



KALASALINGAM
ACADEMY OF RESEARCH & EDUCATION
(DEEMED TO BE UNIVERSITY)
Under sec. 3 of UGC Act 1956. Accredited by NAAC with "A" Grade



***Department of Computer Science and
Engineering***

***DESIGN AND
ANALYSIS OF
ALGORITHMS
(Integrated Course)***

Prepared by,
R. Raja Subramanian, Assistant Professor/ CSE

Student Name :T.M.Dheenadayalan.....

Register Number :9918004024.....

Section :A.....

TABLE OF CONTENTS		
S.No	Topic	Page No.
1	Bonafide Certificate	3
2	Experiment Evaluation Summary	4
3	Course Plan	5
4	Introduction	13
Experiments		
5	Algorithms to compute GCD	12
6	Non-recursive algorithms	20
7	Recursive algorithms	24
8	Greedy algorithms	30
9	Dynamic programming algorithms	41
10	Backtracking algorithms	49
11	Branch and Bound algorithms	57
12	Tractable and Intractable problems	65
13	Randomized Algorithms	70
14	Approximation Algorithms	75
15	Additional Experiments	80
16	References	84



KALASALINGAM
ACADEMY OF RESEARCH & EDUCATION
(DEEMED TO BE UNIVERSITY)
Under sec. 3 of UGC Act 1956. Accredited by NAAC with "A" Grade



SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

BONAFIDE CERTIFICATE

Bonafide record of work done by T.M.Dheenadayalan
of CSE in Design And Analysis Of Algorithm__
during even/odd semester in academic year 2020

Staff In-charge

Head of the Department

Submitted to the practical Examination held at Kalasalingam University, Krishnankoil on
23.11.2020

REGISTER NUMBER

9	9	1	8	0	0	4	0	2	4
---	---	---	---	---	---	---	---	---	---

Internal Examiner

External Examiner

EXPERIMENT EVALUATION SUMMARY

Name:T.M.Dheenadayalan

Reg No:9918004024

Class:III/A

Faculty:R.Raja Subramanian

S.No	Date	Experiment	Marks (100)	Faculty Signature
1	20.08.2020	Algorithms to Compute GCD		
2	27.08.2020	Non-recursive algorithms		
3	03.09.2020	Recursive algorithms		
4	10.09.2020	Greedy algorithms		
5(i)	17.09.2020	Dynamic programming algorithms 1. Implementation of Longest common subsequence problem		
5(ii)	24.09.2020	Implementation of Optimal binary search tree problem.		
6	01.10.2020	Backtracking algorithms		
7	19.10.2020	Branch and Bound algorithms		
8	02.11.2020	Tractable and Intractable problems		
9	09.11.2020	Randomized Algorithms		
10	16.11.2020	Approximation Algorithms		
11				



KALASALINGAM
ACADEMY OF RESEARCH & EDUCATION
(DEEMED TO BE UNIVERSITY)
Under sec. 3 of UGC Act 1956. Accredited by NAAC with "A" Grade



SSCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

COURSE PLAN - ODD SEMESTER 2020-2021

Subject with code	Design and Analysis of Algorithms / CSE18R173
Course	B.Tech (CSE)
Semester / Sec	V / A- D
Course Credit	4
Course Coordinator	Mrs. R. Sumathi
Module Coordinator	Dr.K.Murugeswari
Programme Coordinator	Dr. K. Kartheeban

PRE REQUISITES

- Programming for Problem Solving (CSE18R171)
- Data Structure and Algorithms (CSE18R172)

COURSE DESCRIPTION

This course provides elementary introduction to algorithm design and analysis to study various paradigms and approaches used to analyze and design algorithms and to appreciate the impact of algorithm design in practice. Students can understand how the worst-case time complexity of an algorithm is defined, how asymptotic notation is used to provide a rough classification of algorithms.

They can analyze and study how to apply various number of algorithms for fundamental problems in computer science and engineering work and compare with one another, and how there are still some problems for which it is unknown whether there exist efficient algorithms, and how to design efficient algorithms.

Students can able to Use different computational models (e.g., divide-and-conquer), order notation and various complexity measures (e.g., running time, disk space) to analyze the complexity/performance of different algorithms. Understand the difference between the lower and upper bounds of various problems and their importance in deciding the optimality of an algorithm.

CARRIER OPPORTUNITIES

A degree in complexity theory and analysis of algorithms can be combined with a business degree for employment in management or marketing in the software industry. Combined with an engineering degree, graduates can also work in the fields of architecture, computer engineering, or software engineering.

PROGRAM EDUCATIONAL OBJECTIVES COMPONENTS (PEOS)

- PEO1 :** Graduates will be technically competent to excel in IT industry and to pursue higher studies.
- PEO2 :** The Graduates will possess the skills to design and develop economically and technically feasible computing systems using modern tools and techniques.
- PEO3 :** The Graduates will have effective communication skills, team spirit, ethical principles and the desire for lifelong learning to succeed in their professional career.

PROGRAM SPECIFIC OBJECTIVES (PSOs)

PSO1 : Problem-Solving Skills: The ability to apply mathematics, science and computer engineering knowledge to analyze, design and develop cost effective computing solutions for complex problems with environmental considerations.

PSO2: Professional Skills: The ability to apply modern tools and strategies in software project development using modern programming environments to deliver a quality product for business accomplishment.

PSO3 : Communication and Team Skill : The ability to exhibit proficiency in oral and written communication as individual or as part of a team to work effectively with professional behaviors and ethics.

PSO4 : Successful Career and Entrepreneurship : The ability to create a inventive career path by applying innovative project management techniques to become a successful software professional, an entrepreneur or zest for higher studies.

PROGRAMME OUTCOMES (POS)

- PO1 :** Ability to apply knowledge of mathematics, science and computer engineering to solve computational problems.
- PO2 :** Ability to Identify, formulate, analyze and derive conclusions in complex computing problems.
- PO3 :** Capability to design and develop computing systems to meet the requirement of industry and society with due consideration for public health, safety and environment.
- PO4 :** Ability to apply knowledge of design of experiment and data analysis to derive solutions in complex computing problems.
- PO5 :** Ability to develop and apply modeling, simulation and prediction tools and techniques to engineering problems.
- PO6 :** Ability to assess and understand the professional, legal, security and societal responsibilities relevant to computer engineering practice.

- PO7** : Ability to understand the impact of computing solutions in economic, environmental and societal context for sustainable development.
- PO8** : Applying ethical principles and commitment to ethics of IT and software profession.
- PO9** : Ability to work effectively as an individual as well as in teams.
- PO10** :Ability to communicate effectively with technical community and with society.
- PO11** :Demonstrating and applying the knowledge of computer engineering and management principles in software project development and in multidisciplinary areas.
- PO12** :Understanding the need for technological changes and engage in life-long learning.

COURSE OUTCOMES

CO1 : Understand asymptotic notations to Analyze the performance of algorithms

CO2 : Identify the differences in design techniques and apply to solve optimization problems.

CO3: Apply algorithms for performing operations on graphs and trees.

CO4: Formulate novel problems, by choosing the appropriate algorithm design technique for their solution and justify their selection

CO5: Evaluate deterministic and nondeterministic algorithms to solve complex Problems

PO and PEO Mapping

	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
PSO1	S	S	S	S	S		L			L	S	S
PSO2			L	S	S	S	S				S	S
PSO3					S	S	S	S	S	S	L	S
PSO4								S			L	S

CO & PO MAPPING

	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO1	S											
CO2	L	S									L	
CO3				S								
CO4		S			M						L	
CO5				S							M	M

S –Strong correlation M – Moderate Correlation L – Least Correlation

List of experiments:

Experiment No.	Name of the Experiment	No. of periods	Cumulative No. of periods
1.	Programs to compute GCD a) Implementation of Euclid's Algorithm b) Implementation of Consecutive Integer Checking Algorithm c) Implementation of Middle School Procedure	2	2
2.	Non-recursive Algorithms a) Unique elements in an Array	1	3
3.	Recursive Algorithms a) Tower of Hanoi Problem b) Number of digits in binary using recursion	3	6
4.	Greedy Algorithms a) Single Source Shortest Path Algorithm b) Huffman tree Algorithm c) Task Scheduling Algorithm	4	10
5.	Dynamic Programming Algorithms a) Longest Common Subsequence b) Optimal Binary Tree	4	14
6.	Backtracking Algorithms a) Hamiltonian Circuit Problem b) Subset-Sum Problem	4	18
7.	Branch and Bound Algorithms a) Knapsack Problem b) Traveling Salesman Problem	4	22
8.	Tractable and Intractable Problems a) Vertex Cover Problem	2	24
9.	Randomization Algorithms a) Randomized Quick Sort	1	25
10.	Approximation Algorithms a) Graph Coloring Problem	1	26

New Additional Experiments:

1. Implementation of LV10 Algorithm.
2. Implementation of Set Cover Algorithm using Randomized Rounding.

Assessment Method:

S.no	Assessment	Split up
1	Internal Assessment (20 marks)	Regular Lab Experiment (10)
		Model Lab (10)
2	External Assessment (15 marks)	End semester Lab (15)

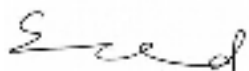
Rubrics for Internal and External Experiment Assessment:

Module	Rubrics for assessment	Marks (100)
Efficiency of Algorithm	<ul style="list-style-type: none"> Poor : 0 - 5 (Not able to understand what is given and what is expected) Normal : 5 - 10 (Understood what is given but can't decide what is expected)) Good : 10 - 15 (Understood what is given and Understood the stated expectation) Very Good : 15 - 20 (Understood what is given and understood the stated expectation as well as the hidden expectation) 	20
Efficiency of program	<ul style="list-style-type: none"> Extraordinary : 35-40 Marks (With good time and space complexity) Used efficient algorithms : 25 - 35 Met problem requirements : 15 - 25 Poor Logic : 0 - 15 Marks 	40
Output	<ul style="list-style-type: none"> Aesthetic Output : 15 - 20 User interactive input and output : 5 - 15 No proper user interactive I/O operation : 0 - 5 	20
Viva questions	<ul style="list-style-type: none"> Answered for more than 80 % Qs : 16 - 20 Marks 50% - 80% - 11 - 15 Marks 25% - 50 % - 6- 10 Marks 0%-25% - 0 - 5 Marks 	20

Mark Distribution for Regular Experiments:

S.No	Experiments	Efficiency of Algorithm	Efficiency of program	Output	VIVA VOICE	
					Technical	Communication
1.	Algorithms to compute GCD	25	25	20	20	10
2.	Non-recursive Algorithms	25	30	15	20	10
3.	Recursive Algorithms	25	30	15	20	10

4.	Greedy Algorithms	30	30	10	20	10
5.	Dynamic Programming Algorithms	30	30	10	20	10
6.	Backtracking Algorithms	30	25	15	20	10
7.	Branch and Bound Algorithms	30	25	15	20	10
8.	Tractable and Intractable Problems	25	25	15	25	10
9.	Randomization Algorithms	30	30	20	10	10
10.	Approximation Algorithms	30	30	20	10	10



Course Coordinator



Module Coordinator



Programme Coordinator



HOD/CSE

INTRODUCTION

In computer science, the **analysis of algorithms** is the determination of the computational complexity of algorithms that is the amount of time, storage and /or other resources necessary to execute them. Usually, this involves determining a function that relates the length of an algorithm's input to the number of steps it takes (its time complexity) or the number of storage locations it uses (its space complexity). An algorithm is said to be efficient when this function's values are small. Since different inputs of the same length may cause the algorithm to have different behavior, the function describing its performance is usually an upper bound on the actual performance, determined from the worst case inputs to the algorithm.

The main objective of this course is to identify and apply suitable data structures and algorithms. Also students may need to make time and space complexity analysis by comparing it with other possible data structures and algorithms. The laboratory course is integrated with theory contents, hence students can use their theoretical strength and apply the same in practical to upgrade their knowledge. The algorithms can possibly be implemented in languages such as C, C++, Java.

