temp=a[i];
            a[i]=a[j];
            a[j]=temp;
        }
    }
    temp=a[i+1];
    a[i+1]=a[high];
    a[high]=temp;
    return i+1;
}

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

int main() {
    int n,i;
    printf("Enter the number of elements:");
```

```
    scanf("%d",&n);
    int a[n];
    printf("Enter the elements:");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);

    clock_t start=clock();
    quickSort(a,0,n-1);
    clock_t end=clock();
    double time_taken=(double)(end-start)/CLOCKS_PER_SEC;

    printf("\nAfter quick sorting:");
    for(i=0;i<n;i++)
        printf("\t%d",a[i]);
    printf("\nNumber of comparisons:%d",count);
    printf("\nTime taken:%f seconds",time_taken);
    printf("\nTime complexity: O(n log n) average, O(n^2) worst");
    printf("\nSpace complexity: O(log n) average, O(n) worst");
    printf("\n");
    return 0;
}
```

INPUT

```
6
42 8 27 14 35 19
```

OUTPUT

```
Enter the number of elements:6
Enter the elements:42 8 27 14 35 19

After quick sorting:    8       14      19      27      35      42
Number of comparisons:9
Time taken:0.000006 seconds
Time complexity: O(n log n) average, O(n^2) worst
Space complexity: O(log n) average, O(n) worst
```

COMPLEXITY

```
Time complexity:  O(n log n) average, O(n^2) worst
Space complexity: O(log n) average, O(n) worst
Best: O(n log n)   Average: O(n log n)   Worst: O(n²)   Paradigm: Divide & conquer
```

PROGRAM 09

Merge sort

Aim: Write a program in c to sort the elements using merge sort and find its execution time using the time function.

ALGORITHM

```
Algorithm: MergeSort(A, low, high)

1. Start
2. If (low < high) then
3.        mid <- (low+high)/2
4.        MergeSort(A, low, mid)
5.        MergeSort(A, mid+1, high)
6.        Merge(A, low, mid, high)
7. Stop

Algorithm: Merge(A, low, mid, high)

1. n1 <- mid-low+1, n2 <- high-mid
2. Copy A[low..mid] into L[0..n1-1] and A[mid+1..high] into R[0..n2-1]
3. i <- 0, j <- 0, k <- low
4. While (i < n1 and j < n2) do
5.        if L[i] <= R[j] then
6.              A[k] <- L[i], i <- i+1
7.        else
8.              A[k] <- R[j], j <- j+1
9.        k <- k+1
10. Copy the remaining elements of L and R into A
11. Return
```

PROGRAM (C)

```c
#include<stdio.h>
#include<time.h>

int count=0;

void merge(int a[],int low,int mid,int high) {
    int n1=mid-low+1,n2=high-mid;
    int left[n1],right[n2];
    int i,j,k;

    for(i=0;i<n1;i++)
        left[i]=a[low+i];
    for(j=0;j<n2;j++)
        right[j]=a[mid+1+j];

    i=0;
    j=0;
    k=low;
    while(i<n1 && j<n2) {
        count++;
        if(left[i]<=right[j])
            a[k++]=left[i++];
        else
            a[k++]=right[j++];
    }
    while(i<n1)
        a[k++]=left[i++];
    while(j<n2)
        a[k++]=right[j++];
}
```

```
void mergeSort(int a[],int low,int high) {
    if(low<high) {
        int mid=(low+high)/2;
        mergeSort(a,low,mid);
        mergeSort(a,mid+1,high);
        merge(a,low,mid,high);
    }
}

int main() {
    int n,i;
    printf("Enter the number of elements:");
    scanf("%d",&n);
    int a[n];
    printf("Enter the elements:");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);

    clock_t start=clock();
    mergeSort(a,0,n-1);
    clock_t end=clock();
    double time_taken=(double)(end-start)/CLOCKS_PER_SEC;

    printf("\nAfter merge sorting:");
    for(i=0;i<n;i++)
        printf("\t%d",a[i]);
    printf("\nNumber of comparisons:%d",count);
    printf("\nTime taken:%f seconds",time_taken);
    printf("\nTime complexity: O(n log n)");
    printf("\nSpace complexity: O(n)");
    printf("\n");
    return 0;
}
```

INPUT

```
6
42 8 27 14 35 19
```

OUTPUT

```
Enter the number of elements:6
Enter the elements:42 8 27 14 35 19

After merge sorting:    8       14      19      27      35      42
Number of comparisons:11
Time taken:0.000008 seconds
Time complexity: O(n log n)
Space complexity: O(n)
```

COMPLEXITY

```
Time complexity:  O(n log n)
Space complexity: O(n)
Best: O(n log n)   Average: O(n log n)   Worst: O(n log n)   Paradigm: Divide & conquer
```

PROGRAM 10

Fibonacci (recursive)

Aim: Write a program in c to find the fibonacci series using recursion and find its execution time using the time function.

ALGORITHM

```
Algorithm: Fib(n)

1. Start
2. If (n <= 1) then
3.        Return n
4. Return Fib(n-1) + Fib(n-2)
5. Stop

Algorithm: FibonacciSeries(n)

1. Start
2. For i <- 0 to (n-1) do
3.        print Fib(i)
4. Stop
```

PROGRAM (C)

```
#include<stdio.h>
#include<time.h>

int calls=0;

int fib(int n) {
    calls++;
    if(n<=1)
        return n;
    return fib(n-1)+fib(n-2);
}

int main() {
    int n,i;
    printf("Enter the number of terms:");
    scanf("%d",&n);

    clock_t start=clock();
    printf("\nFibonacci series:");
    for(i=0;i<n;i++)
        printf("\t%d",fib(i));
    clock_t end=clock();
    double time_taken=(double)(end-start)/CLOCKS_PER_SEC;

    printf("\nNumber of function calls:%d",calls);
    printf("\nTime taken:%f seconds",time_taken);
    printf("\nTime complexity: O(2^n)");
    printf("\nSpace complexity: O(n)");
    printf("\n");
    return 0;
}
```

INPUT

```
10
```

OUTPUT

```
Enter the number of terms:10
```

```
Fibonacci series:       0       1       1       2       3       5       8       13      21      34
Number of function calls:276
Time taken:0.000009 seconds
Time complexity: O(2^n)
Space complexity: O(n)
```

COMPLEXITY

```
Time complexity:  O(2^n)
Space complexity: O(n)
Best: O(2ⁿ)   Average: O(2ⁿ)   Worst: O(2ⁿ)   Paradigm: Recursion
```

PROGRAM 11

Fibonacci (DP)

Aim: Write a program in c to find the fibonacci series using dynamic programming and find its execution time using the time function.

ALGORITHM

```
Algorithm: FibonacciDP(n)

1. Start
2. For i <- 0 to (n-1) do
3.        if (i <= 1) then
4.              F[i] <- i
5.        else
6.              F[i] <- F[i-1] + F[i-2]
7. Print F[0..n-1]
8. Stop
```

PROGRAM (C)

```
#include<stdio.h>
#include<time.h>

int main() {
    int n,i,count=0;
    printf("Enter the number of terms:");
    scanf("%d",&n);
    int f[n];

    clock_t start=clock();
    for(i=0;i<n;i++) {
        count++;
        if(i<=1)
            f[i]=i;
        else
            f[i]=f[i-1]+f[i-2];
    }
    clock_t end=clock();
    double time_taken=(double)(end-start)/CLOCKS_PER_SEC;

    printf("\nFibonacci series:");
    for(i=0;i<n;i++)
        printf("\t%d",f[i]);
    printf("\nNumber of iterations:%d",count);
    printf("\nTime taken:%f seconds",time_taken);
    printf("\nTime complexity: O(n)");
    printf("\nSpace complexity: O(n)");
    printf("\n");
    return 0;
}
```

INPUT

```
10
```

OUTPUT

```
Enter the number of terms:10

Fibonacci series:       0       1       1       2       3       5       8       13      21      34
Number of iterations:10
Time taken:0.000003 seconds
Time complexity: O(n)
Space complexity: O(n)
```

COMPLEXITY

```
Time complexity:  O(n)
Space complexity: O(n)
Best: O(n)   Average: O(n)   Worst: O(n)   Paradigm: Dynamic programming
```

PROGRAM 12

Matrix chain multiplication

Aim: Write a program in c to find the minimum number of scalar multiplications needed for matrix chain multiplication using dynamic programming and find its execution time using the time function.

ALGORITHM

```
Algorithm: MatrixChainOrder(p, n)

1. Start
2. For i <- 1 to n do
3.        m[i][i] <- 0
4. For l <- 2 to n do
5.        For i <- 1 to (n-l+1) do
6.              j <- i+l-1
7.              m[i][j] <- infinity
8.              For k <- i to (j-1) do
9.                    q <- m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j]
10.                   if q < m[i][j] then
11.                         m[i][j] <- q
12.                         s[i][j] <- k
13. Print m[1][n] and PrintOrder(s, 1, n)
14. Stop

Algorithm: PrintOrder(s, i, j)

1. If (i == j) then
2.        print "A"i
3. else
4.        print "("
5.        PrintOrder(s, i, s[i][j])
6.        PrintOrder(s, s[i][j]+1, j)
7.        print ")"
```

PROGRAM (C)

