Design & Analysis of Algorithms Lab

(CSP350)

Practical File

B.TECH 3rd YEAR

SEMESTER : 5th

SESSION : 2024-2025

Name : Nidhi Pandey
System ID : 2022437668
Roll No : 2201010467
Section : CS-C

Submitted To :

Dr. Rani Astya

Associate Professor

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

SHARDA SCHOOL OF ENGINEERING & TECHNOLOGY

SHARDA UNIVERSITY, GREATER NOIDA

List of Experiments

Exp. No.	Aim of Experiment	Date	Signature
1.	Given two sorted arrays nums1 and nums2 of size m and n respectively, return the median of the two sorted arrays.	18-07-2024	
2.	Consider a library has received 1 million books and tomorrow is the inauguration ceremony of library. All the books are required to be arranged alphabetically in racks before the inauguration. The task has to be completed in least possible time. Using quick sort algorithm, implement the situation.	01-08-2024	
3.	Given 35 files containing stock value of each year for a particular share, arrange the values in ascending order using merge sort.	08-08-2024	
4.	Consider there is an array of integers ranging from 1 to 1000, and these elements are to be arranged in ascending order. Write a program to implement the situation using Counting Sort	22-08-2024	
5.	Consider John has recorded weight of 10,000 people and wants to arrange them in descending order. Write a program to implement the situation using Bucket Sort.	29-08-2024	
6.	Consider a scenario where sorting in lexographic order is required. Implement the situation using Radix sort.	12-09-2024	
7.	Given two strings, S1 and S2, the task is to find the length of the Longest Common Subsequence, i.e. longest subsequence present in both of the strings. Write a program to implement Longest Common Subsequences (LCS).	19-09-2024	
8.	We are given N items where each item has some weight and profit associated with it. We are also given a bag with capacity W, [i.e., the bag can hold at most W weight in it]. The target is to put the items into the bag such that the sum of profits associated with them is the maximum possible. Write a program to implement 0/1 Knapsack problem.	03-10-2024	
9.	Given two arrays weight[] and profit[] the weights and profit of N items, we need to put these items in a knapsack of capacity W to get the maximum total value in the knapsack. Write a program to implement fractional Knapsack problem.	10-10-2024	
10.	From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm.	17-10-2024	
11.	The problem is to find the shortest distances between every pair of vertices in a given edge-weighted directed graph. The graph is represented as an adjacency matrix of size n*n. Matrix[i][j] denotes the weight of the edge from i to j. If Matrix[i][j]=-1, it means there is no edge from i to j. Do it in place.	24-10-2024	
12.	Given an integer array nums of unique elements, return all possible subsets (the power set). The solution set must not contain duplicate subsets. Return the solution in any order.	7-11-2024	
13.	WAP to demonstrate concept of Red Black Tree.	14-11-2024	

14.	Given a text of length N txt[0..N-1] and a pattern of length M pat[0..M-1], write a function search(char pat[], char txt[]) that prints all occurrences of pat[] in txt[]. You may assume that N > M. Demonstrate Naïve string matching Algorithm.	14-11-2024	

Experiment No – 1

Aim of Experiment 1: Given two sorted arrays nums1 and nums2 of size m and n respectively, return the median of the two sorted arrays.

Theory:

The median is a measure of central tendency that represents the middle value in a sorted list of numbers. The binary search runs on the smaller of the two arrays, and each step of the binary search takes constant time to check the partition, hence, time Complexity is **O log(n).**

Code:

```
#include <stdio.h>

int maximum(int a, int b) {
   return a > b ? a : b;
}

int minimum(int a, int b) {
   return a < b ? a : b;
}

double findMedianSortedArrays(int* a, int n, int* b, int m) {
   int min_index = 0, max_index = n, i, j, median;

   while (min_index <= max_index) {
      i = (min_index + max_index) / 2;
      j = ((n + m + 1) / 2) - i;

      if (j < 0) {
         max_index = i - 1;
         continue;
      }

      if (i < n && j > 0 && b[j - 1] > a[i]) {
         min_index = i + 1;
      } else if (i > 0 && j < m && b[j] < a[i - 1]) {
         max_index = i - 1;
      } else {
         if (i == 0) {
            median = b[j - 1];
         } else if (j == 0) {
            median = a[i - 1];
         } else {
            median = maximum(a[i - 1], b[j - 1]);
         }
         break;
      }
   }
```

```
    if ((n + m) % 2 == 1) {
        return (double)median;
    }

    if (i == n) {
        return (median + b[j]) / 2.0;
    }

    if (j == m) {
        return (median + a[i]) / 2.0;
    }

    return (median + minimum(a[i], b[j])) / 2.0;
}

int main() {
    int a[] = { 1, 3, 8 };
    int b[] = { 7, 9, 10, 11 };
    int n = sizeof(a) / sizeof(int);
    int m = sizeof(b) / sizeof(int);

    if (n < m) {
        printf("The median is : %.2f\n", findMedianSortedArrays(a, n, b, m));
    } else {
        printf("The median is : %.2f\n", findMedianSortedArrays(b, m, a, n));
    }

    return 0;
}
```