The laboratory contents include Seven categories of algorithms: Recursive and Non-recursive algorithms, Greedy algorithms, Dynamic programming algorithms, Backtracking algorithms, Branch and Bound algorithms, Randomization and approximation algorithms to solve various class of problems. The contents will be dealt in theory classes before being implemented in laboratory. This helps students in mastering algorithm design for various problems and also able to provide efficient solution in terms of time and space.

Being a core course of Computer Science and Engineering, Design and Analysis of Algorithms course has more weight in Graduate Aptitude Test in Engineering (GATE). Through this integrated course, student can get more knowledge and be able to solve GATE questions from Algorithms.

Ex .No 1

Algorithms to Compute GCD

Aim:

- ✓ To design and implement an efficient algorithm to compute GCD of given numbers.
- ✓ To analyze the algorithm for its efficiency.

Greatest Common Divisor:

In mathematics, the greatest common divisor (gcd) of two or more integers, which are not all zero, is the largest positive integer that divides each of the integers. For example, the gcd of 8 and 12 is 4.

The greatest common divisor is also known as the greatest common factor (gcf), highest common factor (hcf), greatest common measure (gcm), or highest common divisor.

This notion can be extended to polynomials (see Polynomial greatest common divisor) and other commutative rings.

List of exercises:

- a) Implementation of Euclid's Algorithm
- b) Implementation of Consecutive Integer Checking Algorithm
- c) Implementation of Middle School Procedure Algorithms

Ex. 1.a.**Euclid's Algorithm to compute GCD****Aim:**

To implement Euclid's algorithm for solving GCD of given numbers.

Algorithm:

1. Enter any two integers.
2. Verify it is a non zero number or not.
3. Find the modulus and store it in r.
4. Verify m,n,r are equal.
5. If it is equal print the GCD as m.

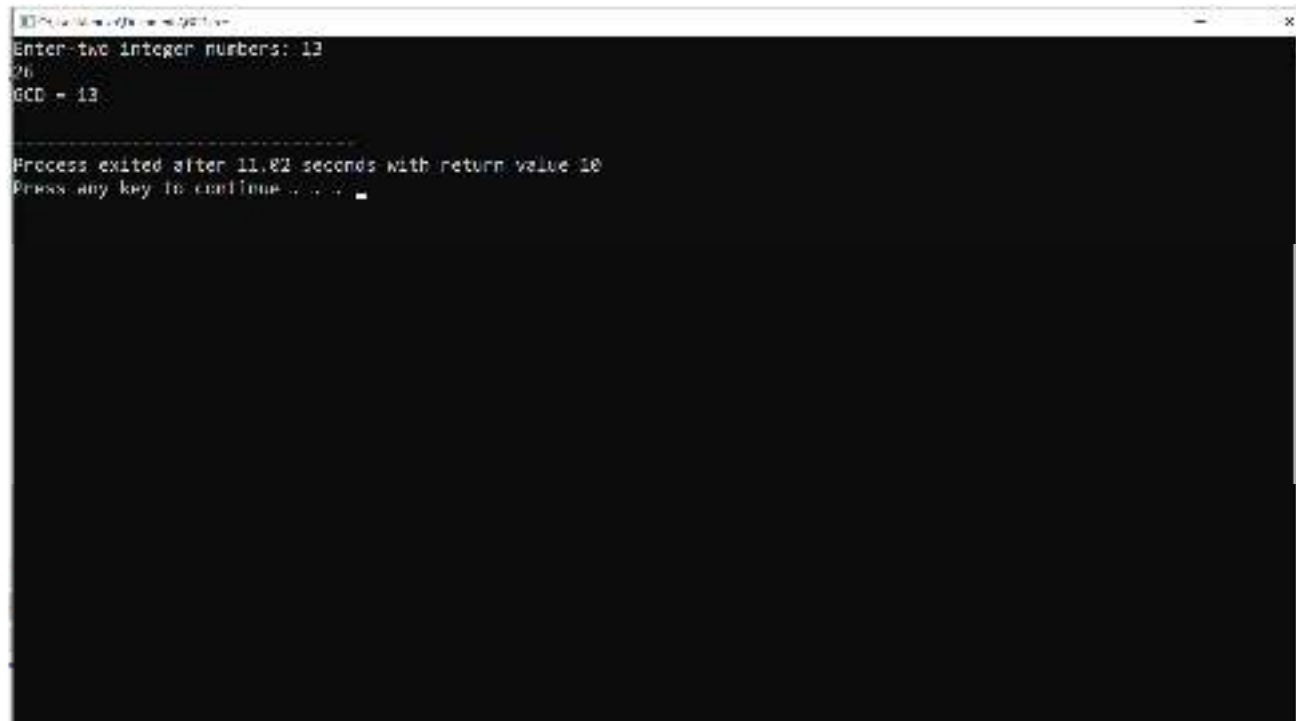
```
while n != 0
r = m mod n
m = n
n = r
return m
```

Program:

```
#include <iostream>
using namespace std;
void main()
{
int m, n;
cout<<"Enter two integer numbers:";
cin>>m>>n;
while (n > 0)
{
int r = m % n;
m = n;

n = r;
}
cout<<"GCD ="<<" "<<m;
}
```

Sample Input and Output:



```
Enter two integer numbers: 12
26
GCD = 13

-----
Process exited after 11.82 seconds with return value 10
Press any key to continue . . .
```

Ex. 1.b.**Consecutive Integer Checking Algorithm to compute GCD****Aim:**

To implement Consecutive integer checking algorithm for solving GCD of given numbers.

Algorithm:

1. Assign value of $\text{sml}\{m, n\}$ to n
2. Divide m by t .
If remainder is 0, go to Step 3;
Otherwise, go to Step 4
3. Divide n by t .
If remainder is 0, return t and stop;
Otherwise, go to Step 4
4. Decrease t by 1 and go to Step 2
5. Print the GCD

Program:

```
#include<iostream>
using namespace std;
int gcd(int m,int n){
int t,sml;
if(m<n){
sml=m;
}
else{
sml=n;
}
t=sml;
while(t>0){
if(m%t==0 && n%t==0){
return t;
}
t=t-1;
}
}
int main(){
int m,n,res;
cout<<"Enter Two Values";
cin>>m>>n;
res=gcd(m,n);
```



```
cout<<"gcd is "<<" "<<res;  
return 0;
```

Sample Input and Output:



```
g++ 12.cpp -std=c++11 -o 12.exe  
Enter Two Values  
12  
16  
gcd is 4  
-----  
Process exited after 18.1 seconds with return value 0  
Press any key to continue . . .
```

Ex. 1.c.**Middle School Procedure to compute GCD****Aim:**

To implement Middle School Procedure for solving GCD of given numbers.

Algorithm:

step 1: Find the prime factors of m. Go to step 2.

step 2: Find the prime factors of n. Go to step 3.

step 3: Identify all the common factors between m and n. proceed to step 4.

step 4: Compute the product of all the common divisors of the two numbers. and the product is the required GCD.

Program:

```
#include<iostream>
using namespace std;
int GCD (int a,int b)
{
    int i,c,n=0,m=0,ans=1;
    c=a>=b?b:a;
    for(i=2;i<=c;i++)
    {
        n=0;
        m=0;
        if(a%i==0){
            n=1;
            a=a/i;
        }
        if(b%i==0){
            m=1;
            b=b/i;
        }
        if(n==1&&m==1){
            ans=ans*i;
        }
        if(n==1|| m==1){
            i--;
        }
    }
    return(ans);
}
int main(){
    int a=12,b=16,gcd;
```

```
gcd=GCD(a,b);  
cout<<"gcd = "<<gcd;  
}
```

Sample Input and Output:



```
gcd-4  
Process exited after 0.00000 seconds with return value 5  
Press any key to continue . . .
```

Viva questions

1. Define algorithms.
2. What is the difference between GCD and LCM?
3. What is the complexity of solving GCD using Euclid's Algorithm?
4. Explain the complexity of solving GCD using Consecutive Integer Checking Algorithm?
5. Explain the complexity of solving GCD using Middle School Procedure?

Sl.No	Module	Max. Marks	Marks Obtained
1	Efficiency of Algorithm	10	
2	Efficiency of program	10	
3	Output	10	
4	Technical Skill	10	
5	Communication Skill	10	
Total		50	
Faculty Signature			

Result:

Efficient algorithm to compute GCD of given numbers is implemented and analyzed the algorithm for its efficiency.

Ex .No 2

Non-recursive Algorithms

Aim

- ✓ To design and implement non-recursive algorithms and analyze the algorithms.

Non-recursive Algorithms

General plan for non-recursive algorithms:

- ✓ Decide on parameters indicating an input size.
- ✓ Identify the algorithm's basic operation.
- ✓ Check the number of times the basic operation is executed.
- ✓ Set-up sum expressing the basic operation is executed.
- ✓ Use standard formulas and rules of sum for manipulation.

List of Exercises

1. Identification of unique elements of an array using non-recursive algorithm.

Ex. 2.a.**Identification of unique elements in an array****Aim:**

To implement and analyze the algorithm to identify unique elements in an array.

Algorithm:

- 1.Create Function unique
- 2.Put For Loop and Give Condition whether given array is unique or not
- 3.Create main function
- 4.Enter Array Elements
- 5.Check Given Array is unique or not

Program:

```
#include<iostream>
using namespace std;
int unique(int a[],int n)
{
    for(int i=0;i<=n-2;i++)
    {
        for(int j=0;j<=n-1;j++)
        {
            if(a[i]==a[j])
                return 0;
        }
    }
    return 1;
}
int main()
{
    int a[5],n;
    cout<<"Enter Number of Elements";
    cin>>n;
    cout<<"Enter the inputs";
    for(int i=0;i<n;i++)
        cin>>a[i];
    if(unique(a,n))
        cout<<"The Array is unique";
    else
        cout<<"The Array is Not Unique";
    return 0;}
```

Sample Input and Output:

```
Enter Number of Elements:5
enter the inputs:
1
1
2
4
The array is not unique
-----
Process exited after 7.228 seconds with return value 0
Press any key to continue . . .
```

Viva questions

1. State some problems that could possibly be solved using non-recursive approach.
2. Explain the complexity of identifying unique elements in an array using non-recursive algorithm.
3. Explain the general procedure for non-recursive algorithm.
4. State the differences between recursive and non-recursive algorithms.
5. List out the advantages and disadvantages of non-recursive algorithms.

Sl.No	Module	Max. Marks	Marks Obtained
1	Efficiency of Algorithm	10	
2	Efficiency of program	10	
3	Output	10	
4	Technical Skill	10	
5	Communication Skill	10	
Total		50	
Faculty Signature			

Result:

Hence Identification of unique elements of an array using non-recursive algorithm is implemented

Ex .No 3

Recursive Algorithms

Aim

- ✓ To design and implement recursive algorithms and analyze the algorithms.

Recursive Algorithms

General plan for Recursive algorithms is:

- ✓ Decide on parameters indicating an input size.
- ✓ Identify the algorithm's basic operation.
- ✓ Check the number of times the basic operation is executed.
- ✓ Set-up a recurrence relation with an appropriate initial condition.
- ✓ Solve the recurrence relation.

List of Exercises

1. Tower of Hanoi Problem
2. Number of digits in binary using recursion

Ex. 3.a.**Tower of Hanoi Problem****Aim:**

To implement and analyze a recursive algorithm to solve Tower of Hanoi problem.

