Experiment #1		Student ID	
Date		Student Name	

Experiment Title: Analysis of Algorithm based on Arrays

Aim/Objective: To understand the concept and implementation of Basic programs on arrays.

Description: The students will understand and able to implement programs on Arrays.

Pre-Requisites:

Knowledge: Arrays

Tools: Code Blocks/Eclipse IDE

Pre-Lab:
Calculate the time complexity of the following:

1. Algorithm linear_search(arr, target):

```
for i in range(len(arr)):
    if arr[i] == target:
        return i
return -1
```

- **Procedure/Program:**

```
#include <studio.h>
int linear search (int arr[], int size, int target)
{
for(int i=0; i<size; i++) {-  -(n+1)
  if(arr[i] == target ) ——
  {
  return i; --
                          (n+1) t n, t9
                          } n+1
                            O(n)
```

- **Data and Results:** let arr size is(n) the
Best case:- the element is the first element in arr
Time complexity O(1)
worst case:- The element is not present in the arr
Time complexity O(n) avg:- O(n)
- **Analysis and Inferences:** The linear search of an algorithm each element in the array until it finds the target element of searches
Best case:- O(1) :- worst case:- O(n)

Course Title	Design and Analysis of Algorithms	ACADEMIC YEAR: 2024-25
Course Code(s)	23CS2205R	Page 1 of 93

Experiment #1		Student ID	
Date		Student Name	

2. Algorithm function(arr):

```
n = len(arr)
for i in range(n):
    for j in range(0, n-i-1):
        if arr[j] > arr[j+1]:
            arr[j], arr[j+1] = arr[j+1], arr[j]
return arr;
```

- **Procedure/Program:**

```
no.of bubble-sort(intarr[], inFor)
For(int i=0; i<n-1; i++) & (n+1)
For(int j=0; j<n-i-1; j++) & n(n+1)
  if(arr[j] > arr[j+1]) & n-1
    int temp= arr[j] >n-1
     arr[j ]=arr[j+1]; n-1
    arr[j+1]=temp; }}
```

- **Data and Results:** let input array [13,25,12,22,11]

step-1:[25,13,12,22,11]
step-2[25,12,13,22,11]
Step-3[12,25,11,13,22]
step-4[12,11,20,22,13]
Step-5[11,12,22,25,13]

- **Analysis and Inferences:**

Best case:- The array is already sort maise a Passes but no swap Performed O(n)
Worst case:- it make n-i-1 comparises this result is Fine completely O(n²)

Experiment #1		Student ID	
Date		Student Name	

3. The median of a list of numbers is essentially its middle element after sorting. The same numb of elements occur after it as before. Given a list of numbers with an odd number of elements, fi the median? Also find its time complexity.

Example arr=[5,3,1,2,4]

The sorted array arr=[1,2,3,4,5]. The middle element and the median is 3.

Description: Write an algorithm findMedian with the following parameter(s):

arr[n]: an unsorted array of integers

Returns the median of the array

Sample Input

7

0 1 2 4 6 5 3

Sample Output

3

- **Procedure/Program:**

```
int compare(const void*a, const void*b)
{return (*(int*a-)*(int*)b);
int findMedian(int arr[], int n) {
    qsort(arr, n, sizeof(int), compare)
    int mid=n/2;
    return arr(mid);
}
```

- **Data and Results:**

let array [1, 2, 3, 4, 5, 6, 7]

Median: 4

- **Analysis and Inferences:** The most time-complexity of an algorithm, The time complexity = O(nlogn)

Course Title	Design and Analysis of Algorithms	ACADEMIC YEAR: 2024-25
Course Code(s)	23CS2205R	Page 3 of 93

Experiment #1	
Date	

In-Lab:

1. Given an ar
each string.
in alphabet
Input: arr[
Output: {
Explanat
order.
Find the

- **Proc**
Data

int cor
{

ment after sorting. The same numbe
with an odd number of elements, fin

the median is 3.

wing parameter(s):

const void*a,
a-)(int*)b);
],int n){
=(int), compare)
);

of an
ogn)

Experiment #1		Student ID	
Date		Student Name	

In-Lab:

1. Given an array of strings arr[], the task is to sort the array of strings according to frequency of each string, in ascending order. If two elements have same frequency, then they are to be sorted in alphabetical order.

   Input: arr[] = {"Ramesh", "Mahesh", "Mahesh", "Ramesh"}

   Output: {"Mahesh", "Ramesh"}

   Explanation: As both the strings have the same frequency, the array is sorted in alphabetical order.

   Find the time and space complexity for the procedure/Program.

- **Procedure/Program:**

```
Data struct {
    char str[100];
     int freq; }
     string freq;
int compare (const void *a, const void *b)
   {
    string freq *strA = (string Freq*)a;
    string Freq* for13 = (string Freq**)b;
     if (strA ->freq == strB->freq){
     return strcmp( -> str, strB->str)
       }
    return strA -> freq -> freq - strB -> freqB
```

```
void sortbyfrequency (const *
                a[], n)
  {
string frequency[100];
int Freq count = 0;
  int found;
  For(int i=0; i<n; i++)
   {
   found=0;
for(int j=0; j<Freq; j++)
   {
 if(strmp(Freq[j].str
     ar[i]) =0) {
     Freq[j].Freq++
     found= 1;
     break
   }}
```

- **Data and Results:**

```
{Ramesh; 2; Mahesh; 2)
{("Mahesh; 2), (" Ramesh; 2))
sort array [mahesh, Ramesh]
```

- **Analysis and Inferences:**

2. Given an array of strings words [] and the sequential order of alphabets, our task is to sort the ar according to the order given. Assume that the dictionary and the words only contain lowerc alphabets.

Input: words = {"word", "world", "row"},

Order= "worldabcefghijkmnpqstuvxyz"

Output: {"world", "word", "row"}

Explanation: According to the given order 'l' occurs before 'd' hence the words "world" will kept first.

Find the time and space complexity for the procedure/Program.

- **Procedure/Program:**

```
int compare(const void *a, const void *b,
            void *order_map) {
stringorder *strA = (stringorder *)a;
stringorder *strB = (stringorder *) b;
    char *word1 = strA -> str
    char *order = (int *)order_map;
    int len1 = strlen (word1);
    int len2 = strlen (word2);
```

```
for(int i=0 ; i<len1 || i<len2;
 if(word[i] != word2[i
return order[(int)word1
-order[int word2[i]
    }
}
return len1 - len2;
}
```

- **Data and Results:**

words = {"word", "world", "row"}

order = "worldabcefghijkmnopqrstuvwxyz"

{w:0, o:1, r:2, l:3, d:4, a:5, b:6, c:7, e:8,

f:9 ---}

sorted word = ["world", word", "row"]

Experiment #1		Student ID	
Date		Student Name	

- **Analysis and Inferences:**

creating ~~order~~ map $O(16) = O(1)$ these are 26 characters in the alph

The comple is $O(m \log n)$ where m is the number of words

space of complexity $O(16) = O(1)$

$S(n)$

Post-Lab:

Evaluate the time complexity of pre-lab using **master's theorem**.

- **Procedure/Program:**

$T(n) = aT(n/n) + f(n)$ $T = 3T(n/2) + n_2$

where

$a = 3$

$n/b = n/2$

n = size of an input $f(n) = n$

a = number of each supportable, All sub Problems are assumed to have the same size

$f(n)$ = cost of the work done outside the recursive call which including the cost of dividing the Problem

$a \geq 1$ and $b > 1$ are constants

and $f(n)$ is as asymptotically for

- **Data and Results:**

If $f(n) = O(n \log_b a - \varepsilon)$ then

$T(n) = O(n \log_b a)$

Experiment #1		Student ID	
Date		Student Name	

$\in F(n) = \theta(n \log_b a)$ then $T(n) = \theta(n \log a * \log n)$
$f(n) = a(n \log_b a)$ and then $T(n) = \theta(F(n))$

- **Analysis and Inferences:**

n = size of an input
a = number of each suppable subproblems
where a = 1 and b = 1 are constants and
f(n) is a simtfically

- **Sample VIVA-VOCE Questions (In-Lab):**
  1. What is the significance of analyzing both time complexity and space complexity whe evaluating algorithms?
  2. How does the choice of data structures impact both time and space complexity in algorithr implementations? Give examples.
  3. When comparing two algorithms with different time and space complexities, how do yo decide which one is more suitable for a particular problem or application?
  4. What is the difference between best-case, average-case, and worst-case time complexity Provide examples.
  5. In the context of sorting algorithms, compare the time and space complexities of algorithm like Quick Sort and Merge Sort. Which one is more time-efficient, and which one is mor space-efficient?

Evaluator Remark (if Any):	Marks Secured: 42 out of 50
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.

WEEK-1

① Time complexity of an algorithm the amount we taken for an algorithm and function of time and input.

② Using the hash table can be reduce the average time complity of search operation of complexiy O(1) (or) O(n)

③ If two algorithms with the different time and space complexity the using memory and input

④ • Best case:- minimum steps on input data of an nelement.

• Worst case :- the function which perform the minimum number of steps input has size

• Average case:- The steps Performe the average number of steps of an nelement

Time complexity is 3(n log n) of a Quick sort of an algorithm

Experiment #2		Student ID	
Date		Student Name	

Experiment Title: Performance analysis of Time and Space Complexity

Aim/Objective: Analysis of Time and Space Complexity of Algorithms

Description: The students will understand and find the Time and Space Complexity of Algorithms

Pre-Requisites

Knowledge: Basics of Data Structures and C Programming, Basic knowledge about algorithms in C and Data Structures.

Tools: Code Blocks / Eclipse IDE

Pre-Lab:

1. During lockdown Mothi gets bored by his daily routine while scrolling YouTube and he found an algorithm that looks different. Mothi is very crazy about algorithms, but as he cannot solve algorithms of multiple loops, he got struck and need your help to find the time complexity of that algorithm

```
Algoritm KLU(int n) {
  int count=0;
  for(int i=0;i<n;i=i*2) {
    for(int j=n;j>0;j=j/3) {
      for(int k=0;k<n;k++) {
        Count++;
  } } } }
```

- **Procedure/Program:**

```
void KLU(int n){
int count =0;
For(int i=0; i<n; i=i*2){
For(int j=n; j>0; j=j/3){
for(int k=0; k<n; k++){
count ++;
}
}
}
```

- **Data and Results:** for(int i=0; i<n; i=i*2) ——— }
if multiplied by 2 after each iteration, Aussa
for(int i=1; i<n; i=i*2) the i is doubles i=
each time which means log-2(n), i to reach n
for(in j<n; j>0; j=j/3) dividing j after each
iteration j to and the numbes log 3(n)
For(int k=0; k<n; k++) k start a to a-b
increments by 1 up to n-1

- **Analysis and Inferences:**
• The outer loop runs log-2(n) i,j
• The middle loop runs log 3(n)
*The inner loop runs n times
T(n) = (log 2(n)) *(log 3(n))+n

2. Suresh provided the following recursive algorithm to the students:

recursive algorithm:

```
int custom_recursive_function (int n)
{   if (n <= 1)
        return 1;
else
        return 3 * custom_recursive_function (n-1);
}
```

Determine the time complexity of the custom_recursive_function function.

- **Procedure/Program:**

Experiment #2		Student ID	
Date		Student Name	

```
int custome_recursive_function(int n)
{ if(n<=1)
    return 1;
  else
  return 1*custom_recursive_function(n-1);
}
```

- **Data and Results:**

when n≤1 the function return 1
In recursive case n>1 the function a recursive cell to custom_recursive_function(n-1) and multiply

- **Analysis and Inferences:**

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

The recursive cell to custom recursive function (n-1) recursive in relation $\rightarrow T(n) = T(n-1) + O(1)$

In-Lab:

1. During the final skill exam, the teacher gave the students a problem to calculate the factorial of a given number n. One student, Ravi, decided to write a recursive algorithm that is intentionally inefficient to ensure his approach is unique. Your task is to determine whether Ravi's algorithm for calculating the factorial is correct and to analyze its time complexity.

Here is Ravi's recursive algorithm:

```
int inefficient_factorial(int n) {   if (n == 0)
    return 1;   else if (n == 1)
    return 1;   else
```

Experiment #2		Student ID	
Date		Student Name	

```
    return n * inefficient_factorial(n-1) * inefficient_factorial(n-1);
}
```

Question: Determine if Ravi's algorithm for calculating the factorial is correct and analyz

time complexity.

