Lecture 3: Recursion 1

1. **Power:** Write a program to find x to the power n (i.e. x^n). Take x and n from the user. You need to return the answer. Do this recursively.

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

int power(int x, int n) {
        if(n==0){
          return 1;
    }
    int SmallOutput = power(x, n-1);
    int Output = x * SmallOutput;
    return Output;
}

int main(){
    int x, n;
    cin >> x >> n;

    cout << power(x, n) << endl;
}
```

2. **Print Numbers:** Given is the code to print numbers from 1 to n in increasing order recursively. But it contains few bugs that you need to rectify such that all the test cases pass.

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

void print(int n){
    if(n == 1){
        cout << n << " ";
        return;
    }
    print(n-1);
    cout << n << " ";
}


int main(){
    int n;
    cin >> n;

    print(n);
}
```

3. **Number of Digits:** Given the code to find out and return the number of digits present in a number recursively. But it contains few bugs, that you need to rectify such that all the test cases should pass.

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

void print(int n){
    if(n == 1){
        cout << n << " ";
        return;
    }
    print(n-1);
    cout << n << " ";
}


int main(){
    int n;
    cin >> n;
    cout << count(n) << endl;
}
```

4. **Sum of Array:** Given an array of length N, you need to find and return the sum of all elements of the array. Do this recursively.

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

int sum(int input[], int n) {
      if(n == 0){
        return 0;
    }
    int SmallOutput = sum(input, n - 1);
    int Output = SmallOutput + input[n-1];
    return Output;
}

int main(){
    int n;
    cin >> n;
    int *input = new int[n];
    for(int i = 0; i < n; i++) {
        cin >> input[i];
    }
    cout << sum(input, n) << endl;
}
```

5. **Check Number:** Given an array of length N and an integer x, you need to find if x is present in the array or not. Return true or false. Do this recursively.

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

```
bool checkNumber(int input[], int size, int x) {
        if (size == 0){
            return false;
        }
        if (input[0] == x){
            return true;
        }
        bool present = checkNumber(input + 1, size - 1, x);
        return present;
}

int main(){
    int n;
    cin >> n;

    int *input = new int[n];

    for(int i = 0; i < n; i++) {
        cin >> input[i];
    }

    int x;

    cin >> x;

    if(checkNumber(input, n, x)) {
        cout << "true" << endl;
    }
    else {
        cout << "false" << endl;
    }
}
```

6. **First Index of Number:** Given an array of length N and an integer x, you need to find and return the first index of integer x present in the array. Return -1 if it is not present in the array. First index means the index of the first occurrence of x in the input array. Do this recursively. Indexing in the array starts from 0.

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

int firstIndex(int input[], int size, int x) {
    if (size == 0){
        return -1;
    }
    if (input[0] == x){
        return 0;
    }
    int SmallAns = firstIndex(input + 1, size - 1, x);
    return SmallAns ==  -1? SmallAns : SmallAns + 1;
}

int main(){
    int n;
    cin >> n;

    int *input = new int[n];

    for(int i = 0; i < n; i++) {
        cin >> input[i];
    }

    int x;

    cin >> x;

    cout << firstIndex(input, n, x) << endl;
}
```

7. **Last Index of Number:** Given an array of length N and an integer x, you need to find and return the last index of integer x present in the array. Return -1 if it is not present in the array. Last index means - if x is present multiple times in the array, return the index at which x comes last in the array.You should start traversing your array from 0, not from (N - 1).Do this recursively. Indexing in the array starts from 0.

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

int lastIndex(int input[], int size, int x) {
      if (size == 0){
        return -1;
    }
    if (input[size] == x){
        return size;
    }
    int SmallAns = lastIndex(input, size - 1, x);
    return SmallAns;
}

int main(){
    int n;
    cin >> n;

    int *input = new int[n];

    for(int i = 0; i < n; i++) {
        cin >> input[i];
    }
    int x;
    cin >> x;
    cout << lastIndex(input, n, x) << end;
}
```

8. **All Indices of Number:** Given an array of length N and an integer x, you need to find all the indexes where x is present in the input array. Save all the indexes in an array (in increasing order). Do this recursively. Indexing in the array starts from 0.

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

int allIndexes(int input[], int size, int x, int output[]) {
        if(size == 0){
         return 0;
    }
    int SmallAns = allIndexes(input + 1, size - 1, x, output);
    if(input[0] == x) {
        for(int i = SmallAns; i >= 0; i--){
            output[i + 1] = output[i] + 1;
        }
        output[0] = 0;
        SmallAns++;
    }
    else {
        for(int i = SmallAns - 1; i >= 0; i--) {
            output[i] = output[i] + 1;
        }
    }
    return SmallAns;
}

int main(){
    int n;
    cin >> n;

    int *input = new int[n];

    for(int i = 0; i < n; i++) {
        cin >> input[i];
    }

    int x;

    cin >> x;

    int *output = new int[n];

    int size = allIndexes(input, n, x, output);
    for(int i = 0; i < size; i++) {
        cout << output[i] << " ";
    }
    delete [] input;
    delete [] output;
}
```

9. **Multiplication (Recursive):** Given two integers M & N, calculate and return their multiplication using recursion. You can only use subtraction and addition for your calculation. No other operators are allowed.

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

int multiplyNumbers(int m, int n) {
       if (n==0){
        return 0;
    }
    int SmallAns = multiplyNumbers(m, n-1);
    int Ans = m + SmallAns;
    return Ans;
}

int main() {
    int m, n;
    cin >> m >> n;
    cout << multiplyNumbers(m, n) << endl;
}
```

10. **Count Zeros:** Given an integer N, count and return the number of zeros that are present in the given integer using recursion.

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

int countZeros(int n) {
    if (n == 0){
        return 0;
    }
    return (n % 10 == 0)? 1 + countZeros(n/10) : countZeros(n/10);
}

int main() {
    int n;
    cin >> n;
    cout << countZeros(n) << endl;
}
```

11. **Geometric Sum:** Given k, find the geometric sum i.e. `1 + 1/2 + 1/4 + 1/8 + ... + 1/(2^k)` using recursion.

```
#include <iostream>
#include <math.h>
#include <iomanip>
using namespace std;

double geometricSum(int k) {
    // Write your code here
       if (k == 0){
         return 1;
    }

    double SmallAns = geometricSum(k - 1);

    double power = 1;            // Calculating power term
    for(int i = 0; i < k ; i++) {
        power = power * 2;
    }

    double Ans = SmallAns + ( 1 / power );
    return Ans;
}

int main() {
    int k;
    cin >> k;
    cout << fixed << setprecision(5);
    cout << geometricSum(k) << endl;
}
```

12. **Check Palindrome (recursive):** Check whether a given String S is a palindrome using recursion. Return true or false.

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

bool helper(char input[], int s, int e) {
    if (s == e) {
        return true;
    }
    if (input[s] != input[e]){
        return false;
    }
    if (s < e + 1) {                    // to ensure for both odd and even
character long strings
    int SmallAns = helper(input, s + 1, e - 1);
    return SmallAns;
    }
}

bool checkPalindrome(char input[]) {
    int s = 0;
    int e = strlen(input) - 1;
    return helper(input, s, e);
}

int main() {
    char input[50];
    cin >> input;

    if(checkPalindrome(input)) {
        cout << "true" << endl;
    }
    else {
        cout << "false" << endl;
    }
}
```

13. **Sum of digits (recursive):** Write a recursive function that returns the sum of the digits of a given integer.

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

int sumOfDigits(int n) {
      if(n < 10){
        return n;
    }
    int SmallAns = sumOfDigits(n / 10);
    int Ans = SmallAns + (n % 10);
    return Ans;
}

int main() {
   int n;
   cin >> n;
   cout << sumOfDigits(n) << endl;
}
```

Assignment: Recursion 1b

1. **Replace pi (recursive):** Given a string, compute recursively a new string where all appearances of "pi" have been replaced by "3.14".

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

// Change in the given string itself. So no need to return or print anything
void replacePi(char input[]) {
        int size = strlen(input);
    if(input[0] == '\0') {
        return;
    }
    else if(input[0] == 'p' && input[1] == 'i') {
        for(int i = size; i > 1; i--) {
            input[i + 2] = input[i];
        }
        input[0] = '3';
        input[1] = '.';
        input[2] = '1';
        input[3] = '4';
        replacePi(input + 4);
    } else {
    replacePi(input + 1);
    }
}

int main() {
    char input[10000];
    cin.getline(input, 10000);
    replacePi(input);
    cout << input << endl;
}
```

2. **Remove X:** Given a string, compute recursively a new string where all 'x' chars have been removed.

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

void removeX(char input[]) {
        int size = strlen(input);
    if (input[0] == '\0') {
        return;
    }
    else if (input[0] == 'x') {
        for(int i = 0; i < size; i++) {
            input[i] = input [i + 1];
        }
        removeX(input);
    }
    else {
    removeX(input + 1);
    }
}

int main() {
    char input[100];
    cin.getline(input, 100);
    removeX(input);
    cout << input << endl;
}
```

3. **String to Integer:** Write a recursive function to convert a given string into the number it represents. That is input will be a numeric string that contains only numbers, you need to convert the string into corresponding integer and return the answer.

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

int helper(char input[], int size) {
    if(size == 1){
        return (input[0] - '0');     // returning int
    }
    int SmallAnswer = helper(input + 1, size - 1);
    return (input[0] - '0')*pow(10, size - 1) + SmallAnswer;
}

int stringToNumber(char input[]) {
    int size = strlen(input);
    return helper(input, size);
}

int main() {
    char input[50];
    cin >> input;
    cout << stringToNumber(input) << endl;
}
```

4. **Pair star:** Given a string S, compute recursively a new string where identical chars that are adjacent in the original string are separated from each other by a "*".

```
#include <iostream>
#include <cstring>
```

```
using namespace std;
void pairStar(char input[]) {
    int size = strlen(input);
    if(input[0] == '\0') {
        return;
    }
    else if(input[0] == input[1]) {
        for(int i = size; i >= 1; i--) {
            input[i + 1] = input[i];
        }
        input[1] = '*';
        pairStar(input + 2);
    } else {
    pairStar(input + 1);
    }
}

int main() {
   char input[100];
   cin.getline(input, 100);
   pairStar(input);
   cout << input << endl;
}
```

5. **Tower of Hanoi:** Tower of Hanoi is a mathematical puzzle where we have three rods and n disks. The objective of the puzzle is to move all disks from source rod to destination rod using third rod (say auxiliary). The rules are :

1) Only one disk can be moved at a time.
2) A disk can be moved only if it is on the top of a rod.
3) No disk can be placed on the top of a smaller disk.

Print the steps required to move n disks from source rod to destination rod. Source Rod is named as 'a', auxiliary rod as 'b' and destination rod as 'c'.

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

void towerOfHanoi(int n, char source, char auxiliary, char destination) {
    if (n == 0) {
      return;
    }
      if (n == 1)
    {
        cout << source << " " << destination <<endl;
        return;
    }
    towerOfHanoi(n - 1, source, destination, auxiliary);
    cout << source << " " << destination << endl;
    towerOfHanoi(n - 1, auxiliary, source, destination);
}

int main() {
    int n;
    cin >> n;
    towerOfHanoi(n, 'a', 'b', 'c');
}
```

Lecture 4: Recursion 2

1. **Replace Character Recursively:** Given an input string S and two characters c1 and c2, you need to replace every occurrence of character c1 with character c2 in the given string. Do this recursively.

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

void replaceCharacter(char input[], char c1, char c2) {
    if(input[0] == '\0'){
        return;
    }
    if(input[0] == c1) {
        input[0] = c2;
    }
    replaceCharacter(input + 1, c1, c2);
}

int main() {
    char input[1000000];
    char c1, c2;
    cin >> input;
    cin >> c1 >> c2;
    replaceCharacter(input, c1, c2);
    cout << input << endl;
}
```

2. **Remove Duplicates Recursively:** Given a string S, remove consecutive duplicates from it recursively.

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

void removeConsecutiveDuplicates(char *input) {
    if(input[0] == '\0'){
        return;
    }
    if(input[0] == input[1]) {
        for(int i = 0; input[i] != '\0'; i++) {
            input[i] = input[i + 1];
        }
        removeConsecutiveDuplicates(&input[0]);
    }
    else {
        removeConsecutiveDuplicates(&input[1]);
    }
}

int main() {
    char s[100000];
    cin >> s;
    removeConsecutiveDuplicates(s);
    cout << s << endl;
}
```

3. **Merge Sort Code:** Sort an array A using Merge Sort. Change in the input array itself. So no need to return or print anything.

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

void merge(int input[], int si, int ei) {
    int mid = (si + ei)/2;
    int i = si, j = mid + 1, k = si;
    int output[1000];
    while(i <= mid && j <= ei) {
        if(input[i] < input[j]) {
            output[k++] = input[i++];
        }
        else {
            output[k++] = input[j++];
        }
    }
    while(i <= mid) {
        output[k++] = input[i++];
    }
    while(j <= ei) {
        output[k++] = input[j++];
    }
    for(int z = si; z <= ei; z++){
        input[z] = output[z];
    }
}

void sort(int input[], int si, int ei) {
    if(si >= ei) {
        return;
    }
    int mid = (si + ei)/2;
    sort(input, si, mid);
    sort(input, mid + 1, ei);
    merge(input, si, ei);
}

void mergeSort(int input[], int size) {
    int si = 0;
    int ei = size - 1;
    sort(input, si, ei);
}
```

```
int main() {
  int length;
  cin >> length;
  int* input = new int[length];
  for(int i=0; i < length; i++)
    cin >> input[i];
  mergeSort(input, length);
  for(int i = 0; i < length; i++) {
    cout << input[i] << " ";
  }
}
```

4. **Quick Sort Code:** Sort an array A using Quick Sort. Change in the input array itself. So no need to return or print anything.