Algorithm:

- 1.Create a function TOH with int n
- 2.Put “if” Loop
- 3.if(n==1) move the disc from “from tower” to the “to tower”
- 4.else call function TOH for “n-1”.Recall the function TOH “n-1” disc and move it from “from tower” to “to tower”
- 5.Recall function again until the number of disc is 1.
- 6.Create Main Function
- 7.Enter No Of Rings

Program:

```
#include<iostream>
using namespace std;
void TOH(int n,char source,char dest,char inter)
{
    if(n==1)
    {
        cout<<"Move ring 1 from tower"<<source<<"to tower"<<dest<<"\n";
        return;
    }
    TOH(n-1,source,inter,dest);
    cout<<"Move ring"<<n<<"from tower"<<source<<"to tower"<<dest<<"\n";
    TOH(n-1,inter,dest,source);
}
int main()
{
    int n;
    cout<<"Enter Number of Rings";
    cin>>n;
    TOH(n,'A','C','B');
    return 0;
}
```

Sample Input and Output:

[illegible]

Ex. 3.b.**Identification of number of digits in Binary****Aim:**

To implement and analyze a recursive algorithm to identify number of digits in binary for a given decimal number.

Algorithm:

- 1.Create a function recbinarydec with int f
- 2.Put "if" loop
- 3.if(f==1) return 1 else return recbinarydec(f/2)+1;
- 4.Create main function
- 5.Enter f value
- 6.Print The no of digits in binary is recbinarydec(f)

Program:

```
#include<iostream>
using namespace std;
int recbinarydec(int f)
{
    if(f==1)
        return 1;
    else
        return recbinarydec(f/2)+1;
}
int main()
{
    int f;
    cout<<"Enter f value";
    cin>>f;
    cout<<"The no of digits in binary is"<<recbinarydec(f);
    return 0;
}
```

Sample Input and Output:

```
8: C:\Users\Arun\Desktop>python3 a.py
Enter a value: 2
The no. of digits in binary is: 2
-----
Process ended after 11.1 seconds with return value 0
Press any key to continue . . .
```

Viva questions

1. Explain the general procedure for recursive algorithms.
2. State some problems that could effectively be solved using recursive algorithms.
3. Explain the time complexity for solving Tower of Hanoi problem.
4. Explain the time complexity for identifying number of digits in binary for a given decimal number using Recursive algorithms.
5. List out the advantages of recursive algorithms.

Sl.No	Module	Max. Marks	Marks Obtained
1	Efficiency of Algorithm	10	
2	Efficiency of program	10	
3	Output	10	
4	Technical Skill	10	
5	Communication Skill	10	
Total		50	
Faculty Signature			

Result:

Recursive algorithms and analyze the algorithms of Tower of Hanoi Problem and Number of digits in binary using recursion is implemented .

Aim

- ✓ To understand the fundamentals of Greedy algorithms.
- ✓ To design and implement various problems using Greedy approach and analyze their complexities.

Greedy Algorithm

A **greedy algorithm** is an algorithmic paradigm that follows the problem solving heuristic of making the locally optimal choice at each stage with the intent of finding a global optimum. In many problems, a greedy strategy does not usually produce an optimal solution, but nonetheless a greedy heuristic may yield locally optimal solutions that approximate a globally optimal solution in a reasonable amount of time.

In graph theory, the **shortest path problem** is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized.

The problem of finding the shortest path between two intersections on a road map (the graph's vertices correspond to intersections and the edges correspond to road segments, each weighted by the length of its road segment) may be modeled by a special case of the shortest path problem in graphs.

In computer science and information theory, a **Huffman code** is a particular type of optimal prefix code that is commonly used for lossless data compression. The output from Huffman's algorithm can be viewed as a variable-length code table for encoding a source symbol (such as a character in a file). The algorithm derives this table from the estimated probability or frequency of occurrence (*weight*) for each possible value of the source symbol. As in other entropy encoding methods, more common symbols are generally represented using fewer bits than less common symbols. Huffman's method can be efficiently implemented, finding a code in time linear to the number of input weights if these weights are sorted.^[2] However, although optimal among methods encoding symbols separately, Huffman coding is not always optimal among all compression methods.

List of Exercises

1. Single Source Shortest Path Algorithm
2. Huffman tree Algorithm
3. Task Scheduling Algorithm

Ex. 4.a.**Single Source Shortest Path Algorithm****Aim:**

To implement and analyze Single source shortest path algorithm using Greedy approach.

Algorithm

1. Declare two 2D array for getting adjacency matrix and to calculate distance matrix.

2. Get input from user for vertices and adjacency matrix

3. `int min(int k,int u)`

```
{  
    int min1,i;  
    min1=dist[k-1][1]+cost[i][u];  
    for(i=2;i<=n;i++)  
        if(min1>(dist[u-1][i]+cost[i][u]))  
            min1=dist[k-1][i]+cost[i][u];  
    return min1;  
}
```

4. From the above pseudo code we can find the min value

5. `for(k=2;k<n;k++)`

```
{  
    for(u=2;u<=n;u++)  
    {  
        m=min(k,u);  
        if(dist[k-1][u]>m)  
            dist[k][u]=m;  
        else  
            dist[k][u]=dist[k-1][u];  
        printf("%d",dist[k][u]);  
    }  
    printf("\n");  
    printf("\n");  
}
```

6. using above pseudo code we can use to calculate the distance matrix

7. then print the values and calculated values

Program:

```
#include<stdio.h>
int cost[8][8],dist[8][8],n;
int main()
{
    int i,j,k,m,u,INF=100;
    printf("\nEnter number of vertices");
    scanf("%d",&n);
    printf("\nEnter the adjacency matrix(Enter if there is no edge)");
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=n;j++)
        {
            printf("\nEnter weight of %d to %d:",i,j);
            scanf("%d",&cost[i][j]);
        }
    }
    for(i=1;i<=n;i++)
    {
        dist[i][i]=0;
    }
    printf("\nDistance matrix\n");
    for(i=2;i<=n;i++)
    {
        dist[1][i]=cost[1][i];
        printf("%d",dist[1][i]);
    }
    printf("\n");
    printf("\n");
    for(k=2;k<n;k++)
    {
        for(u=2;u<=n;u++)
        {
            m=min(k,u);
            if(dist[k-1][u]>m)
                dist[k][u]=m;
            else
                dist[k][u]=dist[k-1][u];
            printf("%d",dist[k][u]);
        }
        printf("\n");
        printf("\n");
    }
    for(i=1;i<=n;i++)
    {
        printf("\nDistance of 1 to %d is",i);
        printf("%d",dist[n-1][i]);
    }
    return 0;
```

```

}
int min(int k,int u)
{
    int min1,i;
    min1=dist[k-1][1]+cost[i][u];
    for(i=2;i<=n;i++)
        if(min1>(dist[u-1][i]+cost[i][u]))
            min1=dist[k-1][i]+cost[i][u];
    return min1;
}

```

Sample Input and Output:

The screenshot shows a C++ IDE with the following code and output:

```

1 // Finding the minimum weight of a path from node 1 to node n, considering a constraint on the number of edges (k).
2 int cost[10][10],dist[10][10];
3 int min1;
4 int n,k;
5 int main()
6 {
7     cin >> n >> k;
8     // Reading the cost matrix
9     for(int i=1;i<=n;i++)
10         for(int j=1;j<=n;j++)
11             cin >> cost[i][j];
12     // Finding the minimum weight of a path from node 1 to node n, considering a constraint on the number of edges (k).
13     for(int k=1;k<=k;k++)
14         for(int u=1;u<=n;u++)
15             min1=dist[k-1][1]+cost[1][u];
16             for(i=2;i<=n;i++)
17                 if(min1>(dist[u-1][i]+cost[i][u]))
18                     min1=dist[k-1][i]+cost[i][u];
19     dist[k][1]=min1;
20     // Printing the minimum weight of a path from node 1 to node n, considering a constraint on the number of edges (k).
21     for(int k=1;k<=k;k++)
22         for(int u=1;u<=n;u++)
23             cout << dist[k][u] << " ";
24     cout << endl;
25 }

```

Output:

```

Enter weight of 1 to 1
Enter weight of 1 to 2
Enter weight of 1 to 3
Enter weight of 2 to 1
Enter weight of 2 to 2
Enter weight of 2 to 3
Enter weight of 3 to 1
Enter weight of 3 to 2
Enter weight of 3 to 3
1 1 1
1 2 2
1 3 3
2 1 2
2 2 1
2 3 2
3 1 3
3 2 3
3 3 1
Process ended after 0.05 seconds with return value 0

```

Ex. 4.b.**Huffman Tree Algorithm****Aim:**

To implement and analyze Huffman Tree algorithm to compress a message.

Algorithm

- 1) Calculate the frequency of each character in the string.
- 2) Sort the characters in increasing order of the frequency. These are stored in a priority queue
- 3) Make each unique character as a leaf node.
- 4) Select 2 minimum frequency symbols and merge them repeatedly
- 5) Build a tree, Created a HeapNode class and used objects to maintain tree structure
- 6) Write the result to an output binary file, which will be our compressed

Program:

```
from collections import Counter
```

```
inp_str = input("enter a string to compress")
frequency = Counter(inp_str)
print(frequency)
for i in frequency.items():
    print(i)
```

```
huff_tree=[]
elements=list(frequency.items())
```

```
k=1
flag=1
while(len(elements)>1 or flag==1):
    huff_tree=[]
    length=len(elements)
    while(length>1):
        left=elements.pop()
        right=elements.pop()
        rootfreq=left[k]+right[k]

        huff_tree.append([left,right,rootfreq])
        length=len(elements)
    if(len(elements)==1):
        skew = elements.pop()
        last=huff_tree.pop()
        skewfreq=skew[k]+last[k]
```

```

    huff_tree.append([last,skew,skewfreq])
elements = list(huff_tree)
flag=0
k=2
for i in huff_tree:
    print(i)
print(huff_tree)

```

Sample Input and Output:

The screenshot shows a Jupyter Notebook with a file explorer on the left, a code editor in the center, and an output window at the bottom.

Code Editor:

```

def huffman_encoding(data):
    elements = list(huffman_tree.items())
    huff_tree = []
    while len(elements) > 1:
        k = 1
        flag = 0
        for i in elements:
            left = i[1][0]
            right = i[1][1]
            left = left + right
            right = left + right
            huff_tree.append([left, right, left + right])
    return huff_tree

```

Output Window:

```

Enter a string to encode: AAAAAAAAAA
Data: AAAAAAAAAA
Huffman code is 00000000000000000000000000000000
Huffman compression ratio is 0.5
Huffman compression ratio is 0.5
Process finished with code 0

```

Ex. 4.c.**Task Scheduling Algorithm****Aim:**

To implement and analyze Task scheduling algorithm using Greedy approach.

Algorithm

- 1) Sort all jobs in decreasing order of profit.
- 2) Iterate on jobs in decreasing order of profit. For each job, do the following :
- 3) Find a time slot i , such that slot is empty and $i < \text{deadline}$ and i is greatest. Put the job in this slot and mark this slot filled.
- 4) If no such i exists, then ignore the job

Program:

```
taskset=[]
n=int(input("Enter no of tasks"))
maxslot=0
for i in range(n):
    inp=input("enter task_num,profit and deadline(separated by commas):")
    inp=inp.split(",")
    inp[1]=int(inp[1])
    inp[2] = int(inp[2])
    if(maxslot < inp[2]):
        maxslot = inp[2]
    taskset.append(inp)

taskset.sort(key = lambda taskset: taskset[1],reverse = True)
print("Sorting the tasks in the decending order of profits")
for i in range(n):
    print(taskset[i])
slots=[]
for i in range(maxslot):
    slots.append(0)
for i in taskset:
    pos = int(i[2])-1
    if(pos<maxslot and pos>=0):
        while(slots[pos]!=0):
            pos=pos-1
            if(pos<0):
                break
        if(pos>=0):
```

```

        slots[pos]=i
print("the task schedule is")
for i in slots:
    print(i)
opt_profit = 0
for i in slots:
    if(i!=0):
        opt_profit += i[1]
print("optional profit is",opt_profit)

```

Sample Input and Output:



The screenshot shows a C++ program running in a terminal window. The program implements a task scheduling algorithm where tasks are sorted by deadline and scheduled in order of increasing deadline. The output shows the tasks being scheduled and the total optional profit.

```

Enter no of tasks:
Enter task name, profit and deadline (separated by comma): t1, 4, 2
Enter task name, profit and deadline (separated by comma): t2, 6, 3
Enter task name, profit and deadline (separated by comma): t3, 4, 4
Sorting the tasks in the decreasing order of profit:
t3, 4, 4
t2, 6, 3
t1, 4, 2
the task schedule is

t1, 4, 2
t2, 6, 3
t3, 4, 4
optional profit is 14

...Program finished with exit code 0
Press ENTER to exit console.