```
#include<stdio.h>
#include<time.h>

int s[20][20];

void printOrder(int i,int j) {
    if(i==j) {
        printf("A%d",i);
        return;
    }
    printf("(");
    printOrder(i,s[i][j]);
    printOrder(s[i][j]+1,j);
    printf(")");
}

int main() {
    int n,i,j,k,l,count=0;
    printf("Enter the number of matrices:");
    scanf("%d",&n);
    int p[n+1];
    printf("Enter the %d dimensions (matrix i is p[i-1] x p[i]):",n+1);
    for(i=0;i<=n;i++)
        scanf("%d",&p[i]);
    int m[20][20];

    clock_t start=clock();
    for(i=1;i<=n;i++)
```

```
        m[i][i]=0;
    for(l=2;l<=n;l++) {
        for(i=1;i<=n-l+1;i++) {
            j=i+l-1;
            m[i][j]=999999999;
            for(k=i;k<j;k++) {
                count++;
                int cost=m[i][k]+m[k+1][j]+p[i-1]*p[k]*p[j];
                if(cost<m[i][j]) {
                    m[i][j]=cost;
                    s[i][j]=k;
                }
            }
        }
    }
    clock_t end=clock();
    double time_taken=(double)(end-start)/CLOCKS_PER_SEC;

    printf("\nCost table:");
    for(i=1;i<=n;i++) {
        printf("\n");
        for(j=1;j<=n;j++)
            if(j<i)
                printf("\t-");
            else
                printf("\t%d",m[i][j]);
    }
    printf("\nMinimum number of multiplications:%d",m[1][n]);
    printf("\nOptimal parenthesization:");
    printOrder(1,n);
    printf("\nNumber of comparisons:%d",count);
    printf("\nTime taken:%f seconds",time_taken);
    printf("\nTime complexity: O(n^3)");
    printf("\nSpace complexity: O(n^2)");
    printf("\n");
    return 0;
}
```

INPUT

```
4
5 4 6 2 7
```

OUTPUT

```
Enter the number of matrices:4
Enter the 5 dimensions (matrix i is p[i-1] x p[i]):5 4 6 2 7

Cost table:
        0       120     88      158
        -       0       48      104
        -       -       0       84
        -       -       -       0
Minimum number of multiplications:158
Optimal parenthesization:((A1(A2A3))A4)
Number of comparisons:10
Time taken:0.000004 seconds
Time complexity: O(n^3)
Space complexity: O(n^2)
```

COMPLEXITY

```
Time complexity:  O(n^3)
Space complexity: O(n^2)
Best: O(n³)   Average: O(n³)   Worst: O(n³)   Paradigm: Dynamic programming
```

PROGRAM 13

0/1 knapsack

Aim: Write a program in c to solve the 0/1 knapsack problem using dynamic programming and find its execution time using the time function.

ALGORITHM

```
Algorithm: Knapsack01(v, wt, n, W)

1. Start
2. For w <- 0 to W do
3.        V[0][w] <- 0
4. For i <- 1 to n do
5.        For w <- 0 to W do
6.              if wt[i] <= w and v[i] + V[i-1][w-wt[i]] > V[i-1][w] then
7.                    V[i][w] <- v[i] + V[i-1][w-wt[i]]
8.                    keep[i][w] <- 1
9.              else
10.                   V[i][w] <- V[i-1][w]
11.                   keep[i][w] <- 0
12. print "Maximum value" V[n][W]
13. w <- W
14. For i <- n down to 1 do
15.       if keep[i][w] == 1 then
16.             print "Item i selected"
17.             w <- w - wt[i]
18. Stop
```

PROGRAM (C)

```
#include<stdio.h>
#include<time.h>

int main() {
    int n,W,i,w,count=0;
    printf("Enter the number of items:");
    scanf("%d",&n);
    int v[n+1],wt[n+1];
    printf("Enter the values of %d items:",n);
    for(i=1;i<=n;i++)
        scanf("%d",&v[i]);
    printf("Enter the weights of %d items:",n);
    for(i=1;i<=n;i++)
        scanf("%d",&wt[i]);
    printf("Enter the capacity of the knapsack:");
    scanf("%d",&W);
    int V[n+1][W+1],keep[n+1][W+1];

    clock_t start=clock();
    for(w=0;w<=W;w++)
        V[0][w]=0;
    for(i=1;i<=n;i++) {
        for(w=0;w<=W;w++) {
            count++;
            if(wt[i]<=w && v[i]+V[i-1][w-wt[i]]>V[i-1][w]) {
                V[i][w]=v[i]+V[i-1][w-wt[i]];
                keep[i][w]=1;
            }
            else {
                V[i][w]=V[i-1][w];
                keep[i][w]=0;
            }
        }
    }
```

```
    clock_t end=clock();
    double time_taken=(double)(end-start)/CLOCKS_PER_SEC;

    printf("\nV[i,w] table:\ni\\w");
    for(w=0;w<=W;w++)
        printf("\t%d",w);
    for(i=0;i<=n;i++) {
        printf("\n%d",i);
        for(w=0;w<=W;w++)
            printf("\t%d",V[i][w]);
    }
    printf("\nMaximum value:%d",V[n][W]);
    printf("\nItems selected:");
    w=W;
    for(i=n;i>=1;i--) {
        if(keep[i][w]==1) {
            printf("\tItem %d(v=%d,w=%d)",i,v[i],wt[i]);
            w=w-wt[i];
        }
    }
    printf("\nNumber of comparisons:%d",count);
    printf("\nTime taken:%f seconds",time_taken);
    printf("\nTime complexity: O(nW)");
    printf("\nSpace complexity: O(nW)");
    printf("\n");
    return 0;
}
```

INPUT

```
4
10 40 30 50
5 4 6 3
10
```

OUTPUT

```
Enter the number of items:4
Enter the values of 4 items:10 40 30 50
Enter the weights of 4 items:5 4 6 3
Enter the capacity of the knapsack:10

V[i,w] table:
i\w     0       1       2       3       4       5       6       7       8       9       10
0       0       0       0       0       0       0       0       0       0       0       0
1       0       0       0       0       0       10      10      10      10      10      10
2       0       0       0       0       40      40      40      40      40      50      50
3       0       0       0       0       40      40      40      40      40      50      70
4       0       0       0       50      50      50      50      90      90      90      90
Maximum value:90
Items selected: Item 4(v=50,w=3)        Item 2(v=40,w=4)
Number of comparisons:44
Time taken:0.000001 seconds
Time complexity: O(nW)
Space complexity: O(nW)
```

COMPLEXITY

```
Time complexity:  O(nW)
Space complexity: O(nW)
Best: O(nW)   Average: O(nW)   Worst: O(nW)   Paradigm: Dynamic programming
```

PROGRAM 14

Longest common subsequence

Aim: Write a program in c to find the longest common subsequence (LCS) of two sequences using dynamic programming and find its execution time using the time function.

ALGORITHM

```
Algorithm: LCS-Length(X, Y, m, n)

1. Start
2. For i <- 1 to m do
3.        c[i][0] <- 0
4. For j <- 0 to n do
5.        c[0][j] <- 0
6. For i <- 1 to m do
7.        For j <- 1 to n do
8.              if X[i] == Y[j] then
9.                    c[i][j] <- c[i-1][j-1] + 1
10.                   b[i][j] <- 'D'
11.             else if c[i-1][j] >= c[i][j-1] then
12.                   c[i][j] <- c[i-1][j]
13.                   b[i][j] <- 'U'
14.             else
15.                   c[i][j] <- c[i][j-1]
16.                   b[i][j] <- 'L'
17. print "Length of LCS" c[m][n]
18. PrintLCS(b, X, m, n)
19. Stop

Algorithm: PrintLCS(b, X, i, j)

1. If (i == 0 or j == 0) then
2.        Return
3. If b[i][j] == 'D' then
4.        PrintLCS(b, X, i-1, j-1)
5.        print X[i]
6. else if b[i][j] == 'U' then
7.        PrintLCS(b, X, i-1, j)
8. else
9.        PrintLCS(b, X, i, j-1)
```

PROGRAM (C)

```
#include<stdio.h>
#include<string.h>
#include<time.h>

int c[50][50];
char b[50][50];
char x[50],y[50];

void printLCS(int i,int j) {
    if(i==0 || j==0)
        return;
    if(b[i][j]=='D') {
        printLCS(i-1,j-1);
        printf("%c",x[i-1]);
    }
    else if(b[i][j]=='U')
        printLCS(i-1,j);
    else
        printLCS(i,j-1);
}
```

```
int main() {
    int m,n,i,j,count=0;
    printf("Enter the first sequence X:");
    scanf("%s",x);
    printf("Enter the second sequence Y:");
    scanf("%s",y);
    m=strlen(x);
    n=strlen(y);

    clock_t start=clock();
    for(i=1;i<=m;i++)
        c[i][0]=0;
    for(j=0;j<=n;j++)
        c[0][j]=0;
    for(i=1;i<=m;i++) {
        for(j=1;j<=n;j++) {
            count++;
            if(x[i-1]==y[j-1]) {
                c[i][j]=c[i-1][j-1]+1;
                b[i][j]='D';
            }
            else if(c[i-1][j]>=c[i][j-1]) {
                c[i][j]=c[i-1][j];
                b[i][j]='U';
            }
            else {
                c[i][j]=c[i][j-1];
                b[i][j]='L';
            }
        }
    }
    clock_t end=clock();
    double time_taken=(double)(end-start)/CLOCKS_PER_SEC;

    printf("\nc[i,j] table (D=diagonal, U=up, L=left):\n\t\t");
    for(j=0;j<n;j++)
        printf("%c\t",y[j]);
    for(i=0;i<=m;i++) {
        printf("\n");
        if(i==0)
            printf("\t");
        else
            printf("%c\t",x[i-1]);
        for(j=0;j<=n;j++)
            if(i==0 || j==0)
                printf("%d\t",c[i][j]);
            else
                printf("%d%c\t",c[i][j],b[i][j]);
    }
    printf("\nLength of LCS:%d",c[m][n]);
    printf("\nLCS:");
    printLCS(m,n);
    printf("\nNumber of comparisons:%d",count);
    printf("\nTime taken:%f seconds",time_taken);
    printf("\nTime complexity: O(mn)");
    printf("\nSpace complexity: O(mn)");
    printf("\n");
    return 0;
}
```

INPUT