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

int partition(int input[], int si, int ei) {
    int SmallCount = 0;
    int x = input[si];
    for(int i = si + 1; i <= ei; i++) {
        if(input[i] < x) {
            SmallCount++;
        }
    }
    int temp1 = input[si];
    input[si] = input[si + SmallCount];
    input[si + SmallCount] = temp1;
    int i = si, j = ei, temp2;
    while (i < j) {
        if(input[i] < x) {
            i++;
        }
        else if (input[j] >= x) {
            j--;
        }
        else {
            temp2 = input[i];
            input[i] = input[j];
            input[j] = temp2;
            i++;
            j--;
        }
    }
    return (si + SmallCount);
}

void sort(int input[], int si, int ei) {
    if(si >= ei) {
        return;
    }
    int pivot = partition(input, si, ei);
    sort(input, si, pivot - 1);
    sort(input, pivot + 1, ei);
}

void quickSort(int input[], int size) {
  int si = 0;
  int ei = size - 1;
  sort(input, si, ei);
}

int main(){
    int n;
    cin >> n;

    int *input = new int[n];

    for(int i = 0; i < n; i++) {
        cin >> input[i];
    }

    quickSort(input, n);
    for(int i = 0; i < n; i++) {
        cout << input[i] << " ";
    }

    delete [] input;
}
```

5. **Return Keypad Code:** Given an integer n, using the phone keypad find out all the possible strings that can be made using digits of input n. Return empty string for numbers 0 and 1.

   Note : 1. The order of strings is not important. 2. Input and output has already been managed for you. You just have to populate the output array and return the count of elements populated in the output array.

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

int keypad(int num, string output[]){
 /* Insert all the possible combinations of the integer number into the output
string array. You do not need to print anything, just return the number of
strings inserted into the array.  */
    if(num == 0) {
        output[0] = "";
        return 1;
    }
    int SizeOfArray = keypad(num/10, output);
    if(num%10 == 2) {
        for(int i = 0; i < SizeOfArray; i++) {
            output[i + 2*SizeOfArray] = output[i] + 'c';
            output[i + SizeOfArray] = output[i] + 'b';
            output[i] = output[i] + 'a';
        }
        return 3*SizeOfArray;
    }
    else if(num%10 == 3) {
        for(int i = 0; i < SizeOfArray; i++) {
            output[i + 2*SizeOfArray] = output[i] + 'f';
            output[i + SizeOfArray] = output[i] + 'e';
            output[i] = output[i] + 'd';
        }
        return 3*SizeOfArray;
    }
    else if(num%10 == 4) {
        for(int i = 0; i < SizeOfArray; i++) {
            output[i + 2*SizeOfArray] = output[i] + 'i';
            output[i + SizeOfArray] = output[i] + 'h';
            output[i] = output[i] + 'g';
        }
        return 3*SizeOfArray;
    }
    else if(num%10 == 5) {
        for(int i = 0; i < SizeOfArray; i++) {
            output[i + 2*SizeOfArray] = output[i] + 'l';
            output[i + SizeOfArray] = output[i] + 'k';
            output[i] = output[i] + 'j';
        }
        return 3*SizeOfArray;
    }
    else if(num%10 == 6) {
        for(int i = 0; i < SizeOfArray; i++) {
            output[i + 2*SizeOfArray] = output[i] + 'o';
            output[i + SizeOfArray] = output[i] + 'n';
            output[i] = output[i] + 'm';
        }
        return 3*SizeOfArray;
    }
    else if(num%10 == 7) {
        for(int i = 0; i < SizeOfArray; i++) {
            output[i + 3*SizeOfArray] = output[i] + 's';
            output[i + 2*SizeOfArray] = output[i] + 'r';
            output[i + SizeOfArray] = output[i] + 'q';
            output[i] = output[i] + 'p';
        }
        return 4*SizeOfArray;
    }
    else if(num%10 == 8) {
        for(int i = 0; i < SizeOfArray; i++) {
            output[i + 2*SizeOfArray] = output[i] + 'v';
            output[i + SizeOfArray] = output[i] + 'u';
            output[i] = output[i] + 't';
        }
        return 3*SizeOfArray;
    }
    else if(num%10 == 9) {
        for(int i = 0; i < SizeOfArray; i++) {
            output[i + 3*SizeOfArray] = output[i] + 'z';
            output[i + 2*SizeOfArray] = output[i] + 'y';
            output[i + SizeOfArray] = output[i] + 'x';
            output[i] = output[i] + 'w';
        }
        return 4*SizeOfArray;
    }
}

int main(){
    int num;
    cin >> num;

    string output[10000];
    int count = keypad(num, output);
    for(int i = 0; i < count && i < 10000; i++){
        cout << output[i] << endl;
    }
    return 0;
}
```

6. **Print Keypad Combinations Code:** Given an integer n, using phone keypad find out and print all the possible strings that can be made using digits of input n.

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

void print(int num, string output) {
    if (num == 0) {
        cout<<output<<endl;
        return;
    }
    if (num%10 == 2) {
        print(num/10, 'a' + output);
        print(num/10, 'b' + output);
        print(num/10, 'c' + output);
    }
    else if (num%10 == 3) {
        print(num/10, 'd' + output);
        print(num/10, 'e' + output);
        print(num/10, 'f' + output);
    }
    else if (num%10 == 4) {
        print(num/10, 'g' + output);
        print(num/10, 'h' + output);
        print(num/10, 'i' + output);
    }
    else if (num%10 == 5) {
        print(num/10, 'j' + output);
        print(num/10, 'k' + output);
        print(num/10, 'l' + output);
    }
    else if (num%10 == 6) {
        print(num/10, 'm' + output);
        print(num/10, 'n' + output);
        print(num/10, 'o' + output);
    }
    else if (num%10 == 7) {
        print(num/10, 'p' + output);
        print(num/10, 'q' + output);
        print(num/10, 'r' + output);
        print(num/10, 's' + output);
    }
    else if (num%10 == 8) {
        print(num/10, 't' + output);
        print(num/10, 'u' + output);
        print(num/10, 'v' + output);
    }
    else if (num%10 == 9) {
        print(num/10, 'w' + output);
        print(num/10, 'x' + output);
        print(num/10, 'y' + output);
        print(num/10, 'z' + output);
    }
}

void printKeypad(int num){
    string output = "";
    print(num, output);
}

int main(){
    int num;
    cin >> num;
    printKeypad(num);
    return 0;
}
```

7. **Check AB:** Suppose you have a string, S, made up of only 'a's and 'b's. Write a recursive function that checks if the string was generated using the following rules:
   a. The string begins with an 'a'
   b. Each 'a' is followed by nothing or an 'a' or "bb"
   c. Each "bb" is followed by nothing or an 'a'
   If all the rules are followed by the given string, return true otherwise return false.

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

bool checkAB(char input[]) {
    if (strlen(input) == 0) {
        return true;
    }
    if (strlen(input) == 1) {
        if(input[0] == 'a') {
            return true;
        }
        else return false;
    }
    if(input[0] == 'a') {
        if(input[1] == 'a') {
            return checkAB(input + 1);
        }
        else if(input[1]=='b' && input[2]=='b')
            return checkAB(input+3);
        else return false;
    }
      return false;
}

int main() {
    char input[100];
    bool ans;
    cin >> input;
    ans=checkAB(input);
    if(ans)
        cout<< "true" << endl;
    else
        cout<< "false" << endl;
}
```

8. **Staircase:** A child is running up a staircase with N steps, and can hop either 1 step, 2 steps or 3 steps at a time. Implement a method to count how many possible ways the child can run up to the stairs. You need to return the number of possible ways W.

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

```
int staircase(int n){
    if (n < 0){
        return 0;
    }
    if (n == 0){
        return 1;
    }
    int x = staircase(n - 1);
    int y = staircase(n - 2);
    int z = staircase(n - 3);
    return x + y +z;
}

int main() {
    int n, output;
    cin >> n;
    output=staircase(n);
    cout<< output <<endl;
}
```

9. **Binary Search (Recursive):** Given an integer sorted array (sorted in increasing order) and an element x, find the x in the given array using binary search. Return the index of x. Return -1 if x is not present in the given array. Note : If given array size is even, take first mid.

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

// input - input array
// size - length of input array
// element - value to be searched
int Search(int input[], int si, int ei, int element) {
    if(si > ei){
        return -1;
    }
    int mid = (si + ei)/2;
    if(input[mid] == element) {
        return mid;
    }
    else if(input[mid] > element) {
        return Search(input, si, mid - 1, element);
    }
    else if(input[mid] < element) {
        return Search(input, mid + 1, ei, element);
    }
}

int binarySearch(int input[], int size, int element) {
    int si = 0;
    int ei = size - 1;
    return Search(input, si, ei, element);
}

int main() {
    int input[100000],length,element, ans;
    cin >> length;
    for(int i =0;i<length;i++)
    {
        cin >> input[i];;
    }

    cin>>element;
    ans = binarySearch(input, length, element);
    cout<< ans << endl;
}
```

10. **Return subset of an array:** Given an integer array (of length n), find and return all the subsets of the input array.
    Subsets are of length varying from 0 to n, that contain elements of the array. But the order of elements should remain the same as in the input array.
    Note : The order of subsets are not important.

```
/***
You need to save all the subsets in the given 2D output array. And return the
number of subsets(i.e. number of rows filled in output) from the given function.

In ith row of the output array, the 1st column contains the length of the ith
subset. And from the 1st column the actual subset follows.
For eg. Input : {1, 2}, then output should contain
       {{0},         // Length of this subset is 0
       {1, 2},               // Length of this subset is 1
       {1, 1},               // Length of this subset is also 1
       {2, 1, 2}}    // Length of this subset is 2

Don't print the subsets, just save them in output.
***/

#include <iostream>
using namespace std;

int subset(int input[], int n, int output[][20]) {
    // Write your code here
      if(n == 0) {
        output[0][0] = 0;    //size of 1st row
        return 1;
    }
    int SmallAns = subset(input + 1, n - 1, output);
    for(int i = 0; i < SmallAns; i++) {
        output[i + SmallAns][0] = 1 + output[i][0];
        output[i + SmallAns][1] = input[0];
        for(int j = 1; j <= output[i][0]; j++) {
            output[i + SmallAns][j + 1] = output[i][j];
        }
    }
    return 2*SmallAns;
}

int main() {
  int input[20],length, output[35000][20];
  cin >> length;
  for(int i=0; i < length; i++)
    cin >> input[i];

  int size = subset(input, length, output);

  for( int i = 0; i < size; i++) {
       for( int j = 1; j <= output[i][0]; j++) {
              cout << output[i][j] << " ";
    }
    cout << endl;
  }
}
```

11. **Print Subsets of Array:** Given an integer array (of length n), find and print all the subsets of the input array.
Subsets are of length varying from 0 to n, that contain elements of the array. But the order of elements should remain the same as in the input array.
Note : The order of subsets are not important. Just print the subsets in different lines.

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

void subset(int input[], int size, int output[], int sizo) {
       if(size == 0) {
           for(int i = 1; i <= sizo; i++) {
               cout<<output[i]<<" ";
           }
           cout<<endl;
           return;
       }
       int newOutput[1000];
       for(int j = 0; j <= sizo; j++) {
           newOutput[j] = output[j];
       }
       newOutput[sizo + 1] = input[0];
       subset(input + 1, size - 1, newOutput, sizo + 1);
       subset(input + 1, size - 1, output, sizo);
}

void printSubsetsOfArray(int input[], int size) {
    int output[1000];
    int sizo = 0;
    subset(input, size, output, sizo);
}

int main() {
  int input[1000],length;
  cin >> length;
  for(int i=0; i < length; i++)
       cin >> input[i];
  printSubsetsOfArray(input, length);
}
```

12. **Return subsets sum to K:** Given an array A of size n and an integer K, return all subsets of A which sum to K.
    Subsets are of length varying from 0 to n, that contain elements of the array. But the order of elements should remain the same as in the input array.

```
/***
You need to save all the subsets in the given 2D output array. And return the
number of subsets(i.e. number of rows filled in output) from the given function.

In ith row of the output array, the 1st column contains the length of the ith
subset. And from the 1st column the actual subset follows.
For eg. Input : {1, 3, 4, 2} and K = 5, then output array should contain
       {{2, 1, 4},  // Length of this subset is 2
       {2, 3, 2}}   // Length of this subset is 2

Don't print the subsets, just save them in output.
***/

#include <iostream>
using namespace std;

int subsetSumToK(int input[], int n, int output[][50], int k) {
       if(n == 0) {
        if(k == 0) {
            output[0][0] = 0;
            return 1;
        }
        else return 0;
    }
    int SmallOutput1[10000][50];                    //With input[0] included
    int SmallAns1 = subsetSumToK(input + 1, n - 1, SmallOutput1, k - input[0]);

    for(int i = 0; i < SmallAns1; i++) {            //copying to output
        output[i][0] = 1 + SmallOutput1[i][0];
        output[i][1] = input[0];
        for(int j = 1; j <= SmallOutput1[i][0]; j++) {
            output[i][j + 1] = SmallOutput1[i][j];
        }
    }

    int SmallOutput2[10000][50];                    //Without input[0] included
    int SmallAns2 = subsetSumToK(input + 1, n - 1, SmallOutput2, k);

    for(int i = 0; i < SmallAns2; i++) {            //copying to output
        for(int j = 0; j <= SmallOutput2[i][0]; j++) {
            output[i + SmallAns1][j] = SmallOutput2[i][j];
        }
    }
    return SmallAns1+SmallAns2;
}

int main() {
  int input[20],length, output[10000][50], k;
  cin >> length;
  for(int i=0; i < length; i++)
    cin >> input[i];

  cin >> k;

  int size = subsetSumToK(input, length, output, k);

  for( int i = 0; i < size; i++) {
      for( int j = 1; j <= output[i][0]; j++) {
            cout << output[i][j] << " ";
    }
    cout << endl;
  }
}
```

13. **Print Subset Sum to K:** Given an array A and an integer K, print all subsets of A which sum to K.
    Subsets are of length varying from 0 to n, that contain elements of the array. But the order of elements should remain the same as in the input array.

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

void print(int input[], int size, int output[], int sizo, int k) {
    if(size == 0) {
        if(k == 0) {
            for(int i = 1; i <= sizo; i++) {
                cout<<output[i]<<" ";
            }
            cout<<endl;
            return;
        }
        else return;
    }

    int newOutput[1000];                  //Creating array with input[0] included
      for(int j = 0; j <= sizo; j++) {
          newOutput[j] = output[j];
      }
      newOutput[sizo + 1] = input[0];

      print(input + 1, size - 1, newOutput, sizo + 1, k - input[0]);
                                                        //With input[0] included
      print(input + 1, size - 1, output, sizo, k);
                                                     //Without input[0] included
}

void printSubsetSumToK(int input[], int size, int k) {
    int output[1000];
    int sizo = 0;
    print(input, size, output, sizo, k);
}

int main() {
  int input[1000],length,k;
  cin >> length;
  for(int i=0; i < length; i++)
    cin >> input[i];
  cin>>k;
  printSubsetSumToK(input, length,k);
}
```

14. **Return all codes - String:** Assume that the value of a = 1, b = 2, c = 3, ... , z = 26. You are given a numeric string S. Write a program to return the list of all possible codes that can be generated from the given string.

```
/* You are given the input text and output string array. Find all possible codes
and store in the output string array. Don't print the codes. Also, return the
number of codes return to the output string. You do not need to print anything.
*/

#include <iostream>
#include <string.h>
using namespace std;

int getCodes(string input, string output[10000]) {
    if(input.empty()) {
        output[0] = "";
        return 1;
    }
    if(input.size() == 1) {
        output[0] = char('a' + (input[0] - '0') - 1);
        return 1;
    }

    string output1[10000], output2[10000];
    int Result1 = getCodes(input.substr(1), output1);

    int Result2 = 0;

    if(input.size() > 1) {
        if ((((input[0] - '0')*10) + (input[1] - '0')) >= 10 && (((input[0] -
'0')*10) + (input[1] - '0')) <= 26) {
            Result2 = getCodes(input.substr(2), output2);
        }
    }

    int k = 0;
    for(int i = 0; i < Result1; i++) {
        output[k++] = char('a' + (input[0] - '0') - 1) + output1[i];
    }
    for(int j = 0; j < Result2; j++) {
        output[k++] = char('a' + ((input[0] - '0')*10) + (input[1] - '0')  - 1)
+ output2[j];
    }

    return Result1 + Result2;
}

int main(){
    string input;
    cin >> input;

    string output[10000];
    int count = getCodes(input, output);
    for(int i = 0; i < count && i < 10000; i++)
        cout << output[i] << endl;
    return 0;
}
```

15. **Print all Codes - String:** Assume that the value of a = 1, b = 2, c = 3, ... , z = 26. You are given a numeric string S. Write a program to print the list of all possible codes that can be generated from the given string.