```

Viva questions

1. Define Greedy algorithms.
2. Explain the components of Greedy algorithms.
3. Explain the complexity of solving Single source shortest path algorithm.
4. Explain the complexity of solving Huffman tree algorithm.
5. Explain the complexity of solving Task scheduling algorithm.

Sl.No	Module	Max. Marks	Marks Obtained
1	Efficiency of Algorithm	10	
2	Efficiency of program	10	
3	Output	10	
4	Technical Skill	10	
5	Communication Skill	10	
Total		50	
Faculty Signature			

Result:

Single Source Shortest Path Algorithm ,Huffman tree Algorithm,Task Scheduling Algorithm is implemented using Greedy approach and analyze.

Aim

- ✓ To understand the fundamentals of Dynamic programming algorithms
- ✓ To design and implement problems using Dynamic programming approach and analyze the algorithms.

Dynamic Programming

Dynamic programming is both a mathematical optimization method and a computer programming method. The method was developed by Richard Bellman in the 1950s and has found applications in numerous fields, from aerospace engineering to economics. In both contexts it refers to simplifying a complicated problem by breaking it down into simpler sub-problems in a recursive manner. While some decision problems cannot be taken apart this way, decisions that span several points in time do often break apart recursively. Likewise, in computer science, if a problem can be solved optimally by breaking it into sub-problems and then recursively finding the optimal solutions to the sub-problems, then it is said to have optimal substructure.

A subsequence is a sequence that can be derived from another sequence by deleting some elements without changing the order of the remaining elements. Longest common subsequence (*LCS*) of 2 sequences is a subsequence, with maximal length, which is common to both the sequences.

In [computer science](#), an **optimal binary search tree (Optimal BST)**, sometimes called a **weight-balanced binary tree**, is a [binary search tree](#) which provides the smallest possible search time (or [expected search time](#)) for a given sequence of accesses (or access probabilities). Optimal BSTs are generally divided into two types: static and dynamic.

List of Exercises

1. Implementation of Longest common subsequence problem.
2. Implementation of Optimal binary search tree problem.

Ex. 5.a.**Longest Common Subsequence Problem****Aim:**

To implement and analyze Longest common subsequence problem using Dynamic programming approach.

Algorithm

1. Get input from the user

2. Assign the required values for the pseudo code

3. for($i=0; i \leq \text{len1}; i++$)

$\text{LCS}[i][0]=0;$

for($j=0; j \leq \text{len2}; j++$)

$\text{LCS}[0][j]=0;$

for($i=1; i \leq \text{len1}; i++$)

{

for($j=1; j \leq \text{len2}; j++$)

{

if($\text{str1}[i-1] == \text{str2}[j-1]$)

{

$\text{LCS}[i][j]=1+\text{LCS}[i-1][j-1];$

}

else

{

$\text{LCS}[i][j]=\max(\text{LCS}[i-1][j], \text{LCS}[i][j-1]);$

}

}

}

return $\text{LCS}[\text{len1}][\text{len2}];$

4. By implementing above pseudo code we can implement the algorithm

Program:

```
#include<iostream>
#include<string>
using namespace std;
int LCS(string str1,string str2,int len1,int len2)
{
    int i,j;
    int LCS[len1+1][len2+1];
    for(i=0;i<=len1;i++)
        LCS[i][0]=0;
    for(j=0;j<=len2;j++)
        LCS[0][j]=0;
    for(i=1;i<=len1;i++)
    {
        for(j=1;j<=len2;j++)
        {
            if(str1[i-1]==str2[j-1])
            {
                LCS[i][j]=1+LCS[i-1][j-1];
            }
            else
            {
                LCS[i][j]=max(LCS[i-1][j],LCS[i][j-1]);
            }
        }
    }
    return LCS[len1][len2];
}

int main()
{
    string str1,str2;
    cout<<"enter string 1";
    cin>>str1;
    cout<<"enter string 2";
    cin>>str2;
    int len1=str1.length();
    int len2=str2.length();
    cout<<"length of the longest common subsequence is"<<LCS(str1,str2,len1,len2);
    return 0;
}
```

Sample Input and Output:

```

1 // Sample Input and Output
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     string s;
8     while (getline(cin, s))
9     {
10         int n = s.length();
11         int dp[n][n];
12         for (int i = 0; i < n; i++)
13             for (int j = 0; j < n; j++)
14                 dp[i][j] = 0;
15
16         // Base case: single character is a palindrome of length 1
17         for (int i = 0; i < n; i++)
18             dp[i][i] = 1;
19
20         // Fill the dp table
21         for (int len = 2; len <= n; len++)
22         {
23             for (int i = 0; i < n - len + 1; i++)
24             {
25                 int j = i + len - 1;
26                 if (s[i] == s[j])
27                     dp[i][j] = dp[i + 1][j - 1] + 1;
28                 else
29                     dp[i][j] = 0;
30             }
31         }
32
33         // Find the maximum value in the dp table
34         int max = 0;
35         for (int i = 0; i < n; i++)
36             for (int j = 0; j < n; j++)
37                 if (dp[i][j] > max)
38                     max = dp[i][j];
39
40         cout << max << endl;
41     }
42     return 0;
43 }
```

Ex. 5.b.**Optimal Binary Search Tree Algorithm****Aim:**

To implement and analyze Optimal binary search tree algorithm using Dynamic programming approach.

Algorithm

```
def findoptimalcost(freq, i, j):
    if(j < i):
        return 0
    if(j == 1):
        return freq[i]
    freqsum = sum(freq[i:j+1])
    min = 10000
    for k in range(i, j+1):
        cost = findoptimalcost(freq, i, k-1)+findoptimalcost(freq, k+1, j)
        if(cost < min):
            min = cost
    return min+freqsum
```

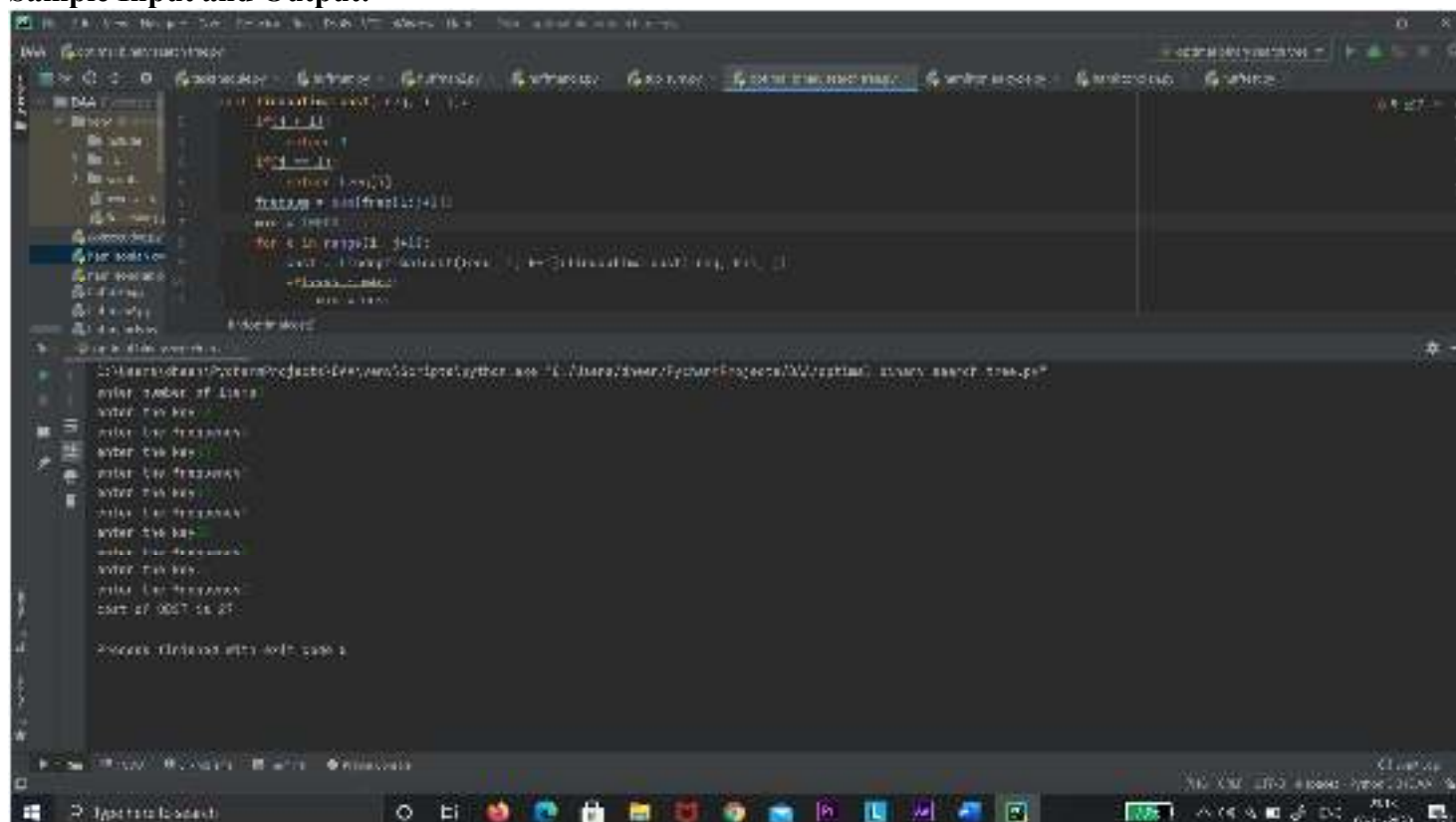
Program:

```
def findoptimalcost(freq, i, j):
    if(j < i):
        return 0
    if(j == 1):
        return freq[i]
    freqsum = sum(freq[i:j+1])
    min = 10000
    for k in range(i, j+1):
        cost = findoptimalcost(freq, i, k-1)+findoptimalcost(freq, k+1, j)
        if(cost < min):
            min = cost
    return min+freqsum
```

```
n = int(input("enter number of items"))
keys = []
freq = []
for i in range(n):
    keys.append(int(input("enter the key")))
    freq.append(int(input("enter the frequency")))

print("cost of OBST is", findoptimalcost(freq, 0, len(freq)-1))
```

Sample Input and Output:



Viva Questions

1. Define Dynamic programming strategy for solving problems.
2. Explain the components of dynamic programming algorithm.
3. Explain the complexity of solving optimal binary search tree algorithm.
4. Explain the complexity of solving longest common subsequence problem.
5. List out the advantages of Dynamic programming algorithm.

Sl.No	Module	Max. Marks	Marks Obtained
1	Efficiency of Algorithm	10	
2	Efficiency of program	10	
3	Output	10	
4	Technical Skill	10	
5	Communication Skill	10	
Total		50	
Faculty Signature			

Result:

Implementation of Longest common subsequence problem and Optimal binary search tree problem is implemented using Dynamic programming approach and analyze the algorithms.

Aim

- ✓ To understand the fundamentals of Backtracking algorithms.
- ✓ To design and implement problems using backtracking approach and analyze the algorithms.

Backtracking Algorithm

Backtracking is a general algorithm for finding all (or some) solutions to some computational problems, notably constraint satisfaction problems, that incrementally builds candidates to the solutions, and abandons a candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution.