```
ABCBDAB
BDCABA
```

OUTPUT

```
Enter the first sequence X:ABCBDAB
Enter the second sequence Y:BDCABA

c[i,j] table (D=diagonal, U=up, L=left):
                B       D       C       A       B       A
        0       0       0       0       0       0       0
A       0       0U      0U      0U      1D      1L      1D
B       0       1D      1L      1L      1U      2D      2L
C       0       1U      1U      2D      2L      2U      2U
B       0       1D      1U      2U      2U      3D      3L
D       0       1U      2D      2U      2U      3U      3U
A       0       1U      2U      2U      3D      3U      4D
B       0       1D      2U      2U      3U      4D      4U
Length of LCS:4
LCS:BCBA
Number of comparisons:42
Time taken:0.000002 seconds
Time complexity: O(mn)
Space complexity: O(mn)
```

COMPLEXITY

```
Time complexity:  O(mn)
Space complexity: O(mn)
Best: O(mn)   Average: O(mn)   Worst: O(mn)   Paradigm: Dynamic programming
```

PROGRAM 15

Fractional knapsack

Aim: Write a program in c to solve the fractional knapsack problem using the greedy method and find its execution time using the time function.

ALGORITHM

```
Algorithm: FractionalKnapsack(v, w, n, W)

1. Start
2. For i <- 1 to n do
3.        r[i] <- v[i]/w[i]
4.        x[i] <- 0
5. Sort the items in decreasing order of r[i] (merge sort)
6. remaining <- W, total <- 0
7. For each item i in sorted order, while remaining > 0 do
8.        if w[i] <= remaining then
9.              x[i] <- 1
10.             remaining <- remaining - w[i]
11.       else
12.             x[i] <- remaining/w[i]
13.             remaining <- 0
14.       total <- total + x[i]*v[i]
15. print "Maximum value" total
16. Stop
```

PROGRAM (C)

```
#include<stdio.h>
#include<time.h>

double r[50];
int count=0;

void merge(int a[],int low,int mid,int high) {
    int n1=mid-low+1,n2=high-mid;
    int left[n1],right[n2];
    int i,j,k;

    for(i=0;i<n1;i++)
        left[i]=a[low+i];
    for(j=0;j<n2;j++)
        right[j]=a[mid+1+j];

    i=0;
    j=0;
    k=low;
    while(i<n1 && j<n2) {
        count++;
        if(r[left[i]]>=r[right[j]])
            a[k++]=left[i++];
        else
            a[k++]=right[j++];
    }
    while(i<n1)
        a[k++]=left[i++];
    while(j<n2)
        a[k++]=right[j++];
}

void mergeSort(int a[],int low,int high) {
    if(low<high) {
        int mid=(low+high)/2;
        mergeSort(a,low,mid);
```

```
        mergeSort(a,mid+1,high);
        merge(a,low,mid,high);
    }
}

int main() {
    int n,i,k;
    double W;
    printf("Enter the number of items:");
    scanf("%d",&n);
    double v[n+1],w[n+1],x[n+1];
    int order[n];
    printf("Enter the values of %d items:",n);
    for(i=1;i<=n;i++)
        scanf("%lf",&v[i]);
    printf("Enter the weights of %d items:",n);
    for(i=1;i<=n;i++)
        scanf("%lf",&w[i]);
    printf("Enter the capacity of the knapsack:");
    scanf("%lf",&W);

    clock_t start=clock();
    for(i=1;i<=n;i++) {
        r[i]=v[i]/w[i];
        x[i]=0;
        order[i-1]=i;
    }
    mergeSort(order,0,n-1);
    double remaining=W,total=0;
    for(k=0;k<n && remaining>0;k++) {
        i=order[k];
        count++;
        if(w[i]<=remaining) {
            x[i]=1;
            remaining=remaining-w[i];
        }
        else {
            x[i]=remaining/w[i];
            remaining=0;
        }
        total=total+x[i]*v[i];
    }
    clock_t end=clock();
    double time_taken=(double)(end-start)/CLOCKS_PER_SEC;

    printf("\nItems in decreasing order of value/weight:\nItem\tValue\tWeight\tv/w\tFraction");
    for(k=0;k<n;k++) {
        i=order[k];
        printf("\n%d\t%g\t%g\t%g\t%g",i,v[i],w[i],r[i],x[i]);
    }
    printf("\nMaximum value:%g",total);
    printf("\nNumber of comparisons:%d",count);
    printf("\nTime taken:%f seconds",time_taken);
    printf("\nTime complexity: O(n log n)");
    printf("\nSpace complexity: O(n)");
    printf("\n");
    return 0;
}
```

INPUT

```
4
10 40 30 50
5 4 6 3
10
```

OUTPUT

```
Enter the number of items:4
Enter the values of 4 items:10 40 30 50
Enter the weights of 4 items:5 4 6 3
Enter the capacity of the knapsack:10

Items in decreasing order of value/weight:
Item    Value   Weight  v/w     Fraction
4       50      3       16.6667 1
2       40      4       10      1
3       30      6       5       0.5
1       10      5       2       0
Maximum value:105
Number of comparisons:8
Time taken:0.000006 seconds
Time complexity: O(n log n)
Space complexity: O(n)
```

COMPLEXITY

```
Time complexity:  O(n log n)
Space complexity: O(n)
Best: O(n log n)   Average: O(n log n)   Worst: O(n log n)   Paradigm: Greedy
```

PROGRAM 16

Job sequencing with deadlines

Aim: Write a program in c to solve the job sequencing with deadlines problem using the greedy method and find its execution time using the time function.

ALGORITHM

```
Algorithm: JobSequencing(p, d, n)

1. Start
2. maxd <- largest deadline
3. For t <- 1 to maxd do
4.        slot[t] <- 0
5. Sort the jobs in decreasing order of profit p[i] (merge sort)
6. total <- 0
7. For each job i in sorted order do
8.        For t <- d[i] down to 1 do
9.              if slot[t] == 0 then
10.                   slot[t] <- i
11.                   total <- total + p[i]
12.                   break
13. print the jobs in slot[1..maxd] and total
14. Stop
```

PROGRAM (C)

```
#include<stdio.h>
#include<time.h>

int p[50];
int count=0;

void merge(int a[],int low,int mid,int high) {
    int n1=mid-low+1,n2=high-mid;
    int left[n1],right[n2];
    int i,j,k;

    for(i=0;i<n1;i++)
        left[i]=a[low+i];
    for(j=0;j<n2;j++)
        right[j]=a[mid+1+j];

    i=0;
    j=0;
    k=low;
    while(i<n1 && j<n2) {
        count++;
        if(p[left[i]]>=p[right[j]])
            a[k++]=left[i++];
        else
            a[k++]=right[j++];
    }
    while(i<n1)
        a[k++]=left[i++];
    while(j<n2)
        a[k++]=right[j++];
}

void mergeSort(int a[],int low,int high) {
    if(low<high) {
        int mid=(low+high)/2;
        mergeSort(a,low,mid);
        mergeSort(a,mid+1,high);
        merge(a,low,mid,high);
```

```
    }
}

int main() {
    int n,i,k,t,maxd=0,total=0;
    printf("Enter the number of jobs:");
    scanf("%d",&n);
    int d[n+1],order[n];
    printf("Enter the profits of %d jobs:",n);
    for(i=1;i<=n;i++)
        scanf("%d",&p[i]);
    printf("Enter the deadlines of %d jobs:",n);
    for(i=1;i<=n;i++) {
        scanf("%d",&d[i]);
        if(d[i]>maxd)
            maxd=d[i];
    }
    int slot[maxd+1];

    clock_t start=clock();
    for(i=1;i<=n;i++)
        order[i-1]=i;
    for(t=1;t<=maxd;t++)
        slot[t]=0;
    mergeSort(order,0,n-1);
    for(k=0;k<n;k++) {
        i=order[k];
        for(t=d[i];t>=1;t--) {
            count++;
            if(slot[t]==0) {
                slot[t]=i;
                total=total+p[i];
                break;
            }
        }
    }
    clock_t end=clock();
    double time_taken=(double)(end-start)/CLOCKS_PER_SEC;

    printf("\nJobs in decreasing order of profit:\nJob\tProfit\tDeadline");
    for(k=0;k<n;k++) {
        i=order[k];
        printf("\nJ%d\t%d\t%d",i,p[i],d[i]);
    }
    printf("\nJob sequence:");
    for(t=1;t<=maxd;t++)
        if(slot[t]!=0)
            printf("\tJ%d",slot[t]);
    printf("\nMaximum profit:%d",total);
    printf("\nNumber of comparisons:%d",count);
    printf("\nTime taken:%f seconds",time_taken);
    printf("\nTime complexity: O(n^2)");
    printf("\nSpace complexity: O(n)");
    printf("\n");
    return 0;
}
```

INPUT

```
5
100 19 27 25 15
2 1 2 1 3
```

OUTPUT

```
Enter the number of jobs:5
Enter the profits of 5 jobs:100 19 27 25 15
Enter the deadlines of 5 jobs:2 1 2 1 3
```

```
Jobs in decreasing order of profit:
Job     Profit  Deadline
J1      100     2
J3      27      2
J4      25      1
J2      19      1
J5      15      3
Job sequence:   J3      J1      J5
Maximum profit:142
Number of comparisons:14
Time taken:0.000003 seconds
Time complexity: O(n^2)
Space complexity: O(n)
```

COMPLEXITY

```
Time complexity:  O(n^2)
Space complexity: O(n)
Best: O(n log n)   Average: O(n²)   Worst: O(n²)   Paradigm: Greedy
```

PROGRAM 17

Huffman coding

Aim: Write a program in c to construct an optimal prefix code using Huffman coding (greedy method) and find its execution time using the time function.

ALGORITHM

```
Algorithm: Huffman(C, f, n)