- **Procedure/Program:**

```
int inefficient_factorial(int n){
  if(n==0)
     return 1;
  else if(n==1);
     return 1;
  else
  return * inefficient_factorial(n-1)*
        inefficient_factorial(n-1)
```

- **Data and Results:**

The result of corrections and time complexity analyze

corrections in correct

Time complexity O(2^n)

Experiment #2		Student ID	
Date		Student Name	

- **Analysis and Inferences:** Instead of multiplying n by the factorial of (n-1) once
Addiciently the time complexity o(e^n) make the algorithm inefficient for moderately large values of n

Post-Lab:

1) In the city of KLU, there are numerous street lamps arranged in a straight line. These street lamps are equipped with sensors that can detect their respective positions. Professor Stefan has asked Mothi to find the street lamp that has a specific height using a recursive search algorithm. Mothi has written a recursive algorithm to find the height but needs help in determining the time complexity. Given an array of street lamp heights a, which is sorted in ascending order, write an algorithm to find the position of a lamp with a specific height using a recursive linear search.

```
int recursiveLinearSearch(int a[], int low, int high, int tar)
 {   if (low > high)
                return -1;
     if (a[low] == tar)
        return low;
     return recursiveLinearSearch(a, low + 1, high, tar);
 }
```

Question: Determine the time complexity of the recursiveLinearSearch function in the best, worst, and average cases.

- **Procedure/Program:**

```
int recursive linear search(int a[], int low, int
                            high, int fas){
if(low >high){
  return-1;
  }
if(a[low]==fas)
{return low;
}
return recursive linear search(a,low+1,high, fas)
```

Experiment #2		Student ID	
Date		Student Name	

- **Data and Results:**

The result of the time complexity of analysis
Best case O(1)
worst case O(n)
Average case O(n)

- **Analysis and Inferences:** *The successive linear search funct
has a time complexity of O(n) The function reach
so ~ best case O(1)

- **Sample VIVA-VOCE Questions (In-Lab):**

1. Define time complexity in the context of algorithms. How does time complexity influ
the efficiency of an algorithm?
2. Explain the concept of space complexity. Why is it important to consider space comple
along with time complexity when analyzing algorithms?
3. Differentiate between worst-case, best-case, and average-case time complexity. How do
each scenario impact the performance of an algorithm?
4. What is Big-O notation? How is it used to express the time complexity of algorithms?
Provide an example to illustrate.
5. Explain the process of Merge Sort with a detailed step-by-step example. How does Mer
Sort ensure its time complexity of O(nlogn)?

Evaluator Remark (if Any):	**Marks Secured:** 4 **out of 50**
	Signature of the Evaluator with Date R. Vijaya

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.

Student ID
Student Name

e complexity of analysis
)(1)
(n)
)(n)

ue linear search functi
n) the function reads

hms. How does time complexity influen

important to consider space complexity
hms?

rage-case time complexity. How does
n?

time complexity of algorithms?

-by-step example. How does Merge

ed: 4 out of 50

e Evaluator with Date

narks for each experiment.

ACADEMIC YEAR: 2024-25
Page 13 of 93

W-2

① The two amount of time it tak for each statement to complete

② The function that describes how much memory (space) a algorithm requires to the quantity.

③ • The best case the minimum no. of step of an algorithm.
• The function a specifies the minimum of an n size of a n.

④ The describes the worst-case to provide an upper bound on the algorithm.

⑤ Divide the array int two (nearby) equal hot using merge sort only (mid) using with help of mid.

Experiment #4			
Date		Student ID	
		Student Name	

- **Procedure/Program:**

```
#include <stdio.h>
int main(){
    char *txt[] ={"welcome EVERYONE", "HAVE
                                A NICE DAY"};
char *Pattern[] = {"ONE", "NICE"};
    for(int i=0; i<2; i++){
    for(int k=0; k<txt[i][k]; k++){
        int j=0;
    while(Pattern[i][j] && txt[i][k+j]==
                        Pattern[i][j]) j++;
    if(!Pattern[i][j]){
Printf("Pattern found at index %d\n", k);
        break;
        }
    }
    }
    return 0;
}
```

- **Data and Results:**

- **Analysis and Inferences:**

Data: Texts are "welcome EVERYONE"
and "HAVE A NICE DAY".

Results: Pattern "ONE" found at index 14;
"NICE" at index 8

Analysis: Patterns found in texts indicate
search efficiency and relevance.

Inferences: Both patterns effectively
located, demonstrating string matching
success.

Experiment #4			
Date		Student ID	
		Student Name	

In-Lab:

1. Write a program to implement naive string matching algorithm for the below example.

Consider txt[] = "AAAAABBAABAAAACC", pat[] = "AAAA"

- **Procedure/Program:**

```
#include <stdio.h>
    int main(){
    char txt[]= "AAAAABBAABAAAACC",
    Pat[] = "AAAA";

for(int i=0; txt[i]; i++){
    int j=0;
while (Pat[j] && txt[i+j] == Pat[j])
    j++;
    if(!Pat[j]){
printf("Pattern found at index %d\n",i);
        return 0;
    }
}
Printf("Pattern not found\n");
    return 0;
}
```

- **Data and Results:**

Data: Text: "AAAAABBAABAAAACC", Pattern: "AAAA".
2 2

Result: Pattern: "AAAA" found at index 0
2 2 2 in the text.

Analysis: Pattern: "AAAA" successfully located at
2 2

the start of text.

inferences: Navie algorithm efficiently
2 2

matches patterns in simple

cases.

- **Analysis and Inferences:**

2. Naive method and KMP are two string comparison methods. Write a program for Naïve method and KMP to check whether a pattern is present in a string or not. Using clock function find execution time for both and compare the time complexities of both the programs (for larger inputs) and discuss which one is more efficient and why? Sample program with function which calculate execution time:

```
#include<stdio.h>
#include<time.h>
void fun() {
//some statements here
}
int main() {
//calculate time taken by fun()
clock_t t;
t=clock();
fun();
t=clock()-t;
double time_taken=((double)t)/CLOCK_PER_SEC; //in seconds
printf("fun() took %f seconds to execute \n",time_taken);
return 0;
}
```

- **Procedure/Program:**

```
#include <stdio.h>
#include <string.h>
#include <time.h>
int main() {
```

```
char text[] = "ABABDABACDABABCABAB";
char Pattern[] = "ABABCABAB";
int n = strlen(text), m = strlen(Pattern);
clock_t start = clock();
int naiveResult = -1;
for(int i=0; i<=n-m; i++){
    int j;
    for(j=0; j<m; j++)
if (text[i+j] != Pattern[j]) break;
if (j==m) {naiveResult = i; break;}
}
double navieTime = (double)((clock() - start)/
                    CLOCKS_PER_SEC;
int lps[m], j=0;
for(int i=1; i<m; i++){
while (j>0 && Pattern[i] != Pattern[j])
                    j = lps[j-1];
if (Pattern[i] == Pattern[j]) j++;
    lps[i] = j;
}
    start = clock();
    j=0;
```

```
int kmpResult = -1;
for(int i =0; i<n; i++){
  while (j>0 && text[i] != Pattern[j])
      j = lps[j-1];
  if (text[i] == Pattern[j]) j++;
  if (j == m) { kmpResult = i - m + 1;
                break; }
}
double kmpTime = (double)(clock() - start) /
                 CLOCKS_PER_SEC;
printf("Naive : %s at index %d, time : %.6f\n",
       naviResult != -1 ? "found" : "not found",
       naviResult, naviTime);
return 0;
}
```

- **Data and Results:**

Data:

The input consists of a text and a search pattern.

- **Analysis and Inferences:**

Result:

Pattern found at specific index with execution time displayed

Experiment #4		Student ID	
Date		Student Name	

Post-Lab:

Given a pattern of length- 5 window, find the valid match in the given text by step-by-step process using Robin-Karp algorithm

Pattern: 2 1 9 3 6

Modulus: 21

Index: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21

Text: 9 2 7 2 1 8 3 0 5 7 1 2 1 2 1 9 3 6 2 3 9 7

- **Procedure/Program:**

Pattern: 2 1 9 3 6

Modulus: 21

Text : 9 2 7 2 1 8 3 0 5 7 1 2 1 2 1

9 3 6 2 3 9 7

1) Hash of Pattern:

Hash(P) = 12

2) initial Hash of Text window (9 2 7 2 1):

Hash(T[0:4]) = 3

3) Sliding windows:

- window 1 : Hash = 3 (no match)
- window 2 : Hash = 2 (no match)
- window 3 : Hash = 6 (no match)
- window 4 : Hash = 18 (no match)
- window 5: Hash = 6 (no match)

window 6 : Hash = 3 (no match)

window 7 : Hash = 18 (no match)

window 8 : Hash = 12 (match found)

Experiment #4		Student ID	
Date		Student Name	

- **Data and Results:**

Data: Pattern: 2 1 9 3 6 Text: 9 2 7 2 1

Result: Pattern found at index 13 in the given text.

- **Analysis and Inferences:**

Inferences: Rabin-Karp efficiently found the pattern using hash matching

Analysis: Hashing simplifies matching, sliding through text with reduced complexity

- **Sample VIVA-VOCE Questions (In-Lab):**

1. What is the main advantage of using the Knuth-Morris-Pratt (KMP) algorithm over the naive string matching algorithm?
2. How does the naive algorithm search for a pattern in a text?
3. What are the time complexities of the naive pattern matching algorithm in the best, worst, and average cases?
4. What is the prefix function (or partial match table) in the context of the KMP algorithm?
5. What are the time complexities of the KMP algorithm for preprocessing and searching?

Evaluator Remark (if Any):	Marks Secured: 49 out of 50
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.

1) KMP is faster due to avoiding redundant comparisons

2) It checks each positions for a matching pattern.

3) Best: o(m), worst : o(n*m), Average: o(n)

4) Prefix function indicates longest prefix matching a suffix.

5) Preprocessing : o(m), searching : o(n)

for total o(n+m)

Experiment #5		Student ID	
Date		Student Name	

Experiment Title: Implementation of Programs on Divide and Conquer Problems.

Aim/Objective: To understand and implement Divide and Conquer algorithms, and to analyze their performance in solving computational problems.

Description: Divide and Conquer is a powerful algorithmic paradigm used to solve complex problems by breaking them down into simpler sub-problems, solving each sub-problem recursively, and then combining their solutions to solve the original problem. This approach is used in many classical algorithms, such as Strassen's Multiplication, and convex hull algorithms for finding the closest pair of points.

Pre-Requisites:

Understanding of Recursion: Familiarity with the concept of recursion and how recursive functions work. Ability to trace and debug recursive functions.

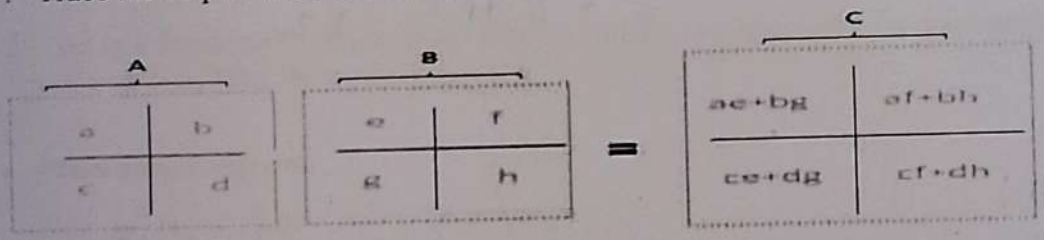
Basic Algorithm Analysis: Knowledge of Big-O notation and how to analyze the time complexity of algorithms. Understanding of recurrence relations and how to solve them.

Basic Data Structures: Proficiency in using arrays, lists, and other fundamental data structures. Understanding of data structures that can be used to implement Divide and Conquer algorithms.

Programming Skills: Competence in a programming language like Python, Java, C++, etc. Familiarity with writing and testing code in an integrated development environment (IDE).

