Name: **Avneet Singh**

Roll Number: **2021UAC1815**

Sem: **3rd**

Subject: **Design and Analysis of Algorithms**

Course Code: **CACSC06**

Course Outcomes:

After completing this course, students will be able to:

1. To be able to analyse a problem in terms of processing steps, time and space complexity.

2. To be able to design and implement the algorithms for any given application.

3. To be able to develop software applications using various programming languages in collaborative groups.

4. To apply the principles in solving problems encountered in career or real-life situations.

TABLE OF CONTENT

EXPERIMENT 1

Problem:
Doctor Johnson has collected the blood pressure rate of a tuberculosis infected patient. He has stored the collected data for observing the blood pressure behaviour of a given patient. During the operation of the patient one of the doctors has asked Doctor Johnson to tell the maximum and minimum blood pressure of the patient. So, this is the routine work of Doctor Johnson. So, perform the following sorting algorithms-Merge sort, Quick sort, Bubble sort, insertion sort and selection sort.

Objective: To work on different sorting algorithms to find the min and max from an array

Merge Sort

Overview:
Merge Sort is a divide and conquer based sorting algorithm where we divide our array into two parts which are sorted individually and then merged to form a final sorted array using recursion.

Algo:
merge sort (array, beg, end)

If beg<end:

set mid = (beg+end)/2

merge sort(arr,beg,mid)

merge sort(arr,mid+1,end)

merge(arr,beg,mid,end)

Complexity: O(n*log(n))

Code:

```
#include<iostream>
using namespace std;

//1. merge sort
void merge(int* A, int s, int m, int e)
{
    // og positions of variables
    // for later use
    int a = s, b = m + 1;

    // making a new array
    int arr[e - s + 1];
    int c = 0;
```

```
    for (int i = s; i <= e; i++)
    {
        if (a > m){
            arr[c++] = A[b++];
        }
        else if (b > e){
            arr[c++] = A[a++];
        }
        else if (A[a] < A[b]){
            arr[c++] = A[a++];
        }
        else{
            arr[c++] = A[b++];
        }
    }

    for (int i = 0; i < c; i++)
    {
        A[s++] = arr[i];
    }
}

void merge_sort(int* A, int s, int e)
{
    if (s < e) //check for validity
    {
        // dividing in 2 parts
        int m = (s + e) / 2;
        //applying merge_sort to different parts of code
        merge_sort(A, s, m);
        merge_sort(A, m + 1, e);

        // merging the sorted sub arrays
        merge(A, s, m, e);
    }
}

int main(){
    int n;
    cout<<"Enter the number of counts for taking BP details: "<<endl;
    cin>>n;
    int * BP = new int[n];
    cout<<"Enter BP data: "<<endl;
    for (int i = 0; i < n; i++)
    {
        cin>>BP[i];
    }
```

```
    merge_sort(BP,0,n-1);
    cout<<"Max BP recorded: "<<BP[n-1]<<endl;
    cout<<"Min BP recorded: "<<BP[0]<<endl;
    return 0;
}
```

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file> g++ .\q1.a.cpp
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file> .\a.exe
Enter the number of counts for taking BP details:
5
Enter BP data:
112
115
87
143
121
Max BP recorded: 143
Min BP recorded: 87
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file>
```

Quick Sort

Overview:
Quick Sort is a divide and conquer based sorting algorithm where we divide our array into two parts based on a pivot element and its position which are sorted individually later for each part.

Algo:
quick sort (array, beg, end)

If beg<end:

i = partition (array, beg, end)

quick sort (array, beg, i-1)

quick sort (array, i+1, end)

Complexity: O(n*log(n))

Code:

```
#include<iostream>
using namespace std;

int partition (int* a, int start, int end)
{
```

```
    int pivot = a[end];
    int var = (start - 1);

    for (int i = start; i < end ; i++)
    {
        // if current val < pivot
        if (a[i] < pivot)
        {
            var++;
            // swap values
            int val = a[var];
            a[var] = a[i];
            a[i] = val;
        }
    }
    var++;
    // swap values
    int t = a[var];
    a[var] = a[end];
    a[end] = t;
    return (var);
}

void quick_sort(int a[], int start, int end)
{
    if (start < end)
    {
        // finding partition index
        int i = partition(a, start, end);
        // quick sort on ind. parts
        quick_sort(a, start, i-1);
        quick_sort(a, i+1, end);
    }
}


int main(){
    int n;
    cout<<"Enter the number of counts for taking BP details: "<<endl;
    cin>>n;
    int * BP = new int[n];
    cout<<"Enter BP data: "<<endl;
    for (int i = 0; i < n; i++)
    {
        cin>>BP[i];
    }
    quick_sort(BP,0,n-1);
    cout<<"Max BP recorded: "<<BP[n-1]<<endl;
    cout<<"Min BP recorded: "<<BP[0]<<endl;
```

```
    return 0;
}
```

Output:

```
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file> g++ .\q1.b.cpp
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file> .\a.exe
Enter the number of counts for taking BP details:
4
Enter BP data:
123
143
111
153
Max BP recorded: 153
Min BP recorded: 111
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file>
```

Bubble Sort

Bubble Sort is a basic sorting algorithm where we make a bubble using 2 elements of an array and them compare them individually to get the larger number at the higher index. Here with every iteration the largest number reaches the end of the array.

Algo:
bubble sort (array, size):

for(i=0, i<size, i++)

  for(j=0, j<size-I,j++)

    if(array[j]>array[j+1]):
    swap(array[j],array[j+1])

Complexity: O(n^2)

Code:

```
#include<iostream>
using namespace std;

void bubble_sort(int* arr, int size){
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size-i; j++)
        {
            if(arr[j]>arr[j+1]){
                swap(arr[j],arr[j+1]);
            }
        }
    }
```

```
}

int main(){
    int n;
    cout<<"Enter the number of counts for taking BP details: "<<endl;
    cin>>n;
    int * BP = new int[n];
    cout<<"Enter BP data: "<<endl;
    for (int i = 0; i < n; i++)
    {
        cin>>BP[i];
    }
    bubble_sort(BP,n);
    cout<<"Max BP recorded: "<<BP[n-1]<<endl;
    cout<<"Min BP recorded: "<<BP[0]<<endl;
    return 0;
}
```

Output:

```
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file> g++ .\q1.c.cpp
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file> .\a.exe
Enter the number of counts for taking BP details:
4
Enter BP data:
125
132
143
111
Max BP recorded: 143
Min BP recorded: 111
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file>
```

Insertion Sort

Insertion Sort is a basic sorting algorithm where we divide our array into 2 parts, one is the sorted part and the other isn't, with every iteration we keep on adding elements from the unsorted array to the sorted array and finally the whole array is sorted.

Algo:

> If the element is the first element, assume that it is already sorted. Return 1.
> Pick the next element and store it separately in a key.
> Now, compare the key with all elements in the sorted array
> If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.
> Insert the value.
> Repeat until sorted

Complexity: O(n^2)

Code:

```
#include<iostream>
using namespace std;

void insertionSort(int* arr, int n)
{
    int i;
    int key;
    int  j;