1. Start
2. Q <- min-priority queue of all n characters, keyed on frequency f
3. For i <- 1 to (n-1) do
4.        allocate a new node z
5.        x <- ExtractMin(Q)
6.        y <- ExtractMin(Q)
7.        left[z] <- x
8.        right[z] <- y
9.        f[z] <- f[x] + f[y]
10.       Insert(Q, z)
11. root <- ExtractMin(Q)
12. PrintCodes(root, "")
13. Stop

Algorithm: PrintCodes(z, code)

1. If z is a leaf then
2.        print character of z and code
3.        Return
4. PrintCodes(left[z], code + "0")
5. PrintCodes(right[z], code + "1")
```

PROGRAM (C)

```
#include<stdio.h>
#include<time.h>

char ch[100];
int f[100],lft[100],rgt[100];
int Q[100],qsize=0,comp=0;
int code[100];

void heapify(int i) {
    int smallest=i,l=2*i+1,r=2*i+2;
    if(l<qsize) {
        comp++;
        if(f[Q[l]]<f[Q[smallest]])
            smallest=l;
    }
    if(r<qsize) {
        comp++;
        if(f[Q[r]]<f[Q[smallest]])
            smallest=r;
    }
    if(smallest!=i) {
        int t=Q[i];
        Q[i]=Q[smallest];
        Q[smallest]=t;
        heapify(smallest);
    }
}

int extractMin() {
    int min=Q[0];
    Q[0]=Q[qsize-1];
```

```
    qsize--;
    heapify(0);
    return min;
}

void insert(int z) {
    int i=qsize;
    Q[qsize++]=z;
    while(i>0) {
        int p=(i-1)/2;
        comp++;
        if(f[Q[i]]>=f[Q[p]])
            break;
        int t=Q[i];
        Q[i]=Q[p];
        Q[p]=t;
        i=p;
    }
}

void printCodes(int z,int depth) {
    int i;
    if(lft[z]==-1 && rgt[z]==-1) {
        printf("\n%c\t%d\t",ch[z],f[z]);
        if(depth==0)
            printf("0");
        for(i=0;i<depth;i++)
            printf("%d",code[i]);
        return;
    }
    code[depth]=0;
    printCodes(lft[z],depth+1);
    code[depth]=1;
    printCodes(rgt[z],depth+1);
}

int main() {
    int n,i;
    printf("Enter the number of characters:");
    scanf("%d",&n);
    printf("Enter the %d characters:",n);
    for(i=0;i<n;i++)
        scanf(" %c",&ch[i]);
    printf("Enter the frequencies of %d characters:",n);
    for(i=0;i<n;i++) {
        scanf("%d",&f[i]);
        lft[i]=rgt[i]=-1;
    }

    clock_t start=clock();
    for(i=0;i<n;i++)
        insert(i);
    int next=n;
    for(i=1;i<=n-1;i++) {
        int z=next++;
        int x=extractMin();
        int y=extractMin();
        lft[z]=x;
        rgt[z]=y;
        f[z]=f[x]+f[y];
        insert(z);
    }
    int root=extractMin();
    clock_t end=clock();
    double time_taken=(double)(end-start)/CLOCKS_PER_SEC;

    printf("\nHuffman codes:\nChar\tFreq\tCode");
    printCodes(root,0);
    printf("\nTotal frequency at root:%d",f[root]);
```

```
    printf("\nNumber of comparisons:%d",comp);
    printf("\nTime taken:%f seconds",time_taken);
    printf("\nTime complexity: O(n log n)");
    printf("\nSpace complexity: O(n)");
    printf("\n");
    return 0;
}
```

INPUT

```
6
a b c d e f
45 13 12 16 9 5
```

OUTPUT

```
Enter the number of characters:6
Enter the 6 characters:a b c d e f
Enter the frequencies of 6 characters:45 13 12 16 9 5

Huffman codes:
Char    Freq    Code
a       45      0
c       12      100
b       13      101
f       5       1100
e       9       1101
d       16      111
Total frequency at root:100
Number of comparisons:27
Time taken:0.000005 seconds
Time complexity: O(n log n)
Space complexity: O(n)
```

COMPLEXITY

```
Time complexity:  O(n log n)
Space complexity: O(n)
Best: O(n log n)   Average: O(n log n)   Worst: O(n log n)   Paradigm: Greedy
```

PROGRAM 18

N-Queen

Aim: Write a program in c to solve the N-Queen problem using backtracking and find its execution time using the time function.

ALGORITHM

```
Algorithm: Place(k, i)
(returns true if a queen can be placed in row k, column i)

1. For j <- 1 to (k-1) do
2.        if x[j] == i or |x[j] - i| == |j - k| then
3.              Return false
4. Return true

Algorithm: NQueens(k, n)

1. Start
2. For i <- 1 to n do
3.        if Place(k, i) then
4.              x[k] <- i
5.              if k == n then
6.                    print x[1..n]
7.              else
8.                    NQueens(k+1, n)
9. Stop
```

PROGRAM (C)

```
#include<stdio.h>
#include<time.h>

int x[20],first[20];
int n,solutions=0,comp=0;

int place(int k,int i) {
    int j;
    for(j=1;j<k;j++) {
        comp++;
        if(x[j]==i || x[j]-i==j-k || x[j]-i==k-j)
            return 0;
    }
    return 1;
}

void nQueens(int k) {
    int i,j;
    for(i=1;i<=n;i++) {
        if(place(k,i)) {
            x[k]=i;
            if(k==n) {
                solutions++;
                printf("\nSolution %d:",solutions);
                for(j=1;j<=n;j++)
                    printf("\t%d",x[j]);
                if(solutions==1)
                    for(j=1;j<=n;j++)
                        first[j]=x[j];
            }
            else
                nQueens(k+1);
        }
    }
}
```

```
int main() {
    int i,j;
    printf("Enter the number of queens:");
    scanf("%d",&n);

    printf("\nColumn of the queen in each row 1..%d:",n);
    clock_t start=clock();
    nQueens(1);
    clock_t end=clock();
    double time_taken=(double)(end-start)/CLOCKS_PER_SEC;

    if(solutions>0) {
        printf("\n\nBoard for solution 1:");
        for(i=1;i<=n;i++) {
            printf("\n");
            for(j=1;j<=n;j++)
                if(first[i]==j)
                    printf("Q ");
                else
                    printf(". ");
        }
    }
    printf("\nTotal solutions:%d",solutions);
    printf("\nNumber of comparisons:%d",comp);
    printf("\nTime taken:%f seconds",time_taken);
    printf("\nTime complexity: O(n!)");
    printf("\nSpace complexity: O(n)");
    printf("\n");
    return 0;
}
```

INPUT

```
4
```

OUTPUT

```
Enter the number of queens:4

Column of the queen in each row 1..4:
Solution 1:     2       4       1       3
Solution 2:     3       1       4       2

Board for solution 1:
. Q . .
. . . Q
Q . . .
. . Q .
Total solutions:2
Number of comparisons:84
Time taken:0.000008 seconds
Time complexity: O(n!)
Space complexity: O(n)
```

COMPLEXITY

```
Time complexity:  O(n!)
Space complexity: O(n)
Best: O(n!)   Average: O(n!)   Worst: O(n!)   Paradigm: Backtracking
```

PROGRAM 19

Travelling salesman (DP)

Aim: Write a program in c to solve the travelling salesman problem using dynamic programming and find its execution time using the time function.

ALGORITHM

```
Algorithm: TSP(cost, n)
(g(i, S) = cheapest path that starts at city i, visits every city in set S once, and ends at cit
y 1)

1. Start
2. For i <- 2 to n do
3.        g(i, {}) <- cost[i][1]
4. For each set S not containing city 1, in increasing size do
5.        For each city i not in S do
6.              g(i, S) <- infinity
7.              For each city k in S do
8.                    if cost[i][k] + g(k, S - {k}) < g(i, S) then
9.                          g(i, S) <- cost[i][k] + g(k, S - {k})
10.                         next(i, S) <- k
11. print "Minimum cost" g(1, {2, 3, ..., n})
12. Follow next(i, S) from city 1 to print the tour
13. Stop
```

PROGRAM (C)

```
#include<stdio.h>
#include<time.h>

int cost[15][15];
int g[1<<15][15],nxt[1<<15][15];

int main() {
    int n,i,j,k,S,full,count=0;
    printf("Enter the number of cities:");
    scanf("%d",&n);
    printf("Enter the %dx%d cost matrix:\n",n,n);
    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
            scanf("%d",&cost[i][j]);

    clock_t start=clock();
    // S is a bitmask of cities; bit 0 (city 1) is never in S
    full=(1<<n)-1;
    for(i=0;i<n;i++)
        g[0][i]=cost[i][0];
    for(S=1;S<=full;S++) {
        if(S&1)
            continue;
        for(i=0;i<n;i++) {
            if(S&(1<<i))
                continue;
            g[S][i]=999999999;
            for(k=1;k<n;k++) {
                if(S&(1<<k)) {
                    count++;
                    int c=cost[i][k]+g[S^(1<<k)][k];
                    if(c<g[S][i]) {
                        g[S][i]=c;
                        nxt[S][i]=k;
                    }
                }
            }
```

```
        }
    }
    S=full^1;
    clock_t end=clock();
    double time_taken=(double)(end-start)/CLOCKS_PER_SEC;

    printf("\nMinimum cost of tour:%d",g[S][0]);
    printf("\nOptimal tour:1");
    i=0;
    while(S!=0) {
        k=nxt[S][i];
        printf(" -> %d",k+1);
        S=S^(1<<k);
        i=k;
    }
    printf(" -> 1");
    printf("\nNumber of comparisons:%d",count);
    printf("\nTime taken:%f seconds",time_taken);
    printf("\nTime complexity: O(n^2 * 2^n)");
    printf("\nSpace complexity: O(n * 2^n)");
    printf("\n");
    return 0;
}
```

INPUT