```
/* Given the input as a string, print all its possible combinations. You do not
need to return anything. */

#include <iostream>
#include <string.h>
using namespace std;

void getCodes(string input, string output) {
    if(input.empty()) {
        cout<<output<<endl;
        return;
    }

    getCodes(input.substr(1), output + char('a' + (input[0] - '0') - 1));

    if(input.size() > 1) {
        if ((((input[0] - '0')*10) + (input[1] - '0')) >= 10 && (((input[0] -
'0')*10) + (input[1] - '0')) <= 26) {
            getCodes(input.substr(2), output + char('a' + ((input[0] - '0')*10)
+ (input[1] - '0')  - 1));
        }
    }
}

void printAllPossibleCodes(string input) {
    string output = "";
    getCodes(input, output);
}

int main(){
    string input;
    cin >> input;

    printAllPossibleCodes(input);
    return 0;
}
```

16. **Return Permutations - String:** Given a string S, find and return all the possible permutations of the input string.
    Note 1:The order of permutations is not important.
    Note 2:If the original string contains duplicate characters, permutations will also be duplicates.

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

int returnPermutations(string input, string output[]){
    if(input.size() == 0) {
          output[0] = "";
          return 1;
      }

      string SmallOutput[1000];
      int k = 0;
      int SmallAns = returnPermutations(input.substr(1), SmallOutput);
      for(int i = 0; i < SmallAns; i++) {
          for(int j = 0; j <= SmallOutput[i].size(); j++) {
              output[k++] = SmallOutput[i].substr(0, j) + input[0] +
SmallOutput[i].substr(j);
```

```
            }
        }
        return k;
}


using namespace std;

int main(){
    string input;
    cin >> input;
    string output[10000];
    int count = returnPermutations(input, output);
    for(int i = 0; i < count && i < 10000; i++){
        cout << output[i] << endl;
    }
    return 0;
}
```

17. **Print Permutations:** Given an input string (STR), find and return all possible permutations of the input string.
    Note: The input string may contain the same characters, so there will also be the same permutations. The order of permutations doesn't matter.

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

void print(string input, string output){
        if(input.size() == 0) {
            cout<<output<<endl;
            return;
        }
        for(int i = 0; i < input.size(); i++) {
            print(input.substr(0, i) + input.substr(i + 1) ,  input[i] + output);
        }
}

void printPermutations(string input){
    string output = "";
    print(input, output);
}

int main() {
    string input;
    cin >> input;
    printPermutations(input);
    return 0;
}
```

Lecture 5 : Time and Space Complexity Analysis

1. **Find the Unique Element:** You have been given an integer array/list(ARR) of size N. Where N is equal to [2M + 1]. Now, in the given array/list, 'M' numbers are present twice and one number is present only once. You need to find and return that number which is unique in the array/list.
   Note: Unique element is always present in the array/list according to the given condition.

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

int findUnique(int *arr, int n) {
    int num = 0;
    for(int i = 0 ; i < n; i++) {
        num = num^arr[i];
    }
    return num;
}

int main() {
    int t;
    cin >> t;

    while (t--) {
        int size;
        cin >> size;
        int *input = new int[size];

        for (int i = 0; i < size; ++i) {
            cin >> input[i];
        }

        cout << findUnique(input, size) << endl;
    }
    return 0;
}
```

2. **Duplicate in array:** You have been given an integer array/list(ARR) of size N which contains numbers from 0 to (N - 2). Each number is present at least once. That is, if N = 5, the array/list constitutes values ranging from 0 to 3, and among these, there is a single integer value that is present twice. You need to find and return that duplicate number present in the array.

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

int findDuplicate(int *arr, int n)
{
    int sumA = 0;
    for(int i = 0; i < n; i++) {
        sumA = sumA + arr[i];
    }
    int sumN = (n - 2)*(n - 1)/2;
    return sumA - sumN;
}


int main()
{
        int t;
        cin >> t;
        while (t--)
        {
                int size;
                cin >> size;
                int *input = new int[size];
                for (int i = 0; i < size; i++)
                {
                        cin >> input[i];
                }
                cout << findDuplicate(input, size) << endl;
        }
        return 0;
}
```

3. **Array Intersection:** You have been given two integer arrays/list(ARR1 and ARR2) of size N and M, respectively. You need to print their intersection; An intersection for this problem can be defined when both the arrays/lists contain a particular value or to put it in other words, when there is a common value that exists in both the arrays/lists.
   Note : Input arrays/lists can contain duplicate elements. The intersection elements printed would be in the order they appear in the first sorted array/list(ARR1).

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

void merge(int *input, int si, int ei) {
    int mid = (si + ei)/2;
    int i = si, j = mid + 1, k = si;
    int output[10000];
    while(i <= mid && j <= ei) {
        if(input[i] < input[j]) {
            output[k++] = input[i++];
        }
        else {
            output[k++] = input[j++];
        }
    }
    while(i <= mid) {
```

```cpp
        output[k++] = input[i++];
    }
    while(j <= ei) {
        output[k++] = input[j++];
    }
    for(int z = si; z <= ei; z++){
        input[z] = output[z];
    }
}

void sort(int *input, int si, int ei) {
    if(si >= ei) {
        return;
    }
    int mid = (si + ei)/2;
    sort(input, si, mid);
    sort(input, mid + 1, ei);
    merge(input, si, ei);
}

void mergeSort(int *input, int size) {
    int si = 0;
    int ei = size - 1;
    sort(input, si, ei);
}

void intersection(int *arr1, int *arr2, int n, int m) {
    mergeSort(arr1, n);
    mergeSort(arr2, m);
    int i = 0, j = 0;
    while(i < n && j < m) {
        if(arr1[i] == arr2[j]) {
            cout<<arr1[i]<<" ";
            i++;
            j++;
        }
        else if(arr1[i] < arr2[j]) {
            i++;
        }
        else if(arr1[i] > arr2[j]) {
            j++;
        }
    }
}

int main()
{
      int t;
      cin >> t;
      while (t--)
      {
```

```
		int size1, size2;

		cin >> size1;
		int *input1 = new int[size1];

		for (int i = 0; i < size1; i++)
		{
			cin >> input1[i];
		}

		cin >> size2;
		int *input2 = new int[size2];

		for (int i = 0; i < size2; i++)
		{
			cin >> input2[i];
		}

		intersection(input1, input2, size1, size2);

		delete[] input1;
		delete[] input2;

		cout << endl;
	}
	return 0;
}
```

4. **Pair sum in array:** You have been given an integer array/list(ARR) and a number 'num'. Find and return the total number of pairs in the array/list which sum to 'num'.

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

void merge(int *input, int si, int ei) {
    int mid = (si + ei)/2;
    int i = si, j = mid + 1, k = si;
    int output[10000];
    while(i <= mid && j <= ei) {
        if(input[i] < input[j]) {
            output[k++] = input[i++];
        }
        else {
            output[k++] = input[j++];
        }
    }
    while(i <= mid) {
        output[k++] = input[i++];
    }
    while(j <= ei) {
        output[k++] = input[j++];
    }
    for(int z = si; z <= ei; z++){
        input[z] = output[z];
    }
}

void sort(int *input, int si, int ei) {
    if(si >= ei) {
        return;
    }
    int mid = (si + ei)/2;
    sort(input, si, mid);
    sort(input, mid + 1, ei);
    merge(input, si, ei);
}

void mergeSort(int *input, int size) {
    int si = 0;
    int ei = size - 1;
    sort(input, si, ei);
}

int countp(int *input, int i) {
    int temp = input[i];
    int countpo = 0;
    while(input[i] == temp) {
        countpo++;
        i++;
    }
    return countpo;
}

int countn(int *input, int j) {
    int temp = input[j];
    int countne = 0;
    while(input[j] == temp) {
        countne++;
        j--;
    }
    return countne;
}

int tripletSum(int *arr, int n, int num)
{
    mergeSort(arr, n);
    int count = 0;
    for(int k = 0; k < n - 2; k++) {
        int sum = num - arr[k];
        int i = k + 1;
        int j = n - 1;
        while(i < j) {
            if(arr[i] + arr[j] == sum){
                if(arr[i] == arr[j]) {
                    count = count + ((countp(arr, i))*(countp(arr, i) - 1)/2);
                    i = i + countp(arr, i);
                }
                else if(countp(arr, i) == 1 && countn(arr, j) == 1) {
                    count++;
                    i++;
                    j--;
                }
                else {
                    count = count + (countp(arr, i) * countn(arr, j));
                    i = i + countp(arr, i);
                    j = j - countn(arr, j);
                }
            }
            else if(arr[i] + arr[j] < sum) {
                i++;
            }
            else if(arr[i] + arr[j] > sum) {
                j--;
            }
        }
    }
    return count;
}

int main()
{
      int t;
      cin >> t;
      while (t--)
      {
             int size;
             int x;

             cin >> size;
             int *input = new int[size];

             for (int i = 0; i < size; i++)
             {
                   cin >> input[i];
             }
             cin >> x;
             cout << pairSum(input, size, x) << endl;
             delete[] input;
      }
      return 0;
}
```

5. **Triplet sum:** You have been given a random integer array/list(ARR) and a number X. Find and return the triplet(s) in the array/list which sum to X.

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

void merge(int *input, int si, int ei) {
    int mid = (si + ei)/2;
    int i = si, j = mid + 1, k = si;
    int output[10000];
    while(i <= mid && j <= ei) {
        if(input[i] < input[j]) {
            output[k++] = input[i++];
        }
        else {
            output[k++] = input[j++];
        }
    }
    while(i <= mid) {
        output[k++] = input[i++];
    }
    while(j <= ei) {
        output[k++] = input[j++];
    }
    for(int z = si; z <= ei; z++){
        input[z] = output[z];
    }
}

void sort(int *input, int si, int ei) {
    if(si >= ei) {
        return;
    }
    int mid = (si + ei)/2;
    sort(input, si, mid);
    sort(input, mid + 1, ei);
    merge(input, si, ei);
}

void mergeSort(int *input, int size) {
    int si = 0;
    int ei = size - 1;
    sort(input, si, ei);
}

int countp(int *input, int i) {
    int temp = input[i];
    int countpo = 0;
    while(input[i] == temp) {
        countpo++;
        i++;
    }
    return countpo;
}

int countn(int *input, int j) {
    int temp = input[j];
    int countne = 0;
    while(input[j] == temp) {
        countne++;
        j--;
    }
    return countne;
}

int tripletSum(int *arr, int n, int num)
{
    mergeSort(arr, n);
    int count = 0;
    for(int k = 0; k < n - 2; k++) {
        int sum = num - arr[k];
        int i = k + 1;
        int j = n - 1;
        while(i < j) {
            if(arr[i] + arr[j] == sum){
                if(arr[i] == arr[j]) {
                    count = count + ((countp(arr, i))*(countp(arr, i) - 1)/2);
                    i = i + countp(arr, i);
                }
                else if(countp(arr, i) == 1 && countn(arr, j) == 1) {
                    count++;
                    i++;
                    j--;
                }
                else {
                    count = count + (countp(arr, i) * countn(arr, j));
                    i = i + countp(arr, i);
                    j = j - countn(arr, j);
                }
            }
            else if(arr[i] + arr[j] < sum) {
                i++;
            }
            else if(arr[i] + arr[j] > sum) {
                j--;
            }
        }
    }
    return count;
}
```

```
int main()
{
	int t;
	cin >> t;

	while (t--)
	{
		int size;
		int x;
		cin >> size;

		int *input = new int[size];

		for (int i = 0; i < size; i++)
		{
			cin >> input[i];
		}
		cin >> x;
		cout << tripletSum(input, size, x) << endl;
		delete[] input;
	}
	return 0;
}
```

6. **Rotate array:** You have been given a random integer array/list(ARR) of size N. Write a function that rotates the given array/list by D elements(towards the left).

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

void rot(int *input, int si, int ei) {
    int temp;
    while(si < ei) {
        temp = input[si];
        input[si] = input[ei];
        input[ei] = temp;
        si++;
        ei--;
    }
}

void rotate(int *input, int d, int n)
{
    rot(input, 0, n - 1);
    rot(input, 0, n - d - 1);
    rot(input, n - d, n - 1);
}
```

```
int main()
{
	int t;
	cin >> t;

	while (t--)
	{
		int size;
		cin >> size;
		int *input = new int[size];

		for (int i = 0; i < size; ++i)
		{
			cin >> input[i];
		}

		int d;
		cin >> d;

		rotate(input, d, size);

		for (int i = 0; i < size; ++i)
		{
			cout << input[i] << " ";
		}
		delete[] input;
		cout << endl;
	}
	return 0;
}
```

7. **Check Array Rotation:** You have been given an integer array/list(ARR) of size N. It has been sorted(in increasing order) and then rotated by some number 'K' in the clockwise direction. Your task is to write a function that returns the value of 'K', that means, the index from which the array/list has been rotated.

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

int arrayRotateCheck(int *input, int size)
{
    int i = 0;
    while(input[i] < input[i + 1]) {
        i++;
    }
    if(i == 0){
        return 0;
    }
    if(i == size - 1){
        return 0;
    }
    else return i + 1;
}


int main()
{
	int t;
	cin >> t;
	while (t--)
	{
		int size;
		cin >> size;
		int *input = new int[size];
		for (int i = 0; i < size; i++)
		{
			cin >> input[i];
		}
		cout << arrayRotateCheck(input, size) << endl;
		delete[] input;
	}
	return 0;
}
```

Lecture 6: OOPS 1

Complex Number Class: A ComplexNumber class contains two data members : one is the real part (R) and the other is imaginary (I) (both integers).
Implement the Complex numbers class that contains following functions -

1. constructor

You need to create the appropriate constructor.

2. plus -

This function adds two given complex numbers and updates the first complex number.

E.g. if C1 = 4 + i5 and C2 = 3 +i1

C1.plus(C2) results in:

C1 = 7 + i6 and C2 = 3 + i1

3. multiply -

This function multiplies two given complex numbers and updates the first complex number.

E.g. if C1 = 4 + i5 and C2 = 1 + i2

C1.multiply(C2) results in:

C1 = -6 + i13 and C2 = 1 + i2

4. print -

This function prints the given complex number in the following format : a + ib

Note : There is space before and after '+' (plus sign) and no space between 'i' (iota symbol) and b.