Output:

```
 /tmp/ZhEjEQdtJX.o
The median is : 8

=== Code Execution Successful ===
```

Experiment No – 2

Aim of Experiment 2: Consider a library has received 1 million books and tomorrow is the inauguration ceremony of library. All the books are required to be arranged alphabetically in racks before the inauguration. The task has to be completed in least possible time. Using quick sort algorithm, implement the situation.

Theory:

The **Quick Sort** algorithm is a sorting algorithm based on the **divide-and-conquer** strategy. It works by selecting a **pivot** element, partitioning the array around this pivot so that elements less than the pivot come before it and elements greater than the pivot come after it, and then recursively applying this process to the sub-arrays. The time complexity of Quick Sort is O(nlogn) on average and O(n^2) in the worst case.

Code:

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

void swap(char** a, char** b) {
    char* temp = *a;
    *a = *b;
    *b = temp;
}

int partition(char* books[], int low, int high) {
    char* pivot = books[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (strcmp(books[j], pivot) <= 0) {
            i++;
            swap(&books[i], &books[j]);
        }
    }
    swap(&books[i + 1], &books[high]);
    return i + 1;
}

void quickSort(char* books[], int low, int high) {
    if (low < high) {
        int pi = partition(books, low, high);

        quickSort(books, low, pi - 1);
        quickSort(books, pi + 1, high);
    }
```

```
}

int main() {
    char* books[] = {
        "A Tale of Two Cities", "Moby Dick", "War and Peace", "Great Expectations", "Ulysses"
        // Add more book titles here, up to 1 million if needed
    };
    int n = sizeof(books) / sizeof(books[0]);

    quickSort(books, 0, n - 1);

    printf("Books sorted alphabetically:\n");
    for (int i = 0; i < n; i++) {
        printf("%s\n", books[i]);
    }

    return 0;
}
```

Output:

```
/tmp/hbYCHZGzAj.o
Books sorted alphabetically:
A Tale of Two Cities
Great Expectations
Moby Dick
Ulysses
War and Peace


=== Code Execution Successful ===
```

Experiment No – 3

Aim of Experiment 3: Given 35 files containing stock value of each year for a particular share, arrange the values in ascending order using merge sort.

Theory:

Merge Sort is a classic sorting algorithm based on the **divide-and-conquer** strategy. It works by recursively dividing the array into two halves, sorting each half, and then merging the sorted halves. The time complexity of Merge Sort is O(nlogn) in all cases (best, average, and worst).

Code:

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

void merge(int* stockValues, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Create temporary arrays
    int* leftArray = (int*)malloc(n1 * sizeof(int));
    int* rightArray = (int*)malloc(n2 * sizeof(int));

    // Copy data to temporary arrays
    for (int i = 0; i < n1; i++)
        leftArray[i] = stockValues[left + i];
    for (int j = 0; j < n2; j++)
        rightArray[j] = stockValues[mid + 1 + j];

    // Merge the temporary arrays back into stockValues
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (leftArray[i] <= rightArray[j]) {
            stockValues[k] = leftArray[i];
            i++;
        } else {
            stockValues[k] = rightArray[j];
            j++;
        }
        k++;
    }

    // Copy any remaining elements of leftArray
    while (i < n1) {
        stockValues[k] = leftArray[i];
        i++;
        k++;
    }
```

```
    // Copy any remaining elements of rightArray
    while (j < n2) {
        stockValues[k] = rightArray[j];
        j++;
        k++;
    }

    // Free the temporary arrays
    free(leftArray);
    free(rightArray);
}

void mergeSort(int* stockValues, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(stockValues, left, mid);
        mergeSort(stockValues, mid + 1, right);
        merge(stockValues, left, mid, right);
    }
}

int main() {
    int stockValues[] = {85, 72, 95, 60, 78, 88, 92, 64, 99, 73, 81, 69, 77, 91, 62, 84};
    int n = sizeof(stockValues) / sizeof(stockValues[0]);

    mergeSort(stockValues, 0, n - 1);

    printf("Stock values sorted in ascending order:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", stockValues[i]);
    }
    printf("\n");

    return 0;
}
```

Output:

```
/tmp/yJ5uLaRsxy.o
Stock values sorted in ascending order:
60 62 64 69 72 73 77 78 81 84 85 88 91 92 95 99


=== Code Execution Successful ===
```

Experiment No – 4

Aim of Experiment 4: Consider there is an array of integers ranging from 1 to 1000, and these elements are to be arranged in ascending order. Write a program to implement the situation using Counting Sort.