The classic textbook example of the use of backtracking is the eight queens puzzle, that asks for all arrangements of eight chess queens on a standard chessboard so that no queen attacks any other. In the common backtracking approach, the partial candidates are arrangements of k queens in the first k rows of the board, all in different rows and columns. Any partial solution that contains two mutually attacking queens can be abandoned.

List of Exercises

1. Implementation of Hamiltonian circuit problem.
2. Implementation of Subset sum problem.

Ex. 6.a.**Hamiltonian Circuit Algorithm****Aim:**

To implement and analyze Hamiltonian circuit algorithm using Backtracking approach.

Algorithm

1. we will try to determine whether a graph contains a Hamiltonian cycle or not. And when a Hamiltonian cycle is present, also print the cycle.
2.

```
def hamiltonian(k):
    while(1):
        nextvalue(k)
        if(x[k]==0):
            return
        if((k==n) and (GRAPH[x[n]][x[1]])):
            x.append(x[1])
            print(x[1:])
            x.pop()
        else:
            hamiltonian(k+1)
```
3. graph G(V, E).
4. The algorithm finds the Hamiltonian path of the given graph.
5. For this case it is (0, 1, 2, 4, 3, 0). This graph has some other Hamiltonian paths.
6. If one graph has no Hamiltonian path, the algorithm should return false
7. Pseudo code for the algorithm
8.

```
def nextvalue(k):
    while(1):
        x[k]=((x[k]+1)%(n+1))
        if (x[k]==0):
            return
        if (k==1):
            return
        t=0
        if((GRAPH[x[k-1]][x[k]])):
            for j in range(1,k):
                t=j
                if(x[j]==x[k]):
                    break
            if((t==k-1)):
                if((k < n)or((k == n) and GRAPH[x[n]][x[1]])):
                    return
```

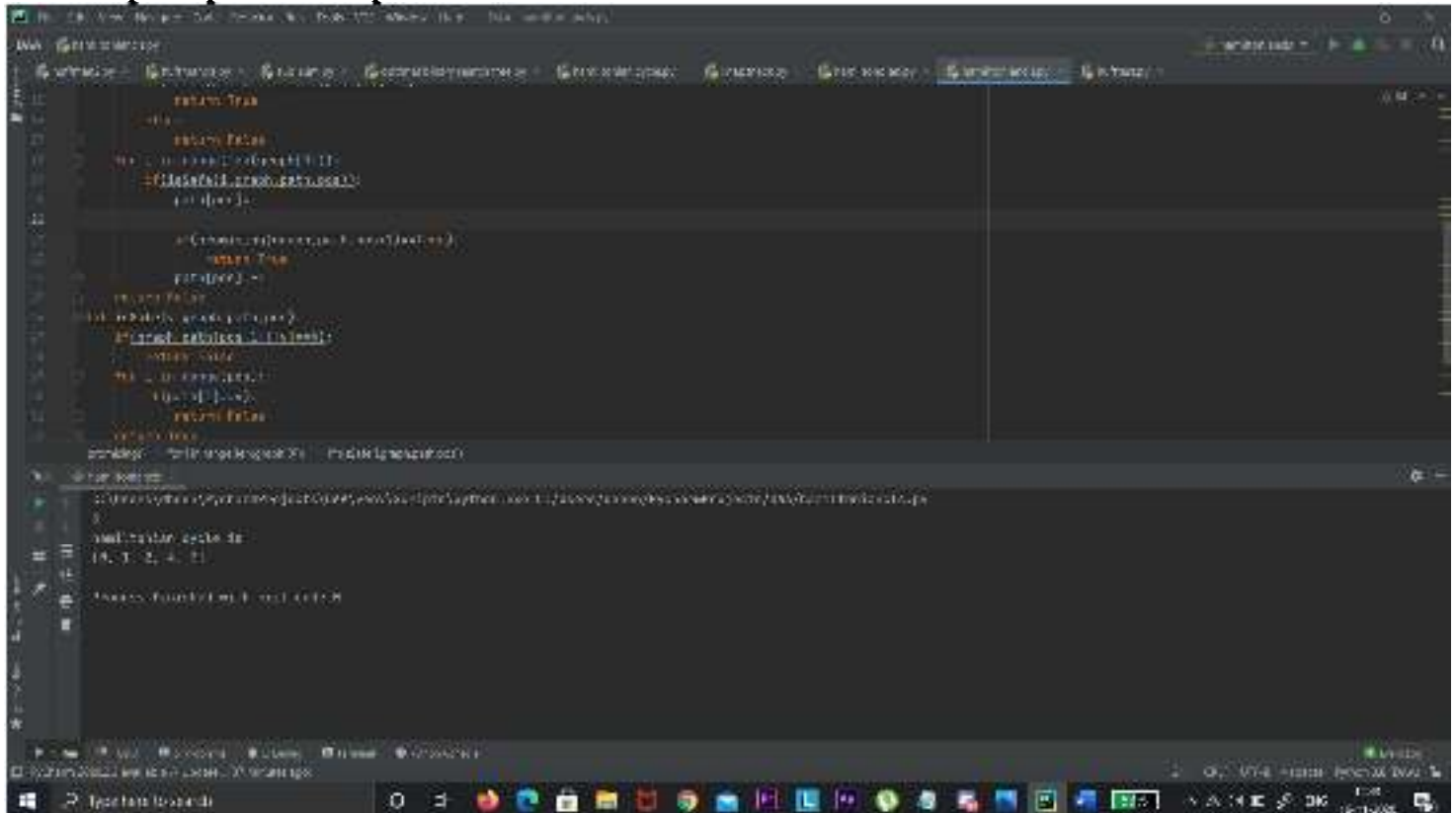

Program:

```
def hamiltonian(graph):
    path=[]
    for i in range(len(graph[0])):
        path.append(-1)
    path[0]=0
    if(promising(graph,path,1)==False):
        print("no possible solution")
        return False
    print("hamiltonian cycle is")
    print(path)
    return True
def promising(graph,path,pos):
    if(pos==(len(graph[0]))):
        if(graph[path[pos - 1]][path[0]]==1):
            return True
        else:
            return False
    for i in range(len(graph[0])):
        if(isSafe(i,graph,path,pos)):
            path[pos]=i

            if(promising(graph,path,pos+1)==True):
                return True
            path[pos]=-1
    return False
def isSafe(v,graph,path,pos):
    if(graph[path[pos-1]][v]==0):
        return False
    for i in range(pos):
        if(path[i]==v):
            return False
    return True

graph=[[0,1,0,1,0],[1,0,1,1,1],[0,1,0,0,1],[1,1,0,0,1],[0,1,1,1,0]]
print(len(graph[0]))
import time
start = time.time()
hamiltonian(graph)
end = time.time()
print(f"Runtime of the program is {end - start}")
```

Sample Input and Output:



```
1 // C++ program to find maximum element in an array using recursion
2
3 #include <iostream>
4 using namespace std;
5
6 // Function to find maximum element in an array using recursion
7 int findMax(int arr[], int n)
8 {
9     // Base case: if array has only one element, return it
10    if (n == 1)
11        return arr[0];
12
13    // Recursive case: find maximum of rest of array and compare with current element
14    int maxOfRest = findMax(arr, n - 1);
15    return max(arr[n - 1], maxOfRest);
16 }
17
18 // Driver code
19 int main()
20 {
21    int arr[] = {14, 3, -2, 4, 1};
22    int n = sizeof(arr) / sizeof(arr[0]);
23    cout << "Maximum Element in array is: " << findMax(arr, n) << endl;
24    return 0;
25 }
```

Output:

```
Maximum Element in array is: 14
```

Ex. 6.b.**Subset Sum Problem****Aim:**

To implement and analyze Subset sum problem using Backtracking approach.

Algorithm

```
for(i=0;i<po_Comb;i++)
{
    res=7;
    par_Sum=0;
    count=0;
    for(j=0;j<n;j++)
    {
        res=res*10;
        if(i & (1<<j))
        {
            res=res+1;
            par_Sum+=set[j];
            count++;
        }
        if(par_Sum>w)
            break;
        if(par_Sum==w)
        {
            answer[k]=res;
            printf("%d\n",answer[k]);
            k++;
            break;
        }
    }
}
```

Program:

```
#include<stdio.h>
#include<math.h>
int main()
{
    int set[30],w,n;
    int po_Comb,par_Sum=0,res=1,count=0;
    long answer[500],k=0;
    int inittotal=0,setttotal=0,i,j;
    printf("Enter number of elements in the set\n");
    scanf("%d",&n);
    po_Comb=pow(2,n);
    for(i=0;i<n;i++)
    {
        scanf("%d",&set[i]);
        setttotal+=set[i];
    }
    printf("Enter the sum value \n");
    scanf("%d",&w);
    for(i=0;i<po_Comb;i++)
    {
        res=7;
        par_Sum=0;
        count=0;
        for(j=0;j<n;j++)
        {
            res=res*10;
            if(i & (1<=<j))
            {
                res=res+1;
                par_Sum+=set[j];
                count++;
            }
            if(par_Sum>w)
                break;
            if(par_Sum==w)
            {
                answer[k]=res;
                printf("%d\n",answer[k]);
                k++;
                break;
            }
        }
    }
    return 0;
}
```

Sample Input and Output:

[illegible]

Viva Questions

1. Define Backtracking approach for solving problems.
2. Explain the complexity of solving Hamiltonian circuit problem using Backtracking approach.
3. Explain the complexity of solving Subset problem using Backtracking approach.
4. List out the advantages of Backtracking algorithms.
5. State some problems that could effectively be solved using Backtracking approach.

Sl.No	Module	Max. Marks	Marks Obtained
1	Efficiency of Algorithm	10	
2	Efficiency of program	10	
3	Output	10	
4	Technical Skill	10	
5	Communication Skill	10	
Total		50	
Faculty Signature			

Result:

Implementation of Hamiltonian circuit problem and Implementation of Subset sum problem is implemented using backtracking approach and analyze the algorithms.

Aim

- ✓ To learn the fundamentals of Branch and Bound algorithms.
- ✓ To design and implement problems using Branch and Bound approach and analyze the algorithms.

Branch and Bound Algorithm

Branch and bound (BB,B&B,orBnB) is an algorithm design paradigm for discrete and combinatorial optimization problems, as well as mathematical optimization. A branch and bound algorithm consists of a systematic enumeration of candidate solutions by means of state space search: the set of candidate solutions is thought of as forming a rooted tree with the full set at the root. The algorithm explores *branches* of this tree, which represent subsets of the solution set. Before enumerating the candidate solutions of a branch, the branch is checked against upper and lower estimated *bounds* on the optimal solution, and is discarded if it cannot produce a better solution than the best one found so far by the algorithm.

The algorithm depends on efficient estimation of the lower and upper bounds of regions/branches of the search space. If no bounds are available, the algorithm degenerates to an exhaustive search.

The **knapsack problem** or **rucksack problem** is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items.

The **Travelling Salesman Problem** (often called **TSP**) is a classic algorithmic problem in the field of computer science and operations research. It is focused on optimization. In this context better solution often means a solution that is cheaper. TSP is a mathematical problem. It is most easily expressed as a graph describing the locations of a set of nodes.

List of Exercises

1. Implementation of Knapsack problem.
2. Implementation of Traveling Salesman Problem.

Ex. 7.a.**Knapsack Problem****Aim:**

To implement and analyze Knapsack problem using Branch and Bound Technique.