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

class ComplexNumbers {
    // Complete this class
    private:
        int Real, Imaginary;
    public:
        ComplexNumbers(int Real, int Imaginary) {
            this -> Real = Real;
            this -> Imaginary = Imaginary;
        }

        void print() {
            cout<< Real << " + i" << Imaginary;
        }

        void plus(ComplexNumbers const &C2) {
            int R = Real + C2.Real;
            int I = Imaginary + C2.Imaginary;
            Real = R;
            Imaginary = I;
        }

        void multiply(ComplexNumbers const &C2) {
            int R = (Real * C2.Real) - (Imaginary * C2.Imaginary);
            int I = (Real * C2.Imaginary) + (Imaginary * C2.Real);
            Real = R;
            Imaginary = I;
        }
};


int main() {
    int real1, imaginary1, real2, imaginary2;

    cin >> real1 >> imaginary1;
    cin >> real2 >> imaginary2;

    ComplexNumbers c1(real1, imaginary1);
    ComplexNumbers c2(real2, imaginary2);

    int choice;
    cin >> choice;

    if(choice == 1) {
        c1.plus(c2);
        c1.print();
    }
    else if(choice == 2) {
        c1.multiply(c2);
        c1.print();
    }
    else {
        return 0;
    }
}
```

Lecture 7: OOPS 2

Polynomial Class: Implement a polynomial class, with following properties and functions.
Properties :
1. An integer (lets say A) which holds the coefficient and degrees. Use array indices as degree and A[i] as coefficient of ith degree.
2. An integer holding total size of array A.

Functions :
1. Default constructor
2. Copy constructor
3. setCoefficient -
This function sets coefficient for a particular degree value. If the given degree is greater than the current capacity of polynomial, increase the capacity accordingly and add then set the required coefficient. If the degree is within limits, then previous coefficient value is replaced by given coefficient value

4. Overload "+" operator (P3 = P1 + P2) :
Adds two polynomials and returns a new polynomial which has result.

5. Overload "-" operator (P3 = p1 - p2) :
Subtracts two polynomials and returns a new polynomial which has result

6. Overload * operator (P3 = P1 * P2) :
Multiplies two polynomials and returns a new polynomial which has result

7. Overload "=" operator (Copy assignment operator) -
Assigns all values of one polynomial to other.

8. print() -
Prints all the terms (only terms with non zero coefficients are to be printed) in increasing order of degree.

Print pattern for a single term : <coefficient>"x"<degree>
And multiple terms should be printed separated by space. And after printing one polynomial, print new line. For more clarity, refer sample test cases

```
/* C++ implementation to convert infix expression to postfix*/
// Note that here we use std::stack  for Stack operations
#include <vector>
#include <climits>
#include <iostream>
using namespace std;

class Polynomial {
    public:
    int *degCoeff;
    int Capacity;

    Polynomial() {
        degCoeff = new int[5];
        for(int i = 0; i < 5; i++) {
            degCoeff[i] = 0;
        }
        Capacity = 5;
    }

    Polynomial(Polynomial const &P) {
        this -> degCoeff = new int[P.Capacity];
        for(int i = 0; i < P.Capacity; i++) {
            this -> degCoeff[i] = P.degCoeff[i];
        }
        this -> Capacity = P.Capacity;
    }

    void setCoefficient(int degree, int coeff) {
        if (degree >= Capacity) {
            int TempCapacity = Capacity;
            while(degree >= TempCapacity) {
                TempCapacity = TempCapacity * 2;
            }
            int *NdegCoeff = new int[TempCapacity];
            for(int i = 0; i < TempCapacity; i++) {
                NdegCoeff[i] = 0;
            }
            for(int i = 0; i < Capacity; i++) {
                NdegCoeff[i] = degCoeff[i];
            }
            delete [] degCoeff;
            degCoeff = NdegCoeff;
            Capacity = TempCapacity;
        }
        degCoeff[degree] = coeff;
    }

    Polynomial  operator=(Polynomial const &P) {
        this -> degCoeff = new int[P.Capacity];
        for(int i = 0; i < P.Capacity; i++) {
            this -> degCoeff[i] = P.degCoeff[i];
        }
        this -> Capacity = P.Capacity;
    }

    Polynomial operator+(Polynomial const &P1) {
        Polynomial PNew;
        for(int i = 0; i < this -> Capacity; i++) {
            PNew.addCoefficient(i, this -> degCoeff[i]);
        }
```

```
        for(int j = 0; j < P1.Capacity; j++) {
            PNew.addCoefficient(j, P1.degCoeff[j]);
        }
        return PNew;
    }

    Polynomial operator-(Polynomial const &P2) {
        Polynomial PNew;
        for(int i = 0; i < this -> Capacity; i++) {
            PNew.addCoefficient(i, this -> degCoeff[i]);
        }
        for(int j = 0; j < P2.Capacity; j++) {
            PNew.addCoefficient(j, - P2.degCoeff[j]);
        }
        return PNew;
    }

    void addCoefficient(int degree, int coeff) {
        if (degree >= Capacity) {
            int TempCapacity = Capacity;
            while(degree >= TempCapacity) {
                TempCapacity = TempCapacity * 2;
            }
            int *NdegCoeff = new int[TempCapacity];
            for(int i = 0; i < TempCapacity; i++) {
                NdegCoeff[i] = 0;
            }
            for(int i = 0; i < Capacity; i++) {
                NdegCoeff[i] = degCoeff[i];
            }
            delete [] degCoeff;
            degCoeff = NdegCoeff;
            Capacity = TempCapacity;
        }
        degCoeff[degree] = degCoeff[degree] + coeff;
    }

    Polynomial operator*(Polynomial const &P3) {
        Polynomial PNew;
        for(int i = 0; i < Capacity; i++) {
            for(int j = 0; j < P3.Capacity; j++) {
                PNew.addCoefficient(i + j, this -> degCoeff[i] *
P3.degCoeff[j]);
            }
        }
        return PNew;
    }

    void print() const {
        for(int i = 0; i < Capacity; i++) {
            if(degCoeff[i] != 0) {
                cout<< degCoeff[i] << "x" << i << " ";
            }
        }
        cout<<endl;
    }
};

//Driver program to test above functions
int main()
{
    int count1,count2,choice;
    cin >> count1;

    int *degree1 = new int[count1];
    int *coeff1 = new int[count1];

    for(int i=0;i < count1; i++) {
        cin >> degree1[i];
    }

    for(int i=0;i < count1; i++) {
        cin >> coeff1[i];
    }

    Polynomial first;
    for(int i = 0; i < count1; i++){
        first.setCoefficient(degree1[i],coeff1[i]);
    }

    cin >> count2;

    int *degree2 = new int[count2];
    int *coeff2 = new int[count2];

    for(int i=0;i < count2; i++) {
        cin >> degree2[i];
    }

    for(int i=0;i < count2; i++) {
        cin >> coeff2[i];
    }

    Polynomial second;
    for(int i = 0; i < count2; i++){
        second.setCoefficient(degree2[i],coeff2[i]);
    }

    cin >> choice;

    switch(choice){
            // Add
        case 1:
        {
            Polynomial result1 = first + second;
            result1.print();
            break;
        }
            // Subtract
        case 2 :
        {
            Polynomial result2 = first - second;
            result2.print();
            break;
        }
            // Multiply
        case 3 :
        {
            Polynomial result3 = first * second;
            result3.print();
            break;
        }
        case 4 : // Copy constructor
        {
            Polynomial third(first);
            if(third.degCoeff == first.degCoeff) {
                cout << "false" << endl;
            }
            else {
                cout << "true" << endl;
            }
            break;
        }

        case 5 : // Copy assignment operator
        {
            Polynomial fourth(first);
            if(fourth.degCoeff == first.degCoeff) {
                cout << "false" << endl;
            }
            else {
                cout << "true" << endl;
            }
            break;
        }

    }

    return 0;
}
```

Lecture 8 : Linked List 1

```
class Node
{
public:
        int data;
        Node *next;
        Node(int data)
        {
                this->data = data;
                this->next = NULL;
        }
};
```

1. **Length of LL:** For a given singly linked list of integers, find and return its length. Do it using an iterative method.

```
int length(Node *head)
{
    if(head == NULL) {            //Empty LL
        return 0;
    }
    else {
        int count = 1;
        while(head->next != NULL) {
            count++;
            head = head->next;
        }
        return count;
    }
}
```

2. **Print ith node:** For a given singly linked list of integers and a position 'i', print the node data at the 'i-th' position.
   Note : Assume that the Indexing for the singly linked list always starts from 0. If the given position 'i', is greater than the length of the given singly linked list, then don't print anything.

```
void printIthNode(Node *head, int i)
{
    int count = 0;
    while(head != NULL) {
        if(count == i) {
            cout<<head->data;
        }
        head = head->next;
        count++;
    }
}
```

3. **Delete node:** You have been given a linked list of integers. Your task is to write a function that deletes a node from a given position, 'pos'.

```
Node *deleteNode(Node *head, int pos)
{
    int count = 0;
    Node *temp = head;

    if(pos == 0) {
        head = head->next;
        return head;
    }
    while(temp->next != NULL && count < pos - 1) {     //Traversal to reach
(i-1)th position
        temp = temp->next;
        count++;
    }
    if(temp->next != NULL) {
        Node *a = temp->next;
        Node *b = a->next;
        temp->next = b;
        delete a;         //Node to be deleted
    }
    return head;
}
```

4. **Length of LL (recursive):** Given a linked list, find and return the length of the given linked list recursively.

```
int helper(Node *head, int count) {
    Node *temp = head;
    if(temp == NULL) {
        return count;
    }
    temp = temp->next;
    count++;
    helper(temp, count);
}

int length(Node *head) {
    int count = 0;
    return helper(head, count);
}
```

5. **Insert node (recursive):** You have been given a linked list of integers. Your task is to write a function that inserts a node at a given position, 'pos'.
   Note: Assume that the Indexing for the linked list always starts from 0. If the given position 'pos' is greater than length of linked list, then you should return the same linked list without

any change. And if position 'pos' is equal to length of input linked list, then insert the node at the last position.

```
Node* insertNode(Node *head, int i, int data) {
        Node* NewNode = new Node(data);
        if (head == NULL) {
            return head;
        }
        if (i == 0) {
            Node* temp = head;
            head = NewNode;
            NewNode->next = temp;
        }
        if (i == 1) {
            Node* a = head;
            Node* b = a->next;
            head->next = NewNode;
            NewNode->next = b;
        }
        insertNode(head->next, i-1, data);
        return head;
}
```

6. **Delete node (recursive):** Given a singly linked list of integers and position 'i', delete the node present at the 'i-th' position in the linked list recursively.

```
Node *deleteNodeRec(Node *head, int pos) {
    if(head == NULL) {
        return NULL;
    }
    if(pos == 0) {
        head = head->next;
        return head;
    }
    if(pos == 1 && head->next!=NULL) {     //head->next!=NULL so that pos=size
of LL, doesnt segmentaion fault
        Node* a = head->next;
        Node* b = a->next;
        head->next = b;
        delete a;
    }
    deleteNodeRec(head->next, pos-1);
    return head;
}
```

7. **Find a Node in Linked List:** You have been given a singly linked list of integers. Write a function that returns the index/position of an integer data denoted by 'N' (if it exists). Return -1 otherwise.

```
int findNode(Node *head, int n)
{
    int pos = 0;
    Node* temp = head;
    if (temp == NULL) return -1;    //Empty linked list
    while(temp != NULL) {
        if(temp->data == n) {
            return pos;
        }
        if(temp->next == NULL) {    //If data not found till end of LL
            return -1;
        }
        pos++;
        temp = temp->next;
    }
}
```

8. **AppendLastNToFirst:** You have been given a singly linked list of integers along with an integer 'N'. Write a function to append the last 'N' nodes towards the front of the singly linked list and returns the new head to the list.

```
int length(Node *head)
{
    if(head == NULL) {          //Empty LL
        return 0;
    }
    else {
        int count = 1;
        while(head->next != NULL) {
            count++;
            head = head->next;
        }
        return count;
    }
}

Node *appendLastNToFirst(Node *head, int n)
{
    if(n == 0 || head == NULL) {
        return head;
    }
    else {
        int N = length(head);
        Node* temp = head;
        for(int i = 1; i < N-n; i++) {
            temp = temp->next;
        }
        Node* h2 = temp->next;
        temp->next = NULL;
        Node* temp2 = h2;
        while(temp2->next != NULL) {
            temp2 = temp2->next;
        }
        temp2->next = head;
        return h2;
    }
}
```

9. **Eliminate duplicates from LL:** You have been given a singly linked list of integers where the elements are sorted in ascending order. Write a function that removes the consecutive duplicate values such that the given list only contains unique elements and returns the head to the updated list.

```
Node *removeDuplicates(Node *head)
{
    if (head == NULL) {
        return NULL;
    }
    Node* t1 = head;
    Node* t2 = head->next;
    while(t2 != NULL) {
        if(t1->data != t2->data) {
            t1->next = t2;         //Connecting unequal nodes
            t1 = t2;               //Moving t1 to t2
            t2 = t1->next;         //Moving t2 to next of t1
        }
        else if(t1->data == t2->data) {
            Node* temp = t2->next;
            delete t2;             //Deallocating duplicate node
            t2 = temp;             //Moving t2 to next node (no change in t1
needed)
        }
    }
    t1->next = t2;        //To connect last node to NULL
    return head;
}
```

10. **Print Reverse LinkedList:** You have been given a singly linked list of integers. Write a function to print the list in a reverse order. To explain it further, you need to start printing the data from the tail and move towards the head of the list, printing the head data at the end.
Note : You can’t change any of the pointers in the linked list, just print it in the reverse order.

```
void printReverse(Node *head)
{
    if(head == NULL) {
        return;
    }
    printReverse(head->next);
    cout<<head->data<<" ";
    return;
}
```

11. **Palindrome LinkedList:** You have been given a head to a singly linked list of integers. Write a function check to whether the list given is a 'Palindrome' or not.

```
Node *Reverse(Node *head) {
    if (head == NULL || head->next == NULL) {
        return head;
    }
    Node *remaining = Reverse(head->next);
    head->next->next = head;
    head->next = NULL;
    return remaining;
}

Node *mid(Node *head) {
    Node* temp1 = head;
    Node* temp2 = head;
    int count = 0;
    while(temp1->next != NULL) {
        temp1 = temp1->next;
        count++;
    }
    for(int i = 0; i < count/2; i++) {
        temp2 = temp2->next;
    }
    return temp2;
}

bool isPalindrome(Node *head)
{
    if(head == NULL) return true;
    Node* temp1 = mid(head);
    Node* temp2 = temp1->next;
    temp1->next = NULL;          //Breaking 1st list
    Node* h1 = head;
    Node* h2 = Reverse(temp2);
    while(h2 != NULL) {
        if(h1->data != h2->data) {
            return false;
        }
        h1 = h1->next;
        h2 = h2->next;
    }
    return true;
}
```

Lecture 9: Linked List 2

```
class Node
{
        public:
        int data;
        Node *next;
        Node(int data)
        {
                this->data = data;
                this->next = NULL;
        }
};
```

1. **Midpoint of LL:** For a given singly linked list of integers, find and return the node present at the middle of the list.
   Note : If the length of the singly linked list is even, then return the first middle node. Example: Consider, 10 -> 20 -> 30 -> 40 is the given list, then the nodes present at the middle with respective data values are, 20 and 30. We return the first node with data 20.

```
Node *midPoint(Node *head)
{
    if(head == NULL) return NULL;
    Node* slow = head;
    Node* fast = head->next;
    while(fast != NULL && fast->next != NULL) {
        slow = slow->next;
        fast = fast->next->next;
    }
    return slow;
}
```

2. **Merge Two Sorted LL:** You have been given two sorted(in ascending order) singly linked lists of integers. Write a function to merge them in such a way that the resulting singly linked list is also sorted(in ascending order) and return the new head to the list.
   Note : Try solving this in O(1) auxiliary space. No need to print the list, it has already been taken care of.

```
Node *mergeTwoSortedLinkedLists(Node *head1, Node *head2)
{
    if(head1 == NULL) {
        return head2;
    }
    else if(head2 == NULL) {
        return head1;
    }
    Node* h1 = head1;
    Node* h2 = head2;
    Node* fh = NULL;
    Node* ft = NULL;
    if(h1->data <= h2->data) {
        fh = h1;
        ft = h1;
        h1 = h1->next;
    }
    else {
        fh = h2;
        ft = h2;
        h2 = h2->next;
    }
    while(h1 != NULL && h2!= NULL) {
        if(h1->data <= h2->data) {
            ft->next = h1;
            ft = h1;
            h1 = h1->next;
        }
        else {
            ft->next = h2;
            ft = h2;
            h2 = h2->next;
        }
    }
    if(h1 != NULL) {
        ft->next = h1;
    }
    else if(h2 != NULL) {
        ft->next = h2;
    }
    return fh;
}
```

3. **Merge Sort:** Given a singly linked list of integers, sort it using 'Merge Sort.'