Theory:

Counting Sort is a non-comparative, integer sorting algorithm that works by counting the occurrences of each distinct element in the input. It then uses this count to determine the positions of each element in the sorted output. The time complexity of Counting Sort is O(n+k), where n is the number of elements and k is the range of the input.

Code:

```
#include <stdio.h>

void counting_sort(int arr[], int n) {
    int max_val = arr[0];
    for (int i = 1; i < n; i++) {
        if (arr[i] > max_val) {
            max_val = arr[i];
        }
    }

    // Create count array with size max_val + 1
    int count[max_val + 1];
    int sorted_arr[n];

    for (int i = 0; i <= max_val; i++) {
        count[i] = 0;
    }

    for (int i = 0; i < n; i++) {
        count[arr[i]]++;
    }

    int k = 0;
    for (int i = 0; i <= max_val; i++) {
        while (count[i]-- > 0) {
            sorted_arr[k++] = i;
        }
    }

    for (int i = 0; i < n; i++) {
        arr[i] = sorted_arr[i];
    }
}

int main() {
```

```
    int arr[] = {345, 123, 789, 432, 645, 234, 876, 563, 129, 999, 654, 231, 100, 432, 678};
    int n = sizeof(arr) / sizeof(arr[0]);

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

    counting_sort(arr, n);

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

    return 0;
}
```

Output:

```
/tmp/KXHi4MYnCH.o
Original array: 345 123 789 432 645 234 876 563 129 999 654 231 100 432
    678
Sorted array: 100 123 129 231 234 345 432 432 563 645 654 678 789 876
    999


=== Code Execution Successful ===
```

Experiment No – 5

Aim of Experiment 5: Consider John has recorded weight of 10,000 people and wants to arrange them in descending order. Write a program to implement the situation using Bucket Sort.

Theory:

Bucket Sort is a distribution-based sorting algorithm that works by dividing the elements of an array into several "buckets." Each bucket is then sorted individually, either using a different sorting algorithm or recursively applying Bucket Sort. Finally, the sorted buckets are concatenated to produce the final sorted array. The time complexity of Bucket Sort is O(n+k) on average, where k is the number of buckets (depends on data distribution). In the worst case, it can be O(n^2).

Code:

```
#include <stdio.h>
#include <stdlib.h>
#define BUCKETS 10

typedef struct Node {
   float value;
   struct Node* next;
} Node;

void insert(Node** bucket, float value) {
   Node* new_node = (Node*)malloc(sizeof(Node));
   new_node->value = value;
   new_node->next = NULL;
   if (*bucket == NULL) {
      *bucket = new_node;
   } else {
      Node* temp = *bucket;
      Node* prev = NULL;
      while (temp != NULL && temp->value > value) {
         prev = temp;
         temp = temp->next;   }
      if (prev == NULL) {
         new_node->next = *bucket;
         *bucket = new_node;
      } else {
         new_node->next = prev->next;
         prev->next = new_node;
      }  } }
void bucketSort(float array[], int n) {
   Node* buckets[BUCKETS];
   for (int i = 0; i < BUCKETS; i++) {
      buckets[i] = NULL;
   }
   float max_value = array[0];
```

```
    for (int i = 1; i < n; i++) {
        if (array[i] > max_value) {
            max_value = array[i];
        }  }
    for (int i = 0; i < n; i++) {
        int bucket_index = (int)(BUCKETS * array[i] / (max_value + 1));
        insert(&buckets[bucket_index], array[i]);  }
    int index = 0;
   for (int i = 0; i < BUCKETS; i++) {
        Node* temp = buckets[i];
        while (temp != NULL) {
            array[index++] = temp->value;
            temp = temp->next;
        }   }
    for (int i = 0; i < n / 2; i++) {
        float temp = array[i];
        array[i] = array[n - i - 1];
        array[n - i - 1] = temp;
    }
    for (int i = 0; i < BUCKETS; i++) {
        Node* temp = buckets[i];
        while (temp != NULL) {
            Node* to_free = temp;
            temp = temp->next;
            free(to_free);
        } } }
int main() {
    int n = 10000;
    float weights[10000];
    for (int i = 0; i < n; i++) {
        weights[i] = (float)(rand() % 10000) / 100;    }
    bucketSort(weights, n);
    printf("Top 20 sorted weights (descending):\n");
    for (int i = 0; i < 20; i++) {
        printf("%.2f\n", weights[i]);    }
    return 0;
}
```