```
4
0 10 15 20
5 0 9 10
6 13 0 12
8 8 9 0
```

OUTPUT

```
Enter the number of cities:4
Enter the 4x4 cost matrix:
0 10 15 20
5 0 9 10
6 13 0 12
8 8 9 0

Minimum cost of tour:35
Optimal tour:1 -> 2 -> 4 -> 3 -> 1
Number of comparisons:24
Time taken:0.000008 seconds
Time complexity: O(n^2 * 2^n)
Space complexity: O(n * 2^n)
```

COMPLEXITY

```
Time complexity:  O(n^2 * 2^n)
Space complexity: O(n * 2^n)
Best: O(n²·2ⁿ)   Average: O(n²·2ⁿ)   Worst: O(n²·2ⁿ)   Paradigm: Dynamic programming
```

PROGRAM 20

Breadth-first search

Aim: Write a program in c to traverse a graph using breadth first search (BFS) and find its execution time using the time function.

ALGORITHM

```
Algorithm: BFS(G, s)

1. Start
2. For each vertex u in G do
3.        visited[u] <- 0
4.        d[u] <- infinity
5. visited[s] <- 1, d[s] <- 0
6. Enqueue(Q, s)
7. While Q is not empty do
8.        u <- Dequeue(Q)
9.        print u
10.       For each vertex v adjacent to u do
11.             if visited[v] == 0 then
12.                   visited[v] <- 1
13.                   d[v] <- d[u] + 1
14.                   Enqueue(Q, v)
15. Stop
```

PROGRAM (C)

```
#include<stdio.h>
#include<time.h>

int main() {
    int n,i,u,v,s,front=0,rear=0,count=0;
    printf("Enter the number of vertices:");
    scanf("%d",&n);
    int adj[n+1][n+1],visited[n+1],d[n+1],Q[n];
    printf("Enter the %dx%d adjacency matrix:\n",n,n);
    for(u=1;u<=n;u++)
        for(v=1;v<=n;v++)
            scanf("%d",&adj[u][v]);
    printf("Enter the source vertex:");
    scanf("%d",&s);

    clock_t start=clock();
    for(u=1;u<=n;u++) {
        visited[u]=0;
        d[u]=-1;
    }
    visited[s]=1;
    d[s]=0;
    Q[rear++]=s;
    printf("\nBFS traversal:");
    while(front<rear) {
        u=Q[front++];
        printf("\t%d",u);
        for(v=1;v<=n;v++) {
            count++;
            if(adj[u][v]==1 && visited[v]==0) {
                visited[v]=1;
                d[v]=d[u]+1;
                Q[rear++]=v;
            }
        }
    }
    clock_t end=clock();
```

```
    double time_taken=(double)(end-start)/CLOCKS_PER_SEC;

    printf("\nVertex\tDistance from %d",s);
    for(i=1;i<=n;i++)
        if(d[i]==-1)
            printf("\n%d\tunreachable",i);
        else
            printf("\n%d\t%d",i,d[i]);
    printf("\nNumber of comparisons:%d",count);
    printf("\nTime taken:%f seconds",time_taken);
    printf("\nTime complexity: O(V+E) with adjacency list, O(V^2) with adjacency matrix");
    printf("\nSpace complexity: O(V+E) with adjacency list, O(V^2) with adjacency matrix");
    printf("\n");
    return 0;
}
```

INPUT

```
6
0 1 1 0 0 0
1 0 0 1 1 0
1 0 0 0 1 0
0 1 0 0 0 1
0 1 1 0 0 1
0 0 0 1 1 0
1
```

OUTPUT

```
Enter the number of vertices:6
Enter the 6x6 adjacency matrix:
0 1 1 0 0 0
1 0 0 1 1 0
1 0 0 0 1 0
0 1 0 0 0 1
0 1 1 0 0 1
0 0 0 1 1 0
Enter the source vertex:1

BFS traversal:  1       2       3       4       5       6
Vertex  Distance from 1
1       0
2       1
3       1
4       2
5       2
6       3
Number of comparisons:36
Time taken:0.000003 seconds
Time complexity: O(V+E) with adjacency list, O(V^2) with adjacency matrix
Space complexity: O(V+E) with adjacency list, O(V^2) with adjacency matrix
```

COMPLEXITY

```
Time complexity:  O(V+E) with adjacency list, O(V^2) with adjacency matrix
Space complexity: O(V+E) with adjacency list, O(V^2) with adjacency matrix
Best: O(V+E)   Average: O(V+E)   Worst: O(V+E)   Paradigm: Graph traversal
```

PROGRAM 21

Depth-first search

Aim: Write a program in c to traverse a graph using depth first search (DFS) and find its execution time using the time function.

ALGORITHM

```
Algorithm: DFS(G, s)

1. Start
2. For each vertex u in G do
3.        visited[u] <- 0
4. time <- 0
5. DFS-Visit(s)
6. For each vertex u in G do
7.        if visited[u] == 0 then
8.              DFS-Visit(u)
9. Stop

Algorithm: DFS-Visit(u)

1. visited[u] <- 1
2. time <- time+1, d[u] <- time
3. print u
4. For each vertex v adjacent to u do
5.        if visited[v] == 0 then
6.              DFS-Visit(v)
7. time <- time+1, f[u] <- time
```

PROGRAM (C)

```
#include<stdio.h>
#include<time.h>

int adj[50][50],visited[50],d[50],f[50];
int n,t=0,count=0;

void dfsVisit(int u) {
    int v;
    visited[u]=1;
    d[u]=++t;
    printf("\t%d",u);
    for(v=1;v<=n;v++) {
        count++;
        if(adj[u][v]==1 && visited[v]==0)
            dfsVisit(v);
    }
    f[u]=++t;
}

int main() {
    int u,v,s;
    printf("Enter the number of vertices:");
    scanf("%d",&n);
    printf("Enter the %dx%d adjacency matrix:\n",n,n);
    for(u=1;u<=n;u++)
        for(v=1;v<=n;v++)
            scanf("%d",&adj[u][v]);
    printf("Enter the source vertex:");
    scanf("%d",&s);

    clock_t start=clock();
    for(u=1;u<=n;u++)
        visited[u]=0;
```

```
    printf("\nDFS traversal:");
    dfsVisit(s);
    for(u=1;u<=n;u++)
        if(visited[u]==0)
            dfsVisit(u);
    clock_t end=clock();
    double time_taken=(double)(end-start)/CLOCKS_PER_SEC;

    printf("\nVertex\tDiscovery\tFinish");
    for(u=1;u<=n;u++)
        printf("\n%d\t%d\t\t%d",u,d[u],f[u]);
    printf("\nNumber of comparisons:%d",count);
    printf("\nTime taken:%f seconds",time_taken);
    printf("\nTime complexity: O(V+E) with adjacency list, O(V^2) with adjacency matrix");
    printf("\nSpace complexity: O(V+E) with adjacency list, O(V^2) with adjacency matrix");
    printf("\n");
    return 0;
}
```

INPUT

```
6
0 1 1 0 0 0
1 0 0 1 1 0
1 0 0 0 1 0
0 1 0 0 0 1
0 1 1 0 0 1
0 0 0 1 1 0
1
```

OUTPUT

```
Enter the number of vertices:6
Enter the 6x6 adjacency matrix:
0 1 1 0 0 0
1 0 0 1 1 0
1 0 0 0 1 0
0 1 0 0 0 1
0 1 1 0 0 1
0 0 0 1 1 0
Enter the source vertex:1

DFS traversal:  1       2       4       6       5       3
Vertex  Discovery       Finish
1       1               12
2       2               11
3       6               7
4       3               10
5       5               8
6       4               9
Number of comparisons:36
Time taken:0.000006 seconds
Time complexity: O(V+E) with adjacency list, O(V^2) with adjacency matrix
Space complexity: O(V+E) with adjacency list, O(V^2) with adjacency matrix
```

COMPLEXITY

```
Time complexity:  O(V+E) with adjacency list, O(V^2) with adjacency matrix
Space complexity: O(V+E) with adjacency list, O(V^2) with adjacency matrix
Best: O(V+E)   Average: O(V+E)   Worst: O(V+E)   Paradigm: Graph traversal
```

PROGRAM 22

Job assignment (branch & bound)

Aim: Write a program in c to solve the job assignment problem using branch and bound and find its execution time using the time function.

ALGORITHM

```
Algorithm: JOB-ASSIGNMENT(C, n)   (Least Cost Branch and Bound)
(lb(X) = cost of the jobs assigned so far in X
         + for every worker not yet assigned, his cheapest job among the jobs still free)

1. Start
2. Create the root node: no worker assigned, cost <- 0
3. lb(root) <- lower bound of root
4. Insert root into the list of live nodes
5. While the list of live nodes is not empty do
6.       E <- the live node with minimum lb
7.       Remove E from the list of live nodes
8.       if all n workers are assigned in E then
9.             print the assignment in E and its cost as the optimal solution
10.            Stop
11.      i <- the next worker to be assigned in E
12.      For each job j not yet assigned in E do
13.            Create a child X of E with worker i assigned job j
14.            cost(X) <- cost(E) + C[i][j]
15.            lb(X) <- lower bound of X
16.            Insert X into the list of live nodes
17. Stop
```

PROGRAM (C)

```
#include<stdio.h>
#include<time.h>
#define INF 99999
#define MAX 100000

int n,c[15][15],count=0,total=0;
int parent[MAX],level[MAX],job[MAX],used[MAX],cost[MAX],lb[MAX],live[MAX];

int bound(int k) {
    int i,j,min,b=cost[k];
    for(i=level[k]+1;i<=n;i++) {
        min=INF;
        for(j=1;j<=n;j++) {
            count++;
            if(!(used[k]&(1<<j)) && c[i][j]<min)
                min=c[i][j];
        }
        b+=min;
    }
    return b;
}

void new_node(int p,int j) {
    int k=total++;
    parent[k]=p;
    if(p==-1) {
        level[k]=0;
        job[k]=0;
        used[k]=0;
        cost[k]=0;
    }
    else {
        level[k]=level[p]+1;
```