```
Node *mergeTwoSortedLinkedLists(Node *head1, Node *head2)
{
    if(head1 == NULL) {
        return head2;
    }
    else if(head2 == NULL) {
        return head1;
    }
    Node* h1 = head1;
    Node* h2 = head2;
    Node* fh = NULL;
    Node* ft = NULL;
    if(h1->data <= h2->data) {
        fh = h1;
        ft = h1;
        h1 = h1->next;
    }
    else {
        fh = h2;
        ft = h2;
        h2 = h2->next;
    }
    while(h1 != NULL && h2!= NULL) {
        if(h1->data <= h2->data) {
            ft->next = h1;
            ft = h1;
            h1 = h1->next;
        }
        else {
            ft->next = h2;
            ft = h2;
            h2 = h2->next;
        }
    }
    if(h1 != NULL) {
        ft->next = h1;
    }
    else if(h2 != NULL) {
        ft->next = h2;
    }
    return fh;
}

Node *mid(Node *head) {
    Node* temp1 = head;
    Node* temp2 = head;
    int count = 0;
    while(temp1->next != NULL) {
        temp1 = temp1->next;
        count++;
    }
    for(int i = 0; i < count/2; i++) {
        temp2 = temp2->next;
    }
    return temp2;
}

Node *mergeSort(Node *head)
{
    if(head == NULL) {
        return NULL;
    }
    if(head->next == NULL) {
        return head;
    }
    Node* h1 = head;
    Node* temp = mid(head);
    Node* h2 = temp->next;
    temp->next = NULL;
    h1 = mergeSort(h1);
    h2 = mergeSort(h2);
    return mergeTwoSortedLinkedLists(h1, h2);
}
```

4. **Reverse LL (Recursive):** Given a singly linked list of integers, reverse it using recursion and return the head to the modified list. You have to do this in O(N) time complexity where N is the size of the linked list.

```
Node *reverseLinkedListRec(Node *head)
{
    if(head == NULL || head->next == NULL) {
        return head;
    }
    Node* SmallAns = reverseLinkedListRec(head->next);
    Node* temp = SmallAns;
    while(temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = head;
    head->next = NULL;
    return SmallAns;
}

/* No need to traverse and find tail; we know that the tail is head->next (As,
the tail is the next node of head before reversing)
Node *reverseLinkedListRec(Node *head)
{
    if(head == NULL || head->next == NULL) {
        return head;
    }
    Node *SmallAns = reverseLinkedListRec(head->next);
    Node* tail = head->next;
    tail->next = head;
    head->next = NULL;
    return SmallAns;
}
This has O(n) time complexity rather than O(n^2) because there is no traversal
*/
```

5. **Reverse LL (Iterative):** Given a singly linked list of integers, reverse it iteratively and return the head to the modified list.

```
Node *reverseLinkedList(Node *head) {
    if(head == NULL) return NULL;
    Node* prv = NULL;
    Node* cur = head;
    Node* nxt = head->next;
    while(cur != NULL) {
        cur->next = prv;
        prv = cur;
        cur = nxt;
        if(nxt !=NULL) nxt = nxt->next;
    }
    return prv;
}
```

6. **Find a node in LL (recursive):** Given a singly linked list of integers and an integer n, find and return the index for the first occurrence of 'n' in the linked list. -1 otherwise. Follow a recursive approach to solve this.

```
int findNodeRec(Node *head, int n)
{
    if(head == NULL) {
        return -1;
    }
    if(head->data == n) {
        return 0;
    }
    int SmallAns = findNodeRec(head->next, n);
    return SmallAns == -1? SmallAns : SmallAns + 1;
}
```

7. **Even after Odd LinkedList:** For a given singly linked list of integers, arrange the elements such that all the even numbers are placed after the odd numbers. The relative order of the odd and even terms should remain unchanged.

```
Node *evenAfterOdd(Node *head)
{
    Node* temp = head;
    Node* OddH = NULL;
    Node* OddT = NULL;
    Node* EvenH = NULL;
    Node* EvenT = NULL;
    while(temp != NULL) {
        if(EvenH == NULL && temp->data % 2 == 0) {
            EvenH = temp;
            EvenT = temp;
        }
        else if(OddH == NULL && temp->data % 2 != 0) {
            OddH = temp;
            OddT = temp;
        }
        else {
            if(temp->data % 2 == 0) {
                EvenT->next = temp;
                EvenT = temp;
            }
            else if(temp->data % 2 != 0) {
                OddT->next = temp;
                OddT = temp;
            }
        }
        temp = temp->next;
    }
    if(EvenH == NULL) {            //No even element
        return head;
    }
    else if(OddH == NULL) {        //No odd element
        return head;
    }
    EvenT->next = NULL;
    OddT->next = EvenH;     //Linking Odd to Even
    return OddH;
}
```

8. **Delete every N nodes:** You have been given a singly linked list of integers along with two integers, 'M,' and 'N.' Traverse the linked list such that you retain the 'M' nodes, then delete the next 'N' nodes. Continue the same until the end of the linked list. To put it in other words, in the given linked list, you need to delete N nodes after every M nodes.

```
Node *swapNodes(Node *head, int i, int j)
{
    Node* term1 = head;
    Node* term2 = head;
    int temp;
      for(int x = 0; x < i; x++) {
        term1 = term1->next;
      }
      for(int y = 0; y < j; y++) {
        term2 = term2->next;
      }
      temp = term2->data;
      term2->data = term1->data;
      term1->data = temp;
      return head;
}
```

9. **kReverse:** Given a singly linked list of integers, reverse the nodes of the linked list 'k' at a time and return its modified list. 'k' is a positive integer and is less than or equal to the length of the linked list. If the number of nodes is not a multiple of 'k,' then left-out nodes, in the end, should be reversed as well.
   Example :
   Given this linked list: 1 -> 2 -> 3 -> 4 -> 5
   For k = 2, you should return: 2 -> 1 -> 4 -> 3 -> 5
   For k = 3, you should return: 3 -> 2 -> 1 -> 5 -> 4

```
Node *kReverse (Node *head, int k)
{
    if(k == 0) {
        return head;
    }
    Node* current = head;
    Node* next = NULL;
    Node* prev = NULL;
    int count = 0;
    while (current != NULL && count < k)
    {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
        count++;
    }
    if (next != NULL)
    head->next = kReverse(next, k);
    return prev;
}
```

10. **Bubble Sort (Iterative):** Given a singly linked list of integers, sort it using 'Bubble Sort.'

```
Node *bubbleSort(Node *head)
{
      Node* current = head;
    int temp;
      if(head == NULL) {
        return NULL;
      }
      while(current != NULL) {
        Node* index = current->next;
        while(index != NULL) {
            if(current->data > index->data) {
                temp = current->data;
                current->data = index->data;
                index->data = temp;
            }
            index = index->next;
        }
        current = current->next;
      }
      return head;
}
```

Lecture 10 : Stacks & Queues

1. **Stack Using LL:** Implement a Stack Data Structure specifically to store integer data using a Singly Linked List. The data members should be private. You need to implement the following public functions :
   1. Constructor: It initialises the data members as required.
   2. push(data) : This function should take one argument of type integer. It pushes the element into the stack and returns nothing.
   3. pop() : It pops the element from the top of the stack and in turn, returns the element being popped or deleted. In case the stack is empty, it returns -1.
   4. top : It returns the element being kept at the top of the stack. In case the stack is empty, it returns -1.
   5. size() : It returns the size of the stack at any given instance of time.
   6. isEmpty() : It returns a boolean value indicating whether the stack is empty or not.
   Operations Performed on the Stack:
   Query-1(Denoted by an integer 1): Pushes an integer data to the stack.
   Query-2(Denoted by an integer 2): Pops the data kept at the top of the stack and returns it to the caller.
   Query-3(Denoted by an integer 3): Fetches and returns the data being kept at the top of the stack but doesn't remove it, unlike the pop function.
   Query-4(Denoted by an integer 4): Returns the current size of the stack.
   Query-5(Denoted by an integer 5): Returns a boolean value denoting whether the stack is empty or not.

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

class Node {
   public:
    int data;
    Node *next;

    Node(int data) {
        this->data = data;
        next = NULL;
    }
};

class Stack {
       Node *head;
       int size;

    public:

    Stack() {
        head = NULL;
        size = 0;
    }

    int getSize() {
        return size;
```

```cpp
    }

    bool isEmpty() {
        if(head == NULL) return true;
        else return false;
    }

    void push(int element) {
        Node* NewNode = new Node(element);
        Node* temp = head;
        head = NewNode;
        head->next = temp;
        size++;
    }

    int pop() {
        if(head == NULL) return -1;
        else {
            int Ans = head->data;
            Node* temp = head->next;
            delete head;
            head = temp;
            size--;
            return Ans;
        }
    }

    int top() {
        if(head == NULL) return -1;
        else return head->data;
    }
};


int main() {
    Stack st;

    int q;
    cin >> q;

    while (q--) {
        int choice, input;
        cin >> choice;
        switch (choice) {
            case 1:
                cin >> input;
                st.push(input);
                break;
            case 2:
                cout << st.pop() << "\n";
                break;
            case 3:
                cout << st.top() << "\n";
                break;
            case 4:
                cout << st.getSize() << "\n";
                break;
            default:
                cout << ((st.isEmpty()) ? "true\n" : "false\n");
                break;
        }
    }
}
```

2. **Balanced Parenthesis:** For a given a string expression containing only round brackets or parentheses, check if they are balanced or not. Brackets are said to be balanced if the bracket which opens last, closes first.
   Example:
   Expression: (()())
   Since all the opening brackets have their corresponding closing brackets, we say it is balanced and hence the output will be, 'true'.
   You need to return a boolean value indicating whether the expression is balanced or not.

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

bool isBalanced(string expression)
{
    stack<char> S;
    int length = expression.size();
    for(int i = 0; i < length; i++) {
        char term = expression[i];
        if(term == '(' || term == '[' || term == '{') {
            S.push(term);
        }
        else if(term == ')' || term == ']' || term == '}') {
            if(S.empty()) return false;
            else if((S.top() == '(' && term == ')') || (S.top() == '[' && term
== ']') || (S.top() == '{' && term == '}')) S.pop();
            else return false;
        }
        else continue;
    }
    if(S.empty()) return true;
    else return false;
}


int main()
```

```
{
    string input;
    cin >> input;
    cout << ((isBalanced(input)) ? "true" : "false");
}
```

3. **Queue Using LL:** Implement a Queue Data Structure specifically to store integer data using a Singly Linked List. The data members should be private.
   You need to implement the following public functions :

   1. Constructor: It initialises the data members as required.

   2. enqueue(data) : This function should take one argument of type integer. It enqueues the element into the queue and returns nothing.

   3. dequeue() : It dequeues/removes the element from the front of the queue and in turn, returns the element being dequeued or removed. In case the queue is empty, it returns -1.

   4. front() : It returns the element being kept at the front of the queue. In case the queue is empty, it returns -1.

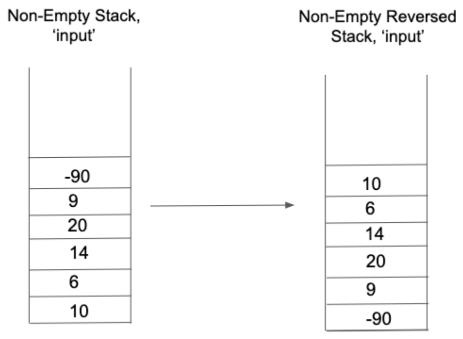
   5. getSize() : It returns the size of the queue at any given instance of time.

   6. isEmpty() : It returns a boolean value indicating whether the queue is empty or not.

   Operations Performed on the Stack:
   Query-1(Denoted by an integer 1): Enqueues an integer data to the queue.
   Query-2(Denoted by an integer 2): Dequeues the data kept at the front of the queue and returns it to the caller.
   Query-3(Denoted by an integer 3): Fetches and returns the data being kept at the front of the queue but doesn't remove it, unlike the dequeue function.
   Query-4(Denoted by an integer 4): Returns the current size of the queue.
   Query-5(Denoted by an integer 5): Returns a boolean value denoting whether the queue is empty or not.

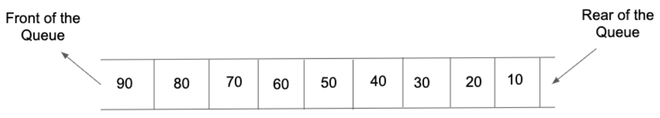
```
#include <iostream>
using namespace std;

class Node {
   public:
    int data;
    Node *next;

    Node(int data) {
        this->data = data;
        next = NULL;
    }
};

class Queue {
     Node *head;
     Node *tail;
     int size;
    public:
    Queue() {
             head = NULL;
             tail = NULL;
             size = 0;
      }

      int getSize() {
             return size;
      }

    bool isEmpty() {
             return size == 0;
      }

    void enqueue(int data) {
             Node *NewNode = new Node(data);
             if(head == NULL) {
            head = NewNode;
            tail = NewNode;
             }
             else {
            tail->next = NewNode;
            tail = NewNode;
             }
             size++;
      }

    int dequeue() {
        if(size == 0) return -1;
        else {
             int Data = head->data;
             Node *temp = head;
             head = head->next;
             size--;
             delete temp;
             return Data;
        }
    }

    int front() {
        if(size == 0) return -1;
        else return head->data;
    }
};


int main() {
    Queue q;

    int t;
    cin >> t;

    while (t--) {
        int choice, input;
        cin >> choice;
        switch (choice) {
            case 1:
                cin >> input;
                q.enqueue(input);
                break;
            case 2:
                cout << q.dequeue() << "\n";
                break;
            case 3:
                cout << q.front() << "\n";
                break;
            case 4:
                cout << q.getSize() << "\n";
                break;
            default:
                cout << ((q.isEmpty()) ? "true\n" : "false\n");
                break;
        }
    }
}
```

4. **Reverse a Stack:** You have been given two stacks that can store integers as the data. Out of the two given stacks, one is populated and the other one is empty. You are required to write a function that reverses the populated stack using the one which is empty.

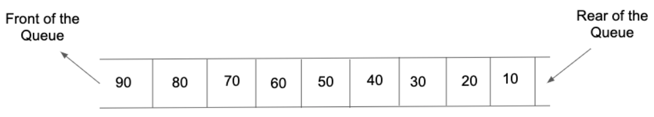


```
#include <iostream>
#include <stack>
using namespace std;

void reverseStack(stack<int> &input, stack<int> &extra) {
    if(input.size() == 0 || input.size() == 1) {
        return;
    }
    int x = input.top();
    input.pop();
    reverseStack(input, extra);        //Recursion (sending input after popping)
    while(input.size() != 0) {         //Pushing to extra stack
        int Data1 = input.top();
        input.pop();
        extra.push(Data1);
    }
    input.push(x);
    while(extra.size() != 0) {         //Pushing back to original stack
        int Data2 = extra.top();
        extra.pop();
        input.push(Data2);
    }
}

int main() {
    stack<int> input, extra;
    int size;
    cin >> size;

    for (int i = 0, val; i < size; i++) {
        cin >> val;
        input.push(val);
    }

    reverseStack(input, extra);

    while (!input.empty()) {
        cout << input.top() << " ";
        input.pop();
    }
}
```

5. **Reverse Queue:** You have been given a queue that can store integers as the data. You are required to write a function that reverses the populated queue itself without using any other data structures.



Input Queue after reversal



Input Queue after reversal

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

void reverseQueue(queue<int> &input) {
       if(input.size() == 0) {         //Base case
         return;
       }
       int Data = input.front();
       input.pop();
       reverseQueue(input);            //Recursive call
       input.push(Data);               //Small Calculation
}

int main() {
    int t;
    cin >> t;

    while (t--) {
        queue<int> q;
        int size;
        cin >> size;

        for (int i = 0, val; i < size; i++) {
            cin >> val;
            q.push(val);
        }

        reverseQueue(q);
        while (!q.empty()) {
            cout << q.front() << " ";
            q.pop();
        }

        cout << "\n";
    }
}
```

6. **Check redundant brackets:** For a given expression in the form of a string, find if there exist any redundant brackets or not. It is given that the expression contains only rounded brackets or parenthesis and the input expression will always be balanced. A pair of the brackets is said to be redundant when a sub-expression is surrounded by unnecessary or needless brackets.