Output:

```
/tmp/4rj0IWB1vA.o
Top 20 sorted weights (descending):
90.90  90.90  90.90  90.91  90.91  90.93  90.94  90.95  90.95  90.97
    90.98  91.00  91.00  91.01  91.03  91.07  91.09  91.11  91.11  91
    .12

=== Code Execution Successful ===
```

Experiment No – 6

Aim of Experiment 6: Consider a scenario where sorting in lexographic order is required Implement the situation using Radix sort.

Theory:

Radix Sort is a non-comparative integer sorting algorithm that sorts numbers by processing individual digits. It works by grouping the numbers based on their digits and using a stable sorting algorithm (such as Counting Sort) to sort them by each digit, starting from the least significant digit to the most significant digit. The time complexity of Radix Sort is $O(d \cdot (n+k))$, where d is the number of digits in the largest number and k is the base.

Code:

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

int find_max_length(char** arr, int n) {
    int max_len = 0;
    for (int i = 0; i < n; i++) {
        if (strlen(arr[i]) > max_len) {
            max_len = strlen(arr[i]);
        }    }
    return max_len;
}
void counting_sort(char** arr, int n, int digit) {
    int count[256] = {0}; // count array for ASCII values
    char** temp = (char**)malloc(sizeof(char*) * n);
    for (int i = 0; i < n; i++) {
        if (strlen(arr[i]) > digit) {
            count[(int)arr[i][digit]]++;
        } else {
            count[0]++;
        }
    }
    for (int i = 1; i < 256; i++) {
        count[i] += count[i - 1];
    }
    for (int i = n - 1; i >= 0; i--) {
        if (strlen(arr[i]) > digit) {
            temp[count[(int)arr[i][digit]] - 1] = arr[i];
            count[(int)arr[i][digit]]--;
        } else {
            temp[count[0] - 1] = arr[i];
            count[0]--;
        }
    }
```

```
    for (int i = 0; i < n; i++) {
        arr[i] = temp[i];
    }
    free(temp);
}
void radix_sort(char** arr, int n) {
    int max_len = find_max_length(arr, n);
    for (int i = max_len - 1; i >= 0; i--) {
        counting_sort(arr, n, i);
    }
}
int main() {
    int n = 5;
    char** arr = (char**)malloc(sizeof(char*) * n);
    arr[0] = "apple";
    arr[1] = "banana";
    arr[2] = "cherry";
    arr[3] = "date";
    arr[4] = "elderberry";
    printf("Original array:\n");
    for (int i = 0; i < n; i++) {
        printf("%s\n", arr[i]);
    }
   radix_sort(arr, n);
   printf("Sorted array:\n");
    for (int i = 0; i < n; i++) {
        printf("%s\n", arr[i]);
    }
    free(arr);
    return 0;
}
```

Output:

```
/tmp/MooD27siYi.o
Original array:
apple banana cherry date elderberry
Sorted array:
apple banana cherry date elderberry

=== Code Execution Successful ===
```

Experiment No – 7

Aim of Experiment 7: Given two strings, S1 and S2, the task is to find the length of the Longest Common Subsequence, i.e. longest subsequence present in both of the strings. Write a program to implement Longest Common Subsequences (LCS).

Theory:

A subsequence is a sequence derived from another sequence by deleting some elements without changing the order of the remaining elements. The time complexity of LCS is O(m X n) in all cases, where m and n are the lengths of the two strings.

Code:

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

int lcs(char *S1, char *S2, int m, int n) {
    int L[m+1][n+1];
    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            if (i == 0 || j == 0)
                L[i][j] = 0;
            else if (S1[i - 1] == S2[j - 1])
                L[i][j] = L[i - 1][j - 1] + 1;
            else
                L[i][j] = (L[i - 1][j] > L[i][j - 1]) ? L[i - 1][j] : L[i][j - 1];
        }
    }
    return L[m][n];
}

int main() {
    char S1[100], S2[100];
    printf("Enter the first string: ");
    scanf("%s", S1);
    printf("Enter the second string: ");
```

```
    scanf("%s", S2);

    int m = strlen(S1);
    int n = strlen(S2);

    printf("Length of Longest Common Subsequence: %d\n", lcs(S1, S2, m, n));
    return 0;
}
```

Output:

```
/tmp/XoFpeT7eDc.o
Enter the first string: abcd
Enter the second string: acbd
Length of Longest Common Subsequence: 3


=== Code Execution Successful ===
```

Experiment No – 8

Aim of Experiment 8: We are given N items where each item has some weight and profit associated with it. We are also given a bag with capacity W, [i.e., the bag can hold at most W weight in it]. The target is to put the items into the bag such that the sum of profits associated with them is the maximum possible. Write a program to implement 0/1 Knapsack problem.

