Vidyavardhini's

College of Engineering & Technology

Vasai Road (W)

Department of Artificial Intelligence & Data Science Engineering

Laboratory Manual

Semester	IV	**Class**	**S.E**
Course Code	CSL401		
Course Name	Analysis of Algorithms Lab		

Vidyavardhini's College of Engineering & Technology

Vision

To be a premier institution of technical education; always aiming at becoming a valuable resource for industry and society.

Mission

- To provide technologically inspiring environment for learning.
- To promote creativity, innovation, and professional activities.
- To inculcate ethical and moral values.
- To cater personal, professional, and societal needs through quality education.

Program Outcomes (POs):

Engineering Graduates will be able to:

- **PO1. Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
- **PO2. Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
- **PO3. Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
- **PO4. Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
- **PO5. Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
- **PO6. The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
- **PO7. Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
- **PO8. Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
- **PO9. Individual and teamwork:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
- **PO10. Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
- **PO11. Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
- **PO12. Life-long learning:** Recognize the need for and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Course Objective

1	To introduce the methods of designing and analyzing algorithms
2	Design and implement efficient algorithms for a specified application
3	Strengthen the ability to identify and apply the suitable algorithm for the given real-world problem.
4	Analyze worst-case running time of algorithms and understand fundamental algorithmic problems.

Course Outcomes

At the end of the course student will be able to:		Action verbs	Bloom's Level
CSL401.1	Analyze time complexity of sorting algorithms	Analyze	Apply (Level 3)
CSL401.2	Analyze the complexity of problems solved using divide and conquer approaches	Analyze	Apply (Level 3)
CSL401.3	Implement greedy algorithms for solving Dijkstras, Minimum spanning tree & fractional knapsack.	Implement	Apply (Level 3)
CSL401.4	Implement dynamic programming algorithm for All pair shortest path and 0/1 knapsack	Apply	Apply (Level 3)
CSL401.5	Implement backtracking and branch and bound for 15 puzzle, N queen and sum of subset problem	Apply	Apply (Level 3)
CSL401.6	Analyze the performance of string-matching techniques	Analyze	Apply (Level 3)

Mapping of Experiments with Course Outcomes

Sr. No	Title	CSL 401.1	CSL401.2	CSL 401.3	CSL 401.4	CSL 401.5	CSL 401.6
1.	To implement Insertion Sort and Comparative analysis for large values of 'n'.	3	-	-	-	-	-
2.	To implement Selection Sort and Comparative analysis for large values of 'n'	3	-	-	-	-	-
3.	To implement Quick Sort and Comparative analysis for large values of 'n' using DAC technique.	-	3	-	-	-	-
4.	To implement Binary Search for 'n' number and perform analysis using DAC technique.	-	3	-	-	-	-
5.	To implement Fractional Knap Sack using Greedy Method.	-	-	3	-	-	-
6.	To implement Prim's MST Algorithm using Greedy Method.	-	-	3	-	-	-
7.	To implement Kruskal's MST Algorithm using Greedy Method.	-	-	3	-	-	-
8.	To implement Single Source Shortest Path Algorithm using Dynamic (Bellman Ford) Method.	-	-	-	3	-	-
9.	To implement Travelling Salesperson Problem using Dynamic Approach.	-	-	-	3	-	-
10.	To implement Sub of Subset problem using Backtracking method.	-	-	-	-	3	-
11.	To implement 15 puzzle problem using Branch and Bound Method.	-	-	-	-	3	-
12.	Implement the Naïve string-matching algorithm and analyse its complexity.	-	-	-	-	-	3

Enter correlation level 1, 2 or 3 as defined below
1: Slight (Low)　　2: Moderate (Medium)　　3: Substantial (High)

If there is no correlation put "— ".

Index

12.	Implement the Naïve string-matching algorithm and analyse its complexity.				

D.O.P: Date of performance

D.O.C : Date of correction

Experiment No.1
Insertion Sort
Date of Performance:
Date of Submission:

Title: Insertion Sort

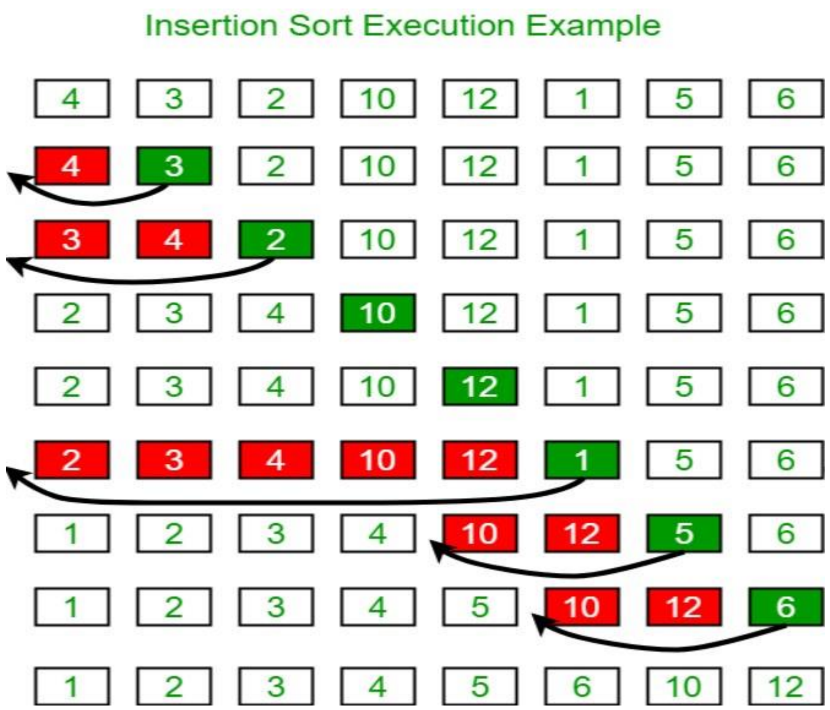
Aim: To implement Selection Comparative analysis for large values of ‘n’

Objective: To introduce the methods of designing and analysing algorithms

Theory:

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

Example:



Algorithm and Complexity:

INSERTION-SORT(A)	*cost*	*times*
for $j = 2$ **to** $A.length$	c_1	n
$key = A[j]$	c_2	$n-1$
// Insert $A[j]$ into the sorted sequence $A[1 \,..\, j-1]$.	0	$n-1$
$i = j - 1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] = key$	c_8	$n-1$

Code :

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

// Function to perform insertion sort
void insertionSort(int arr[], int n) {
   int i, key, j;
   for (i = 1; i < n; i++) {
      key = arr[i];
      j = i - 1;
      // Move elements of arr[0..i-1], that are greater than key, to one position ahead
of their current position
      while (j >= 0 && arr[j] > key) {
         arr[j + 1] = arr[j];
         j = j - 1;
      }
      arr[j + 1] = key;
   }
}

// Function to generate random array of given size
void generateRandomArray(int arr[], int n) {
   srand(time(NULL));
```

```
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 1000; // Generating random numbers between 0 and 999
    }
}

int main() {
    int n, i;

    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int arr[n];

    // Generating random array
    generateRandomArray(arr, n);

    // Sorting the array using Insertion Sort
    clock_t start = clock();
    insertionSort(arr, n);
    clock_t end = clock();
    double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;

    // Displaying sorted array
    printf("Sorted array: ");
    for (i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    // Displaying time taken for sorting
    printf("Time taken for sorting: %f seconds\n", time_taken);

    return 0;
}
```

Output :

```
PS E:\Testing_Lang> cd "e:\Testing_Lang\" ; if ($?) { gcc test.c -o test } ; if ($?) { .\test }
Enter the number of elements: 5
Sorted array: 255 539 591 680 843
Time taken for sorting: 0.000000 seconds
PS E:\Testing_Lang> 
```

Conclusion:

This program prompts the user to enter the number of elements 'n'. It then generates a random array of 'n' elements, sorts the array using the Insertion Sort algorithm, and displays the sorted array along with the time taken for sorting.

To perform a comparative analysis for large values of 'n', you can modify the program to execute multiple times with increasing values of 'n' and measure the time taken for each execution. Additionally, you can compare the performance of Insertion Sort with other sorting algorithms to analyze their efficiency for different input sizes.

Experiment No.2
Selection Sort
Date of Performance:
Date of Submission:

Experiment No. 2

Title: Selection Sort

Aim: To implement Selection Comparative analysis for large values of ‘n’

Objective: To introduce the methods of designing and analyzing algorithms

Theory:

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

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

Example:

arr[] = 64 25 12 22 11

// Find the minimum element in arr[0...4] // and place it at beginning

11 25 12 22 64

// Find the minimum element in arr[1...4] // and place it at beginning of arr[1...4]

11 12 25 22 64

// Find the minimum element in arr[2...4] // and place it at beginning of arr[2...4]

11 12 **22** 25 64

// Find the minimum element in arr[3...4] // and place it at beginning of arr[3...4]

11 12 22 **25** 64

Algorithm and Complexity:

Alg.: SELECTION-SORT(A)		
	cost	Times
n ← length[A]	c_1	1
for j ← 1 to n - 1	c_2	n-1
do smallest ← j	c_3	n-1
for i ← j + 1 to n	c_4	$\sum_{j=1}^{n-1}(n-j+1)$
≈ n2/2 comparisons, do if A[i]<A[smallest]	c_5	$\sum_{j=1}^{n-1}(n-j)$
then smallest ← i	c_6	$\sum_{j=1}^{n-1}(n-j)$
≈ n exchanges, exchange A[j] ↔ A[smallest]	c_7	n-1

Code :

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

// Function to perform selection sort
void selectionSort(int arr[], int n) {
    int i, j, min_idx;
    for (i = 0; i < n-1; i++) {
        min_idx = i;
        for (j = i+1; j < n; j++) {
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
            }
        }
        // Swap the found minimum element with the first element
        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}

// Function to generate random array of given size
```

```c
void generateRandomArray(int arr[], int n) {
    srand(time(NULL));
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 1000; // Generating random numbers between 0 and 999
    }
}

int main() {
    int n, i;

    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int arr[n];

    // Generating random array
    generateRandomArray(arr, n);

    // Sorting the array using Selection Sort
    clock_t start = clock();
    selectionSort(arr, n);
    clock_t end = clock();
    double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;

    // Displaying sorted array
    printf("Sorted array: ");
    for (i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    // Displaying time taken for sorting
    printf("Time taken for sorting: %f seconds\n", time_taken);

    return 0;
}
```

Output :

```
PS E:\Testing_Lang> cd "e:\Testing_Lang\" ; if ($?) { gcc test.c -o test } ; if ($?) { .\test }
Enter the number of elements: 10
Sorted array: 6 39 99 297 458 486 601 761 828 951
Time taken for sorting: 0.000000 seconds
PS E:\Testing_Lang> 
```

Conclusion:

This program prompts the user to enter the number of elements 'n'. It then generates a random array of 'n' elements, sorts the array using the Selection Sort algorithm, and displays the sorted array along with the time taken for sorting.