```
        job[k]=j;
        used[k]=used[p]|(1<<j);
        cost[k]=cost[p]+c[level[k]][j];
    }
    lb[k]=bound(k);
    live[k]=1;
}

void assignment(int k) {
    if(parent[k]==-1) {
        printf("root");
        return;
    }
    if(parent[parent[k]]!=-1) {
        assignment(parent[k]);
        printf(", ");
    }
    printf("W%d->J%d",level[k],job[k]);
}

int main() {
    int i,j,k,E,ans=-1,expanded=0;
    printf("Enter the number of workers/jobs:");
    scanf("%d",&n);
    printf("Enter the %dx%d cost matrix (row = worker, column = job):\n",n,n);
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
            scanf("%d",&c[i][j]);

    int order[MAX],first[MAX];

    clock_t start=clock();
    new_node(-1,0);
    while(1) {
        E=-1;
        for(k=0;k<total;k++) {
            if(live[k]) {
                count++;
                if(E==-1 || lb[k]<lb[E])
                    E=k;
            }
        }
        if(E==-1)
            break;
        live[E]=0;
        if(level[E]==n) {
            ans=E;
            break;
        }
        order[expanded]=E;
        first[expanded]=total;
        expanded++;
        for(j=1;j<=n;j++)
            if(!(used[E]&(1<<j)))
                new_node(E,j);
    }
    clock_t end=clock();
    double time_taken=(double)(end-start)/CLOCKS_PER_SEC;

    printf("\nNodes expanded (least lb first):");
    for(i=0;i<expanded;i++) {
        E=order[i];
        printf("\nE = {");
        assignment(E);
        printf("}  lb = %d",lb[E]);
        for(k=first[i];k<first[i]+n-level[E];k++)
            printf("\n      W%d->J%d   cost = %-4d lb = %d",level[k],job[k],cost[k],lb[k]);
    }
```

```
    printf("\n\nOptimal assignment:");
    for(i=1;i<=n;i++) {
        k=ans;
        while(level[k]!=i)
            k=parent[k];
        printf("\nWorker %d -> Job %d\tcost = %d",i,job[k],c[i][job[k]]);
    }
    printf("\nMinimum total cost:%d",cost[ans]);
    printf("\nNodes generated:%d",total);
    printf("\nNodes expanded:%d",expanded);
    printf("\nNumber of comparisons:%d",count);
    printf("\nTime taken:%f seconds",time_taken);
    printf("\nTime complexity: O(n!) in the worst case, far less when the bound prunes the tree");
    printf("\nSpace complexity: O(n!) in the worst case for the live nodes");
    printf("\n");
    return 0;
}
```

INPUT

```
4
9 2 7 8
6 4 3 7
5 8 1 8
7 6 9 4
```

OUTPUT

```
Enter the number of workers/jobs:4
Enter the 4x4 cost matrix (row = worker, column = job):
9 2 7 8
6 4 3 7
5 8 1 8
7 6 9 4

Nodes expanded (least lb first):
E = {root}  lb = 10
      W1->J1   cost = 9    lb = 17
      W1->J2   cost = 2    lb = 10
      W1->J3   cost = 7    lb = 20
      W1->J4   cost = 8    lb = 18
E = {W1->J2}  lb = 10
      W2->J1   cost = 8    lb = 13
      W2->J3   cost = 5    lb = 14
      W2->J4   cost = 9    lb = 17
E = {W1->J2, W2->J1}  lb = 13
      W3->J3   cost = 9    lb = 13
      W3->J4   cost = 16   lb = 25
E = {W1->J2, W2->J1, W3->J3}  lb = 13
      W4->J4   cost = 13   lb = 13

Optimal assignment:
Worker 1 -> Job 2       cost = 2
Worker 2 -> Job 1       cost = 6
Worker 3 -> Job 3       cost = 1
Worker 4 -> Job 4       cost = 4
Minimum total cost:13
Nodes generated:11
Nodes expanded:4
Number of comparisons:121
Time taken:0.000009 seconds
Time complexity: O(n!) in the worst case, far less when the bound prunes the tree
Space complexity: O(n!) in the worst case for the live nodes
```

COMPLEXITY

```
Time complexity:  O(n!) in the worst case, far less when the bound prunes the tree
```

```
Space complexity: O(n!) in the worst case for the live nodes
Best: O(n²)   Average: O(n!)   Worst: O(n!)   Paradigm: Branch & bound
```

PROGRAM 23

Dijkstra

Aim: Write a program in c to find the shortest path from a source vertex to all other vertices using Dijkstra's algorithm and find its execution time using the time function.

ALGORITHM

```
Algorithm: DA(G, w, s)

1. INITIALIZE-SINGLE-SOURCE(G, s)
2. S <- Ø
3. Q <- V[G]
4. while Q != Ø
5.       do u <- EXTRACT-MIN(Q)
6.          S <- S U {u}
7.          for each vertex v in Adj[u]
8.                do RELAX(u, v, w)

INITIALIZE-SINGLE-SOURCE(G, s)
1. for each vertex v in V[G]
2.       do d[v] <- infinity
3.          p[v] <- NIL
4. d[s] <- 0

EXTRACT-MIN(Q)
1. u <- the vertex in Q with minimum d[u]
2. Q <- Q - {u}
3. return u

RELAX(u, v, w)
1. if d[v] > d[u] + w(u,v)
2.       then d[v] <- d[u] + w(u,v)
3.            p[v] <- u
```

PROGRAM (C)

```
#include<stdio.h>
#include<time.h>
#define INF 99999

int n,count=0;
int w[50][50],d[50],p[50],S[50],Q[50];

void initialize_single_source(int s) {
    int v;
    for(v=1;v<=n;v++) {
        d[v]=INF;
        p[v]=0;
    }
    d[s]=0;
}

int extract_min() {
    int v,u=0;
    for(v=1;v<=n;v++) {
        count++;
        if(Q[v]==1 && (u==0 || d[v]<d[u]))
            u=v;
    }
    Q[u]=0;
    return u;
}

void relax(int u,int v) {
```

```
    count++;
    if(d[v]>d[u]+w[u][v]) {
        d[v]=d[u]+w[u][v];
        p[v]=u;
    }
}

void path(int v) {
    if(p[v]==0) {
        printf("%d",v);
        return;
    }
    path(p[v]);
    printf(" -> %d",v);
}

int main() {
    int i,j,u,v,s,q_size,step=0;
    printf("Enter the number of vertices:");
    scanf("%d",&n);
    printf("Enter the %dx%d weight matrix (0 for no edge):\n",n,n);
    for(u=1;u<=n;u++)
        for(v=1;v<=n;v++)
            scanf("%d",&w[u][v]);
    printf("Enter the source vertex:");
    scanf("%d",&s);

    int order[n+1],pos[n+1],trace[n+1][n+1];
    char cell[12];

    clock_t start=clock();
    initialize_single_source(s);
    for(v=1;v<=n;v++)
        S[v]=0;
    for(v=1;v<=n;v++)
        Q[v]=1;
    q_size=n;
    while(q_size!=0) {
        u=extract_min();
        q_size--;
        S[u]=1;
        step++;
        order[step]=u;
        pos[u]=step;
        for(v=1;v<=n;v++)
            trace[step][v]=d[v];
        for(v=1;v<=n;v++)
            if(w[u][v]!=0)
                relax(u,v);
    }
    clock_t end=clock();
    double time_taken=(double)(end-start)/CLOCKS_PER_SEC;

    printf("\nS = { }");
    printf("\nS = {%d}  (source, d[%d] = 0)\n",s,s);
    printf("\nu     ");
    for(v=1;v<=n;v++)
        if(v!=s)
            printf("%-6d",v);
    printf("S");
    for(i=2;i<=step;i++) {
        if(trace[i][order[i]]==INF)
            break;
        printf("\n%-6d",order[i]);
        for(v=1;v<=n;v++) {
            if(v==s)
                continue;
            if(trace[i][v]==INF)
                printf("%-6s","INF");
```

```
            else if(pos[v]<=i) {
                sprintf(cell,"(%d)",trace[i][v]);
                printf("%-6s",cell);
            }
            else
                printf("%-6d",trace[i][v]);
        }
        printf("{");
        for(j=1;j<=i;j++)
            printf(j==1?"%d":", %d",order[j]);
        printf("}");
    }
    printf("\n( ) = vertex is in S, its shortest distance is final");

    printf("\n\nVertex\tDistance\tPath");
    for(i=1;i<=n;i++) {
        if(d[i]==INF)
            printf("\n%d\tunreachable",i);
        else {
            printf("\n%d\t%d\t\t",i,d[i]);
            path(i);
        }
    }
    printf("\nNumber of comparisons:%d",count);
    printf("\nTime taken:%f seconds",time_taken);
    printf("\nTime complexity: O(V^2) with adjacency matrix, O(E log V) with adjacency list");
    printf("\nSpace complexity: O(V^2) with adjacency matrix, O(V+E) with adjacency list");
    printf("\n");
    return 0;
}
```

INPUT 1

```
6
0 2 4 0 0 0
0 0 1 7 0 0
0 0 0 0 3 0
0 0 0 0 0 1
0 0 0 2 0 5
0 0 0 0 0 0
1
```

OUTPUT 1

```
Enter the number of vertices:6
Enter the 6x6 weight matrix (0 for no edge):
0 2 4 0 0 0
0 0 1 7 0 0
0 0 0 0 3 0
0 0 0 0 0 1
0 0 0 2 0 5
0 0 0 0 0 0
Enter the source vertex:1
S = { }
S = {1}  (source, d[1] = 0)

u     2     3     4     5     6     S
2     (2)   4     INF   INF   INF   {1, 2}
3     (2)   (3)   9     INF   INF   {1, 2, 3}
5     (2)   (3)   9     (6)   INF   {1, 2, 3, 5}
4     (2)   (3)   (8)   (6)   11    {1, 2, 3, 5, 4}
6     (2)   (3)   (8)   (6)   (9)   {1, 2, 3, 5, 4, 6}
( ) = vertex is in S, its shortest distance is final