Theory:

The dynamic programming approach involves creating a two-dimensional table to store the maximum value achievable with a certain weight limit for each item. By iterating through the items and weight capacities, you can fill this table based on whether to include each item or not, ultimately deriving the solution from the last entry of the table. The time complexity is O(m X n) in all cases.

Code:

```
#include <stdio.h>

int knapsack(int weights[], int profits[], int W, int N) {
    int dp[N + 1][W + 1];
    for (int i = 0; i <= N; i++) {
        for (int w = 0; w <= W; w++) {
            dp[i][w] = 0;
        }
    }
    for (int i = 1; i <= N; i++) {
        for (int w = 1; w <= W; w++) {
            if (weights[i - 1] <= w) {
                dp[i][w] = (dp[i - 1][w] > dp[i - 1][w - weights[i - 1]] + profits[i - 1])
                        ? dp[i - 1][w]
                        : dp[i - 1][w - weights[i - 1]] + profits[i - 1];
            } else {
                dp[i][w] = dp[i - 1][w];
            }
        }
    }
    return dp[N][W];
}
int main() {
    int N;
    int W;
    printf("Enter the number of items: ");
    scanf("%d", &N);
    printf("Enter the capacity of the knapsack: ");
    scanf("%d", &W);
    int weights[N];
    int profits[N];
    printf("Enter the weights of the items:\n");
    for (int i = 0; i < N; i++) {
```

```
        scanf("%d", &weights[i]);
    }
    printf("Enter the profits of the items:\n");
    for (int i = 0; i < N; i++) {
        scanf("%d", &profits[i]);
    }
    int maxProfit = knapsack(weights, profits, W, N);
    printf("The maximum profit that can be achieved is: %d\n", maxProfit);
    return 0;
}
```

Output:

```
/tmp/R4u1L8rGkI.o
Enter the number of items: 3
Enter the capacity of the knapsack: 12
Enter the weights of the items:
7
2
4
Enter the profits of the items:
4
2
6
The maximum profit that can be achieved is: 10


=== Code Execution Successful ===
```

Experiment No – 9

Aim of Experiment 9: Given two arrays weight[] and profit[] the weights and profit of N items, we need to put these items in a knapsack of capacity W to get the maximum total value in the knapsack. Write a program to implement fractional Knapsack problem.

Theory:

The solution involves calculating the value-to-weight ratio for each item, sorting the items in descending order based on this ratio, and then using a greedy approach to fill the knapsack. As you iterate through the sorted items, you add the entire item if it fits within the remaining capacity, or take as much as possible of an item if it exceeds the remaining weight limit.

Code:

```
#include <stdio.h>
struct Item {
   int weight;
   int profit;
   float ratio;
};
void sortItems(struct Item items[], int n) {
   for (int i = 0; i < n - 1; i++) {
      for (int j = i + 1; j < n; j++) {
         if (items[i].ratio < items[j].ratio) {
            // Swap items[i] and items[j]
            struct Item temp = items[i];
            items[i] = items[j];
            items[j] = temp;
} } } }
float fractionalKnapsack(struct Item items[], int n, int capacity) {
   sortItems(items, n);
   float totalValue = 0.0;
   int currentWeight = 0;
   for (int i = 0; i < n; i++) {
      if (currentWeight + items[i].weight <= capacity) {
         // If we can include the full item
         currentWeight += items[i].weight;
         totalValue += items[i].profit;
      } else {
         int remainingCapacity = capacity - currentWeight;
         totalValue += items[i].profit * ((float)remainingCapacity / items[i].weight);
         break;
      }
   }
   return totalValue;
}
int main() {
   int n, capacity;
   printf("Enter number of items: ");
```

```
    scanf("%d", &n);
    printf("Enter capacity of the knapsack: ");
    scanf("%d", &capacity);
    struct Item items[n];
    for (int i = 0; i < n; i++) {
        printf("Enter weight and profit of item %d: ", i + 1);
        scanf("%d %d", &items[i].weight, &items[i].profit);
        items[i].ratio = (float)items[i].profit / items[i].weight;
    }
    float maxValue = fractionalKnapsack(items, n, capacity);
    printf("Maximum value in Knapsack = %.2f\n", maxValue);
    return 0;
}
```

Output:

```
/tmp/xYvU1VXxK3.o
Enter number of items: 4
Enter capacity of the knapsack: 50
Enter weight and profit of item 1: 12 4
Enter weight and profit of item 2: 18 6
Enter weight and profit of item 3: 16 5
Enter weight and profit of item 4: 11 7
Maximum value in Knapsack = 19.81


=== Code Execution Successful ===
```

Experiment No – 10

Aim of Experiment 10: From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm.