Algorithm

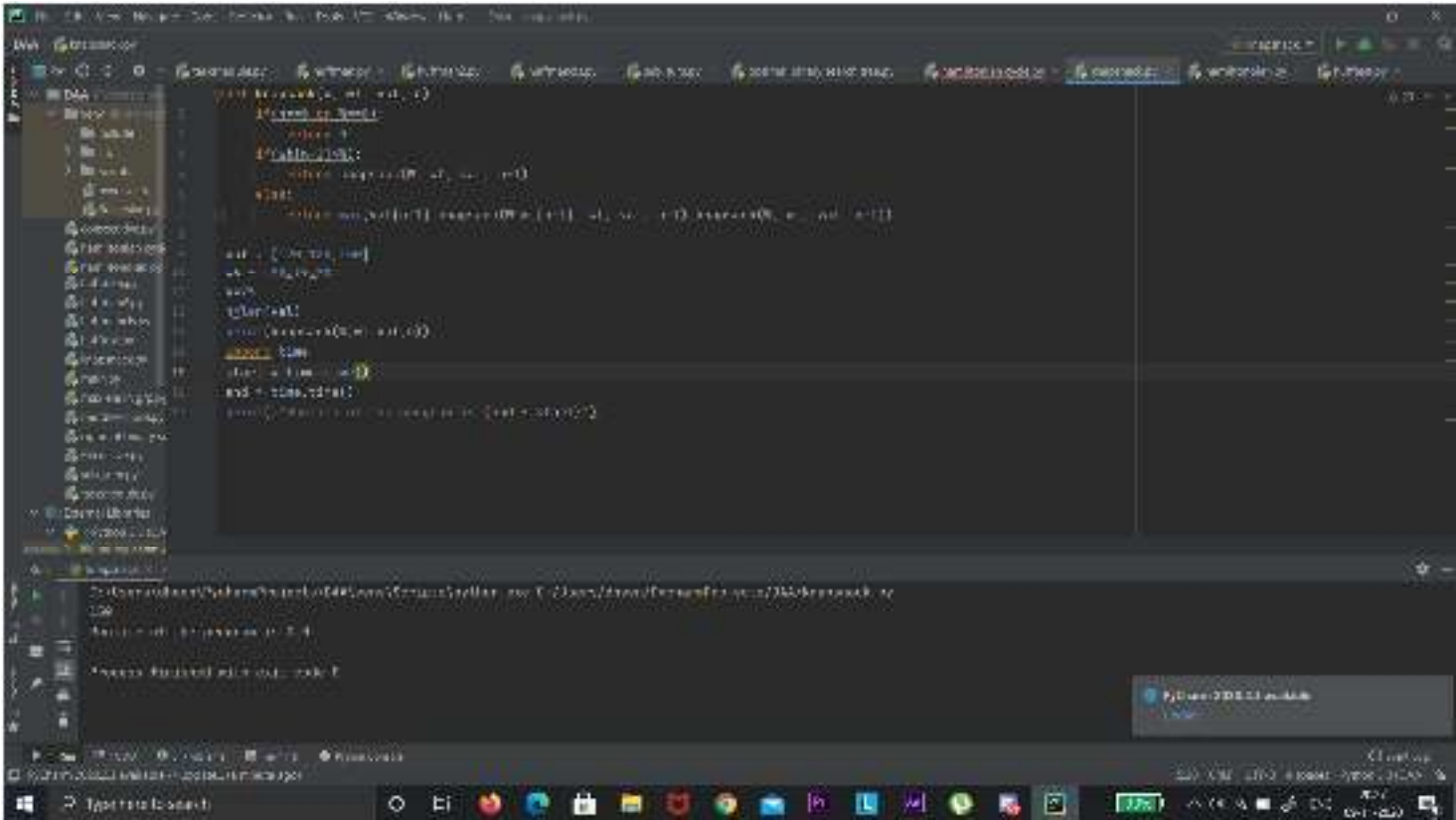
1. To maximize profit and minimize weight in capacity.
2. Each item is taken or not taken.
3. Cannot take a fractional amount of an item taken or take an item more than once.
4. It cannot be solved by the Greedy Approach because it is enable to fill the knapsack to capacity.
5. Greedy Approach doesn't ensure an Optimal Solution.
- 6

```
def knapsack(W, wt, val, n):  
    if(n==0 or W==0):  
        return 0  
    if(wt[n-1]>W):  
        return knapsack(W, wt, val, n-1)  
    else:  
        return max(val[n-1]+knapsack(W-wt[n-1], wt, val, n-1),knapsack(W, wt, val, n-1))
```
- 7 by using above pseudo code we can implement the knapsack problem

Program:

```
def knapsack(W, wt, val, n):  
    if(n==0 or W==0):  
        return 0  
    if(wt[n-1]>W):  
        return knapsack(W, wt, val, n-1)  
    else:  
        return max(val[n-1]+knapsack(W-wt[n-1], wt, val, n-1),knapsack(W, wt, val, n-1))  
  
val = [120,120,150]  
wt = [90,80,70]  
W=75  
n=len(val)  
print(knapsack(W,wt,val,n))  
import time  
start = time.time()  
end = time.time()  
print(f"Runtime of the program is {end - start}")
```

Sample Input and Output:



Ex. 7.b.**Traveling Salesman Algorithm****Aim:**

To implement and analyze Traveling Salesman algorithm using Branch and Bound Technique.

Algorithm

1. Consider city 1 as the starting and ending point.
2. Since the route is cyclic, we can consider any point as a starting point.
3. Generate all $(n-1)!$ permutations of cities.
4. Calculate the cost of every permutation and keep track of the minimum cost permutation.
5. Return the permutation with minimum cost

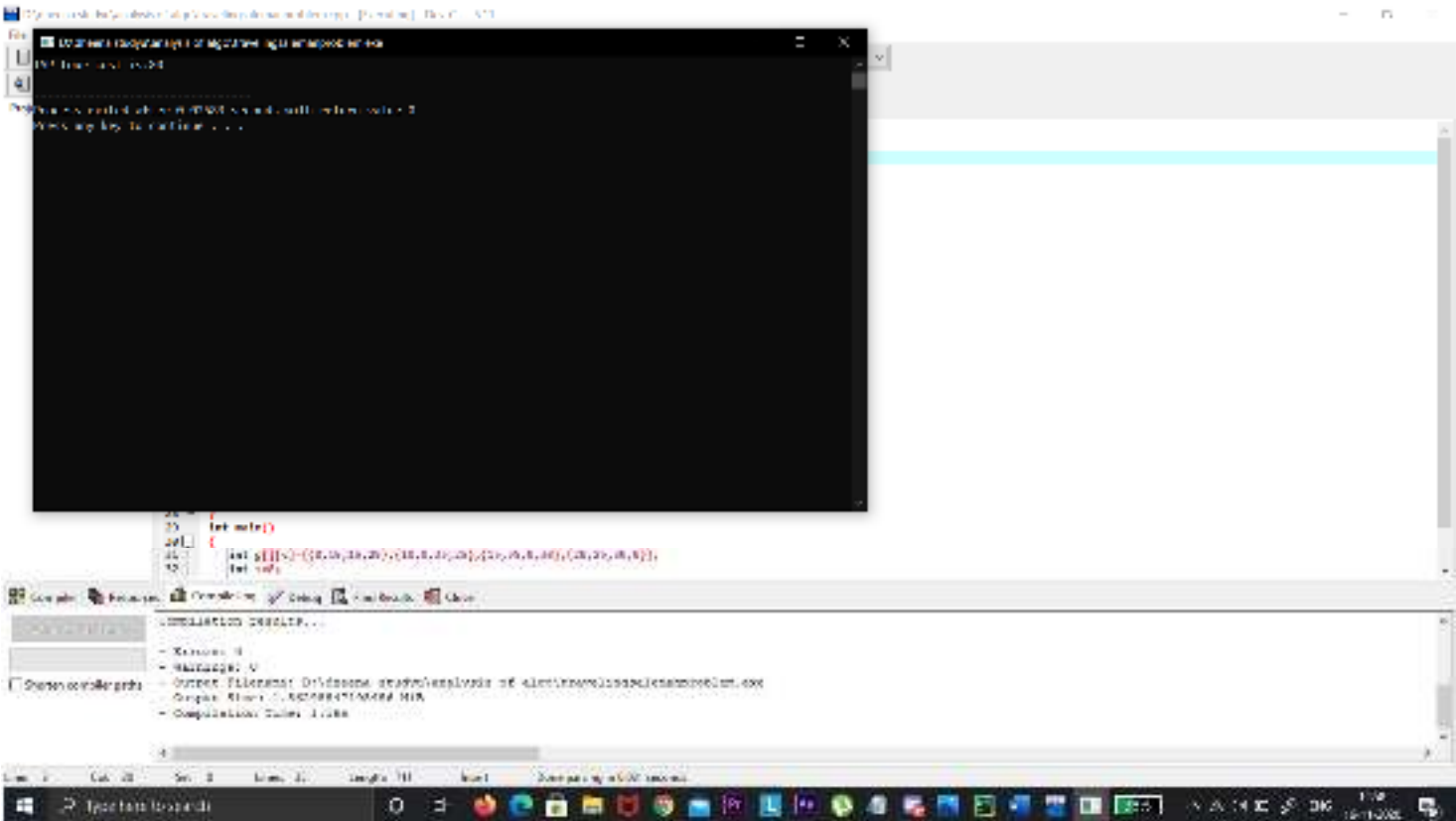
6. Pseudo code

```
int TSP(int g[][v],int s)
{
    vector<int>vertex;
    for(int i=0;i<v;i++)
    {
        if(i!=s)
            vertex.push_back(i);
    }
    int min_path=INT_MAX;
    do
    {
        int current_pathweight=0;
        int k=s;
        for(int i=0;i<vertex.size();i++)
        {
            current_pathweight+=g[k][vertex[i]];
            k=vertex[i];
        }
        current_pathweight+=g[k][s];
        min_path=min(min_path,current_pathweight);
    }while(next_permutation(vertex.begin(),vertex.end()));
    return min_path;
}
```

Program:

```
#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;
#define v 4
int TSP(int g[][v],int s)
{
    vector<int>vertex;
    for(int i=0;i<v;i++)
    {
        if(i!=s)
            vertex.push_back(i);
    }
    int min_path=INT_MAX;
    do
    {
        int current_pathweight=0;
        int k=s;
        for(int i=0;i<vertex.size();i++)
        {
            current_pathweight+=g[k][vertex[i]];
            k=vertex[i];
        }
        current_pathweight+=g[k][s];
        min_path=min(min_path,current_pathweight);
    }while(next_permutation(vertex.begin(),vertex.end()));
    return min_path;
}
int main()
{
    int g[][v]={ {0,10,15,20},{10,0,35,25},{15,35,0,30},{20,25,30,0}};
    int s=0;
    cout<<"TSP tour cost is:"<<TSP(g,s)<<endl;
    return 0;
}
```

Sample Input and Output:



Viva Questions

1. Define Branch and Bound technique for solving problems.
2. Explain the complexity of solving Knapsack problem using Branch and Bound technique.
3. Explain the complexity of solving Traveling Salesman Algorithm using Branch and Bound technique.
4. List out the advantages of Branch and Bound technique.
5. State some problems that could effectively be solved using Branch and Bound technique.

Sl.No	Module	Max. Marks	Marks Obtained
1	Efficiency of Algorithm	10	
2	Efficiency of program	10	
3	Output	10	
4	Technical Skill	10	
5	Communication Skill	10	
Total		50	
Faculty Signature			

Result:

Implementation of Knapsack problem and Traveling Salesman Problem is implemented using Branch and Bound approach and analyze the algorithms.

Ex .No 8

Tractable and Intractable Problems

Aim

- ✓ To understand the fundamentals of Tractable and Intractable problems.
- ✓ To design and implement Np and P problems and analyze the algorithms.

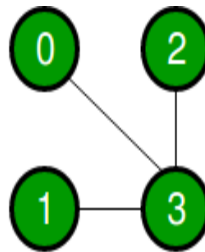
Tractable and Intractable Problems

Tractable Problem: a **problem** that is solvable by a polynomial-time algorithm. The upper bound is polynomial. **Intractable Problem:** a **problem** that cannot be solved by a polynomial-time algorithm. The lower bound is exponential.