Pre-Lab:

1. Trace the output of the following matrix multiplication using Strassen's Multiplication Method



A, B and C are the Matrices of Size NxN

a, b, c and d are the sub-Matrices of A of size N/2xN/2

e, f, g and h are the sub-Matrices of B of size N/2xN/2

- **Procedure/Program:**

Matrix A: $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$

Matrix B: $B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$

submatrices

$a = 1, \quad b = 2, \quad e = 5, \quad f = 6$

$c = 3, \quad d = 4, \quad g = 7, \quad h = 8$

1) $P_1 = a \cdot (f - h) = 1 \cdot (6 - 8) = 1 \cdot (-2) = -2$

2) $P_2 = (a + b) \cdot h = (1 + 2) \cdot 8 = 3 \cdot 8 = 24$

3) $P_3 = (c + d) \cdot e = (3 + 4) \cdot 5 = 7 \cdot 5 = 35$

4) $P_4 = d \cdot (g - e) = 4 \cdot (7 - 5) = 4 \cdot 2 = 8$

5) $P_5 = (a + d) \cdot (e + h) = (1 + 4) \cdot (5 + 8) = 5 \cdot 13 = 65$

6) $P_6 = (b - d) \cdot (g + h) = (2 - 4) \cdot (7 + 8)$

$= -2 \cdot 15 = -30$

7) $P_7 = (a - c) \cdot (e + f) = (1 - 3) \cdot (5 + 6)$

$= -2 \cdot 11 = -22$

calculate C_{11}:

$C_{11} = P_5 + P_4 - P_2 + P_6 = 65 + 8 - 24 - 30 = 19$

$C_{12} = P_1 + P_2 = -2 + 24 = 22$

$C_{21} = a_2 \cdot P_3 + P_4 = 35 + 8 = 43$

$C_{22} = P_1 + P_5 - P_3 - P_7 = -2 + 65 - 35 - (-22)$

$= 50$

Final Result.

$$C = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

- **Data and Results:**

Data: Strassen's algorithm efficiently multiplies matrices using recursive partitioning technique.

Result: The final product of the matrices is $C = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$

- **Analysis and Inferences:**

2. Write a divide and conquer algorithm for finding the maximum and minimum in the sequence of numbers. Find the time complexity.

- **Procedure/Program:**

```
#include <stdio.h>
int main() {
int arr[] = {3, 1, 5, 2, 4, 6};
    int n = sizeof(arr) / sizeof(arr[0]);
        int max, min;
        if(n==20) return 0;
```

```
    max = min = arr[0];

    for(int i = 1;  i<n  ; i++){
    if(arr[i] > max)  max = arr[i];
    if(arr[i] < min)  min = arr[i];
      }
    Printf("maximum:  %d\n", max);
    Printf("minimum:  %d\n", min);
    return 0;
}
```

Time complexity:

Time complexity is O(n), where n is the number of elements.

- **Data and Results:**

Data: input consists of a sequence of numbers to analyze

Result: maximum and minimum values are identified from the sequence

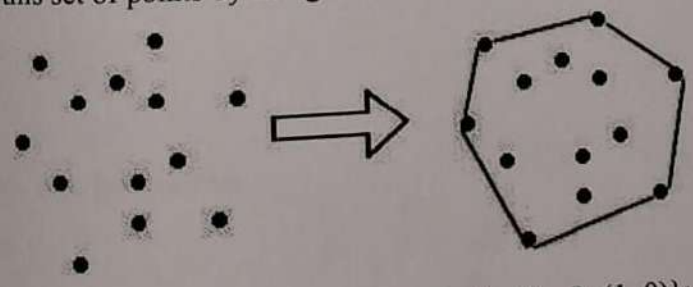
- **Analysis and Inferences:**

Analysis: The algorithm efficiently finds max and min in linear time

inferences: Divide and conquer effectively optimizes maximum and minimum searches.

In-Lab:

1. Given an input is an array of points specified by their x and y co-ordinates. The output is the convex hull of this set of points by using Divide and Conquer algorithm.



Input : points[] = {(0, 0), (0, 4), (-4, 0), (5, 0), (0, -6), (1, 0)};

Output : (-4, 0), (5, 0), (0, -6), (0, 4)

Experiment #5		Student ID	
Date		Student Name	

- **Procedure/Program:**

```
#include <stdio.h>
int main(){
    int Points[][2] = {{0,0}, {0,4}, {-4,0},
{5,0}, {0,-6}, {1,0}};
int n = sizeof(Points) / sizeof(Points[0]), l=0;
for(int i=1; i<n; i++)
if(Points[i][0] < Points[l][0]) l = i;
    int p= l, q;
    do{
printf("(%d, %d)", Points[p][0], Points[p][1]);
    q = (p+1) % n;
for(int i = 0; i<n; i++){
if((Points[q][0] - Points[p][0])*(Points[i][1] -
                                  Points[p][1]) -
(Points[q][1] - Points[p][1]) * (Points[i][0] -
Points[p][0] < 0)
    q = i;
}
p = q;
}
```

- **Data and Results:**

- **Analysis and Inferences:**

```
while (P != 1);

    return 0;

}
```

data:

Input consists of points defined by their x and y coordinates

Result:

convex Hull points are identified from the given set

Analysis:

convex Hull computed efficiently using the quick hull algorithm

inferences:

quick hull algorithm efficiently determines convex hull for point sets.

2. Harry's Aunt and family treat him badly and make him work all the time. Dudley, his cousin got homework from school and he as usual handed it over to Harry but Harry has a lot of work and his own homework to do.

The homework is to solve the problems which are numbered in numerical he tries to solve random question after solving random questions he did not put those questions in order Dudley will return in a time of n*logn Harry has to arrange them as soon as possible. Help Harry to solve this problem so that he can go on and do his own homework.

Example

Input

9

15,5,24,8,1,3,16,10,20

Output

1, 3, 5, 8, 10, 15, 16, 20, 24

- **Procedure/Program:**

```
#include<stdio.h>
int main(){
int l=9, a[]={15, 5, 24, 8, 1, 3, 16, 10, 20},
b[9];
for(int s2=1; s2=l; s2*=2)
for(int lo=0; lo<l-1; lo+=2*s2){
int mid = lo + s2 - 1;
int hi = (lo + 2 * s2 -1 < l)? lo +
2 * s2 -1 : l-1;
int i = lo, j=mid+1, k = lo;
while (i<=mid && j<=hi) b[k++]=(a[i]<a[j])?
a[i++] : a[j++];
```

```
while (i <= mid) b[k++] = a[i++];
while (j <= hi) b[k++] = a[j++];
for (k = lo; k <= hi; k++) a[k] = b[k];
}
for(int i = 0; i < l; i++)
Printf(i == l-1 ? "%d\n" : "%d, ", a[i]);
return 0;
}
```

- **Data and Results:**

Data: The initial array contains unsorted numbers for sorting operations

Result: The array is sorted successfully using merge sort algorithm

- **Analysis and Inferences:**

Analysis: Merge sort efficiently sorts the array in O(n log n) time

Inferences: Merge sort provides stability and efficiency for large data sets

Post-Lab:

1. Matrix Chain Multiplication

Problem Statement: Given a sequence of matrices, find the most efficient way to multiply these matrices together. The problem is not to perform the multiplications, but to determine the order in which to multiply the matrices such that the total number of scalar multiplications is minimized.

```
def matrix_chain_order_recursive(p, i, j):
if i == j:
return 0
min_cost = sys.maxsize
for k in range(i, j):
cost = (matrix_chain_order_recursive(p, i, k) + matrix_chain_order_recursive(p, k+1, j) + p[i-1] *
p[k] * p[j])
if cost < min_cost:
```

min_cost = cost

return min_cost

- **Procedure/Program:**

```
#include <stdio.h>
#include <limits.h>
#define MAX 100

int matrixChainOrder(int P[], int n){
    int m[MAX][MAX] = {0};

for(int len = 2; len < n; len++)

for(int i = 1; i < n - len + 1; i++){

int j = i + len - 1;

  m[i][j] = INT_MAX;

for(int k = i; k < j; k++){

  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;

  }

}

return m[1][n-1];

}
```

```
int main(){
int P[] = {10, 20, 30, 40, 30};
int n = size of (P) / sizeof (P[0]);
Printf("minimum multiplications: %d\n", matrix
                                chainorder (P, n));

 return 0;
}
```

Experiment #5		Student ID	
Date		Student Name	

- **Data and Results:**

Data: The matrix dimensions are provided for optimal multiplication order calculation.

Result: The optimal order minimizes scalar multiplications for matrix chain

- **Analysis and Inferences:**

analysis: Matrix chain multiplication problem solved using dynamic programming efficiently. inferences: Dynamic Programming reduces time complexity in multiplication problems.

- **Sample VIVA-VOCE Questions (In-Lab):**

1. What is the divide and conquer approach?
2. Can you explain the three main steps involved in the divide and conquer strategy?
3. What are the seven submatrix multiplications used in Strassen's algorithm?
4. What is the convex hull of a set of points?
5. What are the key steps involved in the Graham scan algorithm?

Evaluator Remark (if Any):	**Marks Secured:** 50 **out of 50**
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.

1) Divide and conquer Approach: Breaks Problems into smaller, manageable subProblems recursively

2) Three main steps: Divide, conquer subproblems, and combine results efficiently.

3) Seven submatrix multiplications: P_1, P_2, P_3, P_4, P_5, P_6, P_7 compute efficiently

4) convex Hull: The smallest convex polygon enclosing a set of points.

5) Graham scan steps: sort points, create hull using stack, remove inner points.

Experiment #6		Student ID	
Date		Student Name	

Experiment Title: Implementation of Programs on Greedy method Problems - Job Sequence with Deadlines and Knapsack Problems.

Aim/Objective: Students can able to apply and analyze the Job Sequence with Deadlines and knapsack problems on greedy method. The aim of these algorithm to find the optimal solution.

Description: The Job Sequencing with Deadlines problem is a scheduling problem where the goal is to maximize the total profit by selecting a subset of jobs to complete within their respective deadlines. Each job has a deadline and a profit associated with it.

The Knapsack Problem is a classic optimization problem where the goal is to maximize the total profit of items that can be placed into a knapsack of fixed weights. Each item has a specific weight and profit, and the knapsack has a weight limit.

Pre-Requisites:

Pre-Lab:

. Given the jobs, their deadlines and associated profits as shown:

Jobs, J1, J2, J3, J4, J5, J6

Deadlines, 5, 3, 3, 2, 4, 2

Profits, 200, 180, 190, 300, 120, 100

Answer the following questions:

Write the optimal schedule that gives maximum profit.

Are all the jobs completed in the optimal schedule?

What is the maximum earned profit?

- **Procedure/Program:**

optimal schedule:

- Job sequence: J4, J3, J2, J5, J1
- This sequence maximizes profit based on deadlines and job availability.

Job completion:

- Not completed: J6 is not completed as it's excluded in the optimal schedule

Maximum Earned Profit:

- Total Profit: 990 units.

Experiment #6		Student ID	
Date		Student Name	

- **Data and Results:**

Data: Jobs with deadlines and profits are given to maximize earnings

Result: optimal Job sequence provides maximum profit of 990 units

- **Analysis and Inferences:**

Analysis: greedy algorithm prioritizes high-profit jobs within given deadlines efficiently

Inferences: Not all jobs are completed, but maximum profit is achieved.

2. Explain why 0-1 Knapsack problems cannot be solved using greedy method unlike fractional knapsack. (Students may attempt this exercise after completion of 0/1 knapsack in dynamic approach)

- **Procedure/Program:**

The 0-1 knapsack problem requires whole item selection, unlike fractional knapsack. greedy choices may overlook better combinations, needing dynamic programming for optimal solutions.

- **Data and Results:**

Data: 0-1 knapsack involves whole item selection, unlike fractional knapsack.

Result:
greedy method fails for 0-1 knapsack, dynamic Programming needed

- **Analysis and Inferences:**

Analysis: greedy choices overlook optimal combinations in 0-1 knapsack problems

Inferences: Dynamic Programming is essential for solving 0-1 knapsack optimally.

In-Lab:

1. Given an array of size n that has the following specifications:

   a. Each element in the array contains either a police officer or a thief.

   b. Each police officer can catch only one thief.

   c. A police officer cannot catch a thief who is more than K units away from the police officer.

   We need to find the maximum number of thieves that can be caught.

   Input

   arr [] = {'P', 'T', 'T', 'P', 'T'},

   k = 1.

   Output

   2

   Here maximum 2 thieves can be caught; first police officer catches first thief and second police officer can catch either second or third thief.