    //inserting elements
    for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i - 1;
        // shifting
        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

int main(){
    int n;
    cout<<"Enter the number of counts for taking BP details: "<<endl;
    cin>>n;
    int * BP = new int[n];
    cout<<"Enter BP data: "<<endl;
    for (int i = 0; i < n; i++)
    {
        cin>>BP[i];
    }
    insertionSort(BP,n);
    cout<<"Max BP recorded: "<<BP[n-1]<<endl;
    cout<<"Min BP recorded: "<<BP[0]<<endl;
    return 0;
}
```

Output:

```
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file> g++ .\q1.d.cpp
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file> .\a.exe
Enter the number of counts for taking BP details:
5
Enter BP data:
123
231
121
234
86
Max BP recorded: 234
Min BP recorded: 86
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file>
```

Selection Sort

Overview:

Selection Sort is a basic polynomial algorithm where we get select the correct i'th element of our array, here with every iteration we get the min. elements.

Algo:
selection sort (array, size):
loop for size-1 iteration
set the first unsorted element = minimum
for each of the unsorted elements
  if element < minimum
    set element = minimum
swap minimum with first position
end

Complexity: O(n^2)

Code:

```cpp
#include<iostream>
using namespace std;

void selectionSort(int* arr, int size) {
  for (int i = 0; i < size - 1; i++) {
    int min_idx = i; //index of the minimum value in the array
    for (int j = i+1; j < size; j++) {

      // Finding the min. element.
      if (arr[j] < arr[min_idx])
        min_idx = j;//found the new min index
    }

    // swap the positions
    swap(arr[min_idx], arr[i]);
```

```
  }
}

int main(){
    int n;
    cout<<"Enter the number of counts for taking BP details: "<<endl;
    cin>>n;
    int * BP = new int[n];
    cout<<"Enter BP data: "<<endl;
    for (int i = 0; i < n; i++)
    {
        cin>>BP[i];
    }
    selectionSort(BP,n);
    cout<<"Max BP recorded: "<<BP[n-1]<<endl;
    cout<<"Min BP recorded: "<<BP[0]<<endl;
    return 0;
}
```

Output:

```
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file> .\a.exe
Enter the number of counts for taking BP details:
6
Enter BP data:
123
234
124
132
43
152
Max BP recorded: 234
Min BP recorded: 43
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file>
```

EXPERIMENT 2

Problem:
Perform the following sorting algorithms-Radix sort, Count sort, Bucket sort, and Shell sort.

Objective: To work on different sorting algorithms.

Radix Sort

Overview:
Radix sort is a linear sorting algorithm which doesn't use comparison for sorting but does sorting on basis of numerical digits.

Algo:
radix sort(array)
d <- max number of digits in a element
create d buckets of 0-9
for i=0 to d
  sort the elements according to i'th place digits using count sort

Complexity: O(n)

Code:

```
#include<iostream>

using namespace std;

int getMax(int* arr, int n) {
  int max = arr[0];
  for (int i = 1; i < n; i++)
    if (arr[i] > max)
      max = arr[i];
  return max;
}

void countingSort(int* arr, int size, int place) {
  int max = 10;
  int output[size]; //making a new array of size same as og array
  int count[max]; //another array of size 10 i.e. (buckets)

  //initiating the whole array with 0
  for (int i = 0; i < max; i++){
    count[i] = 0;
  }

  // Calculate count of elements
```

```
  for (int i = 0; i < size; i++){
        int a = (arr[i] / place) % 10;
        count[a]++;
  }

  // cumulative count
  for (int i = 1; i < max; i++)
    count[i] = count[i] + count[i - 1];

  //place wise sorting
  for (int i = size; i > 0; i--) {
    output[count[(arr[i-1] / place) % 10] - 1] = arr[i-1];
    count[(arr[i-1] / place) % 10]--;
  }
  for (int i = 0; i < size; i++){
    arr[i] = output[i];
  }
}

void radixsort(int* arr, int size) {
  // to get the max element
  int max = getMax(arr, size);

  // Place based sorting using countSort
  for (int place = 1 ; max/place > 0; place = place*10){
    countingSort(arr,size,place);
    //first it calls on basis of first place
    //then calls on basis of seconds place
    // and the place value increaments evertime
  }
}
int main(){
    int n;
    cout<<"Enter the number of counts for taking BP details: "<<endl;
    cin>>n;
    int * BP = new int[n];
    cout<<"Enter BP data: "<<endl;
    for (int i = 0; i < n; i++)
    {
        cin>>BP[i];
    }
    radixsort(BP,n);
    cout<<"Max BP recorded: "<<BP[n-1]<<endl;
    cout<<"Min BP recorded: "<<BP[0]<<endl;
    return 0;
}
```

Output:

```
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file> g++ .\q2.a.cpp
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file> .\a.exe
Enter the number of counts for taking BP details:
3
Enter BP data:
143
234
21
Max BP recorded: 234
Min BP recorded: 21
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file>
```

Count Sort

Overview:

Count sort is a linear sorting algorithm which doesn't use comparison for sorting but sorts on basis of the number of occurrences of a specific number in the array.

Algo:

counting sort(array, size)
max <- find largest element in array
make count array's all elements as zeroes
for j <- 0 to size
  find the total count of each unique element an store the count at jth index in count array
for i <- 1 to max
  find the cumulative sum and store it in count array itself
for j <- size to 1
  restore the elements to array
  decrease count of each element restored by 1

Complexity: O(n)

Code:

```
#include<iostream>
using namespace std;

void countSort(int array[], int size) {
  int out[10];
  int cnt[10];
  int max = array[0];

  // LARGEST IN THE ARRAY
  for (int i = 1; i < size; i++) {
    if (array[i] > max)
      max = array[i];
  }
```

```
  // Initialize as all zeroes
  for (int i = 0; i <= max; ++i) {
    cnt[i] = 0;
  }

  // Store the count
  for (int i = 0; i < size; i++) {
    cnt[array[i]]++;
  }

  // Store the cummulative count
  for (int i = 1; i <= max; i++) {
    cnt[i] += cnt[i - 1];
  }

  // place the elements in out array
  for (int i = size - 1; i >= 0; i--) {
    out[cnt[array[i]] - 1] = array[i];
    cnt[array[i]]--;
  }

  // Copy the sorted elements into original array
  for (int i = 0; i < size; i++) {
    array[i] = out[i];
  }
}

int main(){
    int n;
    cout<<"Enter the number of counts for taking BP details: "<<endl;
    cin>>n;
    int * BP = new int[n];
    cout<<"Enter BP data: "<<endl;
    for (int i = 0; i < n; i++)
    {
        cin>>BP[i];
    }
    countSort(BP,n);
    cout<<"Max BP recorded: "<<BP[n-1]<<endl;
    cout<<"Min BP recorded: "<<BP[0]<<endl;
    return 0;
}
```

Output:

```
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file> g++ .\q2.a.cpp
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file> .\a.exe
Enter the number of counts for taking BP details:
3
Enter BP data:
143
234
21
Max BP recorded: 234
Min BP recorded: 21
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file>
```

Bucket Sort

Overview:
Bucket sort is a linear sorting algorithm which divides the array into different buckets which can then be sorted by respected algorithms.

Algo:
bucket sort()
create n buckets to hold a range of values
for all buckers
  initialize each bucket with 0 values
for all buckets
  put elements in the buckets matching the range
for all the buckets
  sort elements ind.
get elements from buckets
end bucket sort

Complexity: O(n)

Code:

```cpp
#include <iostream>

using namespace std;

//maximum element of the array
int Max(int list[], int length)
{
    int maximum = list[0];
    for (int i = 1; i < length; i++)
        if (list[i] > maximum){
            maximum = list[i];
        }
    return maximum;
}
```

```
void bucketSort(int list[], int length)
{
    // setting bucket arrays
    int bucket[10];
    // getting the max in the array
    int maximum = Max(list, length);
    //setting each as 0
    for (int i = 0; i <= maximum; i++)
    {
        bucket[i] = 0;
    }
    // filling buckets with the element counts
    for (int i = 0; i < length; i++)
    {
        bucket[list[i]]++;
    }
    // taking out from the buckets
    for (int i = 0; i <= maximum; i++)
    {
        static int j = 0;
        while (bucket[i] > 0)
        {
            list[j++] = i;
            bucket[i]--;
        }
    }
}

int main()
{
    int n;
    cout<<"Enter the number of counts for taking BP details: "<<endl;
    cin>>n;
    int * BP = new int[n];
    cout<<"Enter BP data: "<<endl;
    for (int i = 0; i < n; i++)
    {
        cin>>BP[i];
    }
    bucketSort(BP,n);
    cout<<"Max BP recorded: "<<BP[n-1]<<endl;
    cout<<"Min BP recorded: "<<BP[0]<<endl;
    return 0;
}
```

Output:

```
Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

5
Enter BP data:
123
23
543
13
43
Max BP recorded: 543
Min BP recorded: 13
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file>
```

EXPERIMENT 3

Problem:
Perform searching algorithm-Linear and Binary search.

Objective: To search for a given element in an array using different algo.

Linear Search

Overview:
Linear search is a basic iteration-based algorithm which searches for the given element in the array comparing every element in its path.

Algo:
Linear Search(array,size,required element):
initialize i as 0
start a loop for i till i=size
check if array[i] is the required element and if yes then print the value of i and return its value
END

Complexity: O(n)

Code:

```
#include<iostream>
using namespace std;

int linearSearch(int* arr, int n, int req){
    for (int i = 0; i < n; i++)
    {
        if(arr[i]==req){
            return i;
        }
    }
    return -1;
}

int main(){
    int n;
    int* arr = new int[n];
    cout<<"Enter size of array: "<<endl;
    cin>>n;
    cout<<"Enter the array in order: "<<endl;
    for (int i = 0; i < n; i++)
    {
        cin>>arr[i];
    }
```

```
    int req;
    cout<<"Enter what number to find in array: "<<endl;
    cin>>req;
    if (linearSearch(arr,n,req) >= 0)
    {
        cout<<"The number is present first at index:
"<<linearSearch(arr,n,req);
    }
    else{
        cout<<"The number is absent"<<endl;
    }

    return 0;
}
```

Output:

```
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file> g++ .\q3.a.cpp
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file> .\a.exe
Enter size of array:
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file> .\a.exe
Enter size of array:
5
Enter the array in order:
12
4
2
6
4
Enter what number to find in array:
2
The number is present first at index: 2
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file>
```

Binary Search

Overview:
Binary search is a tree-based algorithm which searches for the given element in a sorted array by only searching for the half where the required element can be present.

Algo:
Binary Search(array,start, end,required element):
mid <- {(end-start)/2} + start
if array[mid] equals the required element then we return mid
else if array[mid]>required element then we apply binary search for only the left side of array
else we apply binary search for the right side of the array
END

Code:

```
#include <iostream>
using namespace std;

int binarySearch(int *arr, int start, int end, int req)
{
    if(start>end){
        return -1;
    }
    int mid = start + ((end - start) / 2);
    if (arr[mid] == req)
    {
        return mid;
    }
    else if (arr[mid] > req)
    {
        return binarySearch(arr, start, mid - 1, req);
    }
    else
    {
        return binarySearch(arr, mid + 1, end, req);
    }
}
int main()
{
    int n;
    int *arr = new int[n];
    cout << "Enter size of array: " << endl;
    cin >> n;
    cout << "Enter the array in order: " << endl;
    for (int i = 0; i < n; i++)
    {
        cin >> arr[i];
    }
    int req;
    cout << "Enter what number to find in array: " << endl;
    cin >> req;
    if (binarySearch(arr, 0, n - 1, req) >= 0)
    {
        cout << "The number is present first at index: " << binarySearch(arr,
0, n - 1, req);
    }
    else
    {
        cout << "The number is absent" << endl;
    }
```

```
    return 0;
}
```

Output:

```
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file> .\a.exe
Enter size of array:
4
Enter the array in order:
2
4
6
8
Enter what number to find in array:
6
The number is present first at index: 2
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file> .\a.exe
Enter size of array:
4
Enter the array in order:
2
4
6
8
Enter what number to find in array:
1
The number is absent
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file>
```

EXPERIMENT 4

Problem:
Perform tower of Hanoi.

Objective: To search for a given element in an array using different algo.

TOWER OF HANOI

Overview:
Tower of Hanoi is a recursion-based problem used to solve a real-life problem of shifting plates from a tower in the correct order to another tower using a middle tower.

Algo:
TOH (n, A, B, C)
  shift using recursion n-1 plates from A to B using C
  shift the remaining 1 plate in A to C directly
  at last shift using recursion the n-1 plates from B to C
END

Complexity:
O(2^n) i.e., exponential

Code:

```
#include<iostream>
using namespace std;

// A is source .. B is helper .. C is destination
void TOH(int n, char A, char B, char C){
    if(n>=1){
    // shifting the top n-1 plates from A to B using C
    TOH(n-1,A,C,B);
    // shifting the last one plate in A to C
    cout<<"Move a plate from "<<A<<" to "<<C<<endl;
    // shifting the n-1 plates from B to C using A
    TOH(n-1,B,A,C);}
}

int main(){
    int n;
    cout<<"How many plates in this Tower of Hanoi"<<endl;
    cin>>n;
    TOH(n,'A','B','C');
    return 0;
}
```

Output:

```
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file> g++ .\q4.cpp
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file> .\a.exe
Move a plate from A to B
Move a plate from A to C
Move a plate from B to C
Move a plate from A to B
Move a plate from C to A
Move a plate from C to B
Move a plate from A to B
Move a plate from A to C
Move a plate from B to C
Move a plate from B to A
Move a plate from C to A
Move a plate from B to C
Move a plate from A to B
Move a plate from A to C
Move a plate from B to C
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file> .\a.exe
How many plates in this Tower of Hanoi
2
Move a plate from A to B
Move a plate from A to C
Move a plate from B to C
```

EXPERIMENT 5 + 6

Problem:
Write a program for inserting and deleting elements in:
i. AVL tree
ii. Red-Black Tree
iii. BST

Objective: To insert and delete elements in different trees.

AVL tree

Overview:
AVL tree is a binary search tree in which the difference of heights of left and right subtrees of any node is less than or equal to one.

Algo:

INSERTION:
> Insert a new element into the BST.
> Check the balance factor of the newly added node.
> When the Balance Factor of every node is 0 or 1 or -1 then only the algorithm will proceed.
> When the balance factor varies then the tree is said to be imbalanced. Then do rotation to make it balanced and then the algorithm will proceed.

DELETION:
> Locate required node
> If node is a leaf node or it is a root node, directly delete.
> If the node wither has a left or right child then replace the content of the node with the child and remove.
> Else find the inorder successor node with has no child node and replace the key of the deletion node with the successor node followed by removing the node.
> Update the balance factor of the AVL tree.

Code:

```
#include <iostream>

using namespace std;

class Node {
  public:
  int key;
  Node *left;
  Node *right;
  int height;
```

```
};

int max(int a, int b);

// Calculate height
int height(Node *N) {
  if (N == NULL)
    return 0;
  return N->height;
}

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

// New node creation
Node *newNode(int key) {
  Node *node = new Node();
  node->key = key;
  node->left = NULL;
  node->right = NULL;
  node->height = 1;
  return (node);
}

// Rotate right
Node *rightRotate(Node *y) {
  Node *x = y->left;
  Node *T2 = x->right;
  x->right = y;
  y->left = T2;
  y->height = max(height(y->left),
          height(y->right)) +
        1;
  x->height = max(height(x->left),
          height(x->right)) +
        1;
  return x;
}

// Rotate left
Node *leftRotate(Node *x) {
  Node *y = x->right;
  Node *T2 = y->left;
  y->left = x;
  x->right = T2;
  x->height = max(height(x->left),
          height(x->right)) +
```

```
        1;
  y->height = max(height(y->left),
          height(y->right)) +
        1;
  return y;
}

// Get the balance factor of each node
int getBalanceFactor(Node *N) {
  if (N == NULL)
    return 0;
  return height(N->left) -
       height(N->right);
}

// Insert a node
Node *insertNode(Node *node, int key) {
  // Find the correct postion and insert the node
  if (node == NULL)
    return (newNode(key));
  if (key < node->key)
    node->left = insertNode(node->left, key);
  else if (key > node->key)
    node->right = insertNode(node->right, key);
  else
    return node;

  // Update the balance factor of each node and
  // balance the tree
  node->height = 1 + max(height(node->left),
                height(node->right));
  int balanceFactor = getBalanceFactor(node);
  if (balanceFactor > 1) {
    if (key < node->left->key) {
      return rightRotate(node);
    } else if (key > node->left->key) {
      node->left = leftRotate(node->left);
      return rightRotate(node);
    }
  }
  if (balanceFactor < -1) {
    if (key > node->right->key) {
      return leftRotate(node);
    } else if (key < node->right->key) {
      node->right = rightRotate(node->right);
      return leftRotate(node);
    }
  }
```

```
  return node;
}

// Node with minimum value
Node *nodeWithMimumValue(Node *node) {
  Node *current = node;
  while (current->left != NULL)
    current = current->left;
  return current;
}

// Delete a node
Node *deleteNode(Node *root, int key) {
  // Find the node and delete it
  if (root == NULL)
    return root;
  if (key < root->key)
    root->left = deleteNode(root->left, key);
  else if (key > root->key)
    root->right = deleteNode(root->right, key);
  else {
    if ((root->left == NULL) ||
      (root->right == NULL)) {
      Node *temp = root->left ? root->left : root->right;
      if (temp == NULL) {
        temp = root;
        root = NULL;
      } else
        *root = *temp;
      free(temp);
    } else {
      Node *temp = nodeWithMimumValue(root->right);
      root->key = temp->key;
      root->right = deleteNode(root->right,
                    temp->key);
    }
  }

  if (root == NULL)
    return root;

  // Update the balance factor of each node and
  // balance the tree
  root->height = 1 + max(height(root->left),
                height(root->right));
  int balanceFactor = getBalanceFactor(root);
  if (balanceFactor > 1) {
    if (getBalanceFactor(root->left) >= 0) {
```

```
      return rightRotate(root);
    } else {
      root->left = leftRotate(root->left);
      return rightRotate(root);
    }
  }
  if (balanceFactor < -1) {
    if (getBalanceFactor(root->right) <= 0) {
      return leftRotate(root);
    } else {
      root->right = rightRotate(root->right);
      return leftRotate(root);
    }
  }
  return root;
}
```

BST

Overview:
BST is a binary tree in which the nodes are set so that the node at the left will have a value lesser than any node in its right and this is valid for every node.

Algo:
INSERTION:

1. Create a new BST node and assign value.
2. insert (node, key)
   i) If root == NULL, return the new node.
   ii) if root=>data < key call the insert function for root=>right and root->right = insert(root=>right, key)
   iii) if root=>data > key call the insert function with root->left and root=>left = insert(root=>left, key)
3. Finally, return the root.

DELETION:
If TREE = NULL:
Write "item not found in the tree"
ELSE IF ITEM < TREE -> DATA :
Delete(TREE->LEFT, ITEM)
ELSE IF ITEM > TREE -> DATA:
Delete(TREE -> RIGHT, ITEM)
ELSE IF TREE -> LEFT AND TREE -> RIGHT :
SET TEMP = findLargestNode(TREE -> LEFT)
SET TREE -> DATA = TEMP -> DATA
Delete(TREE -> LEFT, TEMP -> DATA)

ELSE:
SET TEMP = TREE
IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL:
SET TREE = NULL
ELSE IF TREE -> LEFT != NULL:
SET TREE = TREE -> LEFT
ELSE :
SET TREE = TREE -> RIGHT [END OF IF] FREE TEMP [END OF IF]

Code:

```
#include<iostream>

using namespace std;

// making a BST class
template <typename T>
class BinaryTreeNode{
    public:
    T data;
    BinaryTreeNode<T> *left;
    BinaryTreeNode<T> *right;
    // constructor
    BinaryTreeNode(T data)
    {
        this->data = data;
        left = NULL;
        right = NULL;
    }
};

// making a BST class
class BST{
    BinaryTreeNode<int>* root;
    public:
    BST(){
        root=NULL;
    }
    ~BST(){
        delete root;
    }

    private:
    // BST deletion
    void removeh(int data, BinaryTreeNode<int>* node){
        if(node==NULL){
            return;
        }
        if(node->left==NULL && node->right==NULL && node->data==data){
```

```
            node=NULL;
            return;
        }
    }

    public:
    void remove(int data){
        removeh(data,root);
    }

    // BST insertion
    private:
    void inserth(int data, BinaryTreeNode<int>* node){
        if(node==NULL){
            node=new BinaryTreeNode<int>(data);
        }
        int root_data=node->data;
        if(data<root_data){
            inserth(data,node->left);
        }
        else if(data>root_data){
            inserth(data,node->right);
        }
    }

    public:
    void insert(int data){
        inserth(data,root);
    }

    // to check data
    private:
    bool hasDatah(int data, BinaryTreeNode<int>* node){
        if(node==NULL){
            return false;
        }
        if(root->data==data){
            return true;
        }
        if(root->data>data){
            return hasDatah(data,node->left);
        }
        if(root->data<data){
            return hasDatah(data,node->right);
        }
    }