To perform a comparative analysis for large values of 'n', you can modify the program to execute multiple times with increasing values of 'n' and measure the time taken for each execution. Additionally, you can compare the performance of Selection Sort with other sorting algorithms to analyze their efficiency for different input sizes.

Experiment No. 3
Quick Sort
Date of Performance:
Date of Submission:

Experiment No. 3

Title: Quick Sort

Aim: To implement Quick Sort and Comparative analysis for large values of ‘n’.

Objective: To introduce the methods of designing and analyzing algorithms.

Theory:

The merge sort algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows:

1. Divide: Divide the n-element sequence to be sorted into two subsequences of n=2 elements each.
2. Conquer: Sort the two subsequences recursively using merge sort.
3. Combine: Merge the two sorted subsequence to produce the sorted answer.

Partition-exchange sort or quicksort algorithm was developed in 1960 by Tony Hoare. He developed the algorithm to sort the words to be translated, to make them more easily matched to an already-sorted Russian-to-English dictionary that was stored on magnetic tape.

Quick sort algorithm on average, makes O(n log n) comparisons to sort n items. In the worst case, it makes O(n2) comparisons, though this behavior is rare. Quicksort is often faster in practice than other O(n log n) algorithms. Additionally, quicksort's sequential and localized memory references work well with a cache. Quicksort is a comparison sort and, in efficient implementations, is not a stable sort. Quicksort can be implemented with an in-place partitioning algorithm, so the entire sort can be done with only O(log n) additional space used by the stack during the recursion.

Quicksort is a divide and conquer algorithm. Quicksort first divides a large list into two smaller sub-lists: the low elements and the high elements. Quicksort can then recursively sort the sublists.

1. Elements less than pivot element.

2. Pivot element.

3. Elements greater than pivot element.

Where pivot as middle element of large list. Let's understand through example:

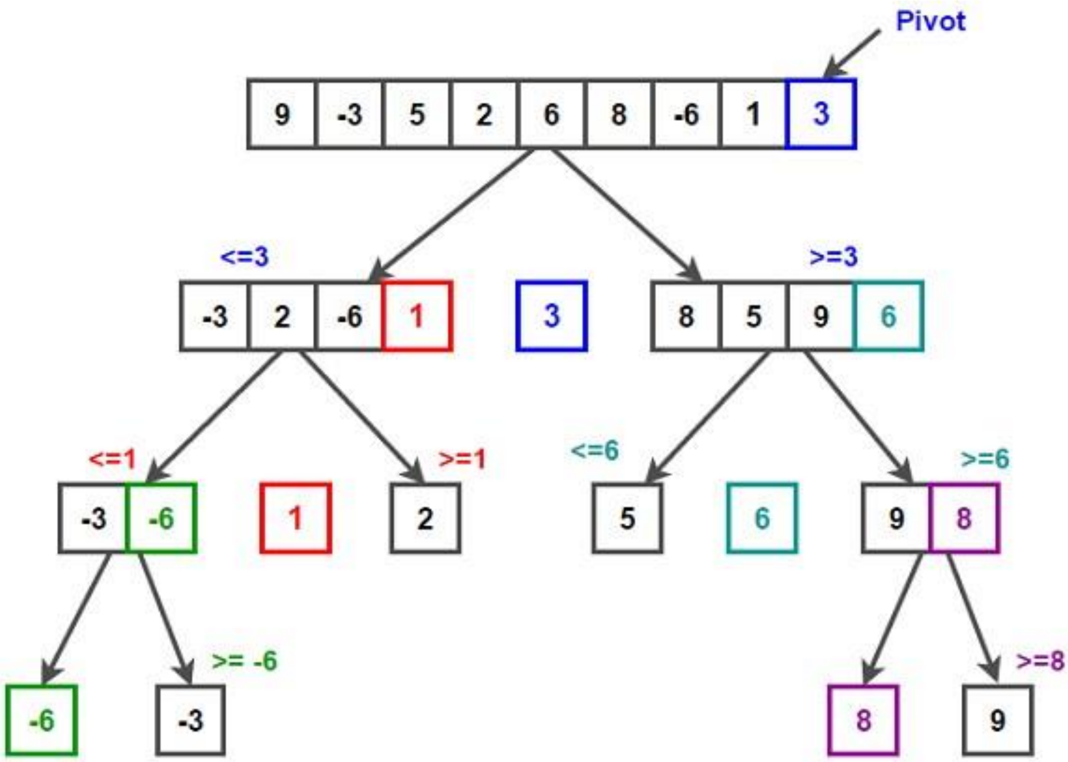
List : 3 7 8 5 2 1 9 5 4

In above list assume 4 is pivot element so rewrite list as:

3 1 2 4 5 8 9 5 7

Here, I want to say that we set the pivot element (4) which has in left side elements are less than and right hand side elements are greater than. Now you think, how's arrange the less than and greater than elements? Be patient, you get answer soon.

Example:

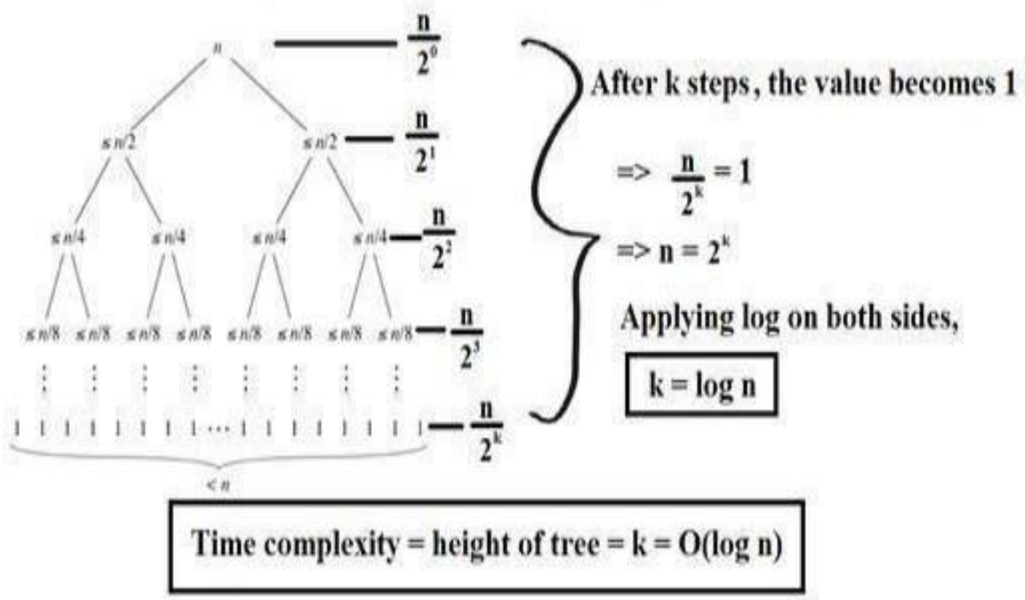



```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
   if (low < high)
   {
```

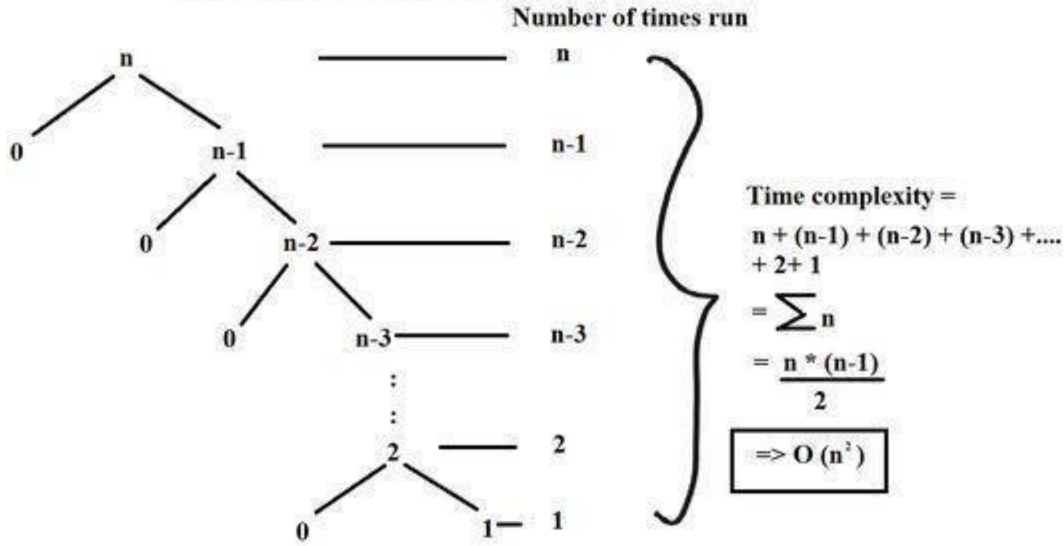
```
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
/* This function takes last element as pivot, places
   the pivot element at its correct position in sorted
    array, and places all smaller (smaller than pivot)
   to left of pivot and all greater elements to right
   of pivot */
partition (arr[], low, high)
{
    // pivot (Element to be placed at right position)
    pivot = arr[high];

    i = (low - 1)  // Index of smaller element and indicates the
                   // right position of pivot found so far
    for (j = low; j <= high- 1; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
            i++;    // increment index of smaller element
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```

Quick Sort: Best case scenario



Quick Sort- Worst Case Scenario



Code :

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

```
// Function to partition the array and return the pivot index
int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // Selecting the last element as pivot
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            // Swap arr[i] and arr[j]
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
    // Swap arr[i+1] and arr[high] (pivot)
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;
    return (i + 1);
}

// Function to perform Quick Sort
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        // Partitioning index
        int pivot_index = partition(arr, low, high);

        // Recursively sort elements before partition and after partition
        quickSort(arr, low, pivot_index - 1);
        quickSort(arr, pivot_index + 1, high);
    }
}

// Function to generate random array of given size
void generateRandomArray(int arr[], int n) {
    srand(time(NULL));
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 1000; // Generating random numbers between 0 and 999
    }
```

```c
}

int main() {
    int n, i;

    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int arr[n];

    // Generating random array
    generateRandomArray(arr, n);

    // Sorting the array using Quick Sort
    clock_t start = clock();
    quickSort(arr, 0, n - 1);
    clock_t end = clock();
    double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;

    // Displaying sorted array
    printf("Sorted array: ");
    for (i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    // Displaying time taken for sorting
    printf("Time taken for sorting: %f seconds\n", time_taken);

    return 0;
}
```

Output :