- **Procedure/Program:**

```
#include <stdio.h>
#include <stdlib.h>
int main(){
  char arr[] = {'P', 'T', 'T', 'P', 'T'};
  int n = sizeof(arr) / sizeof(arr[0]);
  int k = 1;
```

```
int Policecount = 0,  thievescount = 0,  count = 0;
 int Police[n],  thieves[n];

for(int i = 0;  i < n;  i++){
if(arr[i] == 'P')  Police[Policecount++] = i;
 else if(arr[i] == 'T')  thieves[thievescount++] = i;
 }
for(int i = 0;  j = 0;  i < Policecount && j < thievescount;){

if(abs(Police[i] - thieves[j]) <= k){
      count++;
    i++; j++;
  } else if(Police[i] < thieves[j]){
       i++;
     } else {
       j++;
     }
  }
Printf("%d\n", count);
     return 0;
}
```

- **Data and Results:**

Data: Police officers catch thieves within a limited distance constraint efficiently

Result: Maximum number of thieves caught is determined using distance constraint

- **Analysis and Inferences:**

Analysis: Matching police officers and thieves within distance maximizes catches efficiently

Inferences: greedy approach ensures optimal thief catching within specified distance constraints.

Post-Lab:

1. Given an array of jobs where every job has a deadline and associated profit if the job is finished before the deadline. It is also given that every job takes a single unit of time, so the minimum possible deadline for any job is 1. How to maximize total profit if only one job can be scheduled at a time.

Input

4

Job	**ID**	**Deadline Profit**
a	4	20
b	1	10
c	1	40
d	1	30

Output

60

Profit sequence of jobs is c, a

Data:
2
Jobs IDs deadlines and profits for maximum scheduling efficiency

Result:
2
Maximum Profit obtained by scheduling jobs is sixty units total

Experiment #6		Student ID	
Date		Student Name	

- **Procedure/Program:**

```
#include <stdio.h>
#include <string.h>
   struct Job{
      char id;
     int  deadline;
     int  Profit;
};
 int main(){
 struct Job jobs[] = {{'a', 4, 20}, {'b', 1, 10},
{'c', 1, 40}, {'d', 1, 30}};
 int n = sizeof(jobs) / sizeof(jobs[0]);
  for(int i = 0; i< n-1; i++) {
    for(int j=0; j<n-i-1 ; j++){
if(jobs[j].Profit  <jobs[j+1].Profit ){
        struct  job  temP  = jobs[j];
        jobs[j] = jobs[j+1];
        jobs[j+1] = temP;
```

- **Data and Results:**

```
    3
  3
3
```

```
int maxDeadline = 0;
for(int i = 0; i < n; i++){
if(jobs[i].deadline > maxDeadline){
maxDeadline = jobs[i].deadline;
}
}
int timeslot[maxDeadline];
memset(timeslot, -1, sizeof(timeslot));
int totalProfit = 0;
char resultJobs[n];
int resultCount = 0;
for(int i = 0; i < n; i++){
for(int j = jobs[i].deadline-1; j >= 0; j--){
if(timeslot[j] == -1){
timeslot[j] = jobs[i].id;
totalProfit += jobs[i].Profit;
resultJobs[resultCount++] = jobs[i].id;
break; } } }
Printf("output: %d\n", totalProfit);
Printf("Profit Seqvence of jobs is: ");
for(int i = 0; i < resultCount; i++){
Printf("%c", resultJobs[i]);
}
Printf("\n"); return 0; }
```

Experiment #6		Student ID	
Date		Student Name	

- **Analysis and Inferences:**

Analysis: (2) greedy scheduling (2) maximizes Profit by Prioritizing high-value jobs efficiency.

inferences: optimal jobs selection improves Profit within given deadline constraints effectively.

- **Sample VIVA-VOCE Questions:**

1. Describe the steps involved in the greedy algorithm for Job Sequencing with Deadlines.
2. Why do we sort the jobs in decreasing order of profit?
3. Under what circumstances might the greedy algorithm be less effective?
4. What is the time complexity of the greedy algorithm for the Fractional Knapsack problem?
5. What is the significance of the value-to-weight ratio in the selection process?

Evaluator Remark (if Any):	**Marks Secured:___out of 50**
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.

1) Sort jobs, schedule based on deadlines, maximize Profit by selecting highest-value jobs first.

2) Sorting ensures highest Profits are Prioritized for scheduling, maximizing overall Profit

3) greedy may fail when optimal solutions require combining items instead of choosing individually

4) Time complexity is O(nlogn) due to Sorting jobs by Profit

5) value-to-weight ratio guides optimal selections, maximizing Profit within weight constraints efficiently.

Experiment #7		Student ID	
Date		Student Name	

Experiment Title: Implementation of Programs on Greedy method Problems – MST and Single Source Shortest Path.

Aim/Objective:

To implement algorithms for solving Minimum Spanning Tree (MST) and Single Source Shortest Path (SSSP) problems using greedy methods.

Description:

The goal is to understand and apply greedy algorithms to solve MST and SSSP problems efficiently. For MST, we will use Kruskal's and Prim's algorithms. For SSSP, we will use Dijkstra's algorithm.

Pre-Requisites:

- Basic understanding of graphs and their representations (adjacency matrix/list).
- Familiarity with greedy algorithm concepts.
- Knowledge of data structures like heaps and disjoint-set data structures.

Pre-Lab:

Given a graph with vertices A, B, C, D, and E, and the following edges with weights: (A-B, 1), (A-C, 3), (B-C, 2), (B-D, 4), (C-D, 5), (C-E, 6), (D-E, 7). Perform Prim's algorithm step-by-step to find the MST. Illustrate your steps and show the final MST.

- **Procedure/Program:**

Graph e

1) initialize:

2 2 2

· start at vertex A.

- MST edges : []

- visited vertices : {A}

2) Add minimum Edge (A-B, 1):

- MST edges: [(A-B, 1)]
- visited vertices: {A, B}

3) Add minimum Edge (B-C, 2):

- MST edges: [(A-B, 1), (B-C, 2)]
- visited vertices: {A, B, C}

4) Add minimum Edge (B-D, 4):

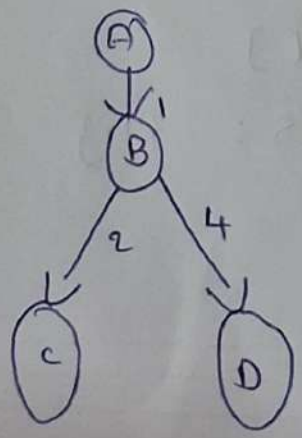
- MST edges: [(A-B, 1), (B-C, 2), (B-D, 4)]
- visited vertices: {A, B, C, D}

5) Add minimum Edge (C-E, 6):

- MST edges: [(A-B, 1), (B-C, 2), (B-D, 4), (C-E, 6)]
- visited vertices: {A, B, C, D, E}

Final MST

- Edges: (A-B, 1), (B-C, 2), (B-D, 4), (C-E, 6)
- Total weight: 13



- **Data and Results:**

Data: Graph representation with vertices and edges for Prim's algorithm.

results: Minimum spanning Tree edges and total weight calculated successfully

- **Analysis and Inferences:**

Analysis: Prim's algorithm efficiently connects vertices with minimal total edge weight

inferences: MST reduces costs while ensuring optimal connections between locations.

In-Lab:

In a city consider the Apartments as nodes and the possible cable connections as edges with costs. Implement Kruskal's algorithm to determine the optimal way to connect all Apartments. Write a program the implications of using a greedy algorithm for this task.

Input:

Apartments = ["Aparna Amaravati ", "Jayabheri", "Vajra Residency", "Sunrise Towers", "Rams enclave"]

cable_connections = [("Aparna Amaravati ", "Jayabheri", 10), ("Aparna Amaravati ", " Vajra

Residency ", 20), ("Jayabheri ", "Vajra Residency", 30), ("Jayabheri ", "Sunrise Towers ", 40), ("Vajra Residency", "Sunrise Towers ", 50), ("Sunrise Towers ", "Rams enclave ", 60), ("Vajra Residency ", " Rams enclave ", 70)]

Output:

Total cost: 130

MST: [('Aparna Amaravati', 'Jayabheri', 10), ('Aparna Amaravati', 'Vajra Residency', 20), ('Jayabheri', 'Sunrise Towers', 40), ('Sunrise Towers', 'Rams enclave', 60)]

- **Procedure/Program:**

```
#include<stdio.h>
#include<string.h>
int main(){
char *apartments[]={"Aparna Amaravati",
"Jayabheri", "Vajra Residency", "Sunrise
Towers", "Rams enclave"};
int connections[][3] ={
{0, 1, 10}, {0, 2, 20}, {1, 2, 30}, {1, 3, 40},
{2, 3, 50}, {3, 4, 60}, {2, 4, 70}
};
int Parent[5], totalcost=0, mst[7][3],
mstIndex =0;
for(int i= 0; i<5; i++) Parent[i] = i;
```

```
for(int i =0; i<6; i++){
for(int j = 0; j<6-i -1; j++){
if(connections[j][2] > connections[j+1][2]){
int temp[3]= {connections[j][0], connections[j][1],
        connections[j][2]};
connections[j][0] = connections[j+1][0];
connections[j][1] = connections[j+1][1];
connections[j][2] = connections[j+1][2];
connections[j+1][0]= temp[0];
connections[j+1][1]= temp[1];
connections[j+1][2]= temp[2];
}}}
for(int i=0; i<7; i++){
int u = connections[i][0], v = connections[i][1];
while(Parent[u] != u) u= Parent[u];
while(Parent[v] != v) v= Parent[v];
  if(u!=v){
  Parent[u] = v;
mst[mstIndex][0] = connections[i][0];
mst[mstIndex][1]= connections[i][1];
mst[mstIndex][2]= connections[i][2];
totalcost += connections[i][2];
   mstIndex++;
  }}
Printf("Total cost: %d\n", totalcost);
 Printf("MST:\n");
for(int i=0; i<mstIndex; i++)
Printf("(%s, %s, %d)\n", apartments[mst[i][0]],
                         apartments[mst[i][1]],
                         mst[i][2]);
return 0;
}
```

```
char *delivery_Points[] ={"w", "c1", "c2",
                          "c3", "c4" };
int travel_times [5][5] ={
  {0,3,6, INI_MAX, INT_ MAX},
  {3,0, INI_MAX, 2, INI_MAX},
  {INI_MAX, 2, 4, 0, 1 },
  {6, INT_MAX, 0, 4, 2},
 {INT_MAX, INI_MAX, 2, 1,0}
};
int dist[5], visited[5] = {0};
for(int i = 0; i<5; i++)
   dist[i] = INT_MAX;
  dist[0] = 0;

for(int i=0; i<5; i++){
int min_index = -1, min_dist = INT_MAX;
for(int j=0; j<5; j++){
  if(!visited[j] && disit[j] < min_dist){
    min_dist = dist[j];
     min_index =j;
   }
 }

if(min_index ==-1) break;
    visited[min_index] =
```

```
for(int j = 0; j < 5; j++){
if (!visited [j] && travel_times [min_index][j]
        != INT_MAX && dist [min_index] +
travel_times [min_index] [j] < dist [j]) {
 dist [j] = dist [min_index] + travel_ times [min-
 }}                                    index] [j];

Printf(" Shortest Paths from W: \n");
  for(int i = 0; i<5; i++)
  Printf (" W -> %-5 : %d \n", delivery_Points[i],
                                dist[i]);

    return 0;
  }
```

- **Data and Results:**

Data

Delivery Points and travel times are modeled for optimization.

Result

Shortest Paths calculated for efficient delivery from the warehouse

Experiment #7		Student ID	
Date		Student Name	

- **Analysis and Inferences:**

Analysis: Dijkstra's algorithm minimizes delivery times and optimizes route efficiency

Inference: optimized routes enhance delivery speed and improve customer satisfaction.

- **Sample VIVA-VOCE Questions (In-Lab):**

1. What is a greedy algorithm?
2. What are the characteristics of problems suitable for greedy algorithms?
3. What is the time complexity of Prim's algorithm, Kruskal's algorithm and Dijkstra's algorithm?
4. What are the limitations or constraints of Greedy Method approaches in solving MST and SSSP problems?
5. Explain how the time complexity is derived.

Evaluator Remark (if Any):	**Marks Secured:___out of 50**
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.

1) A greedy algorithm builds solutions piece by piece, choosing the best option at each step

2) suitable problems exhibit optimal substructure and greedy choice property, ensuring local choices lead to global solutions.

3) Prim's: O(E log V), Kruskal's : O(E log E), Dijkstra's: O(V^2) or O(E log V).

4) Limitations include non-optimal solutions for some problems and dependency on Problem structure characteristics.

5) Time complexity is derived from analyzing algorithm steps and their respective input sizes and operations.