Theory:

The Dijkstra's algorithm begins by initializing distances, assigning zero to the source vertex and infinity to all others. It uses a priority queue to repeatedly select the unvisited vertex with the smallest tentative distance and explores its neighbors to update their distances if a shorter path is found. This process continues until all vertices are marked as visited. The time complexity of Dijkstra's Algorithm is O(V^2) with a simple adjacency matrix and O((V+E)logV) with a priority queue (using Fibonacci or binary heaps), where V is the number of vertices and E is the number of edges.

Code:

```
#include <stdio.h>
#include <limits.h>
#define V 100
int minDistance(int dist[], int processed[], int vertices) {
   int min = INT_MAX, min_index;
   for (int v = 0; v < vertices; v++)
      if (processed[v] == 0 && dist[v] <= min) {
         min = dist[v];
         min_index = v;
      }
   return min_index;
}
void printSolution(int dist[], int vertices) {
   printf("Vertex \t Distance from Source\n");
   for (int i = 0; i < vertices; i++)
      printf("%d \t\t %d\n", i, dist[i]);
}
void dijkstra(int graph[V][V], int src, int vertices) {
   int dist[V];  // Output array, dist[i] holds the shortest distance from src to i
   int processed[V];  // processed[i] will be 1 if vertex i has been processed
   for (int i = 0; i < vertices; i++) {
      dist[i] = INT_MAX;
      processed[i] = 0;
   }
   dist[src] = 0;
   for (int count = 0; count < vertices - 1; count++) {
      int u = minDistance(dist, processed, vertices);
      processed[u] = 1;
      for (int v = 0; v < vertices; v++)
         if (!processed[v] && graph[u][v] && dist[u] != INT_MAX && dist[u] + graph[u][v] < dist[v])
            dist[v] = dist[u] + graph[u][v];
   }
```

```
    printSolution(dist, vertices);
}
int main() {
    int vertices;
    printf("Enter number of vertices: ");
    scanf("%d", &vertices);
    int graph[V][V];
    printf("Enter the adjacency matrix (use 0 for no direct edge):\n");
    for (int i = 0; i < vertices; i++) {
        for (int j = 0; j < vertices; j++) {
            scanf("%d", &graph[i][j]);
        }
    }
    int src;
    printf("Enter the source vertex: ");
    scanf("%d", &src);
    dijkstra(graph, src, vertices);
    return 0;
}
```

Output:

```
/tmp/G1C5eAY5hN.o
Enter number of vertices: 5
Enter the adjacency matrix (use 0 for no direct edge):
0 2 8 1 3
7 0 4 0 0
8 9 0 5 7
7 0 8 2 3
1 2 4 6 8
Enter the source vertex: 0
Vertex   Distance from Source
0        0
1        2
2        6
3        1
4        3


=== Code Execution Successful ===
```

Experiment No – 11

Aim of Experiment 11: The problem is to find the shortest distances between every pair of vertices in a given edge-weighted directed graph. The graph is represented as an adjacency matrix of size n*n. Matrix[i][j] denotes the weight of the edge from I to j. If Matrix[i][j]=-1, it means there is no edge from I to j. Do it in place.

Theory:

In an adjacency matrix, the rows and columns represent the vertices of the graph, and the entries in the matrix indicate whether pairs of vertices are adjacent.