Vertex  Distance        Path
1       0               1
2       2               1 -> 2
3       3               1 -> 2 -> 3
```

```
4       8               1 -> 2 -> 3 -> 5 -> 4
5       6               1 -> 2 -> 3 -> 5
6       9               1 -> 2 -> 3 -> 5 -> 4 -> 6
Number of comparisons:44
Time taken:0.000002 seconds
Time complexity: O(V^2) with adjacency matrix, O(E log V) with adjacency list
Space complexity: O(V^2) with adjacency matrix, O(V+E) with adjacency list
```

INPUT 2

```
6
0 50 45 10 0 0
0 0 10 15 0 0
0 0 0 0 30 0
20 0 0 0 15 0
0 20 35 0 0 0
0 0 0 0 3 0
1
```

OUTPUT 2

```
Enter the number of vertices:6
Enter the 6x6 weight matrix (0 for no edge):
0 50 45 10 0 0
0 0 10 15 0 0
0 0 0 0 30 0
20 0 0 0 15 0
0 20 35 0 0 0
0 0 0 0 3 0
Enter the source vertex:1
S = { }
S = {1}  (source, d[1] = 0)

u     2     3     4     5     6     S
4     50    45    (10)  INF   INF   {1, 4}
5     50    45    (10)  (25)  INF   {1, 4, 5}
2     (45)  45    (10)  (25)  INF   {1, 4, 5, 2}
3     (45)  (45)  (10)  (25)  INF   {1, 4, 5, 2, 3}
( ) = vertex is in S, its shortest distance is final

Vertex  Distance        Path
1       0               1
2       45              1 -> 4 -> 5 -> 2
3       45              1 -> 3
4       10              1 -> 4
5       25              1 -> 4 -> 5
6       unreachable
Number of comparisons:47
Time taken:0.000002 seconds
Time complexity: O(V^2) with adjacency matrix, O(E log V) with adjacency list
Space complexity: O(V^2) with adjacency matrix, O(V+E) with adjacency list
```

COMPLEXITY

```
Time complexity:  O(V^2) with adjacency matrix, O(E log V) with adjacency list
Space complexity: O(V^2) with adjacency matrix, O(V+E) with adjacency list
Best: O(V²)   Average: O(V²)   Worst: O(V²)   Paradigm: Greedy
```

PROGRAM 24

Floyd–Warshall

Aim: Write a program in c to find the shortest paths between all pairs of vertices using the Floyd-Warshall algorithm and find its execution time using the time function.

ALGORITHM

```
Algorithm: FLOYD-WARSHALL(W)

1. Start
2. n <- number of vertices
3. For i <- 1 to n do
4.        For j <- 1 to n do
5.              if i == j then D[i][j] <- 0
6.              else if there is an edge (i,j) then D[i][j] <- W[i][j]
7.              else D[i][j] <- infinity
8. For k <- 1 to n do
9.        For i <- 1 to n do
10.             For j <- 1 to n do
11.                   if D[i][k] + D[k][j] < D[i][j] then
12.                         D[i][j] <- D[i][k] + D[k][j]
13. Print D
14. Stop
```

PROGRAM (C)

```
#include<stdio.h>
#include<time.h>
#define INF 99999

int main() {
    int n,i,j,k,count=0;
    printf("Enter the number of vertices:");
    scanf("%d",&n);
    int D[n+1][n+1];
    printf("Enter the %dx%d weight matrix (0 for no edge):\n",n,n);
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
            scanf("%d",&D[i][j]);

    clock_t start=clock();
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
            if(i!=j && D[i][j]==0)
                D[i][j]=INF;
    for(k=1;k<=n;k++)
        for(i=1;i<=n;i++)
            for(j=1;j<=n;j++) {
                count++;
                if(D[i][k]!=INF && D[k][j]!=INF && D[i][k]+D[k][j]<D[i][j])
                    D[i][j]=D[i][k]+D[k][j];
            }
    clock_t end=clock();
    double time_taken=(double)(end-start)/CLOCKS_PER_SEC;

    printf("\nShortest distance matrix:\n");
    for(j=1;j<=n;j++)
        printf("\t%d",j);
    for(i=1;i<=n;i++) {
        printf("\n%d",i);
        for(j=1;j<=n;j++)
            if(D[i][j]==INF)
                printf("\tINF");
            else
```

```
                printf("\t%d",D[i][j]);
    }
    printf("\nNumber of comparisons:%d",count);
    printf("\nTime taken:%f seconds",time_taken);
    printf("\nTime complexity: O(V^3)");
    printf("\nSpace complexity: O(V^2)");
    printf("\n");
    return 0;
}
```

INPUT

```
4
0 3 0 7
8 0 2 0
5 0 0 1
2 0 0 0
```

OUTPUT

```
Enter the number of vertices:4
Enter the 4x4 weight matrix (0 for no edge):
0 3 0 7
8 0 2 0
5 0 0 1
2 0 0 0

Shortest distance matrix:
        1       2       3       4
1       0       3       5       6
2       5       0       2       3
3       3       6       0       1
4       2       5       7       0
Number of comparisons:64
Time taken:0.000006 seconds
Time complexity: O(V^3)
Space complexity: O(V^2)
```

COMPLEXITY

```
Time complexity:  O(V^3)
Space complexity: O(V^2)
Best: O(V³)   Average: O(V³)   Worst: O(V³)   Paradigm: Dynamic programming
```

PROGRAM 25

Bellman–Ford

Aim: Write a program in c to find the shortest path from a source vertex to all other vertices using the Bellman-Ford algorithm and find its execution time using the time function.

ALGORITHM

```
Algorithm: BELLMAN-FORD(G, w, s)

1. Start
2. For each vertex v in G do
3.        d[v] <- infinity
4.        p[v] <- NIL
5. d[s] <- 0
6. For i <- 1 to n-1 do
7.        For each edge (u,v) in G do
8.              if d[u] + w(u,v) < d[v] then
9.                    d[v] <- d[u] + w(u,v)
10.                   p[v] <- u
11. For each edge (u,v) in G do
12.       if d[u] + w(u,v) < d[v] then
13.             print "Graph contains a negative weight cycle"
14.             Stop
15. Print d[v] and the path to every vertex v using p
16. Stop
```

PROGRAM (C)

```
#include<stdio.h>
#include<time.h>
#define INF 99999

int p[50];

void path(int v) {
    if(p[v]==0) {
        printf("%d",v);
        return;
    }
    path(p[v]);
    printf(" -> %d",v);
}

int main() {
    int n,e,i,j,s,neg=0,count=0;
    printf("Enter the number of vertices:");
    scanf("%d",&n);
    printf("Enter the number of edges:");
    scanf("%d",&e);
    int u[e],v[e],w[e],d[n+1];
    printf("Enter the edges (source destination weight):\n");
    for(j=0;j<e;j++)
        scanf("%d%d%d",&u[j],&v[j],&w[j]);
    printf("Enter the source vertex:");
    scanf("%d",&s);

    clock_t start=clock();
    for(i=1;i<=n;i++) {
        d[i]=INF;
        p[i]=0;
    }
    d[s]=0;
    for(i=1;i<=n-1;i++)
        for(j=0;j<e;j++) {
```

```
            count++;
            if(d[u[j]]!=INF && d[u[j]]+w[j]<d[v[j]]) {
                d[v[j]]=d[u[j]]+w[j];
                p[v[j]]=u[j];
            }
        }
    for(j=0;j<e;j++) {
        count++;
        if(d[u[j]]!=INF && d[u[j]]+w[j]<d[v[j]]) {
            neg=1;
            break;
        }
    }
    clock_t end=clock();
    double time_taken=(double)(end-start)/CLOCKS_PER_SEC;

    if(neg)
        printf("\nGraph contains a negative weight cycle");
    else {
        printf("\nVertex\tDistance\tPath");
        for(i=1;i<=n;i++) {
            if(d[i]==INF)
                printf("\n%d\tunreachable",i);
            else {
                printf("\n%d\t%d\t\t",i,d[i]);
                path(i);
            }
        }
    }
    printf("\nNumber of comparisons:%d",count);
    printf("\nTime taken:%f seconds",time_taken);
    printf("\nTime complexity: O(VE)");
    printf("\nSpace complexity: O(V+E)");
    printf("\n");
    return 0;
}
```

INPUT

```
5
10
1 2 6
1 4 7
2 3 5
2 4 8
2 5 -4
3 2 -2
4 3 -3
4 5 9
5 1 2
5 3 7
1
```

OUTPUT

```
Enter the number of vertices:5
Enter the number of edges:10
Enter the edges (source destination weight):
1 2 6
1 4 7
2 3 5
2 4 8
2 5 -4
3 2 -2
4 3 -3
4 5 9
5 1 2
5 3 7
```

```
Enter the source vertex:1

Vertex  Distance        Path
1       0               1
2       2               1 -> 4 -> 3 -> 2
3       4               1 -> 4 -> 3
4       7               1 -> 4
5       -2              1 -> 4 -> 3 -> 2 -> 5
Number of comparisons:50
Time taken:0.000009 seconds
Time complexity: O(VE)
Space complexity: O(V+E)
```

COMPLEXITY

```
Time complexity:  O(VE)
Space complexity: O(V+E)
Best: O(VE)   Average: O(VE)   Worst: O(VE)   Paradigm: Dynamic programming
```

PROGRAM 26

Prim's MST

Aim: Write a program in c to find the minimum cost spanning tree of a graph using Prim's algorithm and find its execution time using the time function.

ALGORITHM