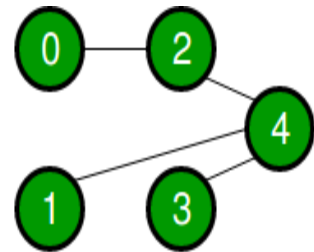
A vertex cover of an undirected graph is a subset of its vertices such that for every edge (u, v) of the graph, either 'u' or 'v' is in vertex cover. Although the name is Vertex Cover, the set covers all edges of the given graph. **Given an undirected graph, the vertex cover problem is to find minimum size vertex cover.** Following are some examples.



Minimum vertex cover is $\text{empty}\{\}$



Minimum vertex cover is $\{3\}$



Minimum vertex cover is $\{4, 2\}$ or $\{4, 0\}$

Vertex Cover Problem is a known NP Complete problem, i.e., there is no polynomial time solution for this unless $P = NP$. There are approximate polynomial time algorithms to solve the problem though.

Ex. 8.a.**Vertex Cover Problem****Aim:**

To implement and analyze Vertex cover problem.

Algorithm

1. Initialize the result as {}
2. Consider a set of all edges in given graph. Let the set be E.
3. Do following while E is not empty
 - a. Pick an arbitrary edge (u, v) from set E and add 'u' and 'v' to result
 - b. Remove all edges from E which are either incident on u or v.
4. Return result

Program:

```
import itertools
class Vertex_Cover:

    def __init__(self, graph):
        self.graph = graph

    def validity_check(self, cover):
        is_valid = True
        for i in range(len(self.graph)):
            for j in range(i+1, len(self.graph[i])):
                if self.graph[i][j] == 1 and cover[i] != '1' and cover[j] != '1':
                    return False

        return is_valid

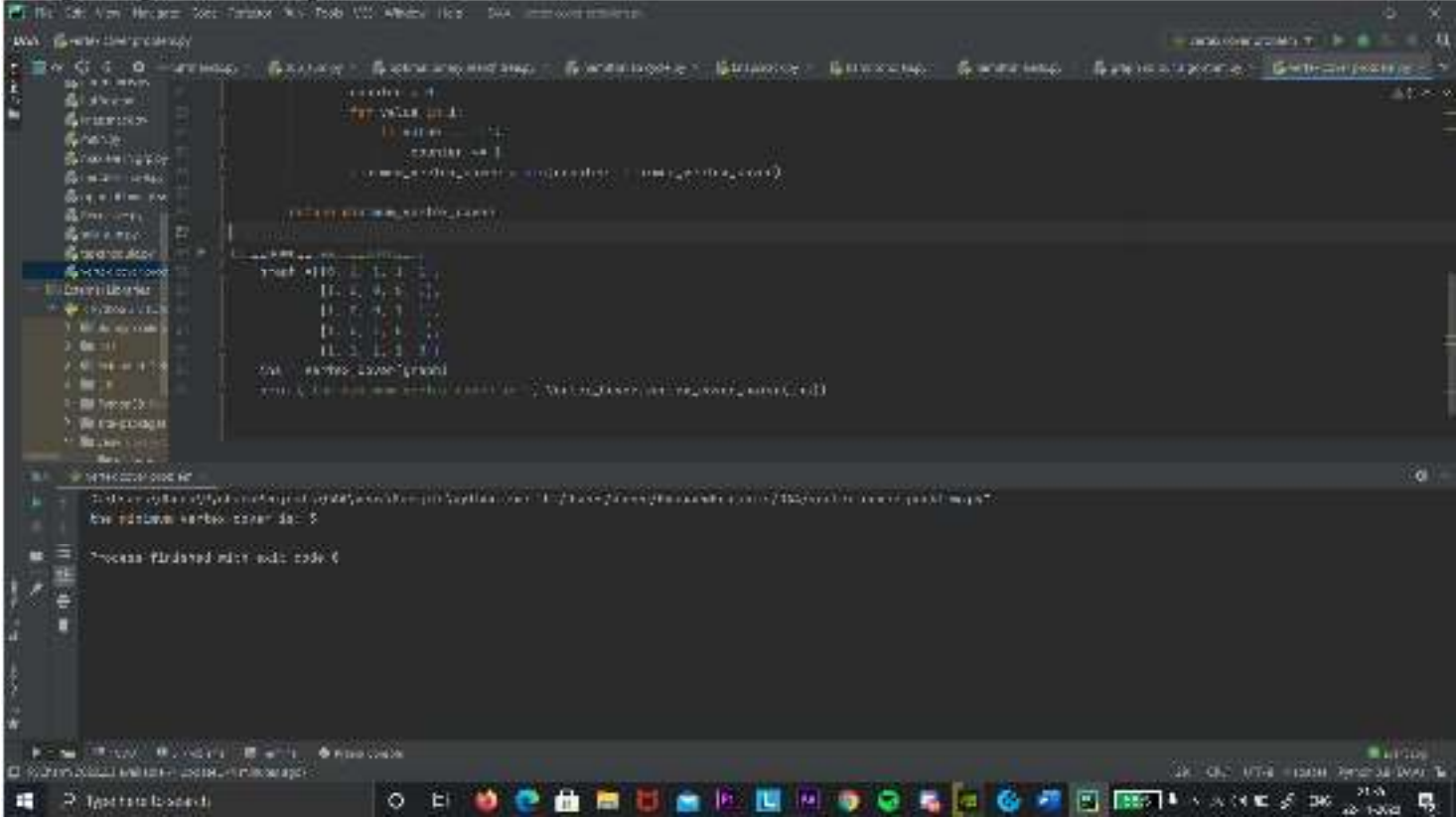
    def vertex_cover_naive(self):
        n = len(self.graph)
        minimum_vertex_cover = n
        a = list(itertools.product(*["01"] * n))
        for i in a:
            if Vertex_Cover.validity_check(self, i):
                counter = 0
                for value in i:
                    if value == '1':
                        counter += 1
                minimum_vertex_cover = min(counter, minimum_vertex_cover)

        return minimum_vertex_cover
```



```
if __name__ == '__main__':  
    graph = [[0, 1, 1, 1, 1],  
             [1, 0, 0, 0, 1],  
             [1, 0, 0, 1, 1],  
             [1, 0, 1, 0, 1],  
             [1, 1, 1, 1, 0]]  
    ins = Vertex_Cover(graph)  
    print('the minimum vertex-cover is:', Vertex_Cover.vertex_cover_naive(ins))
```

Sample Input and Output:



Viva Questions

1. Define Tractable and Intractable problems.
2. Explain NP and P class problems.
3. Explain the complexity of solving vertex cover problem.
4. Give some examples for NP class problems.
5. Give some examples for P class problems.

Sl.No	Module	Max. Marks	Marks Obtained
1	Efficiency of Algorithm	10	
2	Efficiency of program	10	
3	Output	10	
4	Technical Skill	10	
5	Communication Skill	10	
Total		50	
Faculty Signature			

Result:

Vertex cover problem is implemented using Np and P problems and analyze the algorithms.

Aim

- ✓ To learn the fundamentals of Randomized algorithms.
- ✓ To design and implement randomized quick sort and compare the efficiency over normal quick sort algorithm.

Randomized Algorithm

A **randomized algorithm** is an algorithm that employs a degree of randomness as part of its logic. The algorithm typically uses uniformly random bits as an auxiliary input to guide its behavior, in the hope of achieving good performance in the "average case" over all possible choices of random bits. Formally, the algorithm's performance will be a random variable determined by the random bits; thus either the running time, or the output (or both) are random variables.

One has to distinguish between algorithms that use the random input so that they always terminate with the correct answer, but where the expected running time is finite (Las Vegas algorithms, example of which is Quicksort), and algorithms which have a chance of producing an incorrect result (Monte Carlo algorithms, example of which is Monte Carlo algorithm for MFAS) or fail to produce a result either by signaling a failure or failing to terminate.

List of Exercises

1. Implementation of Randomized quick sort algorithm

Ex. 9.a.**Randomized Quick Sort Algorithm****Aim:**

To implement and analyze Randomized quick sort algorithm and compare its efficiency over normal quick sort.

Algorithm

```
public static void QuickSort(int arr[],int start,int end)
{
    if(end-start<0)
        return;
    else
    {
        Random random=new Random();
        i++;
        int pivot=start+random.nextInt(end-start+1);
        int pivotvalue=arr[pivot];
        System.out.print("\nStep :"+i+" Pivot Value: "+pivotvalue+"\n");
        swap(arr,pivot,end);
        int part=Partition(arr,start,end,pivotvalue);
        QuickSort(arr,start,part-1);
        QuickSort(arr,part+1,end);
    }
}

public static void swap(int arr[],int p,int q)
{
    int temp=arr[p];
    arr[p]=arr[q];
    arr[q]=temp;
}

public static int Partition(int arr[],int start,int end,int pivotvalue)
{
    int sptr=start-1;
    int eptr=end;
    while(true)
    {
        while(arr[++sptr]<pivotvalue);
        while(eptr>0&&arr[--eptr]>pivotvalue);
        if(sptr>=eptr)
            break;
        else
            swap(arr,sptr,eptr);
    }
    swap(arr,sptr,end);

    for(int j=0;j<arr.length;j++)
    {
        System.out.print(arr[j]+" ");
    }
}
```

```

    }
    System.out.println();
    return sptr;
}

```

Program:

```

import java.io.*;
import java.util.*;
public class RandomizedQuickSort
{
    public static int i;
    public static void main(String a[])
    {
        Scanner s=new Scanner(System.in);
        System.out.println("Enter No. of Elements");
        int num=s.nextInt();
        int arr[]=new int[num];
        for(int i=0;i<num;i++)
        {
            arr[i]=s.nextInt();
        }
        QuickSort(arr,0,num-1);
        System.out.println("Randomized Quick Sort");
        for(int i=0;i<arr.length;i++)
        {
            System.out.print(arr[i]+" ");
        }
        System.out.println();
    }
    public static void QuickSort(int arr[],int start,int end)
    {
        if(end-start<0)
            return;
        else
        {
            Random random=new Random();
            i++;
            int pivot=start+random.nextInt(end-start+1);
            int pivotvalue=arr[pivot];
            System.out.print("\nStep :"+i+" Pivot Value: "+pivotvalue+"\n");
            swap(arr,pivot,end);
            int part=Partition(arr,start,end,pivotvalue);
            QuickSort(arr,start,part-1);
            QuickSort(arr,part+1,end);
        }
    }
    public static void swap(int arr[],int p,int q)
    {
        int temp=arr[p];
        arr[p]=arr[q];
    }
}