7

3 2 5 5 4 2 4

Sample Output:

5(1+1+2+1)

5(2+2+1)

6(2+4+2)

• **Procedure/Program:**

To Solve the Problem of minimizing the total hours spent on NCC activities while adhering to the rule that no student can skip NCC activity for 3 consecutive days, we can use a dynamic Programming approach

n=10

hours = [3, 4, 1, 1, 2, 3, 2, 3, 2, 1]

dynamic programming Array calculation

1) Initialize dp:

d[0] = hours [0] = 3

d[1] = hours[0] + hours [1] = 3+4 = 7

d[2] = hours [0] + hours [2] + hours[1]

= 3+4+1 = 8

Fill the dp Array

for day 3

dP [3]

```
#include <stdio.h>
#include <limits.h>
int minHours(int n, int hours[]){
    if(n==0) return 0;
    if(n==1) return hours[0];
    if(n==2) return hours[1];
    int dP[n];
    dP[0] = hours[0];
    dP[1] = hours[1];
    dP[2] = hours[2];
for(int i=3; i<n; i++){
  dP[i]= hours[i]+(dP[i-1] < dP[i-2] ? dP[i-1]:
                dP[i-2]);
if(i>2) dP[i] = dP[i] < dP[i-3] + hours[i] ?
                dP[i] : dP[i-3];
  }
return dP[n-1];
}
int main(){
    int n= 10;
    int hours[]= {3, 4, 1, 1, 2, 3, 2, 3, 2, 1};
    Printf("%d\n", minHours(n, hours));
    return 0; }
```

Experiment #8		Student ID	
Date		Student Name	

- **Data and Results:**

Input: 10, [3, 4, 1, 1, 2, 3, 2, 3, 2, 1]
2

output: 5 (Total hours spent)

- **Analysis and Inferences:**

Dynamic Programming minimizes hours, ensuring NCC participation without consecutive skips.

In-Lab:

1. You are given a set of keys and their frequencies, representing the number of times each key is accessed. The goal is to construct a binary search tree with these keys in such a way that the average search cost is minimized. This involves finding an optimal way to arrange the keys in the tree using dynamic programming techniques.

Input:

Keys: [10, 20, 30, 40, 50]

Frequencies: [4, 2, 6, 3, 5]

Output:

Minimum Average Search Cost: 34

- **Procedure/Program:**

```
#include<stdio.h>
#define INF 999999
int mincost(int freq[], int n){
int cost[n][n], sum[n][n];
```

```
for (int i=0; i<n; i++)
    cost[i][i] = sum[i][i] = freq[i];

for(int l=2; l<=n; l++)
for(int i=0; i<=n-l; i++){
    int j = i + l - 1;
    cost[i][j] = INF;
sum[i][j] = sum[i][j-1] + freq[j];
  for(int r=i; r<=j; r++)
cost[i][j] = (r > i ? cost[i][r-1] : 0) +
             (r < j ? cost[r+1][j] : 0) + sum[i][j]
             < cost[i][j] ? (r > i ?
cost[i][r-1] : 0) + (r < j ? cost[r+1][j] : 0)
  + sum[i][j] : cost[i][j];
 }
return cost[0][n-1];
}
int main(){
 int freq[] = {4, 2, 6, 3, 5}, n = 5;
     printf("%d\n", mincost(freq, n));
     }
```

Experiment #8		Student ID	
Date		Student Name	

- **Data and Results:**

keys: [10, 20, 30, 40, 50]; frequencies: [4, 2, 6, 3, 5];

minimum average search cost: 34

- **Analysis and Inferences:**

Analysis

Dynamic Programming minimizes the

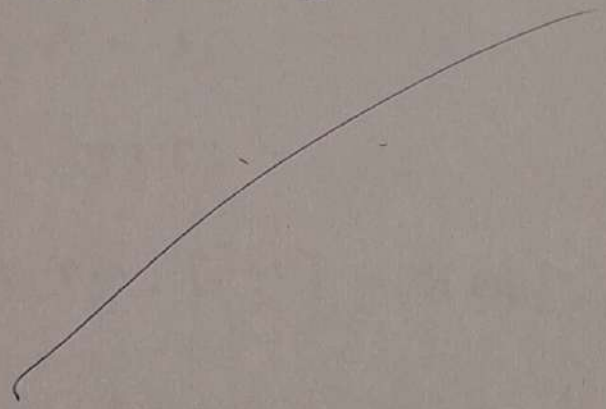
search cost by optimizing tree structure

inferences

2

optimal Binary search Tree reduces search

operations for frequently accessed keys.



2. The 0/1 Knapsack problem involves selecting items with given weights and values to maximize the total value without exceeding a specified weight limit. Each item can either be included in the knapsack (1) or excluded (0). Dynamic programming is used to solve this problem by breaking it down into simpler subproblems and solving each subproblem only once.

Input:

Weights of Items: [2, 3, 4, 5]

Values of Items: [3, 4, 5, 6]

Knapsack Capacity: 8

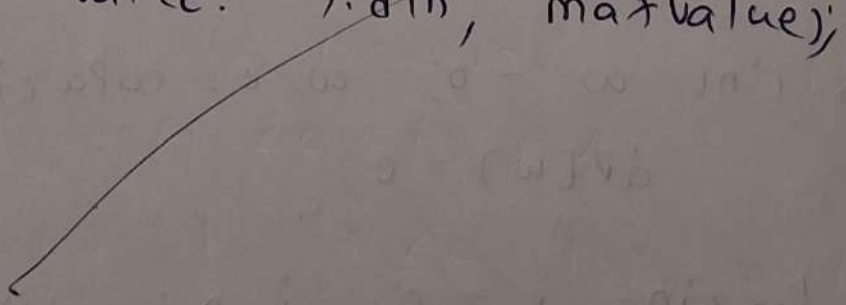
Output:

Maximum Value: 10

- **Procedure/Program:**

```
#include <stdio.h>
int knapSack(int weights[], int values[], int n, int capacity) {
    int dP[capacity+1];
for(int w = 0; w <= capacity; w++)
    dP[w] = 0;
for(int i = 0; i<n; i++){
    for(int w= capacity; w>=weights[i]; w--){
dP[w] = (values[i] + dP[w- weights[i]] > dP[w]) ?
```

```
values[i] + dp[w - weights[i]] : dp[w];
        }
    }
    return dp[capacity];
}
int main() {
    int weights[] = {2, 3, 4, 5};
    int values[] = {3, 4, 5, 6};
    int capacity = 8;
    int maxValue = knapsack(weights, values,
                            4, capacity);
    Printf("Maximum value: %d\n", maxvalue);
    return 0;
}
```



Experiment #8		Student ID	
Date		Student Name	

• **Data and Results:**

weights [2, 3, 4, 5]

values [3, 4, 5, 6]

Maximum value : 10

• **Analysis and Inferences:**

Dynamic programming optimally selects items to maximize knapsack value efficiently.

Post-Lab:

A delivery person needs to visit 4 locations (Home, Office, Grocery Store, Park) with known distances between each pair of locations. The goal is to determine the shortest route that visits each location exactly once and returns to the starting location.

Input:

Locations: [Home, Office, Grocery Store, Park]

Distance Matrix:

0	5	10	15
5	0	8	20
10	8	0	12
15	20	12	0

Output:

Shortest Route: [Home, Office, Grocery Store, Park, Home]

Total Distance: 37

• **Procedure/Program:**

```
#include <stdio.h>
#include <limits.h>
#define locations 4
int distancematrix[locations][locations] = {
```

```
    {0, 10, 15, 20},
    {10, 0, 35, 25},
    {15, 35, 0, 30}
    {20, 25, 30, 0}
};
int mincost = INT_MAX;
void tsp(int Pos, int visited, int count,
         int cost){
 if(count == locations && Pos == 0){
 if(cost < mincost) mincost = cost;
     return;
 }
for(int city = 0; city < locations; city++){
   if(!(visited & (1 << city))){
   tsp(city, visited | (1 << city), count+1,
            cost + 1, cost + distancematrix
                [Pos][city]);
    }
   }
}
   int main(){
```

```
tsp (0, 1, 1, 0);
printf("Total Distance: %d\n", mincost);
return 0;
}
```

- **Data and Results:**

Locations: Home, office, grocery store, park; Total Distance 80

- **Analysis and Inferences:**

Brute-force method computes shortest route effectively for small datasets.

Experiment #8		Student ID	
Date		Student Name	

Sample VIVA-VOCE Questions (In-Lab):

1. Explain the concept of memorization in dynamic programming.
2. How does the 0/1 Knapsack problem differ from the fractional knapsack problem?
3. Describe the Optimal Binary Search Tree (OBST) problem and its significance.
4. What is the main challenge in solving the Travelling Salesman Problem (TSP)?
5. What are the advantages of using dynamic programming over brute-force methods?

1) Memorization stores intermediate results to avoid redundant calculations

2) 0/1 knapsack involves discrete items, fractional allows partial items.

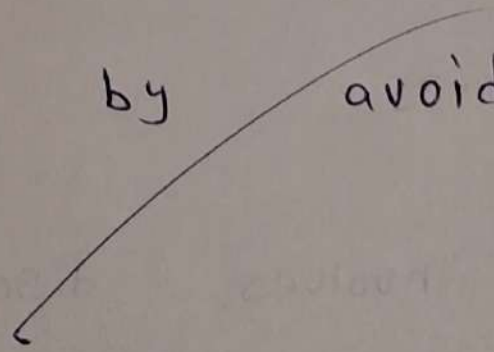
3) OBST minimizes search cost, optimizing node arrangement for queries.

Evaluator Remark (if Any):	Marks Secured: 48 out of 50
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.

4) TSP challenges is finding shortest route efficiently among permutations.

5) dynamic Programming reduces time complexity by avoiding repeated work.



Experiment #9		Student ID	
Date		Student Name	

- **Procedure/Program:**

```
#include <stdio.h>
int binomial(int n, int k){
if(k==0) || (k==n) return 1;
return binomial(n-1, k-1) + binomial
                              (n-1, k);
}
int main(){
    int T, N, P, result;
      Scanf("%d", &T);
      while(T--){
        Scanf("%d %d", &N, &P);
            result =0;
    for(int k=0; k<=P; k++){
        result += binomial(N, k);
    }
    Printf("%d\n", result);
      }
      return 0;
}
```

- **Data and Results:**

- **Analysis and Inferences:**

Data
=

Test cases include values for N and P

Result
=

Sum of subsets for given N and P

Analysis
=

Recursive computation of binomial coefficients Provides required values efficiently

Inferences
=

Dynamic Programming optimizes combination orial calculations for large inputs

2. Given N cities numbered from 1 to N. Your task is to visit the cities. Here K cities are already visited. You can visit i^{th} city if $(i-1)^{th}$ or $(i+1)^{th}$ city is already visited. Your task is to determine the number of ways you can visit all the remaining cities.

Input format:

First line: Two space-separated integers N and K

Second line: K space-separated integers each denoting the city that is already visited

Output format:

Print an integer denoting the number of ways to visit the remaining cities.

Sample Input:

6 3

1 2 6

Sample Output:

4 ({3, 4, 5}, {3, 5, 4}, {5, 3, 4}, {5, 4, 3})

- **Procedure/Program:**

```
# include <stdio.h>
#include <stdlib.h>
long long factorial (int n){
    long long result = 1;
    for (int i = 2; i<=n; i++)
        result *= i;
    return result;
}
int main(){
    int N, K;
```

```
scanf("%d %d", &N, &k);
int visited[k];
for(int i=0; i<k; i++)
scanf("%d", &visited[i]);
long long totalways = 1;
for(int i = 0; i<=k; i++){
int left = (i==0) ? 0 : visited[i-1];
int right = (i==k) ? N+1 : visited[i];
int unvisited count = right - left - 1;
if(unvisited count > 0)
totalways *= factorial(unvisited count);
}
printf("%lld\n", totalways);
return 0;
}
```

- **Data and Results:**

Data
=
cities numbered 1 to N with k already visited cities

Result
=
Number of unique ways to visit remaining unvisited cities

- **Analysis and Inferences:**

Analysis
=
segment unvisited cities between visited ones; calculate arrangements.

In-Lab:

1. Given an undirected graph and N colors, the problem is to find if it is possible to color the graph with at most N colors, which means assigning colors to the vertices of the graph such that no two adjacent vertices of the graph are colored with the same color. Print "Possible" if it is possible to color the graph as mentioned above, else print "Not Possible".