```
PS E:\Testing_Lang> cd "e:\Testing_Lang\" ; if ($?) { gcc test.c -o test } ; if ($?) { .\test }
Enter the number of elements: 6
Sorted array: 172 349 548 795 802 808
Time taken for sorting: 0.000000 seconds
PS E:\Testing_Lang> 
```

Conclusion :

This program prompts the user to enter the number of elements 'n'. It then generates a random array of 'n' elements, sorts the array using the Quick Sort algorithm with the Divide and Conquer technique, and displays the sorted array along with the time taken for sorting.

To perform a comparative analysis for large values of 'n', you can modify the program to execute multiple times with increasing values of 'n' and measure the time taken for each execution. Additionally, you can compare the performance of Quick Sort with other sorting algorithms to analyze their efficiency for different input sizes.

Experiment No. 4
Binary Search Algorithm
Date of Performance:
Date of Submission:

Experiment No. 4

Title: Binary Search Algorithm

Aim: To study and implement Binary Search Algorithm

Objective: To introduce Divide and Conquer based algorithms

Theory:

Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise, narrow it to the upper half. Repeatedly check until the value is found or the interval is empty

- Binary search is efficient than linear search. For binary search, the array must be sorted, which is not required in case of linear search.
- It is divide and conquer based search technique.
- In each step the algorithms divides the list into two halves and check if the element to be searched is on upper or lower half the array
- If the element is found, algorithm returns.

The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(Log n).

- ❑ Compare x with the middle element.
- ❑ If x matches with the middle element, we return the mid index.
- ❑ Else If x is greater than the mid element, then x can only lie in the right half subarray after the mid element. So we recur for the right half.
- ❑ Else (x is smaller) recur for the left half.
- ❑ Binary Search reduces search space by half in every iterations. In a linear search, search space was reduced by one only.
- ❑ n=elements in the array
- ❑ Binary Search would hit the bottom very quickly.

	Linear Search	Binary Search
2nd iteration	n-1	n/2
3rd iteration	n-2	n/4

Example:

Algorithm BINARY_SEARCH(A, key)
// Description: Perform BS on array A
// I/P : array A of size n & key element to be searched.
// O/P : Success/failure.

low ← 1
high ← n
while low < high do
  mid ← (low+high)/2
  if A[mid] == key then
    return mid
  else if A[mid] < key then
    low ← mid+1
  else
    high ← mid-1
  end
end
return 0

A{11, 22, 33, 44, 55, 66, 77, 88}
  (indices 1 2 3 4 5 6 7 8)
key = 33

low = 1
high = 8
mid = 1+8/2
= 4
A[4] == 33 ×
A[4] < 33 ×
44
high = 4-1
high = 3

{11, 22, 33}
1 2 3

low = 1
high = 3
mid = 1+3/2
= 2
A[2] == 33 ×
22 < 33
low = 3

{33} mid = 3+3/2 = 3
A[3] = 33
A[mid] = 33
key = A[3]

Algorithm and Complexity:

The binary search

- Algorithm 3: the binary search algorithm

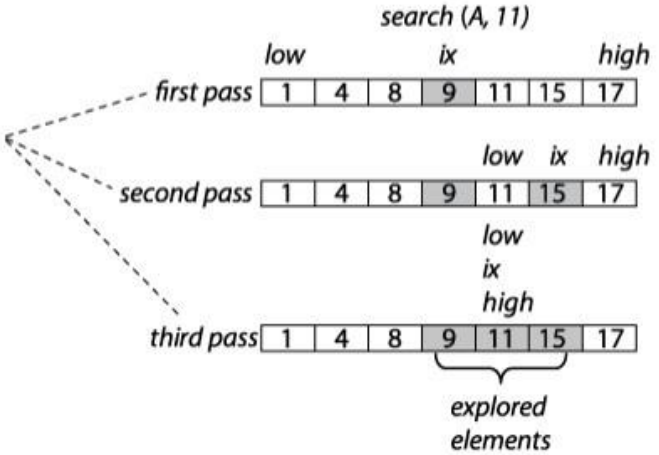
Procedure binary search (x: integer, $a_1, a_2, \ldots, a_n$: increasing integers)
  i :=1 { i is left endpoint of search interval}
  j :=n { j is right endpoint of search interval}
While i < j
begin
  m := $\lfloor (i + j) / 2 \rfloor$
  if $x > a_m$ **then** i := m+1
  else j := m
end
If $x = a_i$ **then** *location* := i
else *location* :=0
{*location* is the subscript of the term equal to x, or 0 if x is not found}

BINARY SEARCH

Best	Average	Worst
O (1)	O (log n)	O (log n)

Array
Divide and Conquer

```
search (A, t)
    low = 0
    high = n−1
    while (low ≤ high) do
      ix = (low + high)/2
      if (t = A[ix]) then
          return true
      else if (t < A[ix]) then
          high = ix − 1
      else low = ix + 1
    return false
end
```



Best Case:
Key is first compared with the middle element of the array.
The key is in the middle position of the array, the algorithm does only one comparison, irrespective of the size of the array.
T(n)=1

Worst Case:
In each iteration search space of BS is reduced by half, Maximum log n(base 2) array divisions are possible.
Recurrence relation is
T(n)=T(n/2) +1
Running Time is O(logn).

Average Case:

Key element neither is in the middle nor at the leaf level of the search tree.
It does half of the log n(base 2).
Base case=O(1)
Average and worst case=O(logn)

Code :

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

// Function to perform Binary Search recursively
int binarySearch(int arr[], int low, int high, int key) {
   if (low <= high) {
      int mid = low + (high - low) / 2;
      if (arr[mid] == key) {
         return mid;
      } else if (arr[mid] < key) {
         return binarySearch(arr, mid + 1, high, key);
      } else {
         return binarySearch(arr, low, mid - 1, key);
      }
   }
   return -1; // Key not found
}
```

```
// Function to generate a sorted array of 'n' numbers
void generateSortedArray(int arr[], int n) {
   srand(time(NULL));
   arr[0] = rand() % 10; // Generating the first element randomly
   for (int i = 1; i < n; i++) {
      arr[i] = arr[i - 1] + (rand() % 10); // Generating subsequent elements by adding a
random number between 0 and 9
   }
}

int main() {
   int n, key;

   printf("Enter the number of elements: ");
   scanf("%d", &n);

   int arr[n];

   // Generating sorted array
   generateSortedArray(arr, n);

   // Displaying the sorted array
   printf("Sorted array: ");
   for (int i = 0; i < n; i++) {
      printf("%d ", arr[i]);
   }
   printf("\n");

   // Prompting the user to enter the key to search
   printf("Enter the key to search: ");
   scanf("%d", &key);

   // Performing binary search
   clock_t start = clock();
   int index = binarySearch(arr, 0, n - 1, key);
   clock_t end = clock();
   double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;

   // Displaying the result of binary search
```

```
    if (index != -1) {
        printf("Key %d found at index %d.\n", key, index);
    } else {
        printf("Key %d not found.\n", key);
    }

    // Displaying time taken for binary search
    printf("Time taken for binary search: %f seconds\n", time_taken);

    return 0;
}
```

Output :

```
PS E:\Testing_Lang> cd "e:\Testing_Lang\" ; if ($?) { gcc test.c -o test } ; if ($?) { .\test }
Enter the number of elements: 4
Sorted array: 1 10 13 19
Enter the key to search:
```

Conclusion:

This program generates a sorted array of 'n' numbers and performs a binary search for a key entered by the user. It measures the time taken for the binary search operation and displays the result along with the time taken.

To perform an analysis for different values of 'n', you can modify the program to execute multiple times with increasing values of 'n' and measure the time taken for each execution. Additionally, you can compare the performance of the binary search algorithm with other searching algorithms to analyze their efficiency for different input sizes.

Experiment No. 5
Fractional Knapsack using Greedy Method
Date of Performance:
Date of Submission:

Experiment No. 5

Title: Fraction Knapsack

Aim: To study and implement Fraction Knapsack Algorithm

Objective: To introduce Greedy based algorithms

Theory:

Greedy method or technique is used to solve Optimization problems. A solution that can be maximized or minimized is called Optimal Solution.

The knapsack problem or rucksack problem is a problem in combinatorial optimization: Given a set of items, each with a mass and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed size knapsack and must fill it with the most valuable items. The most common problem being solved is the 0-1 knapsack problem, which restricts the number xi of copies of each kind of item to zero or one.

In Knapsack problem we are given:1) n objects 2) Knapsack with capacity m, 3) An object i is associated with profit Wi , 4) An object i is associated with profit Pi , 5) when an object i is placed in knapsack we get profit Pi Xi .

Here objects can be broken into pieces (Xi Values) The Objective of Knapsack problem is to maximize the profit.

Example:

In this version of Knapsack problem, items can be broken into smaller pieces. So, the thief may take only a fraction x_i of i^{th} item.

$$0 \leqslant xi \leqslant 1$$

The i^{th} item contributes the weight xi.wi to the total weight in the knapsack and profit xi.pi to the total profit.

greedy_fractional_knapsack (w[1...n], p[1...n], W)

```
for i=1 to n
    do x[i] = 0
weight = 0
for i=1 to n
    if weight + w[i] ≤ W then
        x[i] = 1
        weight = weight + w[i]
    else
        x[i] = (W-weight) / w[i]
        weight = W
        break
return x
```

i=1 → B
0+10 ≤ 60
x[i] = 1
wt = 10

i=2 → A
10+40
50 ≤ 60
x[i] = 2
10+40
wt = 50

i=3 → C
(60-50)/20
10/20 = 1/2
x[i] = 1/2
wt = 60

x = [A, B, 1/2 C]

~~x[i] = 0~~
~~wt = 0~~

Total profit is
100+280+120*(10/20)
380+60 = 440

Total wt
10+40+20*(10/20)
= 60

Ex:
W = 60

Item	A	B	C	D
profit	280	100	120	120
weight	40	10	20	24
Ratio $\left(\frac{p_i}{w_i}\right)$	7	10	6	5

provided items are not sorted based on $\frac{p_i}{w_i}$.

sorted

Item	B	A	C	D
profit	100	280	120	120
weight	10	40	20	24
Ratio $\left(\frac{p_i}{w_i}\right)$	10	7	6	5

Algorithm:

Hence, the objective of this algorithm is to

$$maximize \sum_{n=1}^{n}(x_i . pi)$$

subject to constraint,

$$\sum_{n=1}^{n}(x_i . wi) \leqslant W$$

It is clear that an optimal solution must fill the knapsack exactly, otherwise we could add a fraction of one of the remaining items and increase the overall profit.