Code:

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

void floydWarshall(int graph[MAX][MAX], int n) {
    for (int k = 0; k < n; k++) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (graph[i][k] != INF && graph[k][j] != INF) { //
                    if (graph[i][j] > graph[i][k] + graph[k][j]) {
                        graph[i][j] = graph[i][k] + graph[k][j]; //
                    }}}}}

void printSolution(int graph[MAX][MAX], int n) {
    printf("Shortest distances between every pair of vertices:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (graph[i][j] == INF) {
                printf("INF ");
            } else {
                printf("%d ", graph[i][j]);
```

```
        } }
        printf("\n");
}}
int main() {
    int n;
    printf("Enter number of vertices: ");
    scanf("%d", &n);
    int graph[MAX][MAX];
    printf("Enter the adjacency matrix (-1 for no edge):\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &graph[i][j])
            if (graph[i][j] == -1 && i != j) {
                graph[i][j] = INF;
            } } }
    floydWarshall(graph, n);
    printSolution(graph, n);
    return 0;
}
```

Output:

```
/tmp/9dD000apjE.o
Enter number of vertices: 4
Enter the adjacency matrix (-1 for no edge):
0 3 -1 7
-1 0 -1 2
-1 1 0 -1
-1 -1 4 0
Shortest distances between every pair of vertices:
0 3 9 5
INF 0 6 2
INF 1 0 3
INF 5 4 0


=== Code Execution Successful ===
```

Experiment No – 12

Aim of Experiment 12: Given an integer array nums of unique elements, return all possible subsets (the power set). The solution set must not contain duplicate subsets. Return the solution in any order.

Theory:

The **Power Set** of a set is the collection of all possible subsets of that set, including the empty set and the set itself. For a given set with n elements, the power set contains 2^n subsets.

Code:

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


void generateSubsets(int* nums, int numsSize, int** subsets, int* subset, int index, int* returnSize, int** returnColumnSizes) {
    subsets[*returnSize] = (int*)malloc(index * sizeof(int));
    for (int i = 0; i < index; i++)
        subsets[*returnSize][i] = subset[i];
    (*returnColumnSizes)[*returnSize] = index;
    (*returnSize)++;


    for (int i = index == 0 ? 0 : subset[index - 1] + 1; i < numsSize; i++) {
        subset[index] = nums[i];
        generateSubsets(nums, numsSize, subsets, subset, index + 1, returnSize, returnColumnSizes);
    }
}


int** subsets(int* nums, int numsSize, int* returnSize, int** returnColumnSizes) {
    int maxSize = 1 << numsSize;
    int** result = (int**)malloc(maxSize * sizeof(int*));
    *returnColumnSizes = (int*)malloc(maxSize * sizeof(int));
    int* subset = (int*)malloc(numsSize * sizeof(int));
    *returnSize = 0;
    generateSubsets(nums, numsSize, result, subset, 0, returnSize, returnColumnSizes);
    free(subset);
```

```
    return result;
}

int main() {
    int nums[] = {1, 2, 3};
    int numsSize = sizeof(nums) / sizeof(nums[0]);
    int returnSize;
    int* returnColumnSizes;
    int** result = subsets(nums, numsSize, &returnSize, &returnColumnSizes);

    for (int i = 0; i < returnSize; i++) {
        printf("[");
        for (int j = 0; j < returnColumnSizes[i]; j++) {
            printf("%d", result[i][j]);
            if (j < returnColumnSizes[i] - 1)
                printf(", ");
        }
        printf("]\n");
        free(result[i]);
    }

    free(result);
    free(returnColumnSizes);
    return 0;
}
```

Output:

```
/tmp/9uiDXuHqF7.o
[]
[1]
[1, 3]
[2]
[3]


=== Code Execution Successful ===
```

Experiment No - 13

Aim of Experiment 13 : WAP to demonstrate concept of Red Black Tree.

A **Red-Black Tree** is a self-balancing binary search tree where each node contains an extra bit for denoting the color of the node, either **RED** or **BLACK**. The primary goal of this tree structure is to maintain balance during insertions and deletions, ensuring that the tree's height is kept as small as possible, thus optimizing search time.

Theory:

Properties of a Red-Black Tree

1. **Node Color**: Each node is colored either RED or BLACK.
2. **Root Property**: The root of the tree is always BLACK.
3. **Red-Red Violation**: No two consecutive RED nodes are allowed (i.e., a RED node cannot have a RED parent).
4. **Black-Height Property**: Every path from a node to its descendant NULL pointers must contain the same number of BLACK nodes.
5. **Leaf Nodes**: All leaf nodes (NULL) are considered BLACK.

Time Complexity

Operation	Best	Average	Worst
Search	O(1)	O (log n)	O (log n)
Insertion	O (log n)	O (log n)	O (log n)
Deletion	O (log n)	O (log n)	O (log n)

6.
7.
8.
9.