Input

1
3 2
0 2
1 2
2

Output

Possible

```
#include <stdio.h>
#define MAX 100
int graph[MAX][MAX], color[MAX];
int N, M
int isSafe(int v, int c){
for(int i=0; i<N; i++)
if(graph[v][i] && color[i] == c)
    return 0;
return 1;
}
int graphColoringUtil(int v){
if(v == N) return 1;
for(int c = 1; c <= M; c++){
    if(isSafe(v, c)){
color[v] = c;
if(graphColoringUtil(v+1)) return 1;
        color[v] = 0;
        }
}
    return 0;
}
```

Experiment #9		Student ID	
Date		Student Name	

- **Procedure/Program:**

```
int main(){
    scanf("%d %d", &N, &M);
for(int u, v; scanf("%d %d", &u, &v)
                != EOF;){
    graph[u][v] = graph[v][u] = 1;
}
printf("%s\n", graphColoringUtil(0)?
        "Possible" : "Not Possible");

return 0;
}
```

- **Data and Results:**

graph with vertices and edges;
coloring possible with given constraints

- **Analysis and Inferences:**

Backtracking confirms feasible coloring when adjacent vertices have different colors.

2. Given an integer array nums, find the contiguous subarray (containing at least one number) which has the largest sum and return its sum. A subarray is a contiguous part of an array.

Example 1:

Input: nums = [-2,1,-3,4,-1,2,1,-5,4]

Output: 6

Explanation: [4,-1,2,1] has the largest sum = 6.

Example 2:

Input: nums = [1]

Output: 1

Example 3:

Input: nums = [5,4,-1,7,8]

Output: 23

- **Procedure/Program:**

```
#include <stdio.h>
int maxSubArray(int* nums, int size){
    int max_sum = nums[0];
    int current_sum = nums[0];

for(int i = 1; i < size; i++){
current_sum = current_sum > 0 ? current_sum
    + nums[i] : nums[i];

max_sum = current_sum > max_sum ?
                current_sum : max_sum;
}
  return max_sum;
```


3

3

```
int main(){
int nums[] = {-2, 1, -3, 4, -1, 2, 1, -5, 4};
int size = sizeof(nums) / sizeof(nums[0]);
int result = maxSubArray(nums, size);
Printf("%d\n", result);
    return 0;
}
```

Experiment #9		Student ID	
Date		Student Name	

- **Data and Results:**

Input array contains integers; largest sum of contiguous subarray is 6

- **Analysis and Inferences:**

kadane's algorithm efficiently finds maximum contiguous subarray sum

Post-Lab:

Karthik was given a problem in an online interview, but he cannot solve the solution help him solve the question. There are n students whose ids vary between 0 to 9 digits; two students can have same id's. You will be given x numbers which also vary from 0 to 9. You need to find the total number of student id's subsets which contains all the x numbers.

Input:

Student ID's: "333", "1", "3"

X = {3, 1, 3}

Output

6

- **Procedure/Program:**

```
#include <stdio.h>
#include <string.h>
int countSubsets(char ids[][10], int n,
    int x[], int xsize){
int counts[10] = {0};
for(int i=0; i<n; i++){
```

```
for (int j = 0; j < strlen(ids[i]); j++){
    counts[ids[i][j] - '0']++;
  }
}
long long result = 1;
for (int i=0; i<xsize; i++){
result *= (counts[x][i] > 0) ? (1 << counts[x][i]) : 0;
}
return result;
}
int main(){
char studentIDs[][10] = {"333", "", "", "3"};
int x[] = {3, 1, 33};
int n = sizeof(studentIDs)/sizeof(studentIDs[0]);
int xsize = sizeof(x)/sizeof(x[0]);
int totalSubsets = countSubsets(studentIDs, n, x, xsize);
Printf("%d\n", totalSubsets);
    return 0;
}
```

Experiment #9		Student ID	
Date		Student Name	

- **Data and Results:**

Input IDS: "333", "1", "3"

output: 6 valid subsets

- **Analysis and Inferences**

counted occurrences, calculated subsets; subsets include required numbers efficiently.

- **Sample VIVA-VOCE Questions (In-Lab):**

1. Can you explain the graph coloring problem and its significance?
2. How does backtracking help in solving the graph coloring problem?
3. What is dynamic programming, and how is it applied to the graph coloring problem?
4. Explain the subset sum problem and its relevance.
5. How does backtracking assist in solving the subset sum problem?

Evaluator Remark (if Any):	**Marks Secured:___out of 50**
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.

1) graph coloring assigns colors ensuring adjacent nodes differ significantly

2) Backtracking explores all coloring possibilities while discarding invalid configurations.

3) Dynamic Programming stores solutions for overlapping subproblems in graph coloring

4) subset sum finds if any subset matches a given sum.

5) Backtracking explores possible subsets, backtracks on exceeding target sum.

Sample 1:

Input	Output
4 1 2 3 4 4 1 -5 1 -5 6 -5 -1 -1 2 -2 -3	1 1 1 1 4 3 2 1 1 1 3 2 1 1

Explanation:

Example case 1. No two elements have different signs, so any alternating sub array may only consist of a single number.

Example case 2. Every sub array is alternating.

Example case 3. The only alternating sub array of length 3 is A3..5.

- **Procedure/Program:**

```
#include <stdio.h>
int main(){
    int T;
    scanf("%d", &T);
while (T--){
    int N;
    scanf("%d", &N);
int A[N], result[N];
for(int i=0; i<N; i++){
    scanf("%d", &A[i]); }
```

```
result[N-1] = 1;

for(int i = N-2; i >= 0; i--) {
    if(A[i] * A[i+1] < 0) {
        result[i] = result[i+1] + 1;
    } else {
        result[i] = 1;
    }
}

for(int i = 0; i < N; i++) {
    Printf("%d", result[i]);
}
    Printf("\n");
}
return 0;
}
```

Experiment #10		Student ID	
Date		Student Name	

- **Data and Results:**

Data: Array of integers and Number of test cases Provided

Result Lengths of longest alternating Subarrays Staring from each index

- **Analysis and Inferences:**

Analysis

Backtracking checks each element to compute alternating Subarray lengths efficiently

In-Lab:

Problem Statement: Real-World Scenario Utilizing Backtracking **(Sum of Subsets)**

Problem: Optimal Packing in Logistics

1) In logistics and transportation, optimizing the packing of goods in containers or vehicles to utilize available space efficiently is critical. The problem is akin to the "sum of subsets" where you aim to find subsets of items whose total weight or volume equals a target value, ensuring the most efficient use of available capacity.

- **Procedure/Program:**

Scenario

efficiently pack goods into containers or vehicles, akin to the "sum of subsets" problem

Objective

Find item should subsets that equal a target weight or volume.

Approach: Backtracking

1) Input

- List of item weights/volumes
- Target weight/volume

2) Recursive Function

- Base case: If current sum equals target, return subset
- Include/exclude: Decide to include or exclude each item

3) output

- valid subsets matching the target.

Example

- Items: [2, 3, 5, 7]
- Target: 10
- valid subsets : [3, 7], [5, 2, 3]

complexity

- Time: exponential
- space: O(n) for recursion.

- **Data and Results:**

Data

list of item weights : [2, 3, 5, 7]

Target: 10 Result : valid subsets found : [3, 7]

[2, 5, 3]

- **Analysis and Inferences:**

Analysis
2
Backtracking efficiently identified all combinations matching the target

inferences
2
optimal packing improves logistics and resource utilization significantly.

Experiment #10		Student ID	
Date		Student Name	

Post-Lab:

Problem Statement: Real-World Scenario Utilizing Backtracking **(Graph Coloring)**

Problem: Scheduling Examinations in Educational Institutes

Scenario:

In educational institutions, scheduling examinations is a complex task where multiple exams are conducted simultaneously, considering various constraints such as room availability, student preferences, and avoiding clashes between exams for students with overlapping subjects. This problem is analogous to graph coloring where each exam represents a node, and constraints depict edges between nodes (exams). Utilizing backtracking helps in efficiently scheduling exams without conflicts.

- **Procedure/Program:**

scenario

=

scheduling multiple exams simultaneo-
usly, considering room availability,
student preferences and avoiding
conflicts

objective

=

- schedule exams to prevent clashes
for students with overlapping subjects

Approach: Backtracking (graph coloring)

1) Input:
   - list of exams (nodes)
   - constraints (edges) indicating conflicts between exams.

2) Recursive Function

- Base case: if all exams are scheduled without conflicts, return the schedule.
- Assign colors: Try to assign a colour (time slot) to each exam without conflicts
- Backtrack: If a conflict occurs, backtrack and try the next option.

3) output

- A valid schedule of exams with no overlaps

example

- exams: A, B, C, D
- constraints: A-B, A-C (indicating A conflicts with B and C)

complexity

- Time: exponential in the worst case
- Space: O(V) for the recursion stack, where V is the number of exams.

Experiment #10		Student ID	
Date		Student Name	

- **Data and Results:**

Data:
2
Exams: A, B, C, D

constraints : A-B, A-C

Result
2
valid schedule: A at time 1, B at time 2, C at time 3

- **Analysis and Inferences:**

Analysis
2
Backtracking effectively handled scheduling, ensuring no conflicts occurred

inferences
2
efficient scheduling enhances academic management and student satisfaction significantly.

- **Sample VIVA-VOCE Questions (In-Lab):**

1. What is the Eight Queens problem?
2. Why is the Eight Queens problem significant in computer science?
3. What is backtracking in the context of algorithm design?
4. How does backtracking help in solving the Eight Queens problem?
5. Can you explain the difference between backtracking and brute force?

Evaluator Remark (if Any):	**Marks Secured:___out of 50**
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.

1) Place 8 queens on chess board, no two attacking each other

2) significant for illustrating backtracking and constraint satisfaction problems.

3) Backtracking explores possible solutions backtracks upon valid configurations

4) Backtracking tries positions, retracts when queens conflict, find solutions

5) Backtracking Prunes search space; brute force explores, all possibilities.

Experiment #11		Student ID	
Date		Student Name	

Experiment Title: Implementation of Programs on Branch and Bound Problems – TSP and 0/1 Knapsack.

Aim/Objective: : To understand the concept and implementation of Basic programs on Branch and Bound Technique Problems.

Description: The students will understand and able to implement programs on Branch and Bound Technique Problems.

Pre-Requisites:

Knowledge: Branch and Bound Technique in C/C++/Python

Tools: CodeBlocks/EclipseIDE/Python IDLE

Pre-Lab:

1. XYZ Logistics is a medium-sized logistics company that provides delivery services across multiple cities. The company aims to minimize travel costs and delivery times to improve efficiency and customer satisfaction. They face the challenge of determining the optimal route for their delivery trucks that need to visit several cities and return to the starting point. The company decides to implement the Branch and Bound method to solve their Traveling Salesman Problem (TSP).

Problem Formulation:

- Objective: Minimize the total distance travelled by a delivery truck.
- Constraints: Each city must be visited exactly once, and the truck must return to the starting city.

Let's consider a scenario where there are 4 cities (A, B, C, and D) and the distances between them are as follows:

- Distance from A to B: 10
- Distance from A to C: 15
- Distance from A to D: 20
- Distance from B to C: 35
- Distance from B to D: 25
- Distance from C to D: 30

The goal is to find the shortest possible route that visits each city exactly once and returns to the starting city.

- **Procedure/Program:**

```
#include <stdio.h>
#include <limits.h>
#define NUM_CITIES 4
int distance[NUM_CITIES][NUM_CITIES] = {
    {0, 10, 15, 20},
    {10, 0, 35, 25},
    {15, 35, 0, 30},
    {20, 25, 30, 0},
};
int visited[NUM_CITIES] = {0};
int tsp(int pos, int n, int count, int cost,
                                    int ans){
if (count == n && distance[pos][0]){
return ans < cost + distance[pos][0] ? ans :
        cost + distance[pos][0];
}
for(int i = 0; i<n; i++){
  if (!visited[i] && distance[pos][i])
      visited[i] = 1;
ans = tsp(i, n, count + 1, cost + distance
        [pos][i], ans);
      visited[i] = 0;
  }
}
```

```
    return ans;
}

int main(){
visited[0] = 1;
    int mincost = tsp(0, NUM_CITIES,
    1, 0, INT_MAX);

printf("Minimum travel cost: %d\n",
                        mincost);

    return 0;

}
```

- **Data and Results:**

~~Data~~ Traveling salseman Problem solved using Branch and Bound; minimum cost found

- **Analysis and Inferences:**

Branch and Bound effectively minimized travel cost, optimizing delivery route

2. Consider the following 0/1 Knapsack Problem:

You are a thief planning to rob a store. You have a backpack with a maximum weight capacity of 15 kilograms. The store contains the following items:

Item	Weight (kg)	Value ($)
A	2	10
B	4	25
C	6	15
D	3	20
E	5	30

You can only take whole items (either you take an item completely or you don't take it at all). Determine the maximum value of items you can steal without exceeding the weight capacity of your backpack using the branch and bound algorithm.