Thus, an optimal solution can be obtained by

$$\boxed{\sum_{n=1}^{n}(x_i . wi) = W}$$

In this context, first we need to sort those items according to the value of $\frac{p_i}{w_i}$, so that $\frac{p_i+1}{w_i+1}$ ≤ $\frac{p_i}{w_i}$. Here, ***x*** is an array to store the fraction of items.

```
Algorithm: Greedy-Fractional-Knapsack (w[1..n], p[1..n], W)
for i = 1 to n
   do x[i] = 0
weight = 0
for i = 1 to n
   if weight + w[i] ≤ W then
      x[i] = 1
      weight = weight + w[i]
   else
      x[i] = (W - weight) / w[i]
      weight = W
      break
return x
```

Code :

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

// Structure to represent an item
struct Item {
   int value;   // Value of the item
   int weight;  // Weight of the item
};

// Function to compare items based on their value-to-weight ratio
int compare(const void *a, const void *b) {
   double ratio1 = ((struct Item *)a)->value / (double)((struct Item *)a)->weight;
   double ratio2 = ((struct Item *)b)->value / (double)((struct Item *)b)->weight;
   return ratio2 > ratio1 ? 1 : -1;
}

// Function to solve Fractional Knapsack problem
void fractionalKnapsack(struct Item items[], int n, int capacity) {
   // Sort items based on their value-to-weight ratio
```

```
    qsort(items, n, sizeof(struct Item), compare);

    int curWeight = 0;  // Current weight in knapsack
    double finalValue = 0.0;  // Final value of items selected

    // Loop through sorted items and add to knapsack until capacity is reached
    for (int i = 0; i < n; i++) {
        // If adding the entire item exceeds capacity, add fractional part
        if (curWeight + items[i].weight <= capacity) {
            curWeight += items[i].weight;
            finalValue += items[i].value;
        } else {
            int remaining = capacity - curWeight;
            finalValue += items[i].value * ((double)remaining / items[i].weight);
            break;
        }
    }

    // Print the final value of items selected
    printf("Maximum value in knapsack = %.2lf\n", finalValue);
}

int main() {
    int n, capacity;

    printf("Enter the number of items: ");
    scanf("%d", &n);

    struct Item items[n];

    // Input values and weights of items
    for (int i = 0; i < n; i++) {
        printf("Enter value and weight of item %d: ", i + 1);
        scanf("%d %d", &items[i].value, &items[i].weight);
    }

    printf("Enter the capacity of knapsack: ");
    scanf("%d", &capacity);
```

```
    // Solve Fractional Knapsack problem
    fractionalKnapsack(items, n, capacity);

    return 0;
}
```

Output :

```
PS E:\Testing_Lang> cd "e:\Testing_Lang\" ; if ($?) { gcc test.c -o test } ; if ($?) { .\test }
Enter the number of items: 5
Enter value and weight of item 1: 2
5
Enter value and weight of item 2: 4
6
Enter value and weight of item 3: 4
9
Enter value and weight of item 4: 7
8
Enter value and weight of item 5: 7

4
Enter the capacity of knapsack: 50
Maximum value in knapsack = 24.00
PS E:\Testing_Lang> 
```

Conclusion:

This program prompts the user to enter the number of items, their values, weights, and the capacity of the knapsack. It then solves the Fractional Knapsack problem using the Greedy method and prints the maximum value that can be obtained in the knapsack.

In the Fractional Knapsack problem, items can be taken in fractional amounts, so the greedy approach is to always select the item with the maximum value-to-weight ratio first. This ensures that the value obtained per unit of weight is maximized.

Experiment No. 6
Prim's Algorithm
Date of Performance:

Date of Submission:

Experiment No. 6

Title: Prim's Algorithm.

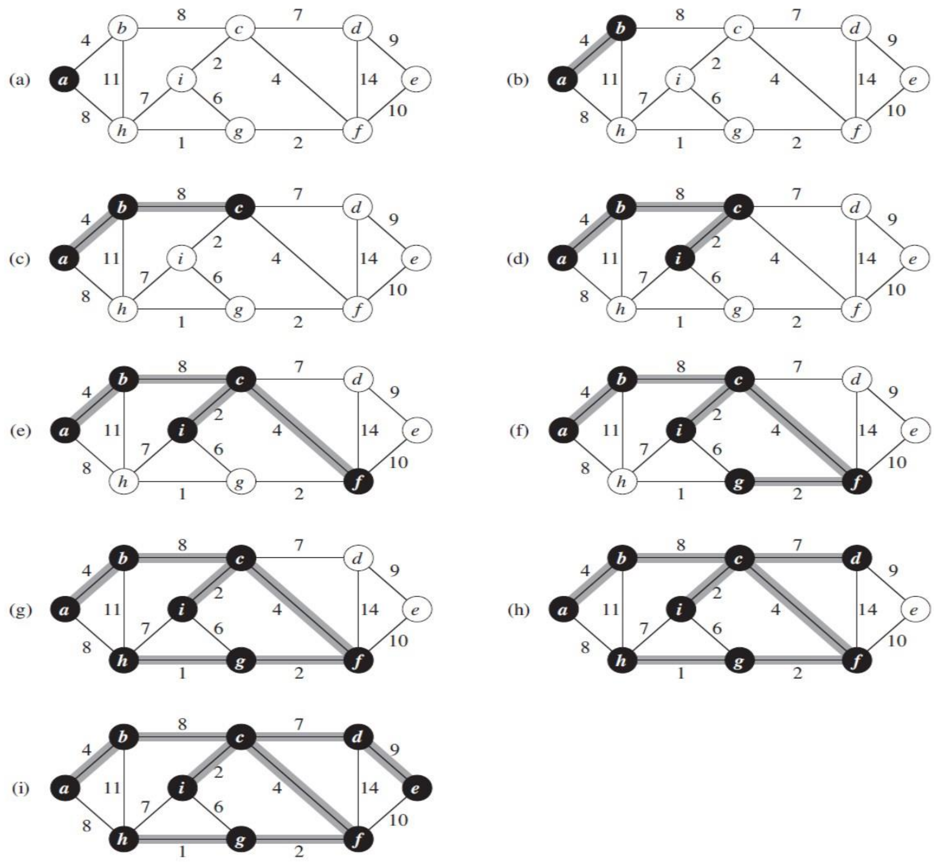
Aim: To study and implement Prim's Minimum Cost Spanning Tree Algorithm.

Objective: To introduce Greedy based algorithms

Theory:

Prim's algorithm is a greedy algorithm that finds a minimum spanning tree for a weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. The algorithm operates by building this tree one vertex at a time, from an arbitrary starting vertex, at each step adding the cheapest possible connection from the tree to another vertex.

Example:



Algorithm and Complexity:

```
Algorithm Prim(E, cost, n, t)
// E is the set of edges in G. cost[1 : n, 1 : n] is the cost
// adjacency matrix of an n vertex graph such that cost[i, j] is
// either a positive real number or ∞ if no edge (i, j) exists.
// A minimum spanning tree is computed and stored as a set of
// edges in the array t[1 : n − 1, 1 : 2]. (t[i, 1], t[i, 2]) is an edge in
// the minimum-cost spanning tree. The final cost is returned.
{
    Let (k, l) be an edge of minimum cost in E;
    mincost := cost[k, l];
    t[1, 1] := k; t[1, 2] := l;
    for i := 1 to n do  // Initialize near.
        if (cost[i, l] < cost[i, k]) then near[i] := l;
        else near[i] := k;
    near[k] := near[l] := 0;
    for i := 2 to n − 1 do
    { // Find n − 2 additional edges for t.
        Let j be an index such that near[j] ≠ 0 and
        cost[j, near[j]] is minimum;
        t[i, 1] := j; t[i, 2] := near[j];
        mincost := mincost + cost[j, near[j]];
        near[j] := 0;
        for k := 1 to n do // Update near[ ].
            if ((near[k] ≠ 0) and (cost[k, near[k]] > cost[k, j]))
                then near[k] := j;
    }
    return mincost;
}
```

Time Complexity is O(n2), Where, n = number of vertices **Theory:**

Code :

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

#define V 5  // Number of vertices in the graph

// Function to find the vertex with the minimum key value
int minKey(int key[], int mstSet[]) {
   int min = INT_MAX, min_index;
   for (int v = 0; v < V; v++) {
      if (mstSet[v] == 0 && key[v] < min) {
         min = key[v];
         min_index = v;
```

```
        }
    }
    return min_index;
}

// Function to print the MST
void printMST(int parent[], int graph[V][V]) {
    printf("Edge   Weight\n");
    for (int i = 1; i < V; i++) {
        printf("%d - %d    %d \n", parent[i], i, graph[i][parent[i]]);
    }
}

// Function to construct and print MST for a graph represented using adjacency
matrix
void primMST(int graph[V][V]) {
    int parent[V];   // Array to store constructed MST
    int key[V];      // Key values used to pick minimum weight edge in cut
    int mstSet[V];   // To represent set of vertices not yet included in MST

    // Initialize all keys as INFINITE
    for (int i = 0; i < V; i++) {
        key[i] = INT_MAX;
        mstSet[i] = 0;
    }

    // Always include first vertex in MST.
    key[0] = 0;     // Make key 0 so that this vertex is picked as first vertex
    parent[0] = -1; // First node is always root of MST

    // The MST will have V vertices
    for (int count = 0; count < V - 1; count++) {
        // Pick the minimum key vertex from the set of vertices not yet included in MST
        int u = minKey(key, mstSet);

        // Add the picked vertex to the MST set
        mstSet[u] = 1;
```

```
      // Update key value and parent index of the adjacent vertices of the picked
vertex.
      // Consider only those vertices which are not yet included in MST
      for (int v = 0; v < V; v++) {
        // graph[u][v] is non zero only for adjacent vertices of m
        // mstSet[v] is false for vertices not yet included in MST
        // Update the key only if graph[u][v] is smaller than key[v]
        if (graph[u][v] && mstSet[v] == 0 && graph[u][v] < key[v]) {
          parent[v] = u;
          key[v] = graph[u][v];
        }
      }
    }

    // Print the constructed MST
    printMST(parent, graph);
}

int main() {
    /* Let us create the following graph
         2    3
      (0)--(1)--(2)
      |   / \   |
     6| 8/   \5 |7
      | /     \ |
      (3)-------(4)
           9        */
    int graph[V][V] = {{0, 2, 0, 6, 0},
                       {2, 0, 3, 8, 5},
                       {0, 3, 0, 0, 7},
                       {6, 8, 0, 0, 9},
                       {0, 5, 7, 9, 0}};

    // Print the solution
    primMST(graph);

    return 0;
}
```

Output :

```
PS E:\Testing_Lang> cd "e:\Testing_Lang\" ; if ($?) { gcc test.c -o test } ; if ($?) { .\test }
Edge   Weight
0 - 1    2
1 - 2    3
0 - 3    6
1 - 4    5
PS E:\Testing_Lang> 
```

Conclusion:

This program implements Prim's MST algorithm using the Greedy method for a given graph represented using an adjacency matrix. It constructs and prints the Minimum Spanning Tree (MST) of the graph.

Prim's algorithm works by starting from an arbitrary vertex and repeatedly adding the minimum-weight edge that connects a vertex in the MST to a vertex outside the MST. It continues this process until all vertices are included in the MST.

Experiment No. 7
Kruskal's Algorithm
Date of Performance:
Date of Submission:

Experiment No. 7

Title: Kruskal’s Algorithm.