   Example:
   Expression: (a+b)+c
   Since there are no needless brackets, hence, the output must be 'false'.

   Expression: ((a+b))
   The expression can be reduced to (a+b). Hence the expression has redundant brackets and the output will be 'true'.

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

bool checkRedundantBrackets(string expression) {
       stack<char> S;
       int length = expression.size();
       for(int i = 0; i < length; i++) {
        int count  = 0;
        if(expression[i] == ')') {
            while(S.top() != '(') {
                S.pop();
                count++;
            }
            S.pop();                         //Popping ( too
            if (count <= 1) return true;    //Assuming >1 terms b/w non-redundant
        }
        S.push(expression[i]);
       }
       return false;
}

int main() {
    string input;
    cin >> input;
    cout << ((checkRedundantBrackets(input)) ? "true" : "false");
}
```

7. **Stock Span:** Afzal has been working with an organization called 'Money Traders' for the past few years. The organization is into the money trading business. His manager assigned him a task. For a given array/list of stock's prices for N days, find the stock's span for each day. The span of the stock's price today is defined as the maximum number of consecutive days(starting from today and going backwards) for which the price of the stock was less than today's price.
   For example, if the price of a stock over a period of 7 days are [100, 80, 60, 70, 60, 75, 85], then the stock spans will be [1, 1, 1, 2, 1, 4, 6].

   ### Explanation:
   On the sixth day when the price of the stock was 75, the span came out to be 4, because the last 4 prices(including the current price of 75) were less than the current or the sixth day's price.
   Similarly, we can deduce the remaining results.
   Afzal has to return an array/list of spans corresponding to each day's stock's price. Help him to achieve the task.

```cpp
#include <iostream>
#include <stack>
```

```
using namespace std;

int* helper(int* price, int size, int *span) {
    stack<int> s;
    s.push(0);     //Index of 1st element
    span[0] = 1;   //Span of 1st element always 1

    for(int i = 1; i < size; i++) {      //finding span of elements other than 1
        while(s.size() != 0 && price[i] > price[s.top()]) {
            s.pop();             //popping indexes of terms smaller than ith term
        }
        span[i] = (s.empty()) ? (i + 1) : (i - s.top());
        //if stack empty then ith term greater than all other terms before it
thus (i + 1) not empty then span is (ith term index) - (the index of term
greater than ith term)
        s.push(i);               //Pushing ith index to stack
    }
    return span;
}

int* stockSpan(int *price, int size)  {
      int *span = new int[size];
      return helper(price, size, span);
}

int main() {
    int size;
    cin >> size;

    int *input = new int[size];
    for (int i = 0; i < size; i++) {
        cin >> input[i];
    }

    int *output = stockSpan(input, size);
    for (int i = 0; i < size; i++) {
        cout << output[i] << " ";
    }

    cout << "\n";

    delete[] input;
    delete[] output;
}
```

8. **Minimum bracket Reversal:** For a given expression in the form of a string, find the minimum number of brackets that can be reversed in order to make the expression balanced. The expression will only contain curly brackets. If the expression can't be balanced, return -1.
   Example:
   Expression: {{{{

If we reverse the second and the fourth opening brackets, the whole expression will get balanced. Since we have to reverse two brackets to make the expression balanced, the expected output will be 2.

Expression: {{{
In this example, even if we reverse the last opening bracket, we would be left with the first opening bracket and hence will not be able to make the expression balanced and the output will be -1.

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

int countBracketReversals(string input) {
      if(input.size() % 2 != 0) return -1;    //Odd cant be balanced
      stack<int> s;
      for(int i = 0; i < input.size(); i++) {
       if (input[i] == '{') {
           s.push(input[i]);
       }
       else if (input[i] == '}') {
           if(s.empty()) s.push(input[i]);
           else if(s.top() == '{') s.pop();
           else if(s.top() != '{') s.push(input[i]);
       }
      }
    int count = 0;
    while(s.size() != 0) {
        char ch1 = s.top();
        s.pop();
        char ch2 = s.top();
        s.pop();
        if(ch1 == ch2) count++;
        if(ch1 != ch2) count += 2;
    }
    return count;
}

int main() {
    string input;
    cin >> input;
    cout << countBracketReversals(input);
}
```

Lecture 11 : Trees

```
template <typename T>
class TreeNode {
   public:
    T data;
    vector<TreeNode<T>*> children;

    TreeNode(T data) { this->data = data; }

    ~TreeNode() {
        for (int i = 0; i < children.size(); i++) {
            delete children[i];
        }
    }
};

/* Input format :
The first line of input contains data of the nodes of the tree in level order
form. The order is: data for root node, number of children to root node, data of
each of child nodes and so on and so forth for each node. The data of the nodes
of the tree is separated by space.
*/

TreeNode<int>* takeInputLevelWise() {
    int rootData;
    cin >> rootData;
    TreeNode<int>* root = new TreeNode<int>(rootData);

    queue<TreeNode<int>*> pendingNodes;

    pendingNodes.push(root);
    while (pendingNodes.size() != 0) {
        TreeNode<int>* front = pendingNodes.front();
        pendingNodes.pop();
        int numChild;
        cin >> numChild;
        for (int i = 0; i < numChild; i++) {
            int childData;
            cin >> childData;
            TreeNode<int>* child = new TreeNode<int>(childData);
            front->children.push_back(child);
            pendingNodes.push(child);
        }
    }

    return root;
}
```

1. **Print Level Wise:** Given a generic tree, print the input tree in level wise order.
   For printing a node with data N, you need to follow the exact format -

   N:x1,x2,x3,...,xn

   where, N is data of any node present in the generic tree. x1, x2, x3, ..., xn are the children of node N. Note that there is no space in between.

   You need to print all nodes in the level order form in different lines.

```
void printLevelWise(TreeNode<int>* root) {
    queue<TreeNode<int>*> Nodes;
    Nodes.push(root);
    while(Nodes.size() != 0) {
        TreeNode<int>* front = Nodes.front();
        Nodes.pop();
        cout<<front->data<<":";
        for(int i = 0; i < front->children.size(); i++) {
            if(i == (front->children.size() - 1))
cout<<front->children[i]->data;
            else cout<<front->children[i]->data<<",";
            Nodes.push(front->children[i]);
        }
        cout<<endl;
    }
}
```

2. **Find sum of nodes:** Given a generic tree, find and return the sum of all nodes present in the given tree.

```
int sumOfNodes(TreeNode<int>* root) {
    int SmallAns = root->data;
    for(int i = 0; i < root->children.size(); i++) {
        SmallAns += sumOfNodes(root->children[i]);
    }
    return SmallAns;
}
```

3. **Max data node:** Given a generic tree, find and return the node with maximum data. You need to return the node which is having maximum data. Return null if the tree is empty.

```
TreeNode<int>* maxDataNode(TreeNode<int>* root) {
    if(root == NULL) return NULL;
    TreeNode<int>* Max = root;
    for(int i = 0; i < root->children.size(); i++) {
        TreeNode<int>* SmallAns = maxDataNode(root->children[i]);
//Assuming recurssion sends us Max from sub-tree
        if(SmallAns->data > Max->data) Max = SmallAns;
    }
    return Max;
}
```

4. **Find height:** Given a generic tree, find and return the height of given tree.

```
int getHeight(TreeNode<int>* root) {
    int max = 0;
    for(int i = 0; i < root->children.size(); i++) {
        int SmallHeight = getHeight(root->children[i]);
        if(SmallHeight > max) max = SmallHeight;
    }
    return max + 1;
}
```

5. **Count leaf nodes:** Given a generic tree, count and return the number of leaf nodes present in the given tree.

```
int getLeafNodeCount(TreeNode<int>* root) {
    if(root->children.size() == 0) return 1;
    int SmallAns = 0;
    for(int i = 0; i < root->children.size(); i++) {
        SmallAns = SmallAns + getLeafNodeCount(root->children[i]);
    }
    return SmallAns;
}
```

6. **PostOrder Traversal:** Given a generic tree, print the post-order traversal of given tree. The post-order traversal is: visit child nodes first and then root node.

```
void printPostOrder(TreeNode<int>* root) {
    if(root == NULL) return;
    for(int i = 0; i < root->children.size(); i++) {
        printPostOrder(root->children[i]);
    }
    cout<<root->data<<" ";
}
```

7. **Contains x:** Given a generic tree and an integer x, check if x is present in the given tree or not. Return true if x is present, return false otherwise.

```
bool isPresent(TreeNode<int>* root, int x) {
    if(root->data == x) return true;
    for(int i = 0; i < root->children.size(); i++) {
        if(isPresent(root->children[i], x) == true) return true;
    }
    return false;
}
```

8. **Count nodes:** Given a tree and an integer x, find and return the number of nodes which contains data greater than x.

```
int getLargeNodeCount(TreeNode<int>* root, int x) {
    if(root == NULL) return 0;
    int count = 0;
    if(root->data > x) count++;
    for(int i = 0; i < root->children.size(); i++) {
        count = count + getLargeNodeCount(root->children[i], x);
    }
    return count;
}
```

9. **Count nodes:** Given a tree and an integer x, find and return the number of nodes which contains data greater than x.

```
int getLargeNodeCount(TreeNode<int>* root, int x) {
    if(root == NULL) return 0;
    int count = 0;
    if(root->data > x) count++;
    for(int i = 0; i < root->children.size(); i++) {
        count = count + getLargeNodeCount(root->children[i], x);
    }
    return count;
}
```

10. **Node with maximum child sum:** Given a generic tree, find and return the node for which sum of its data and data of all its child nodes is maximum. In the sum, data of the node and data of its immediate child nodes has to be taken.

```
int sum(TreeNode<int>* root) {
    int sum = root->data;
    for(int i = 0; i < root->children.size(); i++) {
        sum = sum + root->children[i]->data;
    }
    return sum;
}

TreeNode<int>* maxSumNode(TreeNode<int>* root) {
    TreeNode<int>* Max = root;
    for(int i = 0; i < root->children.size(); i++) {
        TreeNode<int>* SmallAns = maxSumNode(root->children[i]);
        if(sum(Max) < sum(SmallAns)) Max = SmallAns;
    }
    return Max;
}
```

11. **Structurally identical:** Given two generic trees, return true if they are structurally identical. Otherwise return false.
    Structural Identical -> If the two given trees are made of nodes with the same values and the nodes are arranged in the same way, then the trees are called identical.

```
bool areIdentical(TreeNode<int> *root1, TreeNode<int> * root2) {
    if(root1->data != root2->data) return false;
    for(int i = 0; i < root1->children.size(); i++) {
        if(areIdentical(root1->children[i], root2->children[i]) == false) return
false;
    }
    return true;
}
```

12. **Next larger:** Given a generic tree and an integer n. Find and return the node with next larger element in the tree i.e. find a node with value just greater than n.
    Note: Return NULL if no node is present with the value greater than n.

```
TreeNode<int>* getNextLargerElement(TreeNode<int>* root, int x) {
    TreeNode<int>* ans;
    if(root->data > x) ans = root;
    else ans = NULL;
    for(int i = 0; i < root->children.size(); i++) {
        TreeNode<int>* temp = getNextLargerElement(root->children[i], x);
        if(temp == NULL) continue;                           //No term > x in sub-tree
        else if(ans == NULL) ans = temp;                     //ans NULL, assigning
value(without comaprision)
        else if(ans->data > temp->data) ans = temp;     //temp closer to x(the
smaller, the closer[since term > x])
    }
    return ans;
}
```

13. **Second Largest Element In Tree:** Given a generic tree, find and return the node with second largest value in given tree.
    Note: Return NULL if no node with required value is present.

```
//Hint uses pair class node to save and update both largest and second largest
node

int maxDataNode(TreeNode<int>* root) {
    int Max = root->data;
    for(int i = 0; i < root->children.size(); i++) {
        int SmallAns = maxDataNode(root->children[i]);
        if(SmallAns > Max) Max = SmallAns;
    }
    return Max;
}

TreeNode<int>* getNextSmallerElement(TreeNode<int>* root, int x) {    //Sending
Max and finding Node
    TreeNode<int>* ans;                                               //just
smaller than it
    if(root->data < x) ans = root;
    else ans = NULL;
    for(int i = 0; i < root->children.size(); i++) {
        TreeNode<int>* temp = getNextSmallerElement(root->children[i], x);
        if(temp == NULL) continue;
        else if(ans == NULL) ans = temp;
        else if(ans->data < temp->data) ans = temp;
    }
    return ans;
}

TreeNode<int>* getSecondLargestNode(TreeNode<int>* root) {
    int Max = maxDataNode(root);
    return getNextSmallerElement(root, Max);
}
```

14. **Replace with depth:** You are given a generic tree. You have to replace each node with its depth value. You just have to update the data of each node, there is no need to return or print anything.

```
void helper(TreeNode<int>* root, int n) {
    root->data = n;
    for(int i = 0; i < root->children.size(); i++) {
        helper(root->children[i], n + 1);
    }
}

void replaceWithDepthValue(TreeNode<int>* root) {
    int depth = 0;
    helper(root, depth);
}
```

Lecture 12 : Binary Trees

```
template <typename T>
class BinaryTreeNode {
   public:
    T data;
    BinaryTreeNode<T>* left;
    BinaryTreeNode<T>* right;
    BinaryTreeNode(T data) {
        this->data = data;
        left = NULL;
        right = NULL;
    }
};

/* The first and the only line of input will contain the node data, all
separated by a single space. Since -1 is used as an indication whether the left
or right node data exist for root, it will not be a part of the node data. */

BinaryTreeNode<int>* takeInput() {
    int rootData;
    cin >> rootData;
    if (rootData == -1) {
        return NULL;
    }
    BinaryTreeNode<int>* root = new BinaryTreeNode<int>(rootData);
    queue<BinaryTreeNode<int>*> q;
    q.push(root);
    while (!q.empty()) {
        BinaryTreeNode<int>* currentNode = q.front();
        q.pop();
        int leftChild, rightChild;

        cin >> leftChild;
        if (leftChild != -1) {
            BinaryTreeNode<int>* leftNode = new BinaryTreeNode<int>(leftChild);
            currentNode->left = leftNode;
            q.push(leftNode);
        }

        cin >> rightChild;
        if (rightChild != -1) {
            BinaryTreeNode<int>* rightNode =
                new BinaryTreeNode<int>(rightChild);
            currentNode->right = rightNode;
            q.push(rightNode);
        }
    }
    return root;
}
```

1. **Print Level Wise:** For a given a Binary Tree of type integer, print the complete information of every node, when traversed in a level-order fashion.
   To print the information of a node with data D, you need to follow the exact format :
   D:L:X,R:Y (Print -1 if the child doesn't exist)

```
void printLevelWise(BinaryTreeNode<int> *root) {
       queue<BinaryTreeNode<int>*> pendingNodes;
       pendingNodes.push(root);
       while(pendingNodes.size() != 0) {
         BinaryTreeNode<int>* front = pendingNodes.front();
         pendingNodes.pop();
         int leftChild = -1, rightChild = -1;
         if(front->left != NULL) {
                 leftChild = front->left->data;
                 pendingNodes.push(front->left);
         }
         if(front->right != NULL) {
                 rightChild = front->right->data;
                 pendingNodes.push(front->right);
         }
         cout << front->data << ":L:" << leftChild << ",R:" << rightChild <<
endl;
       }
}
```

2. **Find a node:** For a given Binary Tree of type integer and a number X, find whether a node exists in the tree with data X or not.

```
bool isNodePresent(BinaryTreeNode<int> *root, int x) {
    if(root == NULL) return false;
    if(root->data == x) return true;
    if(isNodePresent(root->left, x) == true || isNodePresent(root->right, x) ==
true) return true;
    else return false;
}
```