```
Algorithm: MST-PRIM(G, w, r)

1. for each u in V[G]
2.       do key[u] <- infinity
3.          p[u] <- NIL
4. key[r] <- 0
5. Q <- V[G]
6. while Q != Ø
7.       do u <- EXTRACT-MIN(Q)
8.          for each v in Adj[u]
9.                do if v in Q and w(u,v) < key[v]
10.                     then p[v] <- u
11.                          key[v] <- w(u,v)

EXTRACT-MIN(Q)
1. u <- the vertex in Q with minimum key[u]
2. Q <- Q - {u}
3. return u
```

PROGRAM (C)

```
#include<stdio.h>
#include<time.h>
#define INF 99999

int n,count=0;
int w[50][50],key[50],p[50],Q[50];

int extract_min() {
    int v,u=0;
    for(v=1;v<=n;v++) {
        count++;
        if(Q[v]==1 && (u==0 || key[v]<key[u]))
            u=v;
    }
    Q[u]=0;
    return u;
}

int main() {
    int i,u,v,r,q_size,step=0,cost=0;
    printf("Enter the number of vertices:");
    scanf("%d",&n);
    printf("Enter the %dx%d weight matrix (0 for no edge):\n",n,n);
    for(u=1;u<=n;u++)
        for(v=1;v<=n;v++)
            scanf("%d",&w[u][v]);
    printf("Enter the root vertex:");
    scanf("%d",&r);

    int order[n+1];

    clock_t start=clock();
    for(u=1;u<=n;u++) {
        key[u]=INF;
        p[u]=0;
    }
```

```
    key[r]=0;
    for(u=1;u<=n;u++)
        Q[u]=1;
    q_size=n;
    while(q_size!=0) {
        u=extract_min();
        q_size--;
        if(key[u]==INF)
            break;
        step++;
        order[step]=u;
        for(v=1;v<=n;v++)
            if(w[u][v]!=0) {
                count++;
                if(Q[v]==1 && w[u][v]<key[v]) {
                    p[v]=u;
                    key[v]=w[u][v];
                }
            }
    }
    clock_t end=clock();
    double time_taken=(double)(end-start)/CLOCKS_PER_SEC;

    if(step<n)
        printf("\nGraph is disconnected, no spanning tree exists");
    else {
        printf("\nEdges in the minimum spanning tree (in the order added):");
        printf("\nEdge\t\tWeight");
        for(i=2;i<=n;i++) {
            v=order[i];
            printf("\n%d - %d\t\t%d",p[v],v,key[v]);
            cost+=key[v];
        }
        printf("\nMinimum cost of spanning tree:%d",cost);
    }
    printf("\nNumber of comparisons:%d",count);
    printf("\nTime taken:%f seconds",time_taken);
    printf("\nTime complexity: O(V^2) with adjacency matrix, O(E log V) with adjacency list and binary
 heap");
    printf("\nSpace complexity: O(V^2) with adjacency matrix, O(V+E) with adjacency list");
    printf("\n");
    return 0;
}
```

INPUT

```
9
0 4 0 0 0 0 0 8 0
4 0 8 0 0 0 0 11 0
0 8 0 7 0 4 0 0 2
0 0 7 0 9 14 0 0 0
0 0 0 9 0 10 0 0 0
0 0 4 14 10 0 2 0 0
0 0 0 0 0 2 0 1 6
8 11 0 0 0 0 1 0 7
0 0 2 0 0 0 6 7 0
1
```

OUTPUT

```
Enter the number of vertices:9
Enter the 9x9 weight matrix (0 for no edge):
0 4 0 0 0 0 0 8 0
4 0 8 0 0 0 0 11 0
0 8 0 7 0 4 0 0 2
0 0 7 0 9 14 0 0 0
0 0 0 9 0 10 0 0 0
0 0 4 14 10 0 2 0 0
```

```
0 0 0 0 0 2 0 1 6
8 11 0 0 0 0 1 0 7
0 0 2 0 0 0 6 7 0
Enter the root vertex:1

Edges in the minimum spanning tree (in the order added):
Edge            Weight
1 - 2           4
2 - 3           8
3 - 9           2
3 - 6           4
6 - 7           2
7 - 8           1
3 - 4           7
4 - 5           9
Minimum cost of spanning tree:37
Number of comparisons:109
Time taken:0.000005 seconds
Time complexity: O(V^2) with adjacency matrix, O(E log V) with adjacency list and binary heap
Space complexity: O(V^2) with adjacency matrix, O(V+E) with adjacency list
```

COMPLEXITY

```
Time complexity:  O(V^2) with adjacency matrix, O(E log V) with adjacency list and binary heap
Space complexity: O(V^2) with adjacency matrix, O(V+E) with adjacency list
Best: O(V²)   Average: O(V²)   Worst: O(V²)   Paradigm: Greedy
```

PROGRAM 27

Kruskal's MST

Aim: Write a program in c to find the minimum cost spanning tree of a graph using Kruskal's algorithm and find its execution time using the time function.

ALGORITHM

```
Algorithm: MST-KRUSKAL(G, w)

1. A <- Ø
2. for each vertex v in V[G]
3.       do MAKE-SET(v)
4. sort the edges of E into nondecreasing order by weight w
5. for each edge (u,v) in E, taken in nondecreasing order by weight
6.       do if FIND-SET(u) != FIND-SET(v)
7.             then A <- A U {(u,v)}
8.                  UNION(u,v)
9. return A

MAKE-SET(x)
1. parent[x] <- x
2. rank[x] <- 0

FIND-SET(x)
1. if x != parent[x]
2.       then parent[x] <- FIND-SET(parent[x])
3. return parent[x]

UNION(x, y)
1. x <- FIND-SET(x), y <- FIND-SET(y)
2. if rank[x] > rank[y]
3.       then parent[y] <- x
4.       else parent[x] <- y
5.            if rank[x] == rank[y]
6.                  then rank[y] <- rank[y] + 1
```

PROGRAM (C)

```
#include<stdio.h>
#include<time.h>

int parent[50],rnk[50];

void make_set(int x) {
    parent[x]=x;
    rnk[x]=0;
}

int find_set(int x) {
    if(x!=parent[x])
        parent[x]=find_set(parent[x]);
    return parent[x];
}

void union_set(int x,int y) {
    x=find_set(x);
    y=find_set(y);
    if(rnk[x]>rnk[y])
        parent[y]=x;
    else {
        parent[x]=y;
        if(rnk[x]==rnk[y])
            rnk[y]++;
    }
```

```
}

int main() {
    int n,e,i,j,k=0,cost=0,count=0,tu,tv,tw;
    printf("Enter the number of vertices:");
    scanf("%d",&n);
    printf("Enter the number of edges:");
    scanf("%d",&e);
    int u[e],v[e],w[e],au[n],av[n],aw[n];
    printf("Enter the edges (vertex vertex weight):\n");
    for(i=0;i<e;i++)
        scanf("%d%d%d",&u[i],&v[i],&w[i]);

    clock_t start=clock();
    for(i=1;i<=n;i++)
        make_set(i);
    for(i=1;i<e;i++) {
        tu=u[i];
        tv=v[i];
        tw=w[i];
        j=i-1;
        while(j>=0) {
            count++;
            if(w[j]<=tw)
                break;
            u[j+1]=u[j];
            v[j+1]=v[j];
            w[j+1]=w[j];
            j--;
        }
        u[j+1]=tu;
        v[j+1]=tv;
        w[j+1]=tw;
    }
    for(i=0;i<e && k<n-1;i++) {
        count++;
        if(find_set(u[i])!=find_set(v[i])) {
            au[k]=u[i];
            av[k]=v[i];
            aw[k]=w[i];
            k++;
            union_set(u[i],v[i]);
        }
    }
    clock_t end=clock();
    double time_taken=(double)(end-start)/CLOCKS_PER_SEC;

    if(k<n-1)
        printf("\nGraph is disconnected, no spanning tree exists");
    else {
        printf("\nEdges in the minimum spanning tree (in the order added):");
        printf("\nEdge\t\tWeight");
        for(i=0;i<k;i++) {
            printf("\n%d - %d\t\t%d",au[i],av[i],aw[i]);
            cost+=aw[i];
        }
        printf("\nMinimum cost of spanning tree:%d",cost);
    }
    printf("\nNumber of comparisons:%d",count);
    printf("\nTime taken:%f seconds",time_taken);
    printf("\nTime complexity: O(E log E) with merge sort, O(E^2) with the insertion sort used here");
    printf("\nSpace complexity: O(V+E)");
    printf("\n");
    return 0;
}
```

INPUT

```
9
14
1 2 4
1 8 8
2 3 8
2 8 11
3 4 7
3 6 4
3 9 2
4 5 9
4 6 14
5 6 10
6 7 2
7 8 1
7 9 6
8 9 7
```

OUTPUT

```
Enter the number of vertices:9
Enter the number of edges:14
Enter the edges (vertex vertex weight):
1 2 4
1 8 8
2 3 8
2 8 11
3 4 7
3 6 4
3 9 2
4 5 9
4 6 14
5 6 10
6 7 2
7 8 1
7 9 6
8 9 7

Edges in the minimum spanning tree (in the order added):
Edge            Weight
7 - 8           1
3 - 9           2
6 - 7           2
1 - 2           4
3 - 6           4
3 - 4           7
1 - 8           8
4 - 5           9
Minimum cost of spanning tree:37
Number of comparisons:71
Time taken:0.000010 seconds
Time complexity: O(E log E) with merge sort, O(E^2) with the insertion sort used here
Space complexity: O(V+E)
```

COMPLEXITY

```
Time complexity:  O(E log E) with merge sort, O(E^2) with the insertion sort used here
Space complexity: O(V+E)
Best: O(E²)   Average: O(E²)   Worst: O(E²)   Paradigm: Greedy
```