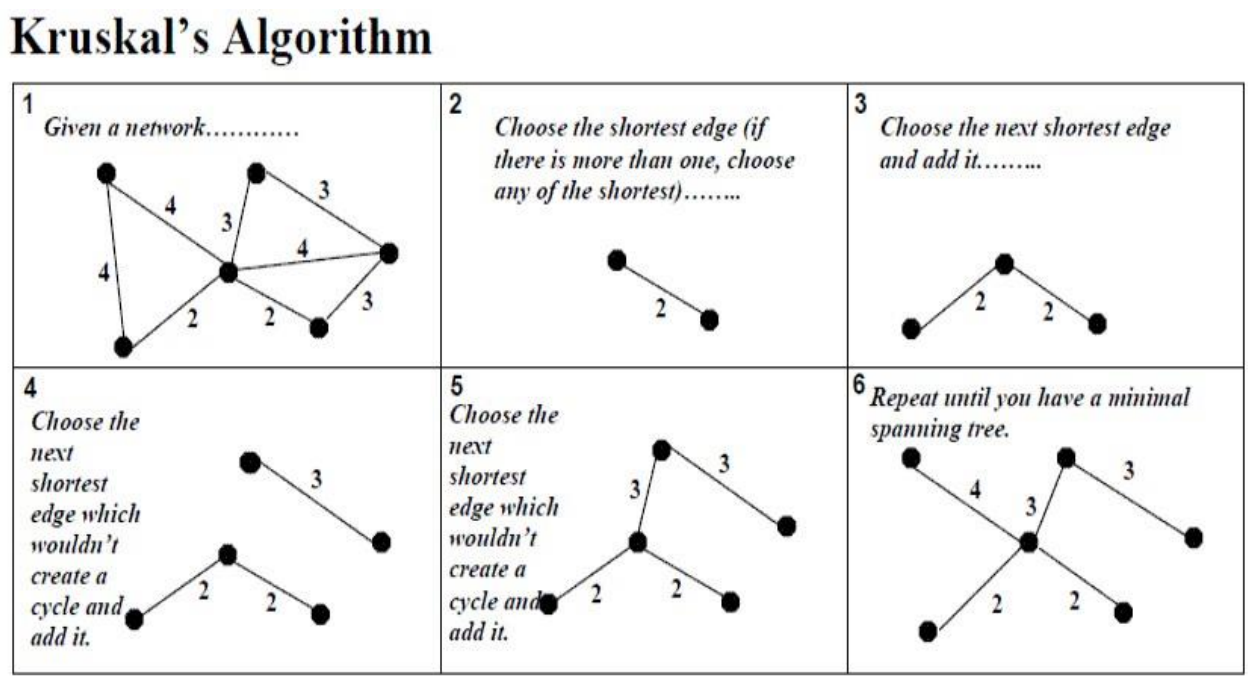
Aim: To study and implement Kruskal’s Minimum Cost Spanning Tree Algorithm.

Objective: To introduce Greedy based algorithms

Theory:

Kruskal's algorithm finds a minimum spanning forest of an undirected edge-weighted graph. If the graph is connected, it finds a minimum spanning tree. (A minimum spanning tree of a connected graph is a subset of the edges that forms a tree that includes every vertex, where the sum of the weights of all the edges in the tree is minimized. For a disconnected graph, a minimum spanning forest is composed of a minimum spanning tree for each connected component.) It is a greedy algorithm in graph theory as in each step it adds the next lowest-weight edge that will not form a cycle to the minimum spanning forest.

Example:



Algorithm and Complexity:

```
Algorithm Kruskal(E, cost, n, t)
// E is the set of edges in G. G has n vertices. cost[u, v] is the
// cost of edge (u, v). t is the set of edges in the minimum-cost
// spanning tree. The final cost is returned.
{
    Construct a heap out of the edge costs using Heapify;
    for i := 1 to n do parent[i] := -1;
    // Each vertex is in a different set.
    i := 0; mincost := 0.0;
    while ((i < n - 1)  and (heap not empty)) do
    {
        Delete a minimum cost edge (u, v) from the heap
        and reheapify using Adjust;
        j := Find(u); k := Find(v);
        if (j ≠ k) then
        {
            i := i + 1;
            t[i, 1] := u; t[i, 2] := v;
            mincost := mincost + cost[u, v];
            Union(j, k);
        }
    }
    if (i ≠ n - 1) then write ("No spanning tree");
    else return mincost;
}
```

Time Complexity is O(nlog n), Where, n = number of Edges

Code :

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

// Structure to represent an edge in the graph
struct Edge {
   int src, dest, weight;
};

// Structure to represent a subset for union-find
struct Subset {
   int parent;
   int rank;
};
```

```c
// Function to find set of an element i
int find(struct Subset subsets[], int i) {
   if (subsets[i].parent != i)
      subsets[i].parent = find(subsets, subsets[i].parent);
   return subsets[i].parent;
}

// Function that does union of two sets of x and y
void Union(struct Subset subsets[], int x, int y) {
   int xroot = find(subsets, x);
   int yroot = find(subsets, y);

   // Attach smaller rank tree under root of high rank tree (Union by Rank)
   if (subsets[xroot].rank < subsets[yroot].rank)
      subsets[xroot].parent = yroot;
   else if (subsets[xroot].rank > subsets[yroot].rank)
      subsets[yroot].parent = xroot;
   else {
      subsets[yroot].parent = xroot;
      subsets[xroot].rank++;
   }
}

// Comparator function to sort edges based on weight
int compare(const void *a, const void *b) {
   struct Edge *a1 = (struct Edge *)a;
   struct Edge *b1 = (struct Edge *)b;
   return a1->weight > b1->weight;
}

// Function to construct and print MST using Kruskal's algorithm
void KruskalMST(struct Edge edges[], int V, int E) {
   struct Edge result[V];  // Store the result MST
   int e = 0;  // An index variable used for result[]
   int i = 0;  // An index variable used for sorted edges

   // Step 1: Sort all the edges in non-decreasing order of their weight
   qsort(edges, E, sizeof(edges[0]), compare);
```

```
    // Allocate memory for creating V subsets
    struct Subset *subsets = (struct Subset *)malloc(V * sizeof(struct Subset));

    // Create V subsets with single elements
    for (int v = 0; v < V; v++) {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    // Number of edges to be taken is equal to V-1
    while (e < V - 1 && i < E) {
        // Step 2: Pick the smallest edge. And increment the index for next iteration
        struct Edge next_edge = edges[i++];

        int x = find(subsets, next_edge.src);
        int y = find(subsets, next_edge.dest);

        // If including this edge doesn't cause cycle, include it in result and increment
the index
        // of result for next edge
        if (x != y) {
            result[e++] = next_edge;
            Union(subsets, x, y);
        }
    }

    // Print the constructed MST
    printf("Following are the edges in the constructed MST:\n");
    for (i = 0; i < e; ++i)
        printf("%d -- %d == %d\n", result[i].src, result[i].dest, result[i].weight);
}

int main() {
    /* Let us create the following graph
            10
        0--------1
        |  \     |
       6|   5\   |15
        |      \ |
```

```
      2--------3
         4       */
    int V = 4;  // Number of vertices in graph
    int E = 5;  // Number of edges in graph
    struct Edge edges[] = {{0, 1, 10}, {0, 2, 6}, {0, 3, 5}, {1, 3, 15}, {2, 3, 4}};

    // Function call to construct and print MST
    KruskalMST(edges, V, E);

    return 0;
}
```

Output :

```
PS E:\Testing_Lang> cd "e:\Testing_Lang\" ; if ($?) { gcc test.c -o test } ; if ($?) { .\test }
Following are the edges in the constructed MST:
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
PS E:\Testing_Lang>
```

Conclusion:

This program implements Kruskal's MST algorithm using the Greedy method for a given graph represented using an array of edges. It constructs and prints the Minimum Spanning Tree (MST) of the graph.

Kruskal's algorithm works by sorting all the edges in non-decreasing order of their weight and selecting edges one by one in this sorted order, adding the edge to the MST if it doesn't create a cycle. It continues this process until all vertices are included in the MST.

Experiment No. 8
Single Source Shortest Path using Dynamic Programming (Bellman-Ford Algorithm)
Date of Performance:
Date of Submission:

Experiment No: 8

Title: Single Source Shortest Path: Bellman Ford

Aim: To study and implement Single Source Shortest Path using Dynamic Programming: Bellman Ford

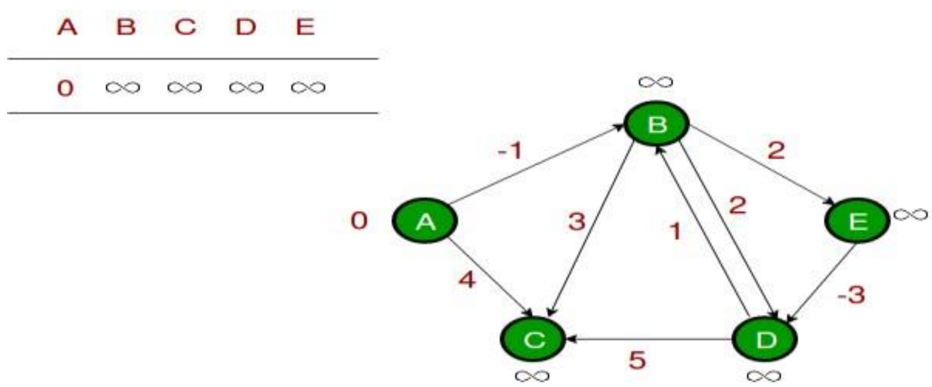
Objective: To introduce Bellman Ford method

Theory:

Given a graph and a source vertex source in graph, find shortest paths from src to all vertices in the given graph. The graph may contain negative weight edges.We have discussed Dijkstra's algorithm for this problem. Dijkstra's algorithm is a Greedy algorithm and time complexity is O(VLogV) (with the use of Fibonacci heap). Dijkstra doesn't work for Graphs with negative weight edges, Bellman-Ford works for such graphs. Bellman-Ford is also simpler than Dijkstra and suites well for distributed systems. But time complexity of Bellman-Ford is O(VE), which is more than Dijkstra.

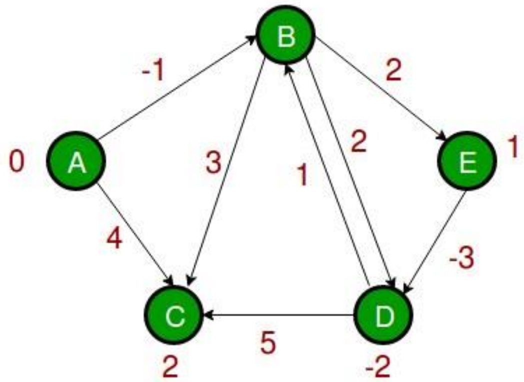
Example:

Let the given source vertex be 0. Initialize all distances as infinite, except the distance to the source itself. Total number of vertices in the graph is 5, so all edges must be processed 4 times.

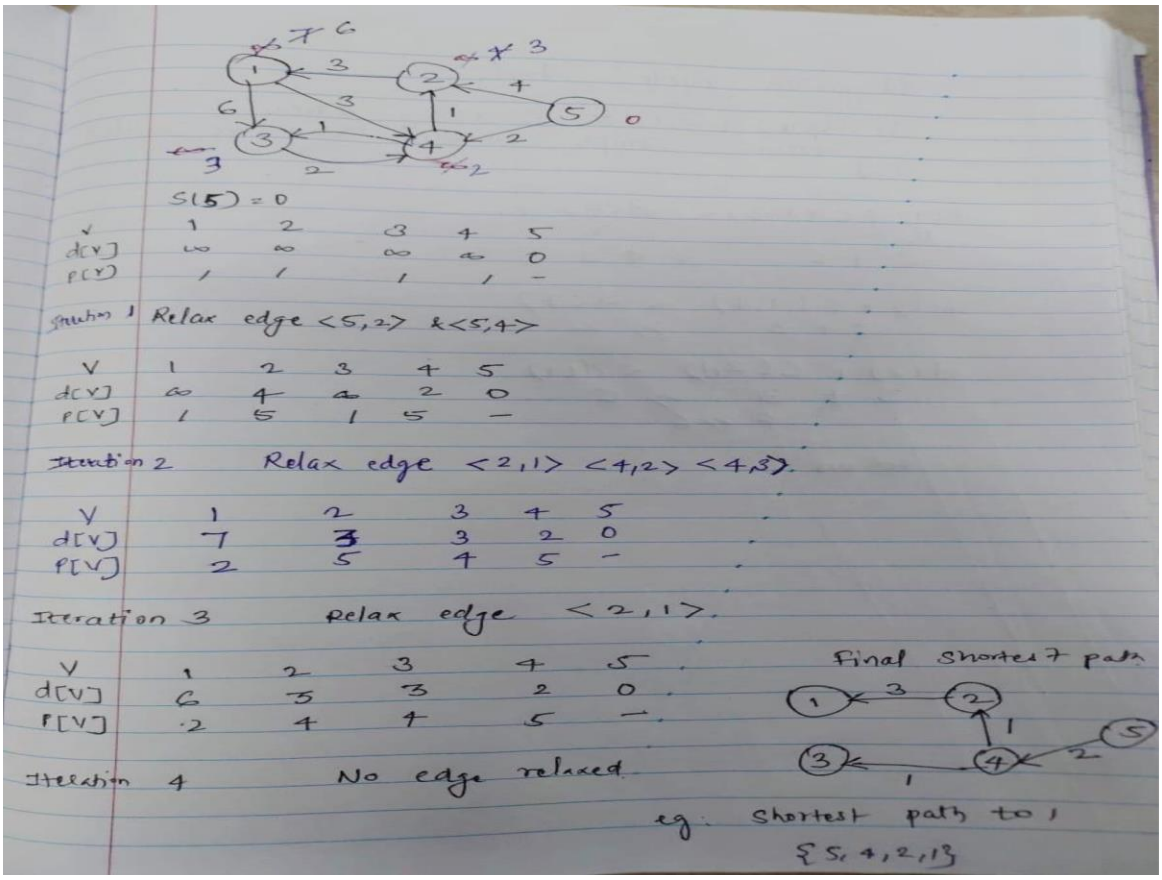


Let all edges are processed in the following order: (B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D). We get the following distances when all edges are processed the first time. The first row shows initial distances. The second row shows distances when edges (B, E), (D, B), (B, D) and (A, B) are processed. The third row shows distances when (A, C) is processed. The fourth row shows when (D, C), (B, C) and (E, D) are processed.

A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1	2	∞	1
0	-1	2	1	1
0	-1	2	-2	1



The second iteration guarantees to give all shortest paths which are at most 2 edges long. The algorithm processes all edges 2 more times. The distances are minimized after the second iteration, so third and fourth iterations don't update the distances.



Algorithm:

```
function Bellman_Ford(list vertices, list edges, vertex source, distance[], parent[])

// Step 1 – initialize the graph. In the beginning, all vertices weight of
// INFINITY and a null parent, except for the source, where the weight is 0

for each vertex v in vertices
   distance[v] = INFINITY
   parent[v] = NULL

distance[source] = 0
// Step 2 – relax edges repeatedly
   for i = 1 to V-1    // V – number of vertices
      for each edge (u, v) with weight w
         if (distance[u] + w) is less than distance[v]
            distance[v] = distance[u] + w
            parent[v] = u

// Step 3 – check for negative-weight cycles
for each edge (u, v) with weight w
   if (distance[u] + w) is less than distance[v]
      return "Graph contains a negative-weight cycle"

return distance[], parent[]
```

Output:

Shortest path from source (5)

Vertex 5 -> cost=0 parent=0

Vertex 1-> cost=6 parent=2

Vertex 2-> cost=3 parent=4

Vertex 3-> cost =3 parent =4

Vertex 4-> cost =2 paren=5

Code :

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

// Structure to represent an edge in the graph
struct Edge {
   int src, dest, weight;
};

// Structure to represent a graph
struct Graph {
   int V, E;
   struct Edge *edge;
};

// Function to create a graph with V vertices and E edges
struct Graph* createGraph(int V, int E) {
   struct Graph* graph = (struct Graph*) malloc(sizeof(struct Graph));
   graph->V = V;
   graph->E = E;
   graph->edge = (struct Edge*) malloc(graph->E * sizeof(struct Edge));
   return graph;
}

// Function to find the shortest path from source to all other vertices using Bellman-
Ford algorithm
void BellmanFord(struct Graph* graph, int src) {
   int V = graph->V;
   int E = graph->E;
   int dist[V];

   // Initialize distances from source to all other vertices as INFINITE
   for (int i = 0; i < V; i++)
      dist[i] = INT_MAX;
```

```
    dist[src] = 0;

    // Relax all edges |V| - 1 times
    for (int i = 1; i <= V - 1; i++) {
        for (int j = 0; j < E; j++) {
            int u = graph->edge[j].src;
            int v = graph->edge[j].dest;
            int weight = graph->edge[j].weight;
            if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
                dist[v] = dist[u] + weight;
        }
    }

    // Check for negative-weight cycles
    for (int i = 0; i < E; i++) {
        int u = graph->edge[i].src;
        int v = graph->edge[i].dest;
        int weight = graph->edge[i].weight;
        if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
            printf("Graph contains negative weight cycle.\n");
            return;
        }
    }

    // Print the shortest distances
    printf("Vertex   Distance from Source\n");
    for (int i = 0; i < V; i++)
        printf("%d \t\t %d\n", i, dist[i]);
}

int main() {
    int V, E, src;
    printf("Enter number of vertices and edges: ");
    scanf("%d %d", &V, &E);

    struct Graph* graph = createGraph(V, E);

    printf("Enter source vertex: ");
    scanf("%d", &src);
```

```
    printf("Enter edges (src dest weight):\n");
    for (int i = 0; i < E; i++)
        scanf("%d %d %d", &graph->edge[i].src, &graph->edge[i].dest, &graph->edge[i].weight);

    BellmanFord(graph, src);

    return 0;
}
```

Output :

```
PS E:\Testing_Lang> cd "e:\Testing_Lang\" ; if ($?) { gcc test.c -o test } ; if ($?) { .\test }
Enter number of vertices and edges: 2 3
Enter source vertex: 0
Enter edges (src dest weight):
5 4
6 1
2 3
4 8
5 1
Vertex   Distance from Source
0               0
1               2147483647
PS E:\Testing_Lang> 
```

Conclusion:

This program prompts the user to input the number of vertices and edges of the graph, the source vertex, and the details of each edge (source, destination, and weight). It then uses the Bellman-Ford algorithm to find the shortest path from the source vertex to all other vertices in the graph and prints the shortest distances. If the graph contains a negative-weight cycle, it will detect and print a message indicating that.

Experiment No. 9
Travelling Salesperson Problem using Dynamic Approach
Date of Performance:
Date of Submission:

Experiment No. 9

Title: Travelling Salesman Problem

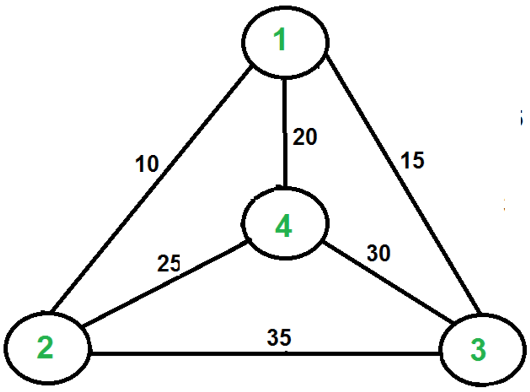
Aim: To study and implement Travelling Salesman Problem.

Objective: To introduce Dynamic Programming approach

Theory:

The **Traveling Salesman Problem (TSP)** is a classic optimization problem in which a salesperson needs to visit a set of cities exactly once and return to the starting city while minimizing the total distance traveled.

Given a set of cities and the distance between every pair of cities, find the **shortest possible route** that visits every city exactly once and returns to the starting point.



For example, consider the graph shown in the figure on the right side. A TSP tour in the graph is 1-2-4-3-1. The cost of the tour is 10+25+30+15 which is 80. The problem is a famous NP-hard problem. There is no polynomial-time know solution for this problem. The following are different solutions for the traveling salesman problem.

Naive Solution:

1) Consider city 1 as the starting and ending point.

2) Generate all (n-1)! Permutations of cities.

3) Calculate the cost of every permutation and keep track of the minimum cost permutation.

4) Return the permutation with minimum cost.

Time Complexity: ?(n!)

Dynamic Programming:

Let the given set of vertices be {1, 2, 3, 4,.n}. Let us consider 1 as starting and ending point of output. For every other vertex I (other than 1), we find the minimum cost path with 1 as the starting point, I as the ending point, and all vertices appearing exactly once. Let the cost of this path cost (i), and the cost of the corresponding Cycle would cost (i) + dist(i, 1) where dist(i, 1) is the distance from I to 1. Finally, we return the minimum of all [cost(i) + dist(i, 1)] values. This looks simple so far.

Now the question is how to get cost(i)? To calculate the cost(i) using Dynamic Programming, we need to have some recursive relation in terms of sub-problems.