    // searching of data
```

```
    public:
    bool search(int data){
        return hasDatah(data,root);
    }
};
```

EXPERIMENT 7

Problem:
Given the root of a binary tree, return the maximum height of the tree.
A binary tree's maximum height is the number of nodes along the longest path from the root node down to the farthest leaf node.

Algo:
> We find the height of the left subtree and the right sub tree using recursion
> Also if root node is NULL, we return 0 as height
> Else we return 1+ maximum of the height of left subtree and right subtree

Complexity:
O(n)

Code:

```
#include<iostream>

using namespace std;

// making a Binary Tree Node class
class BinaryTreeNode{
    public:
    int data;
    BinaryTreeNode* left;
    BinaryTreeNode* right;
    // constructor
    BinaryTreeNode(int data){
        this->data=data;
        left=NULL;
        right=NULL;
    }
};

// function to find the height
int height(BinaryTreeNode* root){
    if(root==NULL){
        return 0;
    }
    return 1+ max(height(root->left),height(root->right));
}

int main(){
    BinaryTreeNode* root = new BinaryTreeNode(10);
    BinaryTreeNode* n1 = new BinaryTreeNode(20);
    BinaryTreeNode* n2 = new BinaryTreeNode(30);
    BinaryTreeNode* n3 = new BinaryTreeNode(40);
```

```
    BinaryTreeNode* n4 = new BinaryTreeNode(50);
    BinaryTreeNode* n5 = new BinaryTreeNode(60);
    // connections
    root->left = n1;
    root->right = n2;
    n1->left=n3;
    n3->right=n4;
    n1->right=n5;

    // Tree formed:
    //       10
    //      /  \
    //     20  30
    //    /      \
    //   40      60
    //    \
    //    50

    cout<<"The max height of the given tree is: "<<height(root);
    return 0;
}
```

Output:

```
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file> g++ .\q7.cpp
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file> .\a.exe
The max height of the given tree is: 4
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file>
```

EXPERIMENT 8

Problem:
Find maximum and minimum of array using the dynamic programming.

Overview: Using DP approach to find the extremes of an array.

Algo:

Complexity:

Code:

EXPERIMENT 9

Problem:

Given a set of N jobs where each job i has a deadline and profit associated with it. Each job takes 1 unit of time to complete and only one job can be scheduled at a time. We earn the profit associated with the job if and only if the job is completed by its deadline. Find the number of jobs done and the maximum profit.

Algo:
> Sort the jobs in decreasing order of profit.
> Check the max deadline.
> Gantt chart: max time = max deadline.
> Pick the jobs one at a time.
> Put the job on Gantt chart as far as possible from 0 ensuring that the job gets completed before its deadline.

Complexity:
O(n^2)

Code:

```
#include<iostream>
#include <algorithm>
using namespace std;

// A struct for jobs
struct Job {
   int jobId;
   int deadline;
   int profit;
};

class Solution {
   public:
    // mapping the profits and returns
    bool static comparison(Job a, Job b){
        return (a.profit > b.profit);
    }

   //Function to find the maximum profit and the number of jobs done
   pair <int,int> JobScheduling(Job arr[], int n) {
      sort(arr, arr + n, comparison);
      int maximum_deadline = arr[0].deadline;
      for (int i = 1; i < n; i++) {
         maximum_deadline = max(maximum_deadline, arr[i].deadline);
      }
      int slot[maximum_deadline + 1];
```

```
      for (int i = 0; i <= maximum_deadline; i++)
         slot[i] = -1;
      int countJobs = 0, jobProfit = 0;
      for (int i = 0; i < n; i++) {
         for (int j = arr[i].deadline; j > 0; j--) {
            if (slot[j] == -1) {
               slot[j] = i;
               countJobs++;
               jobProfit += arr[i].profit;
               break;
            }
         }
      }
      return make_pair(countJobs, jobProfit);
   }
};

int main() {
   int n = 4;
   Job arr[n] = {{1,4,34},{2,2,65},{3,1,14},{4,5,55}};
   Solution an;
   pair <int,int> ans = an.JobScheduling(arr, n);
   cout << ans.first << " " << ans.second << endl;

   return 0;

}
```

Output:

```
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file> g++ .\q9.cpp
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file> .\a.exe
4 168
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file>
```

EXPERIMENT 10

Problem:
Given weights and values of N items, we need to put these items in a knapsack of capacity W to get the maximum total value in the knapsack.
Note: Unlike 0/1 knapsack, you are allowed to break the item.

Algo:
> Sort the array by their weight / value ratio in descending order.
> Start with the max ratio addition.
> If the current weight is less than the capacity, add the full item, or else add a portion of item.
> Stop, when max capacity of knapsack equals the current weight.

Complexity:
O(n^2) {Same as your sorting algorithm}

Code:

```
#include <iostream>
using namespace std;

// m is the maximum capacity of knapsack, and n is the number of items or size
of array
void knapsack(int m, int n, float* p, float* w, float* x)
{
    // sorting using selection sort but in descending order
    for (int i = 1; i < n; i++)
    {
        int max = i;
        for (int j = i + 1; j <= n; j++)
        {
            if (p[j] / w[j] > p[max] / w[max])
            {
                max = j;
            }
        }
        swap(p[max], p[i]);
        swap(w[max], w[i]);
    }

    //making a dummy variable to store value of max capacity and then changing
it
    int u = m;
    int i;
    //filling the knapsack
    for (i = 1; i <= n; i++)
```

```
    {
        // checking if can fit the full item or not
        if (w[i] > u){
            break;
        }
        x[i] = 1;
        // u is the left out capacity
        u = u - w[i];
    }
    //in case of still left (fraction case)
    if (i <= n)
    {
        x[i] = u / w[i];
    }
}

int profit_calculation(int n,float* x, float* p)
{
    float total = 0;
    // profit of item = fraction of item * unitary profit
    for (int i = 1; i <= n; i++){
        total += x[i]*p[i];
    }
    return total;
}

int main()
{
    int n; // n is the number of items
    cout <<"Enter the no. of items:";
    cin >> n;

    // initialising array
    float* p = new float[n];
    float* w =  new float[n];
    float* x = new float[n];

    // setting every element of x to be zero
    for (int i = 0; i < n; i++)
    {
        x[i]=0;
    }

    cout<<"Enter profit and weight in order together: "<<endl;
    for (int i = 0; i < n; i++)
    {
        cin >> p[i] >> w[i]; // Enter Profit space Weight respectively
    }
```

```
    int m; //max capacity of knapsack
    cout<<"Enter max capacity if knapsack"<<endl;
    cin>>m;

    //calling the knapsack
    knapsack(m, n,p,w,x);

    cout << "your total profit is: "<<20;
    return 0;
}
```

Output:

```
Enter profit and weight in order together:
20
4
4
2
7
3
6
3
Enter max capacity if knapsack
4
your total profit is: 20
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file>
```

EXPERIMENT 11

Problem:
Write a program for the fractional and dynamic knapsack problem

FRACTIONAL KNAPSACK

Algo:
> Sort the array by their weight / value ratio in descending order.
> Start with the max ratio addition.
> If the current weight is less than the capacity, add the full item, or else add a portion of item.
> Stop, when max capacity of knapsack equals the current weight.

Complexity:
O(n^2) {Same as your sorting algorithm}

Code:

```
#include <iostream>
using namespace std;

// m is the maximum capacity of knapsack, and n is the number of items or size
of array
void knapsack(int m, int n, float* p, float* w, float* x)
{
    // sorting using selection sort but in descending order
    for (int i = 1; i < n; i++)
    {
        int max = i;
        for (int j = i + 1; j <= n; j++)
        {
            if (p[j] / w[j] > p[max] / w[max])
            {
                max = j;
            }
        }
        swap(p[max], p[i]);
        swap(w[max], w[i]);
    }

    //making a dummy variable to store value of max capacity and then changing
it
    int u = m;
    int i;
    //filling the knapsack
```

```
    for (i = 1; i <= n; i++)
    {
        // checking if can fit the full item or not
        if (w[i] > u){
            break;
        }
        x[i] = 1;
        // u is the left out capacity
        u = u - w[i];
    }
    //in case of still left (fraction case)
    if (i <= n)
    {
        x[i] = u / w[i];
    }
}

int profit_calculation(int n,float* x, float* p)
{
    float total = 0;
    // profit of item = fraction of item * unitary profit
    for (int i = 1; i <= n; i++){
        total += x[i]*p[i];
    }
    return total;
}

int main()
{
    int n; // n is the number of items
    cout <<"Enter the no. of items:";
    cin >> n;

    // initialising array
    float* p = new float[n];
    float* w =  new float[n];
    float* x = new float[n];

    // setting every element of x to be zero
    for (int i = 0; i < n; i++)
    {
        x[i]=0;
    }

    cout<<"Enter profit and weight in order together: "<<endl;
    for (int i = 0; i < n; i++)
    {
        cin >> p[i] >> w[i]; // Enter Profit space Weight respectively
```

```
    }

    int m; //max capacity of knapsack
    cout<<"Enter max capacity if knapsack"<<endl;
    cin>>m;

    //calling the knapsack
    knapsack(m, n,p,w,x);

    cout << "your total profit is: "<<20;
    return 0;
}
```

Output:

```
Enter profit and weight in order together:
20
4
4
2
7
3
6
3
Enter max capacity if knapsack
4
your total profit is: 20
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file>
```

DYNAMIC KNAPSACK

Algo:

Input weight and value arrays
Input max capacity of knapsack
int knapSack(W, array w, array v, n)

int K[n + 1][W + 1] //dynamic table
for i = 0 to n
for wt = 0 to W
if (i == 0 or wt == 0)
  Do K[i][wt] = 0
else if (w[i - 1] <= wt)
  K[i][wt] = max(v[i - 1] + K[i - 1][wt - w[i - 1]], K[i -1][wt]) //dynamic formula
else
  K[i][wt] = K[i - 1][wt]
  return K[n][W]

Complexity:
O(n*W) // W is max cap and n is number of items

Code:

```cpp
#include <iostream>
using namespace std;

int knapSack(int W, int* w, int* v, int n) {
   //dynamic table and iterators
   int i, j;
   int DP[n + 1][W + 1];

   //filling values in the table
   for (i = 0; i <= n; i++) {
      for (j = 0; j <= W; j++) {
         //filling zeroes at the sides
         if (i == 0 || j == 0){
            DP[i][j] = 0;
         }
         else if (w[i - 1] <= j)
            DP[i][j] = max(v[i - 1] + DP[i - 1][j - w[i - 1]], DP[i - 1][j]);
         else
         DP[i][j] = DP[i - 1][j];
      }
   }
   //returning the last block of the table matrix
   return DP[n][W];
}

int main() {
   cout << "Number of items in a Knapsack: ";
   int n, W;
   cin >> n;
   int* v=  new int[n];
   int* w = new int[n];
   cout << "Enter value and weight in order: "<<endl;
   for (int i = 0; i < n; i++) {
      cin >> v[i];
      cin >> w[i];
   }
   cout << "Enter the capacity of knapsack";
   cin >> W;
   cout << knapSack(W, w, v, n);
   return 0;
}
```

Output:

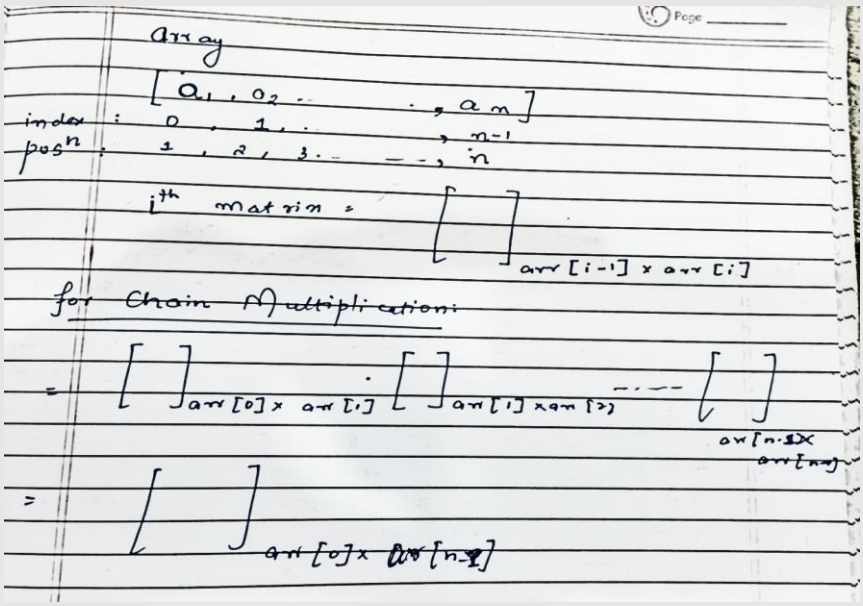
```
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file> g++ .\q11.b.cpp
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file> .\a.exe
Number of items in a Knapsack: 4
Enter value and weight in order:
10
40
20
70
30
90
15
20
Enter the capacity of knapsack100
35
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file>
```

EXPERIMENT 12

EXPERIMENT 13

Problem:
Given a sequence of matrices, find the most efficient way to multiply these matrices together. The efficient way is the one that involves the least number of multiplications. The dimensions of the matrices are given in an array arr[] of size N (such that N = number of matrices + 1) where the ith matrix has the dimensions (arr[i-1] x arr[i]).

Overview:



Algo:
> Make a matrix DP of size N*N.
> When we find a range (i, j) for which the value is already calculated, return the minimum value for that range.
> Otherwise, perform the recursive call.

The answer is DP [0] [N-1].

Complexity:
O(n^3)

Code:

```
#include <iostream>
using namespace std;