3. **Height of Binary Tree:** For a given Binary Tree of integers, find and return the height of the tree. (Height is defined as the total number of nodes along the longest path from the root to any of the leaf nodes.)

```
int height(BinaryTreeNode<int>* root) {
    if (root == NULL) return 0;
    int LeftH = height(root->left);
    int RightH = height(root->right);
    return LeftH >= RightH ? LeftH + 1 : RightH + 1;
}
```

4. **Mirror:** For a given Binary Tree of type integer, update it with its corresponding mirror image.

```
       1              1
      / \            / \
     2   3          3   2
    / \ / \        / \ / \
   4  5 6  7      7  6 5  4

  Original BT     Mirrored BT
```

```
void mirrorBinaryTree(BinaryTreeNode<int>* root) {
    BinaryTreeNode<int>* temp = root->left;
    root->left = root->right;
    root->right = temp;
    if(root->right != NULL) mirrorBinaryTree(root->right);
    if(root->left != NULL) mirrorBinaryTree(root->left);
}
```

5. **Preorder Binary Tree:** For a given Binary Tree of integers, print the pre-order traversal.

```
void preOrder(BinaryTreeNode<int> *root) {
      if (root == NULL) return;
    cout<<root->data<<" ";
    preOrder(root->left);
    preOrder(root->right);
}
```

6. **Postorder Binary Tree:** For a given Binary Tree of integers, print the post-order traversal.

```
void postOrder(BinaryTreeNode<int> *root) {
      if (root == NULL) return;
    postOrder(root->left);
    postOrder(root->right);
    cout<<root->data<<" ";
}
```

7. **Construct Tree from Preorder and Inorder:** For a given preorder and inorder traversal of a Binary Tree of type integer stored in an array/list, create the binary tree using the given two arrays/lists. You just need to construct the tree and return the root.
   Note: Assume that the Binary Tree contains only unique elements.

```
BinaryTreeNode<int>* helper(int *preorder, int preS, int preE, int *inorder, int
inS, int inE) {
    BinaryTreeNode<int>* root = new BinaryTreeNode<int>(preorder[preS]);
    if(preS > preE || inS > inE) {
        return NULL;
    }

    int RootIndex = 0;
    while(inorder[inS + RootIndex] != preorder[preS]) {
        RootIndex++;
    }

    int LpreS = preS + 1;       //Dividing pre-order into left and right part
    int LpreE = preS + RootIndex;
    int RpreS = preS + RootIndex + 1;
    int RpreE = preE;

    int LinS  = inS;            //Dividing in-order into left and right part
    int LinE  = inS + RootIndex - 1;
    int RinS  = inS + RootIndex + 1;
    int RinE  = inE;

    root->left  = helper(preorder, LpreS, LpreE, inorder, LinS, LinE);
    root->right = helper(preorder, RpreS, RpreE, inorder, RinS, RinE);
    return root;
}

BinaryTreeNode<int>* buildTree(int *preorder, int preLength, int *inorder, int
inLength) {
    return helper(preorder, 0, preLength - 1, inorder, 0, inLength - 1);
}
```

8. **Construct Tree from Postorder and Inorder:** For a given postorder and inorder traversal of a Binary Tree of type integer stored in an array/list, create the binary tree using the given two arrays/lists. You just need to construct the tree and return the root.
   Note: Assume that the Binary Tree contains only unique elements.

```
BinaryTreeNode<int>* helper(int *postorder, int postS, int postE, int *inorder,
int inS, int inE) {
    BinaryTreeNode<int>* root = new BinaryTreeNode<int>(postorder[postE]);
    if(postS > postE || inS > inE) {
        return NULL;
    }

    int RootIndex = 0;
    while(inorder[inS + RootIndex] != postorder[postE]) {
        RootIndex++;
    }

    int LpostS = postS;
    int LpostE = postS + RootIndex - 1;
    int RpostS = postS + RootIndex;
    int RpostE = postE - 1;

    int LinS   = inS;
    int LinE   = inS + RootIndex - 1;
    int RinS   = inS + RootIndex + 1;
    int RinE   = inE;
    root->left  = helper(postorder, LpostS, LpostE, inorder, LinS, LinE);
    root->right = helper(postorder, RpostS, RpostE, inorder, RinS, RinE);
    return root;
}

BinaryTreeNode<int>* buildTree(int *postorder, int postLength, int *inorder, int
inLength) {
    return helper(postorder, 0, postLength - 1, inorder, 0, inLength - 1);
}
```

9. **Min and Max of Binary Tree:** For a given Binary Tree of type integer, find and return the minimum and the maximum data values.
   Return the output as an object of Pair class, which is already created.

```
#include<climits>
pair<int, int> getMinAndMax(BinaryTreeNode<int> *root) {
      if(root == NULL) {
             pair<int, int> p;
             p.first  = INT_MAX;
             p.second = INT_MIN;
             return p;
       }

       pair<int, int> leftAns  = getMinAndMax(root->left );
       pair<int, int> rightAns = getMinAndMax(root->right);
       pair<int, int> p;
       p.first  = min(root->data, min(leftAns.first , rightAns.first ));
       p.second = max(root->data, max(leftAns.second, rightAns.second));
       return p;
}
```

10. **Sum Of Nodes:** For a given Binary Tree of integers, find and return the sum of all the nodes data.

```
int getSum(BinaryTreeNode<int>* root) {
    if(root == NULL) return 0;
    return root->data + getSum(root->left) + getSum(root->right);
}
```

11. **Check Balanced:** Given a binary tree, check if it is balanced. Return true if given binary tree is balanced, false otherwise.
    Balanced Binary Tree:
    An empty binary tree or binary tree with zero nodes is always balanced. For a non empty binary tree to be balanced, following conditions must be met:
    1. The left and right subtrees must be balanced.
    2. |hL - hR| <= 1, where hL is the height or depth of the left subtree and hR is the height or depth of the right subtree.

```
pair<int, bool> helper(BinaryTreeNode<int> * root) {
    if(root == NULL) {
        pair<int, bool> p;
        p.first = 0;
        p.second = true;
        return p;
    }

    pair<int, bool> leftAns  = helper(root->left);
    pair<int, bool> rightAns = helper(root->right);

    int hL  = leftAns.first;
    int hR  = rightAns.first;
    bool bL = leftAns.second;
    bool bR = rightAns.second;

    pair<int, bool> p;
    p.first = 1 + max(hL, hR);
    if(bL == false || bR == false) {
        p.second = false;
    }
    else {
        if(max(hL, hR) - min(hL, hR) <= 1) {
            p.second = true;
        }
        else p.second = false;
    }
    return p;
}

bool isBalanced(BinaryTreeNode<int> *root) {
     return helper(root).second;
}
```

12. **Level order traversal:** For a given a Binary Tree of type integer, print it in a level order fashion where each level will be printed on a new line. Elements on every level will be printed in a linear fashion and a single space will separate them.
    the output would look like:
    ```
    10
    20 30
    40 50 60
    ```

```
void printLevelWise(BinaryTreeNode<int> *root) {
    queue<BinaryTreeNode<int>*> pendingNodes;
    pendingNodes.push(root);
    pendingNodes.push(NULL);
    while(pendingNodes.size() != 0) {
        BinaryTreeNode<int>* Front = pendingNodes.front();
        pendingNodes.pop();
        if(Front == NULL) {
```

```
                cout<<endl;
                if(pendingNodes.size() != 0) pendingNodes.push(NULL);
            }
            else {
                cout<<Front->data<<" ";
                if(Front->left != NULL) pendingNodes.push(Front->left);
                if(Front->right != NULL) pendingNodes.push(Front->right);
            }
        }
    }
```

13. **Remove Leaf nodes:** Given a binary tree, remove all leaf nodes from it. Leaf nodes are those nodes, which don't have any children.
    Note:

    1. Root will also be a leaf node if it doesn't have left and right child.

    2. You don't need to print the tree, just remove all leaf nodes and return the updated root.

```
BinaryTreeNode<int>* removeLeafNodes(BinaryTreeNode<int> *root) {
    if(root == NULL) {                                   //Base case
        return NULL;
    }
    if(root->left == NULL && root->right == NULL) {      //Small Calc.
        return NULL;
    }
    root->left  = removeLeafNodes(root->left);           //Recursive calls
    root->right = removeLeafNodes(root->right);
    return root;
}
```

14. **Level wise linkedlist:** Given a binary tree, write code to create a separate linked list for each level. You need to return the array which contains the head of each level linked list.

```
vector<Node<int>*> constructLinkedListForEachLevel(BinaryTreeNode<int> *root) {
    vector<Node<int>*> vecLL;
    if(root == NULL) {
        vecLL.push_back(NULL);
        return vecLL;
    }
    queue<BinaryTreeNode<int>*> pendingNodes;
    pendingNodes.push(root);
    pendingNodes.push(NULL);
    Node<int>* head = NULL;
    Node<int>* tail = NULL;
    while(pendingNodes.size() != 0) {
        BinaryTreeNode<int>* Front = pendingNodes.front();
        pendingNodes.pop();
        if(Front == NULL) {
            tail->next = NULL;
            vecLL.push_back(head);
            head = NULL;
            tail = NULL;
            if(pendingNodes.size() != 0) pendingNodes.push(NULL);
        }
        else {
            Node<int>* node = new Node<int>(Front->data);
            if(head == NULL) {
                head = node;
                tail = node;
            }
            else {
                tail->next = node;
                tail = tail->next;
            }
            if(Front->left != NULL) pendingNodes.push(Front->left);
            if(Front->right != NULL) pendingNodes.push(Front->right);
        }
    }
    return vecLL;
}
```

15. **ZigZag tree:** Given a binary tree, print the zig zag order. In zigzag order, level 1 is printed from left to right, level 2 from right to left and so on. This means odd levels should get printed from left to right and even level right to left.

```
#include <stack>
void zigZagOrder(BinaryTreeNode<int> *root) {
    stack<BinaryTreeNode<int>*> s1, s2;
    s1.push(root);
    while(s1.size() != 0 || s2.size() != 0) {
        while(s1.size() != 0) {
            BinaryTreeNode<int>* tops1 = s1.top();
            s1.pop();
            cout<<tops1->data<<" ";
            if(tops1->left != NULL) s2.push(tops1->left);
            if(tops1->right != NULL) s2.push(tops1->right);
        }
        cout<<endl;
        while(s2.size() != 0) {
            BinaryTreeNode<int>* tops2 = s2.top();
            s2.pop();
            cout<<tops2->data<<" ";
            if(tops2->right != NULL) s1.push(tops2->right);
            if(tops2->left != NULL) s1.push(tops2->left);
        }
        cout<<endl;
    }
}
```

16. **Nodes without sibling:** For a given Binary Tree of type integer, print all the nodes without any siblings.

```
void printNodesWithoutSibling(BinaryTreeNode<int> *root) {
    if(root == NULL) return;
    if(root->left != NULL && root->right == NULL) {
        cout<<root->left->data<<" ";
        printNodesWithoutSibling(root->left);
    }
    else if(root->left == NULL && root->right != NULL) {
        cout<<root->right->data<<" ";
        printNodesWithoutSibling(root->right);
    }
    else if(root->left != NULL && root->right != NULL) {
        printNodesWithoutSibling(root->left);
        printNodesWithoutSibling(root->right);
    }
}
```

Lecture 13 : BST

```
template <typename T>
class BinaryTreeNode {
   public:
    T data;
    BinaryTreeNode<T> *left;
    BinaryTreeNode<T> *right;

    BinaryTreeNode(T data) {
        this->data = data;
        left = NULL;
        right = NULL;
    }
    ~BinaryTreeNode() {
        if (left) delete left;
        if (right) delete right;
    }
};

/* The first line of input contains data of the nodes of the tree in level order form.
The data of the nodes of the tree is separated by space. If any node does not have a
left or right child, take -1 in its place. Since -1 is used as an indication whether the
left or right nodes exist, therefore, it will not be a part of the data of any node. The
following line of input contains an integer, that denotes the value of k. */

BinaryTreeNode<int> *takeInput() {
    int rootData;
    cin >> rootData;
    if (rootData == -1) {
        return NULL;
    }
    BinaryTreeNode<int> *root = new BinaryTreeNode<int>(rootData);
    queue<BinaryTreeNode<int> *> q;
    q.push(root);
    while (!q.empty()) {
        BinaryTreeNode<int> *currentNode = q.front();
        q.pop();
        int leftChild, rightChild;
        cin >> leftChild;
        if (leftChild != -1) {
            BinaryTreeNode<int> *leftNode = new BinaryTreeNode<int>(leftChild);
            currentNode->left = leftNode;
            q.push(leftNode);
        }
        cin >> rightChild;
        if (rightChild != -1) {
            BinaryTreeNode<int> *rightNode =
                new BinaryTreeNode<int>(rightChild);
            currentNode->right = rightNode;
            q.push(rightNode);
        }
    }
    return root;
}
```

1. **Search in BST:** Given a BST and an integer k. Find if the integer k is present in given BST or not. You have to return true, if node with data k is present, return false otherwise.
   Note: Assume that BST contains all unique elements.

```
bool searchInBST(BinaryTreeNode<int> *root , int k) {
      if(root == NULL) {
        return false;
      }
      if(root->data == k) return true;
      if(root->data > k) return searchInBST(root->left, k);
      if(root->data < k) return searchInBST(root->right, k);
}
```

2. **Print Elements in Range:** Given a Binary Search Tree and two integers k1 and k2, find and print the elements which are in range k1 and k2 (both inclusive).
   Print the elements in increasing order.

```
void elementsInRangeK1K2(BinaryTreeNode<int>* root, int k1, int k2) {
      if(root == NULL) return;
      if(root->data >= k1 && root->data <= k2) {
        elementsInRangeK1K2(root->left, k1, k2);     //left->R->right
        cout<<root->data<<" ";                        //Printing increasing order
        elementsInRangeK1K2(root->right, k1, k2);
      }
      else if(root->data > k2) elementsInRangeK1K2(root->left, k1, k2);
      else if(root->data < k1) elementsInRangeK1K2(root->right, k1, k2);
}
```

3. **Check if a Binary Tree is BST:** Given a binary tree with N number of nodes, check if that input tree is BST (Binary Search Tree). If yes, return true, return false otherwise.
   Note: Duplicate elements should be kept in the right subtree.

```
#include <limits.h>
class isBSTclass {
    public:
        bool isBST;
        int minimum;
        int maximum;
};

isBSTclass isBSTfunc(BinaryTreeNode<int> *root) {
      if(root == NULL) {
        isBSTclass output;
        output.isBST = true;
        output.minimum = INT_MAX;
        output.maximum = INT_MIN;
        return output;
      }
```

```
        isBSTclass leftAns = isBSTfunc(root->left);
        isBSTclass rightAns = isBSTfunc(root->right);

        int minimum = min(root->data, min(leftAns.minimum, rightAns.minimum));
        int maximum = max(root->data, max(leftAns.maximum, rightAns.maximum));
        bool isBST = (root->data > leftAns.maximum) && (root->data <=
rightAns.minimum) && leftAns.isBST && rightAns.isBST;

        isBSTclass Ans;
        Ans.minimum = minimum;
        Ans.maximum = maximum;
        Ans.isBST = isBST;
        return Ans;
}

bool isBST(BinaryTreeNode<int> *root) {
        return isBSTfunc(root).isBST;
}
```

4. **Construct BST from a Sorted Array:** Given a sorted integer array A of size n, which contains all unique elements. You need to construct a balanced BST from this input array. Return the root of constructed BST.
   Note: If array size is even, take first mid as root.

```
BinaryTreeNode<int>* helper(int *input, int si, int ei) {
    if(si > ei) {
        return NULL;
    }
    int mid = (si + ei)/2;
    BinaryTreeNode<int>* root = new BinaryTreeNode<int>(input[mid]);
    BinaryTreeNode<int>* leftChild = helper(input, si, mid - 1);
    BinaryTreeNode<int>* rightChild = helper(input, mid + 1, ei);
    root->left = leftChild;
    root->right = rightChild;
    return root;
}

BinaryTreeNode<int>* constructTree(int *input, int n) {
        return helper(input, 0, n - 1);
}
```

5. **BST to Sorted LL:** Given a BST, convert it into a sorted linked list. You have to return the head of LL.

```
pair<Node<int>*, Node<int>*> helper(BinaryTreeNode<int>* root) {
    if(root == NULL) {
        pair<Node<int>*, Node<int>*> output;
        output.first = NULL;
        output.second = NULL;
        return output;
    }
    Node<int>* Lroot = new Node<int>(root->data);

    pair<Node<int>*, Node<int>*> leftAns = helper(root->left);
    pair<Node<int>*, Node<int>*> rightAns = helper(root->right);
    pair<Node<int>*, Node<int>*> SmallAns;

    if(leftAns.first == NULL) {          //Exception when Left returns NULL
        SmallAns.first = Lroot;          //root = head
        Lroot->next = rightAns.first;
    }
    else {
        SmallAns.first = leftAns.first;
        leftAns.second->next = Lroot;
        Lroot->next = rightAns.first;
    }

    if(rightAns.first == NULL) {         //Exception when Right returns NULL
        SmallAns.second = Lroot;         //root = tail
    }
    else {
        SmallAns.second = rightAns.second;
    }

    return SmallAns;
}

Node<int>* constructLinkedList(BinaryTreeNode<int>* root) {
      return helper(root).first;
}
```

6. **Find Path in BST:** Given a BST and an integer k. Find and return the path from the node with data k and root (if a node with data k is present in given BST) in a list. Return empty list otherwise.
   Note: Assume that BST contains all unique elements.