Let us define a term *C(S, i) be the cost of the minimum cost path visiting each vertex in set S exactly once, starting at 1 and ending at i*. We start with all subsets of size 2 and calculate

C(S, i) for all subsets where S is the subset, then we calculate C(S, i) for all subsets S of size 3 and so on. Note that 1 must be present in every subset.

```
If size of S is 2, then S must be {1, i},

 C(S, i) = dist(1, i)

Else if size of S is greater than 2.

 C(S, i) = min { C(S-{i}, j) + dis(j, i)} where j belongs to S, j !=
i and j != 1.
```

Code :

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

#define V 4  // Number of vertices in the graph

// Function to find the minimum of two numbers
int min(int a, int b) {
   return (a < b) ? a : b;
}

// Function to solve TSP using dynamic programming
int tsp(int graph[][V], int mask, int pos, int n, int dp[][V]) {
   // If all vertices have been visited
   if (mask == (1 << n) - 1) {
      return graph[pos][0]; // Return cost of going back to the starting city
   }

   // If this subproblem has already been computed
   if (dp[mask][pos] != -1) {
      return dp[mask][pos];
   }

   int ans = INT_MAX;
```

```
    // Try to go to an unvisited city
    for (int city = 0; city < n; city++) {
        if ((mask & (1 << city)) == 0) { // If city has not been visited
            int newAns = graph[pos][city] + tsp(graph, mask | (1 << city), city, n, dp);
            ans = min(ans, newAns);
        }
    }

    return dp[mask][pos] = ans;
}

int main() {
    int graph[V][V] = {{0, 10, 15, 20},
                       {10, 0, 35, 25},
                       {15, 35, 0, 30},
                       {20, 25, 30, 0}};

    int dp[1 << V][V]; // Dynamic programming table to store results of subproblems

    // Initialize dp table with -1
    for (int i = 0; i < (1 << V); i++) {
        for (int j = 0; j < V; j++) {
            dp[i][j] = -1;
        }
    }

    int minCost = tsp(graph, 1, 0, V, dp); // Starting from city 0

    printf("Minimum cost of TSP: %d\n", minCost);

    return 0;
}
```

Output :

```
PS E:\Testing_Lang> cd "e:\Testing_Lang\" ; if ($?) { gcc test.c -o test } ; if ($?) { .\test }
Minimum cost of TSP: 80
PS E:\Testing_Lang> 
```

Conclusion:

Travelling Salesman Problem has been successfully implemented. This program demonstrates solving the TSP using dynamic programming. It initializes a 2D array dp to store the results of subproblems. The tsp function recursively explores all possible permutations of cities and computes the minimum cost of the tour. It uses bitmasking to keep track of visited cities efficiently.

Experiment No. 10
Sum of Subset using Backtracking
Date of Performance:
Date of Submission:

Title: Sum of Subset

Aim: To study and implement Sum of Subset problem

Objective: To introduce Backtracking methods

Theory:

Backtracking is finding the solution of a problem whereby the solution depends on the previous steps taken. For example, in a maze problem, the solution depends on all the steps you take one-by-one. If any of those steps is wrong, then it will not lead us to the solution. In a maze problem, we first choose a path and continue moving along it. But once we understand that the particular path is incorrect, then we just come back and change it. This is what backtracking basically is.

In backtracking, we first take a step and then we see if this step taken is correct or not i.e., whether it will give a correct answer or not. And if it doesn't, then we just come back and change our first step. In general, this is accomplished by recursion. Thus, in backtracking, we first start with a partial sub-solution of the problem (which may or may not lead us to the solution) and then check if we can proceed further with this sub-solution or not. If not, then we just come back and change it.

Thus, the general steps of backtracking are:

- start with a sub-solution
- check if this sub-solution will lead to the solution or not
- If not, then come back and change the sub-solution and continue again.

The subset sum problem is a classic optimization problem that involves finding a subset of a given set of positive integers whose sum matches a given target value. More formally, given a set of non-negative integers and a target sum, we aim to determine whether there exists a subset of the integers whose sum equals the target.

Let's consider an example to better understand the problem. Suppose we have a set of integers [1, 4, 6, 8, 2] and a target sum of 9. We need to determine whether there exists a subset within the given set whose sum equals the target, in this case, 9. In this example, the subset [1, 8] satisfies the condition, as their sum is indeed 9.

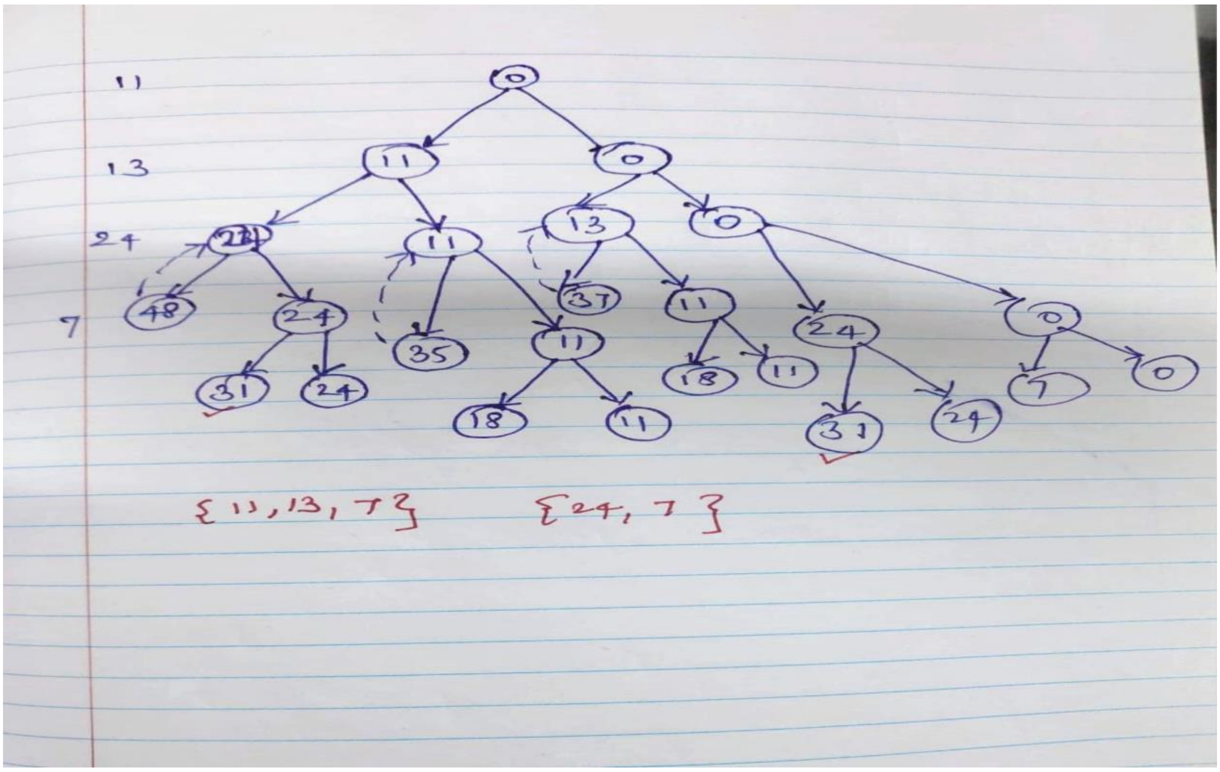
Solving Subset Sum with Backtracking

To solve the subset, sum problem using backtracking, we will follow a recursive approach. Here's an outline of the algorithm:

1. Sort the given set of integers in non-decreasing order.
2. Start with an empty subset and initialize the current sum as 0.
3. Iterate through each integer in the set:

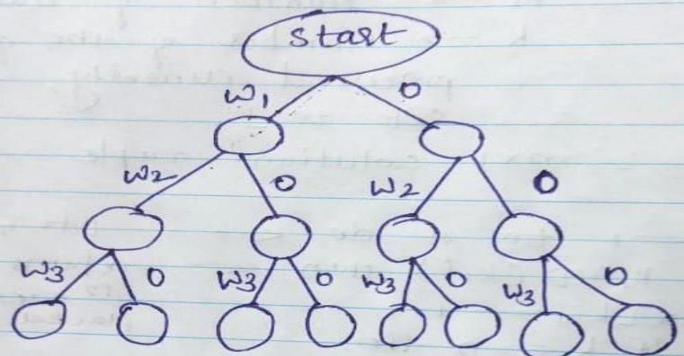
- Include the current integer in the subset.
- Increment the current sum by the value of the current integer.
- Recursively call the algorithm with the updated subset and current sum.
- If the current sum equals the target sum, we have found a valid subset.
- Backtrack by excluding the current integer from the subset.
- Decrement the current sum by the value of the current integer.

4. If we have exhausted all the integers and none of the subsets sum up to the target, we conclude that there is no valid subset.



State space tree for $n = 3$

* Any path from the root to leaf forms a subset.



Q. Solve the sum of subset problem using backtracking strategy for the following data $n = 4$

$W = (w_1, w_2, w_3, w_4) = (11, 13, 24, 7)$ and $M = 31$

Subset Items	condition	comment
{ }	0	Initial condition
{11}	11 < 31	Add next element
{11, 13}	24 < 31	Add next element
{11, 13, 24}	48 < 31	Sub set exceeds sum so backtrack
{11, 13, 7}	31	Solution found

Code :

```
#include <stdio.h>

#define MAX_SIZE 100

// Function to print subsets that sum to the target
void subsetSum(int arr[], int subset[], int subsetSize, int index, int target) {
   if (target == 0) {
      // Print the subset that sums to the target
      printf("Subset found: ");
      for (int i = 0; i < subsetSize; i++) {
```

```
            printf("%d ", subset[i]);
        }
        printf("\n");
        return;
    }

    if (index >= 0 && target >= 0) {
        // Include the current element in the subset
        subset[subsetSize] = arr[index];
        subsetSum(arr, subset, subsetSize + 1, index - 1, target - arr[index]);

        // Exclude the current element from the subset and move to the next element
        subsetSum(arr, subset, subsetSize, index - 1, target);
    }
}

int main() {
    int arr[] = {2, 3, 7, 8, 10};
    int n = sizeof(arr) / sizeof(arr[0]);
    int target = 10;
    int subset[MAX_SIZE];

    printf("Subsets that sum to %d:\n", target);
    subsetSum(arr, subset, 0, n - 1, target);

    return 0;
}
```

Output :