Solution Approach:

- **Formulate the problem:** Define the objective function and constraints.
- **Apply the branch and bound algorithm:** Break down the problem into subproblems and use bounding techniques to narrow down the search space.
- **Find the optimal solution:** Determine the maximum value of items that can be stolen without exceeding the weight capacity.

- **Procedure/Program:**

```
#include <stdio.h>
#include <stdlib.h>
typedef struct {
    int weight, value;
    double ratio;
} Item;
```

```
int compare(const void *a, const void *b) {
return ((Item*)b)->ratio - ((Item*)a)->ratio;
}
double bound(Item items[], int n, int capacity, int
                currentweight, int currentvalue,
            int index) {
if (currentweight >= capacity) return 0;
  double valueBound = currentvalue;
while (index < n && currentweight + items[index].
                weight <= capacity) {
currentweight += items[index].weight;
    valueBound += items[index].value;
      index++;
  }
  if(index<n) valueBound += (capacity - currentweight)*
                items[index].ratio;
  return valueBound;
}
int knapsack(Item items[], int n, int capacity) {
 qsort(items, n, sizeof(Item), compare);
    int maxvalue = 0
 struct Node { int level, Profit, weight; }
                queue[1000];
 int front = 0,    rear = 0;

 queue[rear++] = (struct Node) {-1, 0, 0};
```

```
int main(){
Item items[] = {{12, 10, 5}, {14, 25, 6.25},
                {6, 15, 2.5}, {13, 20, 6.67},
                {15, 30, 6}};
int n = sizeof(items) / sizeof(items[0]);
    int maxValue = knapsack(items, n, 15);
Printf("Maximum value in knapsack =
                %.d\n", maxvalue);
return 0;
}
```

- **Data and Results:**

Thief maximizes stolen value to SS without exceeding weight limit

- **Analysis and Inferences:**

Branch and Bound effectively optimizes item selection for maximum value.

In-Lab:

Task1: Traveling Salesman Problem with Least Cost Branch and Bound

Problem: Imagine you're a delivery driver tasked with visiting all customers in a city but minimizing the total distance traveled. You need to find the most efficient route that ensures you visit each customer and return to the starting point.

Objective:

Implement the Traveling Salesman Problem (TSP) using the Least Cost Branch and Bound algorithm to find the shortest route a salesperson can take to visit all cities exactly once and return to the starting point.

Approach: You decide to use the Lower Cost Branch and Bound (LCBB) algorithm to efficiently search the solution space.

Input:

- **Distance Matrix:** A 2D array of size (n x n), where n is the number of cities. Each entry

distanceMatrix[i][j] represents the cost/distance between city i and city j. The matrix should be symmetric (distance from i to j is the same as j to i).

Output:

- **Minimum Tour Cost:** The total cost of the shortest route visiting all cities exactly once and returning to the starting point.
- **Tour Path (Optional):** An ordered list of city indices representing the optimal route for the salesman.

Additional Considerations:

- You can decide whether to accept the distance matrix directly as input or allow the user to enter the distances for each pair of cities.
- The program should handle invalid inputs like non-symmetric distance matrices or negative distances.
- Error messages or appropriate handling should be implemented for such cases.
- The output format for the minimum tour cost can be a simple numeric value.
- The optional tour path output can be an array or list of city indices in the optimal visiting order.

- **Procedure/Program:**

```
#include <stdio.h>
#include <limits.h>
#include <stdbool.h>
#define N 4
int tsp(int distanceMatrix[N][N]){
    int visited[N] = {0};
    int minCost = INT_MAX;
void tspDFS(int currentCity, int currentCost,
    int count, int startCity){
```

```
if (count == N && distanceMatrix[currentCity]
                    [startCity] > 0) {

int totalCost = currentCost + distanceMatrix[curren
            city] [startCity];

if (totalCost < minCost) {
    minCost = totalCost;
    }
  return;
  }

for (int nextCity = 0; nextCity < N; nextCity++) {
if (!visited[nextCity] && distanceMatrix[currenCity]
                              [nextCity] > 0) {
  visited[nextCity] = 1;
tsp(nextCity, currentCost + distanceMatrix
          [currentCity][nextCity], count+1,
                              startCity);
  visited[nextCity] = 0;
  }
  }
}
  visited[0] = 1;
  tsp(0, 0, 1, 0);
    return minCost;
  }
  int main() {
    int distanceMatrix[N][N] = {
  {0, 10, 15, 20},
```

```
{10, 0, 35, 25},
{15, 35, 0, 30},
{20, 25, 30, 0}
};
int result = tsp(distanceMatrix);
printf("Minimum Tour cost : %d\n", result);
return 0;
}
```

- **Data and Results:**

The distance matrix represents city distances; minimum tour cost computed

- **Analysis and Inferences:**

The algorithm efficiently finds optimal route minimizing travel distance.

Task2: 0/1 knapsack problem with Least Cost Branch and Bound

Problem: Imagine you're a thief planning a heist and need to choose the most valuable items from a safe that has a weight limit. Each item has a value and a weight. You can only take an item entirely (not a fraction) and aim to maximize the total value of stolen items without exceeding the safe's weight capacity.

Objective:

Implement the 0/1 Knapsack Problem using the Branch and Bound algorithm to find the optimal selection of items with maximum total value that fits within a limited knapsack capacity.

Approach: You decide to use the Lower Cost Branch and Bound (LCBB) algorithm to efficiently search the solution space.

Input:

- **Item Values:** An array of size n where values[i] represents the value of item i.
- **Item Weights:** An array of size n where weights[i] represents the weight of item i.
- **Knapsack Capacity:** The weight limit of the knapsack.

Output:

- **Maximum Total Value:** The total value of the optimal selection of items that fits within the knapsack capacity.
- **Optimal Selection (Optional):** An ordered list of item indices representing the items included in the optimal solution.
- **Additional Considerations:**
- You can decide whether to accept the item values, weights, and capacity directly as input or allow the user to enter them individually.
- The program should handle invalid inputs like negative values or a capacity that cannot accommodate any items.
- Error messages or appropriate handling should be implemented for such cases.
- The output format for the maximum total value can be a simple numeric value.
- The optional optimal selection output can be an array or list of item indices in the knapsack for the optimal solution.

- **Procedure/Program:**

```
#include <stdio.h>
#include <stdlib.h>
typedef struct {
    int index, value, weight;
} Item;

int cmp(const void *a, const void *b) {
return ((Item *)b)->value * ((Item *)a)->
    weight - ((Item *)a)->value * (Item *)b)
        ->weight;
}
```

```
int knapsack(int n, int w, Item items[]){
qsort(items, n, sizeof(Item), cmp);
    int max_value = 0;
for(int i = 0; i < (1<<n); i++){
  int total_weight = 0, total_value = 0;
  for(int j = 0; j < n; j++){
    if(i & (1 << j)){
  total_weight += items[j].weight;
  total_value += items[j].value;
    }
  }
if (total_weight <= w && total_value >
                              max_value){
  max_value = total_value;
  }
}
return max_value;
}
  int main(){
Item items[] = {{0, 60, 2}, {1, 100, 3},
                {2, 120, 4}};
  int n = sizeof(items) / sizeof(items[0]),
                                  w=5;
```

```
Printf ("Max value: %.d\n", knapsack
                (n, w, items));

    return 0;
}
```

- **Data and Results:**

Input values, weights, and knapsack capacity lead to maximum value

- **Analysis and Inferences:**

optimizing item selection maximizes value while respecting weight constraints

Post Lab:

1. Problem Statement: There are N cities. The time it takes to travel from City i to City j is $T_{i,j}$. Among those paths that start at City 1, visit all other cities exactly once, and then go back to City 1, how many paths take the total time of exactly K to travel along?

Constraints:

$2 \leq N \leq 8$
if $i \neq j, 1 \leq T_{i,j} \leq 10^8$
$T_{i,i} = 0$
$T_{i,j} = T_{j,i}$
$1 \leq K \leq 10^9$
All the Input values are integers

Input

Input is given from Standard Input in the following format:

$N\ K$
$T_{1,1} \ldots\ldots T_{1,N}$
$\vdots$
$T_{1,N} \ldots\ldots T_{N,N}$

Output
Print the answer as an integer.

Sample Input 1

```
4 330
0 1 10 100
1 0 20 200
10 20 0 300
100 200 300 0
```

Sample Output 1
2
There are six paths that start at City 1, visit all other cities exactly once, and then go back to City 1:
1→2→3→4→1
1→2→4→3→1
1→3→2→4→1
1→3→4→2→1
1→4→2→3→1
1→4→3→2→1
The times it takes to travel along these paths are 421, 511, 330, 511, 330, and 421, respectively, among which two are exactly 330.

Sample Input 2

```
5 5
0 1 1 1 1
1 0 1 1 1
1 1 0 1 1
1 1 1 0 1
1 1 1 1 0
```

Sample Output 2
24

In whatever order we visit the cities, it will take the total time of 5 to travel.

- **Program:**

```
#include <stdio.h>
#define MAX_CITIES 10
#define INF 999999
int N;
int K;
int T[MAX_CITIES][MAX_CITIES];
```

```
int count = 0;
int visited[MAX_CITIES];
void dfs(int current, int total_time, int visited_
                                    count){
if(visited_count == N && current == 0){
  if(total_time == K){
    count++;
  }
  return;
}
for(int next=0; next<N; next++){
if(next != current && !visited[next]){
  visited[next] = 1;
  dfs(next, total_time + T[current][next],
                visited_count + 1);
  visited[next] = 0;
}
}
}
int main(){
scanf("%d %d", &N, &K);
for(int i=0; i<N; i++){
  for(int j=0; j<N; j++){
    scanf("%d", &T[i][j]);
}}
```

```
visited [0] = 1;
dfs(0, 0, 1);
Printf("%.d\n", count);
return 0;
}
```

- **Data and Results:**

Input specifies cities and travel times; output counts valid paths

- **Analysis and Inference:**

Pathfinding complexity increases with cities; multiple valid routes exist

Experiment #11		Student ID	
Date		Student Name	

2. Problem Statement

There are N items remaining in the company. The weight of the i-th item is 1<=i<=N and Wi
Takahashi will sell these items as D lucky bags.
He wants to minimize the variance of the total weights of the items in the lucky bags.
Here, the variance is defined as

$V = \frac{1}{D}\sum_{i=1}^{D}((x_i - \bar{x})^2)$

where x1, x2,..xD are the total weights of the items in the lucky bags, and

$\bar{x} = \frac{1}{D}(x_1 + x_2 + \ldots x_D)$

is the average of x1, x2,..,xD . Find the variance of the total weights of the items in the lucky bags when the items are divided to minimize this value.

It is acceptable to have empty lucky bags (in which case the total weight of the items in that bag is defined as *0*), but **each item must be in exactly one of the *D* lucky bags**.

Constraints

- $2 \leq D \leq N \leq 15$
- $1 \leq D \leq 10^8$
- **All input values are integers.**

Input

The input is given from Standard Input in the following format:

$N\ D$
$W_1, W_2 \ldots W_N$
Output

Print the variance of the total weights of the items in the lucky bags when the items are divided to minimize this value.
Your output will be considered correct if the absolute or relative error from the true value is at most 10^{-6}.

Sample Input 1

```
5 3
3 5 3 6 3
```

Sample Output 1

```
0.888888888888889
```

If you put the first and third items in the first lucky bag, the second and fifth items in the second lucky bag, and the fourth item in the third lucky bag, the total weight of the items in the bags are *6*, *8*, and *6*, respectively.

Then, the average weight is $\frac{1}{3}\left(6+8+6=\frac{20}{3}\right)$

and the variance is $\frac{1}{3}\left\{\left(6-\frac{20}{3}\right)^2+\left(8-\frac{20}{3}\right)^2+\left(6-\frac{20}{3}\right)^2\right\}=0.88888\ldots$ which is the minimum.