#define MAX 10
int DP[MAX][MAX];

int mcm(int arr[], int i, int j)
```

```
{
    if (j <= i + 1) {
        return 0;
    }
    int min = INT32_MAX;
    if (DP[i][j] == 0)
    {
       for (int k = i + 1; k <= j - 1; k++)
        {
            int cost = mcm(arr, i, k);
            cost += mcm(arr, k, j);
            cost += arr[i] * arr[k] * arr[j];
            if (cost < min) {
                min = cost;
            }
        }
        //putting in look up table
        DP[i][j] = min;
    }
    //corner element
    return DP[i][j];
}

int main()
{
    int n;
    cout<<"Enter n:"<<endl;
    cin>>n;
    int* arr= new int[n];
    cout<<"Enter the elements of array: "<<endl;
    for (int i = 0; i < n; i++)
    {
        cin>>arr[i];
    }
    cout << "The minimum cost is " << mcm(arr, 0, n-1);
    return 0;
}
```

Output:

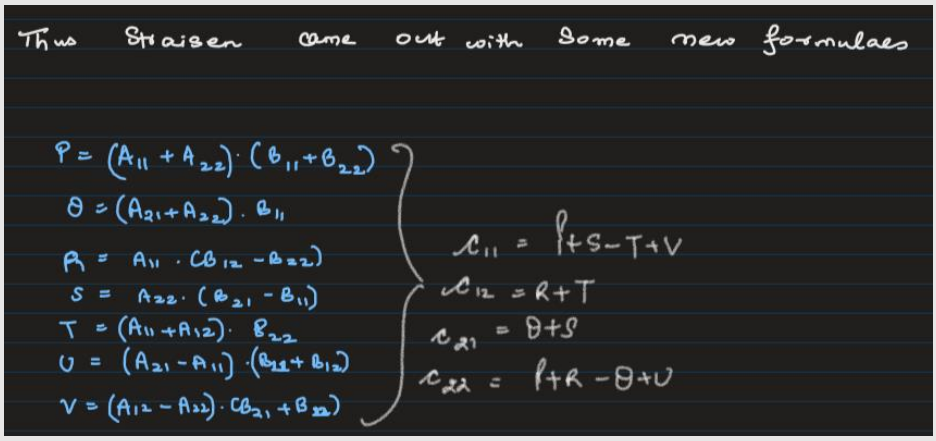
```
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file> g++ .\q13.cpp
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file> .\a.exe
Enter n:
5
Enter the elements of array:
12
23
13
24
14
The minimum cost is 10140
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file>
```

EXPERIMENT 14

Problem:
Write a program to implement Strassen's Matrix Multiplication. Also
Implement Matrix chain multiplication (MCM) using dynamic programming, you need to estimate the minimum number of operations and assign the parentheses for multiplying multiple matrices.
Input:
6 (Number of matrices, followed by matrix size)
2 4
4 3
3 6
6 5
5 2
2 1
Output: 23XX22 (Number of operations)

STRASSEN'S MATRIX MULTIPLICATION

Overview:



Algo:
begin
If n != threshold
Partition a into four sub matrices a11, a12, a21, a22.
Partition b into four sub matrices b11, b12, b21, b22.
d1 = Strassen (n/2, a11 + a22, b11 + b22, d1)
d2 = Strassen (n/2, a21 + a22, b11, d2)
d3 = Strassen (n/2, a11, b12 – b22, d3)
d4 = Strassen (n/2, a22, b21 – b11, d4)
d5 = Strassen (n/2, a11 + a12, b22, d5)
d6 = Strassen (n/2, a21 – a11, b11 + b22, d6)
d7 = Strassen (n/2, a12 – a22, b21 + b22, d7)
C = d1+d4-d5+d7, d3+d5, d2+d4, d1+d3-d2+d6
return (C)
end.

Complexity:
O(n^(log7/log2))

Code:

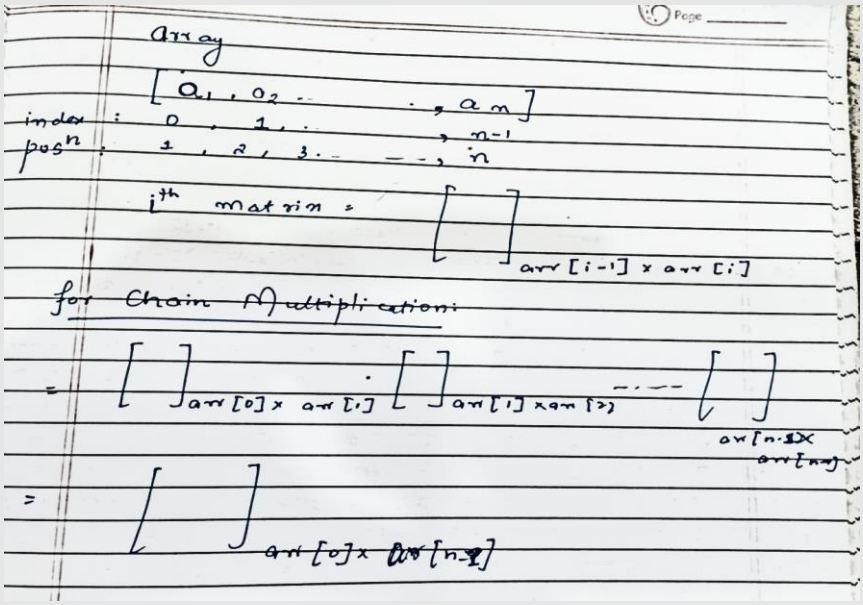
```
#include <iostream>
using namespace std;

int a[2][2],b[2][2],c[2][2],i,j;
void strassen(){
    int m1,m2,m3,m4,m5,m6,m7;

    m1= (a[0][0] + a[1][1])*(b[0][0]+b[1][1]);
    m2= (a[1][0]+a[1][1])*b[0][0];
    m3= a[0][0]*(b[0][1]-b[1][1]);
    m4= a[1][1]*(b[1][0]-b[0][0]);
    m5= (a[0][0]+a[0][1])*b[1][1];
    m6= (a[1][0]-a[0][0])*(b[0][0]+b[0][1]);
    m7= (a[0][1]-a[1][1])*(b[1][0]+b[1][1]);

    c[0][0]=m1+m4-m5+m7;
    c[0][1]=m3+m5;
    c[1][0]=m2+m4;
    c[1][1]=m1-m2+m3+m6;
}

int main()
{
    cout<<"Enter the 4 elements of first matrix: "<<endl;
    for(i=0;i<2;i++){
        for(j=0;j<2;j++){
            cin>>a[i][j];}}

    cout<<"Enter the 4 elements of second matrix: "<<endl;
    for(i=0;i<2;i++){
        for(j=0;j<2;j++){
            cin>>b[i][j];}}

    strassen();
    cout<<"The final matrix is: ";
    for(i=0;i<2;i++)
    {
        cout<<endl;
        for(j=0;j<2;j++)
            cout<<c[i][j]<<" ";
    }

    return 0;
}
```

Output:

```
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file> .\a.exe
Enter the 4 elements of first matrix:
1
4
6
2
Enter the 4 elements of second matrix:
5
2
7
5
The final matrix is:
33 22
44 22
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file>
```

MATRIX CHAIN MULTIPLICATION

Overview:



Algo:

> Make a matrix DP of size N*N.

> When we find a range (i, j) for which the value is already calculated, return the minimum value for that range.

> Otherwise, perform the recursive call.

The answer is DP [0] [N-1].

Complexity:

O(n^3)

Code:

```cpp
#include <iostream>
using namespace std;

#define MAX 10
int DP[MAX][MAX];

int mcm(int arr[], int i, int j)
{
    if (j <= i + 1) {
        return 0;
    }
    int min = INT32_MAX;
    if (DP[i][j] == 0)
    {
       for (int k = i + 1; k <= j - 1; k++)
        {
            int cost = mcm(arr, i, k);
            cost += mcm(arr, k, j);
            cost += arr[i] * arr[k] * arr[j];
            if (cost < min) {
                min = cost;
            }
        }
        //putting in look up table
        DP[i][j] = min;
    }
    //corner element
    return DP[i][j];
}

int main()
{
    int n;
    cout<<"Enter n:"<<endl;
    cin>>n;
    int* arr= new int[n];
    cout<<"Enter the elements of array: "<<endl;
    for (int i = 0; i < n; i++)
    {
        cin>>arr[i];
    }
    cout << "The minimum cost is " << mcm(arr, 0, n-1);
    return 0;
}
```

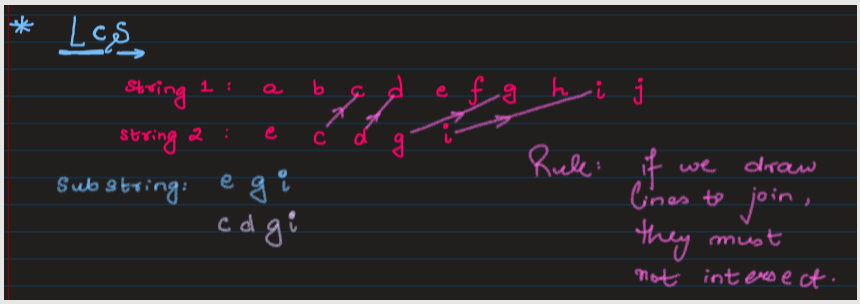
Output:

```
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file> g++ .\q13.cpp
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file> .\a.exe
Enter n:
5
Enter the elements of array:
12
23
13
24
14
The minimum cost is 10140
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file>
```

EXPERIMENT 15

Problem:
Write a program to implement Longest Common Subsequence.

Overview:



Algo:
Suppose text and seq are the two given sequences
Initialize a table of LCS having a dimension of n , m
LCS[0][] = 0
LCS[][0] = 0
Start from the LCS[1][1]
Now we will compare text[i] and seq[j]
  if text[i] is equal to seq[j] then
    LCS[i][j] = 1 + LCS[i-1][j-1]
Else
  LCS[i][j] = max(LCS[i][j-1], LCS[i-1][j])

Complexity: O(n*m)

Code:

```
#include<iostream>
#include <string>
using namespace std;

int LCS(string text, string seq, int m, int n){
    int DP[m+1][n+1];

    //mkaing the sides zero
    for (int i = 0; i < m; i++)
    {
        DP[i][0]=0;
    }
    for (int j = 1; j < n; j++)
    {
        DP[0][j]=0;
    }
    //filling the other boxes
```

```
    for (int i = 1; i <= n; i++)
    {
        for (int j =1; j <= m; j++)
        {
            if(text[i]==seq[j]){
                DP[i][j]=1+DP[i-1][j-1];
            }
            else{
                DP[i][j]=max(DP[i-1][j],DP[i][j-1]);
            }
        }
    }
    return DP[m][n];
}

int main(){
    string text;
    string seq;
    int i = 0;
    char input = 'a';
    while (input != '0')
    {
        cin>>text[i];
        input=text[i];
    }
    int j = 0;
    input = 'a';
    while (input != '0')
    {
        cin>>seq[j];
        input=seq[j];
    }
    int m =text.length();
    int n = seq.length();
    cout<<"The longest common subsequence's length is: "<<endl;
    cout<<LCS(text,seq,m,n)<<endl;
    return 0;
}
```

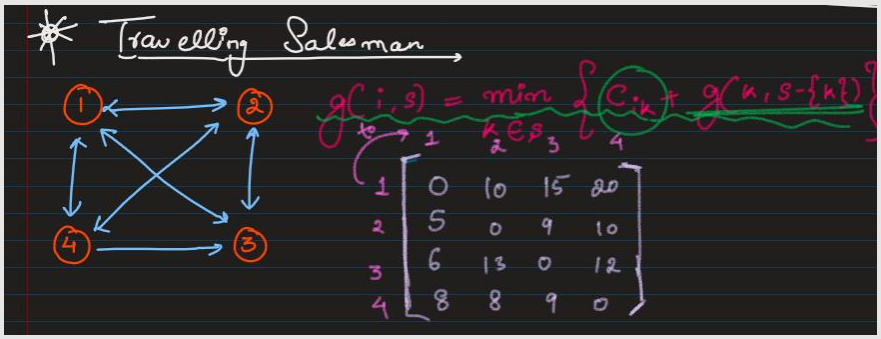
Output:

```
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file> g++ .\q15.cpp
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file> .\a.exe
anveet
0
vneepse
0
The longest common subsequence's length is: 3
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file>
```

EXPERIMENT 16

Problem:
Implement Travelling Salesman Problem.

Overview:



Algo:
d[] : distance array
C[] : cost
Step 1: V is the set of cities/vertices in given graph. We aim to minimize the cost function.
Step 2: Graph contains n vertices. TSP finds a path covering all vertices exactly once, and minimize the overall traveling distance.
Step 3: Mathematical formula:
C (i, V) = min {d [i, j] + C(j, V – { j }) }, j ∈ V and i ∉ V.

Complexity: O((N^2) * (2^N))

Code:

```
// travelling salesman problem
#include <bits/stdc++.h>
#include <vector>
using namespace std;
#define V 4 // Total no of cities

int TSP(int Adj_matrix[][V], int s)
{
    // Store all cities apart from the source city in a vector.
    vector<int> cities;
    for (int i = 0; i < V; i++)
    if (i != s)
    cities.push_back(i);

    int min_distance = INT32_MAX;
    do
    {
        // Taking starting Path distance as zero
        int curr_distance = 0;
```

```
        // compute current path distance
        int k = s;
        for (int i = 0; i < cities.size(); i++)
            {
        curr_distance += Adj_matrix[k][cities[i]];
        k = cities[i];
        }
        curr_distance += Adj_matrix[k][s];

        // update minimum
        min_distance = min(min_distance, curr_distance);

    }
    //using following permutation method of C++
    while(next_permutation(cities.begin(), cities.end()));
    return min_distance;
}

int main()
{
    //The adjacent matrix of the graph given:
    int matrix[][V] = { { 0, 10, 15, 20 },{ 10, 0, 35, 25 },{ 15, 35, 0, 30
},{ 20, 25, 30, 0 } };
    int start = 0;
    cout <<"The minimum cost is: "<<TSP(matrix, start) << endl;
    return 0;
}
```

Ouput:

```
Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file> g++ .\q16.cpp
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file> .\a.exe
The minimum cost is: 80
PS C:\Users\Avneet\Desktop\NSUT\Sem3\DAA\DAA file>
```