```

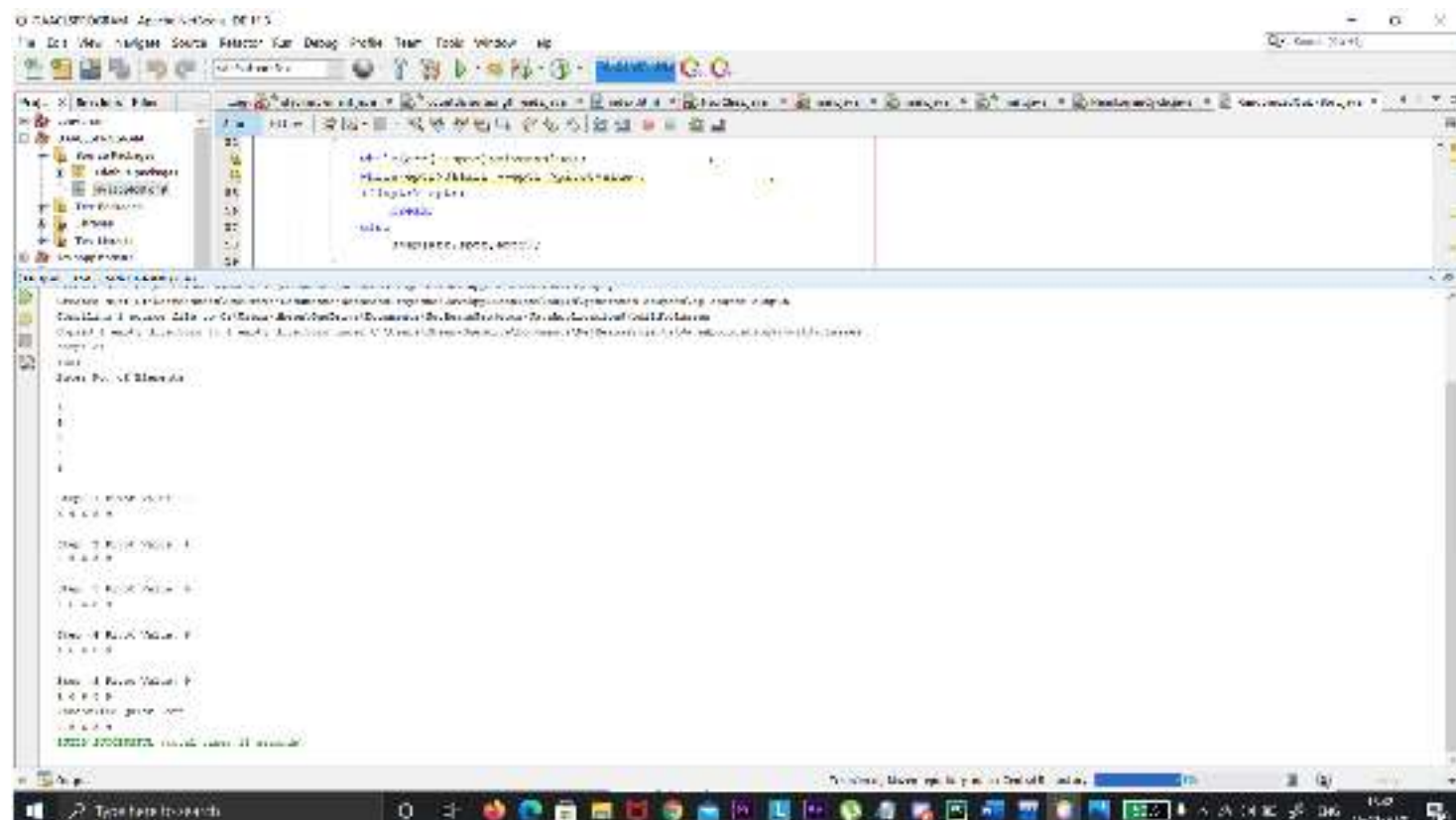
```

    arr[q]=temp;
}
public static int Partition(int arr[],int start,int end,int pivotvalue)
{
    int sptr=start-1;
    int eptr=end;
    while(true)
    {
        while(arr[++sptr]<pivotvalue);
        while(eptr>0&&arr[--eptr]>pivotvalue);
        if(sptr>=eptr)
            break;
        else
            swap(arr,sptr,eptr);
    }
    swap(arr,sptr,end);

    for(int j=0;j<arr.length;j++)
    {
        System.out.print(arr[j]+" ");
    }
    System.out.println();
    return sptr;
}

```

Sample Input and Output:



Viva Questions

1. State Randomized algorithms.
2. Explain the need for randomized algorithms.
3. State some problems that could efficiently be solved using randomization.
4. Explain the complexity of randomized quick sort.
5. Compare normal quick sort with randomized quick sort algorithm.

Sl.No	Module	Max. Marks	Marks Obtained
1	Efficiency of Algorithm	10	
2	Efficiency of program	10	
3	Output	10	
4	Technical Skill	10	
5	Communication Skill	10	
Total		50	
Faculty Signature			

Result:

Hence randomized quick sort and compare the efficiency over normal quick sort algorithm is implemented.

Aim

- ✓ To understand the fundamentals of Approximation algorithms.
- ✓ To design and implement Graph coloring algorithm using Approximation algorithm and analyze the same.

Approximation algorithms

In computer science and operations research, **approximation algorithms** are efficient algorithms that find approximate solutions to NP-hard optimization problems with **provable guarantees** on the distance of the returned solution to the optimal one. Approximation algorithms naturally arise in the field of theoretical computer science as a consequence of the widely believed $P \neq NP$ conjecture. Under this conjecture, a wide class of optimization problems cannot be solved exactly in polynomial time. The field of approximation algorithms, therefore, tries to understand how closely it is possible to approximate optimal solutions to such problems in polynomial time. In an overwhelming majority of the cases, the guarantee of such algorithms is a multiplicative one expressed as an approximation ratio or approximation factor i.e., the optimal solution is always guaranteed to be within a (predetermined) multiplicative factor of the returned solution. However, there are also many approximation algorithms that provide an additive guarantee on the quality of the returned solution.

List of Exercises

1. Implementation of Graph coloring algorithm.

Ex. 10.a.**Graph Coloring Algorithm****Aim:**

To implement and analyze Graph coloring algorithm using Approximation technique.

Algorithm

```
def promising(vertex, color):
    for adedge in adjedges.get(vertex):
        color_adedge=color_vertices.get(adedge)
        if color_adedge==color:
            return False
    return True

def getVertexColor(vertex):
    for color in colors:
        if(promising(vertex, color)):
            return color

def main():
    for vertex in vertices:
        color_vertices[vertex]=getVertexColor(vertex)
    print(color_vertices)
```

Program:

```
colors = ['Red', 'Blue', 'Green', 'Yellow', 'Black']
vertices = ['A', 'B', 'C', 'D']
adjedges = {}
adjedges['A']={'B','C'}
adjedges['B']={'A','C','D'}
adjedges['C']={'A','B','D'}
adjedges['D']={'B','C'}
color_vertices={}

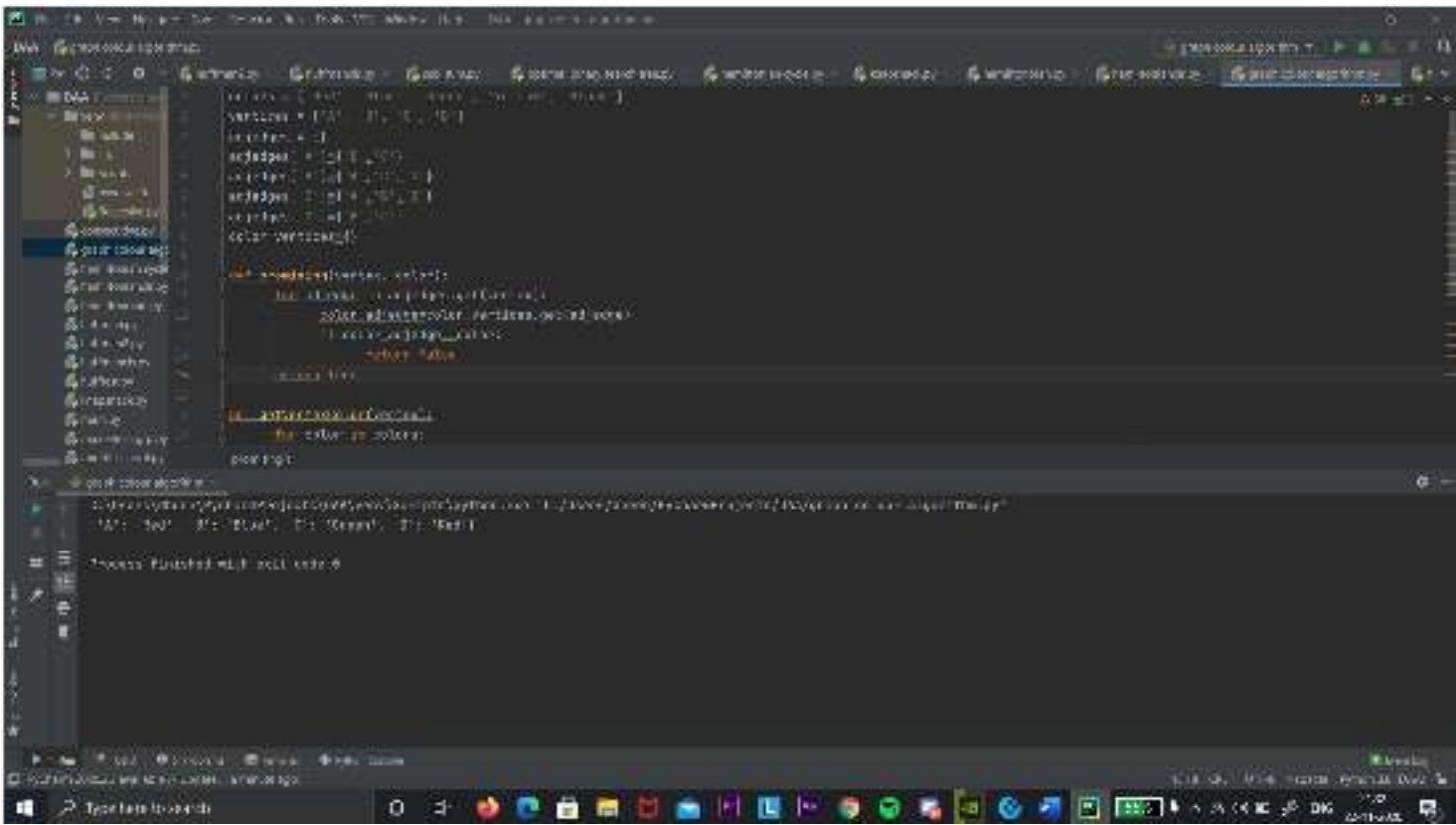
def promising(vertex, color):
    for adgedge in adjedges.get(vertex):
        color_adgedge=color_vertices.get(adgedge)
        if color_adgedge==color:
            return False
    return True

def getVertexColor(vertex):
    for color in colors:
        if(promising(vertex, color)):
            return color

def main():
    for vertex in vertices:
        color_vertices[vertex]=getVertexColor(vertex)
    print(color_vertices)

main()
```

Sample Input and Output:



Viva Questions

1. Define Approximation algorithms.
2. Explain the difference between randomization and approximation algorithms.
3. Explain the complexity of solving Graph coloring problem using approximation.
4. State some problems that could effectively be solved using Approximation.
5. Explain the need for Approximation algorithms.

Sl.No	Module	Max. Marks	Marks Obtained
1	Efficiency of Algorithm	10	
2	Efficiency of program	10	
3	Output	10	
4	Technical Skill	10	
5	Communication Skill	10	
Total		50	
Faculty Signature			

Result:

Therefore Graph coloring algorithm using Approximation algorithm and analyze the same is implemented.

Additional Experiments

Additional Experiments

Additional Experiments

References

1. T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, "Introduction to Algorithms", PHI Pvt. Ltd., 2009.
2. Sara Baase and Allen Van Gelder, "Computer Algorithms - Introduction to Design and Analysis", Pearson Education Asia, 2005.
3. A.V.Aho, J.E. Hopcroft and J.D.Ullman, "The Design and Analysis Of Computer Algorithms", Pearson Education Asia, 2010.
4. Anany Levitin, "Introduction to the Design and Analysis of Algorithm", Pearson Education Asia, 3rd Edition, 2011.
5. <https://www.geeksforgeeks.org/job-sequencing-problem-set-1-greedy-algorithm/>
6. <https://www.geeksforgeeks.org/greedy-algorithms-set-3-huffman-coding/>
7. <https://www.geeksforgeeks.org/dynamic-programming-subset-sum-problem/>
8. <https://www.geeksforgeeks.org/backtracking-set-3-n-queen-problem/>
9. <https://www.geeksforgeeks.org/quicksort-using-random-pivoting/>
10. <https://www.geeksforgeeks.org/vertex-cover-problem-set-1-introduction-approximate-algorithm-2/>