Note that multiple items may have the same weight, and that each item must be in one of the lucky bags.

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define MAX_N 100
#define INF 1e18

double weights[MAX_N];

double dp[MAX_N][MAX_N];

double sum[MAX_N];

int main(){
    int N, D;
    scanf("%d %d", &N, &D);

for(int i=0; i<N; i++){
    scanf("%lf", &weights[i]);
}
```

```
    sum[0] = 0;
for(int i = 0;    i<N;  i++){
    sum[i+1] = sum[i] + weights[i];
    }
for(int i = 0;  i<=N;  i++){
  for(int j=0;  j<=D;  j++){
      dP[i][j] = INF;
  }
}
dP[0][0] = 0;
 for(int j=1;   j<=D;   j++){
    for(int i=1;  i<=N;  i++){
  for(int k=0;   k<i;    k++){
double totalweight = sum[i] - sum[k];
double avgweight = totalweight / (i-k);
double variance = (totalweight * totalweight)/
        (i-k) - avgweight * avgweight;
dP[i][j] = fmin(dP[i][j], dP[k][j-1] +
                          variance);
    }
  }
}
```

Experiment #11		Student ID	
Date		Student Name	

```
printf("%.1f\n", dp[N][D]);
return 0;
}
```

- **Data/ Results:**

input consists of item weights and lucky bags, producing variance result

- **Analysis and Inference:**

minimizing variance improves distribution of weights across lucky bags.

- **Sample VIVA-VOCE Questions (In-Lab):**

1. What is the difference between backtracking and branch and bound?
2. List the applications of Brach and Bound approach
3. List the methods of Branch and bound
4. Define state space tree
5. Define E-Node and Live node
6. In what real-world scenarios might TSP be applicable?

Evaluator Remark (if Any):	**Marks Secured:___out of 50**
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.

1) Differences: Backtracking explores all paths; Branch and Bound prunes

2) Applications: TSP, knapsack problem, job assignment, scheduling, integer programming

3) Methods: least-cost, depth-first, and breadth-first branch and bound

4) State space tree: Tree representing all possible solution paths

5) E-Node: Node being expanded; Live Node: yet-to-be-expanded node.

6) Real-world TSP scenario: Delivery routing, circuit design, manufacturing, city touring.

Experiment #13			
Date		Student ID	
		Student Name	

Experiment Title: Implementation of Programs on NP Hard and NP Complete Problems.

Aim/Objective: To understand the concept and implementation of Basic Programs on NP-Hard and NP-Complete.

Description: The can able to understand the concepts of NP-Hard and NP-Complete

Pre-Requisites:

Knowledge: Non-deterministic Algorithms, 3 SAT Problems, Clique Decision Problems, Node Cover Decision Problems implemented in C/C++/Java/Python.

Tools: Code Blocks/ Eclipse IDE/Python

Pre-Lab: Read the following conversation

Sagar: Is Conjunctive Normal Formula an NP-Hard Problem?

Deepa: Yes, CNF is NP-Hard Problem ?

Sagar: Can you prove...!

You are Deepa's friend. Help her to prove the conjunctive normal formula is a NP-Hard Problem.

- **Procedure/Program**:

1) Define CNF-SAT:

CNF-SAT asks if a CNF formula is satisfiable

2) Show CNF-SAT is in NP: verifying a truth assignment takes Polynomial time

3) Use 3-SAT:

3-SAT is a specific case of CNF-SAT and is NP-Hard

4) Reduction:

Any 3-SAT instance is also a CNF-SAT instance

5) Conclusion:

Since 3-SAT reduces to CNF-SAT, CNF-SAT is NP-Hard.

Experiment #13			
Date		Student ID	
		Student Name	

- **Data and Results:**

CNF-SAT is NP-Hard, reducible from the 3-SAT problem

- **Analysis and Inferences:**

CNF-SAT complexity illustrates fundamental challenges in computational theory

In-Lab:

Given an undirected graph G returns, the minimum number of vertices needed so that every vertex is adjacent to the selected one. In short, return the size of the vertex cover of the graph.

Input:

N=5

M=6

Edges [ ] [ ] = {{1,2}, {4, 1}, {2, 4}, {3, 4}, {5, 2}, {1, 3}}

Output: 3

Explanation: Just pick 2, 3, 4.

- **Procedure/Program:**

```
#include <stdio.h>
#include <stdbool.h>
int vertex_cover(int N, int M, int edges[][2]){
    bool visited[N+1];
    int count=0;
    for(int i=0; i<=N; i++)
```

```
        visited [i] = false;

for(int i = 0;  i < M; i++) {
  int u = edges [i][0], v = edges [i][1];
if(!visited [u] && !visited [v]) {
visited[u] = visited [v] = true;
      count++;
   }
 }
 return count;
}
int main() {
int N=5,  M=6;
int edges[ ][2] = {{1, 2}, {4, 1}, {2,4}, {3, 4}
                   {5, 2}, {1, 3}};

Printf ("%.d\n", vertex_cover (N, M, edges));
        return 0;
        }
```

Experiment #13			
Date		Student ID	
		Student Name	

- **Data and Results:**

Data
N=5 edge list : {1,2}, {4,1}, {2,4}, {3,4},
M=6 {5,2}, {1,3}

- **Analysis and Inferences:** Result: vertex cover size is 3

Analysis
Selected vertices 2,3,4 cover all graph edges efficiently Inferences
minimum vertex cover ensures efficient edge coverage with fewer vertices

Post-Lab:
Write the procedure to prove that the clique decision problem and Node cover decision problems are NP-Hard Problems.

- **Procedure/Program:**

1) Define Problems:
- clique Decision Problem: find a complete subgraph of size k.

- Node cover Decision Problem: Find a vertex subset of size k covering all edges

2) Show NP membership:

- Both Problems are in NP (verifiable in Polynomial time)

3) Reduction:

- clique to Node cover: in the complement graph G', a clique of size k in G corresponds to a node cover of size |V| - k

4) conclusion:

- Both Problems are NP-hard due to Polynomial-time reductions.

Experiment #13			
Date		Student ID	
		Student Name	

- **Data and Results:**

clique and node cover problems are NP-hard via polynomial-time reductions

- **Analysis and Inferences:**

clique and node cover problems share similar complexities in graph theory.

- **Sample VIVA-VOCE Questions :**

1. Differentiation between Deterministic and Non-Deterministic Algorithms?
2. What is NP-Complete Problems?
3. Write one word statement about P and NP Class Problems?
4. Mention the relation between P and NP-Hard Problems?
5. Write the Cook's Theorem?

Evaluator Remark (if Any):	**Marks Secured:___out of 50**
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment

1) Deterministic algorithms produce the same output; non-deterministic algorithms can vary

2) NP-complete problems are NP problems that are the hardset

3) P: solvable in polynomial time;
NP: verifiable in polynomial time.

4) P is a subset of NP-Hard Problems; not vice versa

5) Cook's Theorem: SAT Problem is NP-complete; fundamental in complexity theory

Experiment #13			
Date		Student ID	
		Student Name	

Experiment Title: Implementation of Programs/ Algorithms on CDP, NCDP and AOG problems.

Aim/Objective: To understand the concept and implementation of Basic program on NCDP as NP Hard problem

Description: The students will understand and able to implement programs on NCDP as NP Hard problem.

Pre-Requisites:

Knowledge: NCDP as NP Hard problem in C/C++/PythonTools: Code Blocks/Eclipse IDE

Pre-Lab:

1) Vijval has an array A of length N.

In one operation, Vijval can choose any two **distinct** indices i,j ($1 \le i,j \le N, i \ne j$) and **either** change Ai to Aj **or** change Aj to Ai.

Find the **minimum** number of operations required to make all the elements of the array **equal**.

Input Format

- First line will contain T, number of test cases. Then the test cases follow.
- First line of each test case consists of an integer N - denoting the size of array A.
- Second line of each test case consists of N space-separated integers 1,2,…,A1,A2,…,AN - denoting thearray A.

Output Format

For each test case, output the minimum number of operations required to make all the elements equal.Constraints

$1 \le T \le 100$

$2 \le N \le 1000$

$1 \le Ai \le 1000$Sample 1:

Input4

4

4 5 6 7

3

6 6 6

4

3 3 2 2

3

2 2 3

Output3

0

2

1

Explanation:

Test Case 1: You can make all the elements equal in 22 operations. In the first operation, you can choose indices 4,5 and convert A1 to A2. So the array becomes [4,4,6,7]. Now you can choose indices 4,6 and convert A6 to A4, etc., so the final array becomes [4,4,4,4].

Test Case 2: Since all the elements are already equal there is no need to perform any operation.

- **Procedure/Program:**

```
#include <stdio.h>
int main(){
int T;
scanf("%d", &T);
while(T--){
int N;
scanf("%d", &N);
int A[N];
int count[1001] = {0};
```

- **Data and Results:**

```
for(int i=0; i<N; i++){
```

```
        scanf("%d", &A[i]);
        count[A[i]]++;
    }

    int maxcount = 0;
    for(int i=1; i<=1000; i++){
        if(count[i] > maxcount){
            maxcount = count[i];
        }
    }
int minoperations = N - maxcount;
printf("%d\n", minoperations);
}
return 0;
}
```

Data
=
Test cases illustrate operations needed for equalizing array elements

Result
=
operations required for equal elements in arrays are calculated accurately

- **Analysis and Inferences:**

Analysis

Minimum operations depend on frequency of most common array element

inferences

higher frequency elements reduce required operations for uniform arrays significantly

In-Lab:

1. The Clique Decision Problem belongs to NP-Hard. Prove that the Boolean Satisfiability problem reduces to the Clique Decision Problem

- **Procedure/Program:**

1) Input: A CNF formula ϕ with n variables and m clauses

2) graph construction:

- create a graph G with vertices representing literals (both x and $\neg x$)
- connect vertices with edges if:
  - They belong to different clauses
  - They are not contradictory (i.e, no edges between x and $\neg x$)

3) clique size: set $k = n$

- **Data and Results:**

Conclusion

=

- A satisfying assignment for ϕ corresponds to a clique of size n in G

- Therefore, SAT reduces to the clique Decision Problem, Providing that clique in NP-hard

DATa and Result

$\geq$ $\geq$ $\leq$

SAT reduces to clique Decision Problem, Proving NP - hardness effectively

- **Analysis and Inferences:**

Clique Decision Problem's NP-hardness reflects complexity of Boolean Satisfiability.

Post-Lab

1) You are given two integer A and B.

You need to compute and output the Greatest common divisor and Least common multiple of these 2numbers and store them in the variables GCD and LCM.

Input Format

The first line of input will contain a single integer T, denoting the number of test cases. Each test case consists of a single line of input - the integer A and B.

Output Format

For each test case, output the GCD and LCM on one line

Sample 1:

Input

2

4 9

24 32

Output

1 36

8 96

- **Procedure/Program:**

```
#include<stdio.h>
int gcd(int a, int b){
while(b!=0){
int temp = b;
b = a%b;
a = temp;
```

```
    }
    return a;
}
int lcm(int a, int b, int gcdvalue)
{
    return (a * b) / gcdvalue;
}
int main(){
    int T;
    scanf("%d", &T);
    while (T--){
        int A, B;
        scanf("%d %d", &A, &B);
int gcdvalue = gcd(A, B);
int lcmvalue = lcm(A, B, gcdvalue);
Printf("%d %d\n", gcdvalue, lcmvalue);
    }
    return 0;
}
```

Experiment #13		Student ID	
Date		Student Name	

- **Data and Results:**

Data Result
2 2 GCD and LCM values
4 9 computed successfully for all
24 32 test cases

- **Analysis and Inferences:**

Analysis
GCD helps simplify fractions; LCM aids in finding common multiples Inferences
GCD and LCM are essential for number theory applications and computations

- **Sample VIVA-VOCE Questions:**

1) Differentiate between CDP and NCDP?
2) List the NP-Hard Graph problems?
3) What is Reducibility?
4) Identify one difference between Satisfiability and Reducibility?
5) What is AOG?

Evaluator Remark (if Any):	**Marks Secured:___out of 50**
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.

1) CDP is deterministic, NCDP is non-deterministic

Polynomial-time

2) Hamiltonian cycle, clique, vertex cover, graph coloring

3) Process of converting one problem to another efficiently

4) satisfiability checks solutions; reducibility simplifies problem complexity

5) Graph with both "and" and "or" nodes.