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
typedef enum { RED, BLACK } Color;
typedef struct Node {
    int data;
    Color color;
    struct Node *left, *right, *parent;
} Node;
Node* createNode(int data) {
```

```
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->color = RED;
    newNode->left = newNode->right = newNode->parent = NULL;
    return newNode;
}
void leftRotate(Node** root, Node* x) {
    Node* y = x->right;
    x->right = y->left;
    if (y->left != NULL)
        y->left->parent = x;
    y->parent = x->parent;
    if (x->parent == NULL)
        *root = y;
    else if (x == x->parent->left)
        x->parent->left = y;
    else
        x->parent->right = y;
    y->left = x;
    x->parent = y;
}
void rightRotate(Node** root, Node* x) {
    Node* y = x->left;
    x->left = y->right;
    if (y->right != NULL)
        y->right->parent = x;
    y->parent = x->parent;
    if (x->parent == NULL)
        *root = y;
    else if (x == x->parent->right)
        x->parent->right = y;
    else
        x->parent->left = y;
```

```
    y->right = x;
    x->parent = y;
}
void fixViolation(Node** root, Node* z) {
    while (z != *root && z->parent->color == RED) {
        Node* grandparent = z->parent->parent;
        if (z->parent == grandparent->left) {
            Node* uncle = grandparent->right;
            if (uncle != NULL && uncle->color == RED) {
                z->parent->color = BLACK;
                uncle->color = BLACK;
                grandparent->color = RED;
                z = grandparent;
            } else {
                if (z == z->parent->right) {
                    z = z->parent;
                    leftRotate(root, z);
                }
                z->parent->color = BLACK;
                grandparent->color = RED;
                rightRotate(root, grandparent);
            }
        } else {
            Node* uncle = grandparent->left;
            if (uncle != NULL && uncle->color == RED) {
                z->parent->color = BLACK;
                uncle->color = BLACK;
                grandparent->color = RED;
                z = grandparent;
            } else {
                if (z == z->parent->left) {
                    z = z->parent;
                    rightRotate(root, z);
```

```
            }
            z->parent->color = BLACK;
            grandparent->color = RED;
            leftRotate(root, grandparent);
        }
    }
  }
  (*root)->color = BLACK;
}
void insert(Node** root, int data) {
  Node* newNode = createNode(data);
  Node* y = NULL;
  Node* x = *root;
  while (x != NULL) {
    y = x;
    if (data < x->data)
      x = x->left;
    else
      x = x->right;
  }
  newNode->parent = y;
  if (y == NULL)
    *root = newNode;
  else if (data < y->data)
    y->left = newNode;
  else
    y->right = newNode;
  fixViolation(root, newNode);
}
void printSpaces(int count) {
  for (int i = 0; i < count; i++) {
    printf(" ");
  }
```

```
}
void printTree(Node* root, int space) {
    if (root == NULL)
        return;
    space += 10;
    printTree(root->right, space);
    printf("\n");
    printSpaces(space - 10);
    printf("%d (%s)\n", root->data, (root->color == RED) ? "RED" : "BLACK");
    printTree(root->left, space);
}
int main() {
    Node* root = NULL;
    int arr[] = {10, 20, 30, 15, 25, 5, 1};
    int n = sizeof(arr) / sizeof(arr[0]);
    for (int i = 0; i < n; i++) {
        insert(&root, arr[i]);
    }
    printf("Tree-like structure of the Red-Black Tree:\n");
    printTree(root, 0);
    return 0;
}
```

Output-

```
Tree-like structure of the Red-Black Tree:
          30 (BLACK)
                    25 (RED)
20 (BLACK)
                    15 (BLACK)
          10 (RED)
                    5 (BLACK)
                              1 (RED)
```

Experiment No – 14

Aim of Experiment 14: Given a text of length N txt[0..N-1] and a pattern of length M pat[0..M-1], write a function search(char pat[], char txt[]) that prints all occurrences of pat[] in txt[]. You may assume that N > M. Demonstrate Naïve string matching Algorithm.

Theory:

The **Naive String Matching Algorithm** is a method for finding all occurrences of a pattern string within a larger text string. This algorithm checks for a match by comparing the pattern to every possible substring of the text. The time complexity of the Naive String Matching Algorithm is O(m·n), where m is the length of the pattern and n is the length of the text.

Code:

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

void search(char pat[], char txt[]) {
    int N = strlen(txt);
    int M = strlen(pat);

    for (int i = 0; i <= N - M; i++) {
        int j;
        for (j = 0; j < M; j++)
            if (txt[i + j] != pat[j])
                break;
        if (j == M)
            printf("Pattern found at index %d\n", i);
    }
}

int main() {
    char txt[] = "ABABDABACDABABCABAB";
    char pat[] = "ABABCABAB";
    search(pat, txt);
    return 0;
```

}

Output:

```
/tmp/2OcatWYoSo.o
Pattern found at index 10


=== Code Execution Successful ===
```