```
PS E:\Testing_Lang> cd "e:\Testing_Lang\" ; if ($?) { gcc test.c -o test } ; if ($?) { .\test }
Subsets that sum to 10:
Subset found: 10
Subset found: 8 2
Subset found: 7 3
PS E:\Testing_Lang>
```

Conclusion:

The Sum of Subset problem has been implemented. In this program, the subsetSum function recursively explores all possible subsets of the given set arr[] and checks whether each subset sums to the target value. If a subset is found whose sum equals the target, it is printed.

Experiment No. 11
15 puzzle problem
Date of Performance:
Date of Submission:

Experiment No. 11

Title: 15 Puzzle

Aim: To study and implement 15 puzzle problem

Objective: To introduce Backtracking and Branch-Bound methods

Theory:

The 15 puzzle problem is invented by sam loyd in 1878.

- In this problem there are 15 tiles, which are numbered from 0 – 15.
- The objective of this problem is to transform the arrangement of tiles from initial arrangement to a goal arrangement.
- The initial and goal arrangement is shown by following figure.

1	2	3	4
5	6	7	
9	10	12	8
11	13	14	15

Initial arrangement

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

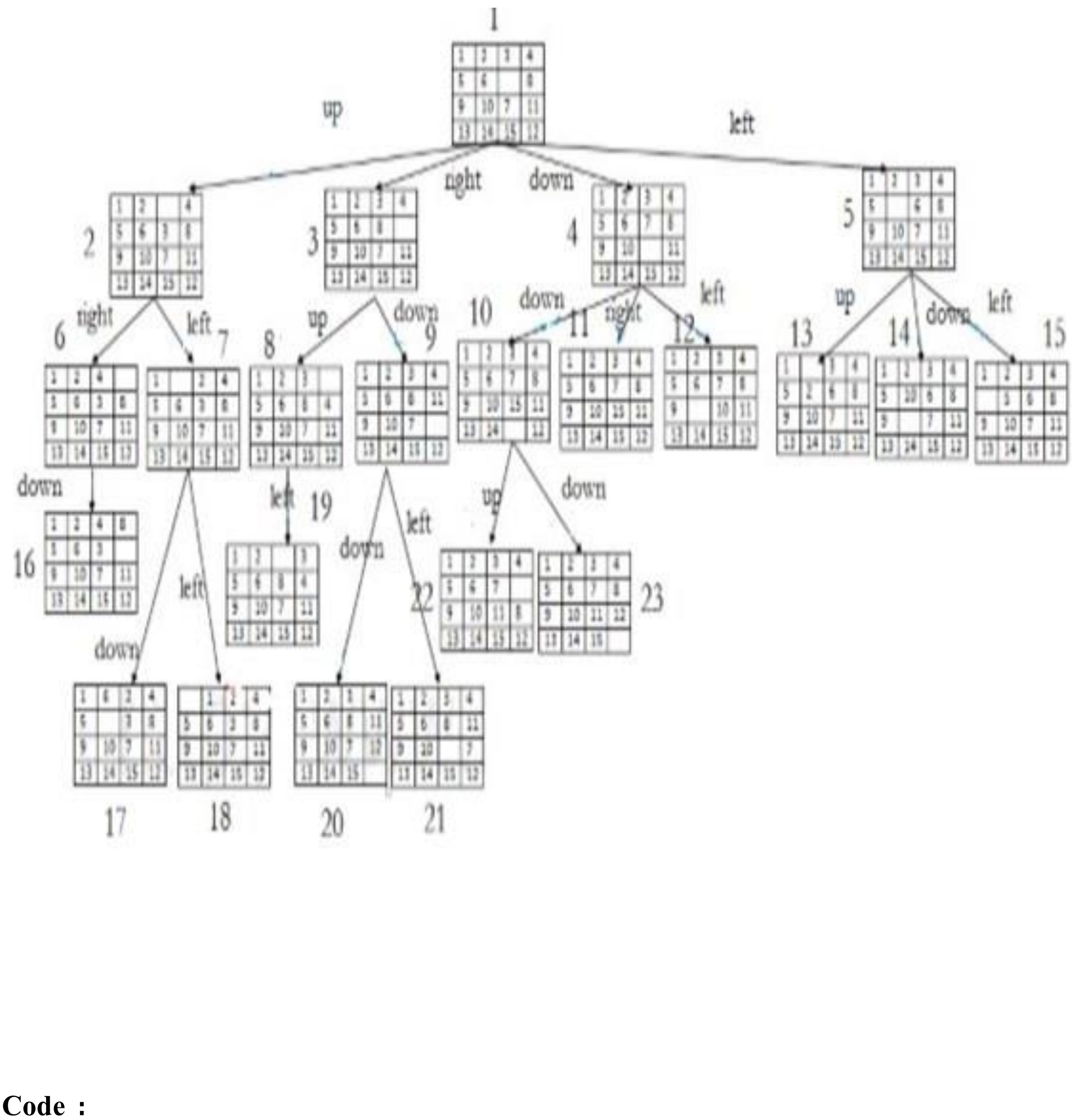
Final arrangement

Figure 12

- There is always an empty slot in the initial arrangement.
- The legal moves are the moves in which the tiles adjacent to ES are moved to either left, right, up or down.
- Each move creates a new arrangement in a tile.
- These arrangements are called as states of the puzzle.
- The initial arrangement is called as initial state and goal arrangement is called as goal state.
- The state space tree for 15 puzzle is very large because there can be 16! Different arrangements.
- A partial state space tree can be shown in figure.
- In state space tree, the nodes are numbered as per the level.
- Each next move is generated based on empty slot positions.
- Edges are label according to the direction in which the empty space moves.
- The root node becomes the E – node.
- The child node 2, 3, 4 and 5 of this E – node get generated.
- Out of which node 4 becomes an E – node. For this node the live nodes 10, 11, 12 gets generated.

- Then the node 10 becomes the E – node for which the child nodes 22 and 23 gets generated.
- Finally we get a goal state at node 23.
- We can decide which node to become an E – node based on estimation formula.

Example:



Code :

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

```
#define N 8

// Function to check if a queen can be placed at board[row][col]
bool isSafe(int board[N][N], int row, int col) {
    int i, j;

    // Check this row on the left side
    for (i = 0; i < col; i++) {
        if (board[row][i]) {
            return false;
        }
    }

    // Check upper diagonal on left side
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--) {
        if (board[i][j]) {
            return false;
        }
    }

    // Check lower diagonal on left side
    for (i = row, j = col; j >= 0 && i < N; i++, j--) {
        if (board[i][j]) {
            return false;
        }
    }

    return true;
}

// Recursive function to solve N-Queens problem using branch and bound
bool solveNQueensUtil(int board[N][N], int col) {
    // If all queens are placed then return true
    if (col >= N) {
        return true;
    }

    // Consider this column and try placing this queen in all rows one by one
    for (int i = 0; i < N; i++) {
```

```
        // Check if the queen can be placed on board[i][col]
        if (isSafe(board, i, col)) {
            // Place this queen in board[i][col]
            board[i][col] = 1;

            // Recur to place the rest of the queens
            if (solveNQueensUtil(board, col + 1)) {
                return true;
            }

            // If placing queen in board[i][col] doesn't lead to a solution then backtrack
            board[i][col] = 0; // Backtrack
        }
    }

    // If the queen cannot be placed in any row in this column, then return false
    return false;
}

// Function to solve N-Queens problem using branch and bound
void solveNQueens() {
    int board[N][N] = { {0, 0, 0, 0, 0, 0, 0, 0},
                        {0, 0, 0, 0, 0, 0, 0, 0},
                        {0, 0, 0, 0, 0, 0, 0, 0},
                        {0, 0, 0, 0, 0, 0, 0, 0},
                        {0, 0, 0, 0, 0, 0, 0, 0},
                        {0, 0, 0, 0, 0, 0, 0, 0},
                        {0, 0, 0, 0, 0, 0, 0, 0},
                        {0, 0, 0, 0, 0, 0, 0, 0} };

    if (solveNQueensUtil(board, 0) == false) {
        printf("Solution does not exist");
        return;
    }

    // Print the solution
    printf("Solution:\n");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
```

```
            printf(" %d ", board[i][j]);
        }
        printf("\n");
    }
}

int main() {
    solveNQueens();
    return 0;
}
```

Output :

```
PS E:\Testing_Lang> cd "e:\Testing_Lang\" ; if ($?) { gcc test.c -o test } ; if ($?) { .\test }
Solution:
 1  0  0  0  0  0  0  0
 0  0  0  0  0  0  1  0
 0  0  0  0  1  0  0  0
 0  0  0  0  0  0  0  1
 0  1  0  0  0  0  0  0
 0  0  0  1  0  0  0  0
 0  0  0  0  0  1  0  0
 0  0  1  0  0  0  0  0
PS E:\Testing_Lang>
```

Conclusion: The 15 Puzzle problem has been implemented.

Experiment No. 12
Naïve String matching
Date of Performance:
Date of Submission:

Experiment No. 12

Title: Naïve String matching

Aim: To study and implement Naïve string matching Algorithm

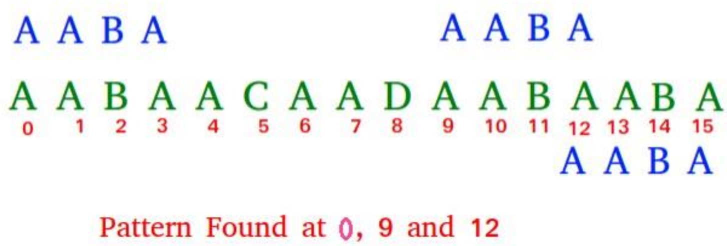
Objective: To introduce String matching methods

Theory:

The naïve approach tests all the possible placement of Pattern P [1.......m] relative to text T [1......n]. We try shift s = 0, 1.......n-m, successively and for each shift s. Compare T [s+1.......s+m] to P [1......m].

The naïve algorithm finds all valid shifts using a loop that checks the condition P [1.......m] = T [s+1.......s+m] for each of the n - m +1 possible value of s.

Example:



Algorithm:

THE NAIVE ALGORITHM

The naive algorithm finds all valid shifts using a loop that checks
the condition P[1….m]=T[s+1…. s+m] for each of the n-m+1
possible values of s.(P=pattern , T=text/string , s=shift)

NAIVE-STRING-MATCHER(T,P)

1) n = T.length
2) m = P.length
3) **for** s=0 to n-m
4) **if** P[1…m]==T[s+1….s+m]
5) printf" Pattern occurs with shift " s

Code :

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

// Function to perform naive string matching
void naiveStringMatch(char text[], char pattern[]) {
   int n = strlen(text);
   int m = strlen(pattern);
   int count = 0;

   // Iterate through each position of the text
   for (int i = 0; i <= n - m; i++) {
      int j;
      // Check if the pattern matches the substring starting at position i
      for (j = 0; j < m; j++) {
         if (text[i + j] != pattern[j])
            break; // mismatch found, break the loop
      }
      if (j == m) {
```

```
            printf("Pattern found at index %d\n", i);
            count++;
        }
    }

    if (count == 0)
        printf("Pattern not found in the text.\n");
}

int main() {
    char text[] = "AABAACAADAABAAABAA";
    char pattern[] = "AABA";

    printf("Text: %s\n", text);
    printf("Pattern: %s\n", pattern);

    printf("Occurrences of pattern in the text:\n");
    naiveStringMatch(text, pattern);

    return 0;
}
```

Output :

```
PS E:\Testing_Lang> cd "e:\Testing_Lang\" ; if ($?) { gcc test.c -o test } ; if ($?) { .\test }
Text: AABAACAADAABAAABAA
Pattern: AABA
Occurrences of pattern in the text:
Pattern found at index 0
Pattern found at index 9
Pattern found at index 13
PS E:\Testing_Lang>
```

Conclusion: Naïve string-matching algorithm has been successfully implemented. In this program, the naiveStringMatch function iterates through each position of the text and checks if the pattern matches the substring starting at that position. If a match is found, it prints the index where the match occurred.

The time complexity of the Naive String-Matching Algorithm depends on the length of the text n and the length of the pattern m. In the worst-case scenario, where the pattern matches at every position of the text, the time complexity is O((n-m+1)m). However, in typical cases, the algorithm tends to have a time complexity closer to O(nm) because the average number of comparisons made is proportional to the length of both the text and the pattern.