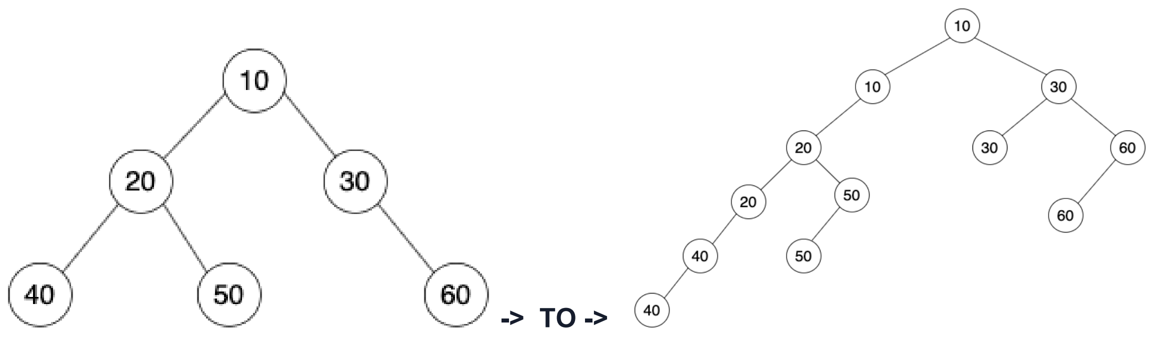
```
vector<int>* getPath(BinaryTreeNode<int> *root , int data) {
      if(root == NULL) return NULL;

      if(root->data == data) {
        vector<int>* vecPath = new vector<int>();
        vecPath->push_back(root->data);
        return vecPath;
      }

      vector<int>* leftPath = getPath(root->left, data);
      if(leftPath != NULL) {
        leftPath->push_back(root->data);
        return leftPath;
      }

      vector<int>* rightPath = getPath(root->right, data);
      if(rightPath != NULL) {
        rightPath->push_back(root->data);
        return rightPath;
      } else {
        return NULL;
      }
}
```

7. **BST Class:** Implement the BST class which includes following functions -
   1. search: Given an element, find if that is present in BST or not. Return true or false.
   2. insert - Given an element, insert that element in the BST at the correct position. If element is equal to the data of the node, insert it in the left subtree.
   3. delete - Given an element, remove that element from the BST. If the element which is to be deleted has both children, replace that with the minimum element from right sub-tree.
   4. printTree (recursive) - Print the BST in ithe following format -
   For printing a node with data N, you need to follow the exact format - N:L:x,R:y
   wherer, N is data of any node present in the binary tree. x and y are the values of left and right child of node N. Print the children only if it is not null. There is no space in between. You need to print all nodes in the recursive format in different lines.

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

template <typename T>
class BinaryTreeNode {
   public:
    T data;
    BinaryTreeNode<T> *left;
    BinaryTreeNode<T> *right;

    BinaryTreeNode(T data) {
        this->data = data;
        left = NULL;
        right = NULL;
    }
};

class BST {
    BinaryTreeNode<int>* root;     // Define the data members

    /*-------------- Private helper Functions of BST --------------*/

    BinaryTreeNode<int>* remove(int data, BinaryTreeNode<int>* node) {
        if(node == NULL) return NULL;
        if(node->data > data) {
            BinaryTreeNode<int>* leftAns = remove(data, node->left);
            node->left = leftAns;
            return node;
        }
        if(node->data < data) {
            BinaryTreeNode<int>* rightAns = remove(data, node->right);
            node->right = rightAns;
            return node;
        }
        if(node->data == data) {
            if(node->left == NULL && node->right == NULL) {
                                                    //Leaf node(return NULL)
                delete node;
                return NULL;
            }
            else if(node->left == NULL) {
                BinaryTreeNode<int>* temp = node->right; //Storing right subtree
                node->right = NULL;                      //Breaking connection
                delete node;
                        //(or destructor will delete entire subtree with itself)
                return temp;
            }
            else if(node->right == NULL) {
                BinaryTreeNode<int>* temp = node->left;  //Storing left subtree
                node->left = NULL;                       //Breaking connection
                delete node;
                        //(or destructor will delete entire subtree with itself)
                return temp;
            }
            else {
                BinaryTreeNode<int>* minNode = node->right;
                while(minNode->left != NULL) {
                //finding minimum node in right subtree to act as new node
                    minNode = minNode->left;
                //(since we need replacement for node who has both left and
right child)
                }
                int rightMin = minNode->data;
                //Updating our node with minimum we found
                node->data = rightMin;
                node->right = remove(rightMin, node->right);
                //Deleting node we took for replacement
                return node;
            }
        }
    }

    void print(BinaryTreeNode<int>* node) {
        if(node == NULL) return;
        cout<<node->data<<":";
        if(node->left != NULL) {
            cout<<"L:"<<node->left->data<<",";
        }
        if(node->right != NULL) {
            cout<<"R:"<<node->right->data;
        }
        cout<<endl;
        print(node->left);
        print(node->right);
    }

    BinaryTreeNode<int>* insert(int data, BinaryTreeNode<int>* node) {
        if(node == NULL) {
            BinaryTreeNode<int>* NewNode = new BinaryTreeNode<int>(data);
            return NewNode;
        }
        if(node->data >= data) {
            BinaryTreeNode<int>* leftAns = insert(data, node->left);
            node->left = leftAns;
        }
        else {
            BinaryTreeNode<int>* rightAns = insert(data, node->right);
            node->right = rightAns;
        }
        return node;
    }

    bool search(int data, BinaryTreeNode<int>* node) {
        if(node == NULL) {
            return false;
        }
        if(node->data == data) {
            return true;
        }
        else if(node->data > data) {
            return search(data, node->left);
        }
        else if(node->data < data) {
            return search(data, node->right);
        }
    }

    public:
    BST() {                     // Implement the Constructor
        root = NULL;
    }

      /*----------------- Public Functions of BST -----------------*/
    void remove(int data) {    // Implement the remove() function
        root = remove(data, root);
    }

    void print() {              // Implement the print() function
```

```
        print(root);
    }

    void insert(int data) {    // Implement the insert() function
        root = insert(data, root);
    }

    bool search(int data) {    // Implement the search() function
             return search(data, root);
    }
};

int main() {
    BST *tree = new BST();
    int choice, input, q;
    cin >> q;
    while (q--) {
        cin >> choice;
        switch (choice) {
            case 1:
                cin >> input;
                tree->insert(input);
                break;
            case 2:
                cin >> input;
                tree->remove(input);
                break;
            case 3:
                cin >> input;
                cout << ((tree->search(input)) ? "true\n" : "false\n");
                break;
            default:
                tree->print();
                break;
        }
    }
}
```

8. **Create & Insert Duplicate Node:** For a given a Binary Tree of type integer, duplicate every node of the tree and attach it to the left of itself. The root will remain the same. So you just need to insert nodes in the given Binary Tree.

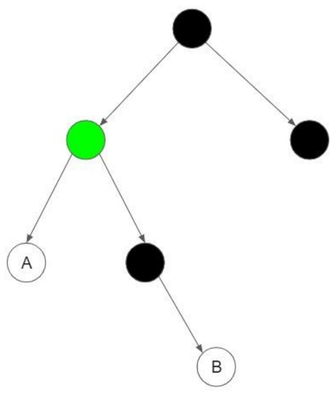


```
void insertDuplicateNode(BinaryTreeNode<int> *root) {
    if(root == NULL) return;
    insertDuplicateNode(root->left);
    insertDuplicateNode(root->right);
    BinaryTreeNode<int>* newNode = new BinaryTreeNode<int>(root->data);
    newNode->left = root->left;
    root->left = newNode;
}
```

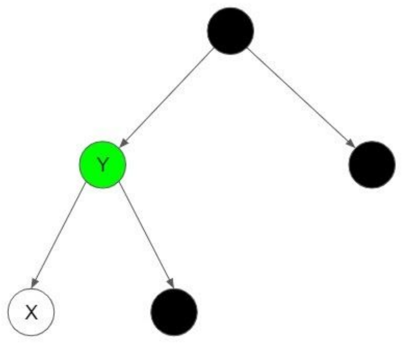
9. **Pair Sum Binary Tree:** Given a binary tree and an integer S, print all the pair of nodes whose sum equals S.
   Note:
   1. Assume the given binary tree contains all unique elements.
   2. In a pair, print the smaller element first. Order of different pairs doesn't matter.

```
#include<bits/stdc++.h>
void save(BinaryTreeNode<int> *root,int arr[],int &n)
{
    if(root==NULL)
    {
        return;
    }
    arr[n]=root->data;
    n++;
    save(root->left,arr,n);
    save(root->right,arr,n);

}

void pairSum(BinaryTreeNode<int> *root, int sum)
{
    int arr[1000000];
    int n=0;
    save(root,arr,n);
    sort(arr,arr+n);
    for(int i=0,j=n-1;i<j;)
    {
        int s=arr[i]+arr[j];
        if(s==sum)
        {
            cout<<arr[i]<<' '<<arr[j]<<endl;
            i++;
            j--;
        }
        else if(s<sum) i++;
        else j--;
    }
}
```

10. **LCA of Binary Tree:** Given a binary tree and data of two nodes, find 'LCA' (Lowest Common Ancestor) of the given two nodes in the binary tree.
LCA -> LCA of two nodes A and B is the lowest or deepest node which has both A and B as its descendants.



Green LCA of A and B



Y LCA of X and Y

```
int getLCA(BinaryTreeNode <int>* root , int a, int b) {
    if(root == NULL) return -1;
    if(root->data == a || root->data == b) return root->data;
    int leftAns  = getLCA(root->left, a, b);
    int rightAns = getLCA(root->right, a, b);
    if(leftAns == -1 && rightAns == -1) return -1;                //No term in
both subtree
    else if(leftAns != -1 && rightAns != -1) return root->data;   //if 1 term in
left and another in right then root is LCA
    else if(leftAns != -1)  return leftAns;            //returning the term that
is not NULL
    else if(rightAns != -1) return rightAns;           //(either 1 term is not in
tree or a term is LCA of the other term)
}
```

11. **LCA of BST:**

```
int getLCA(BinaryTreeNode<int>* root , int val1 , int val2){
    if(root == NULL) return -1;
    if(root->data == val1 || root->data == val2) return root->data;

    int leftAns = -1;       //optimizing for BST
    int rightAns = -1;
    if(val1 < root->data && val2 < root->data) leftAns  = getLCA(root->left,
val1, val2);
    else if(val1 > root->data && val2 > root->data) rightAns =
getLCA(root->right, val1, val2);
    else {
        leftAns  = getLCA(root->left, val1, val2);
        rightAns = getLCA(root->right, val1, val2);
    }
    if(leftAns == -1 && rightAns == -1) return -1;
    else if(leftAns != -1 && rightAns != -1) return root->data;
    else if(leftAns != -1)  return leftAns;
    else if(rightAns != -1) return rightAns;
}
```

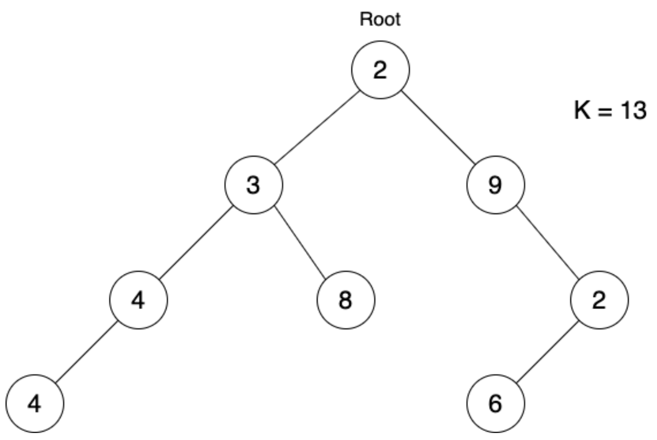
12. **Largest BST:** Given a Binary tree, find the largest BST subtree. That is, you need to find the BST with maximum height in the given binary tree. You have to return the height of largest BST.

```
#include <limits.h>
class isBSTclass {
    public:
        bool isBST;
        int minimum;
        int maximum;
        int height;
};

isBSTclass isBSTfunc(BinaryTreeNode<int> *root) {
      if(root == NULL) {
        isBSTclass output;
        output.isBST = true;
        output.minimum = INT_MAX;
        output.maximum = INT_MIN;
        output.height = 0;
        return output;
      }

      isBSTclass leftAns = isBSTfunc(root->left);
      isBSTclass rightAns = isBSTfunc(root->right);

      int minimum = min(root->data, min(leftAns.minimum, rightAns.minimum));
      int maximum = max(root->data, max(leftAns.maximum, rightAns.maximum));
      bool isBST = (root->data > leftAns.maximum) && (root->data <=
rightAns.minimum) && leftAns.isBST && rightAns.isBST;
    int height = max(leftAns.height, rightAns.height);
    if(isBST) height++;   //if this subtree is BST returning +1(to include root)

      isBSTclass Ans;
      Ans.minimum = minimum;
      Ans.maximum = maximum;
      Ans.isBST = isBST;
      Ans.height = height;
      return Ans;
}

int largestBSTSubtree(BinaryTreeNode<int> *root) {
    return isBSTfunc(root).height;
}
```

13. **Replace with Sum of greater nodes:** Given a binary search tree, you have to replace each node's data with the sum of all nodes which are greater or equal than it. You need to include the current node's data also.
    That is, if in given BST there is a node with data 5, you need to replace it with sum of its data (i.e. 5) and all nodes whose data is greater than or equal to 5.
    Note: You don't need to return or print, just change the data of each node.

```
void replace(BinaryTreeNode<int>* root, int* sum) {
    if (root == NULL) return;     //reverse in-order traversal
    replace(root->right, sum);
//traversing in descending order and simultaneously updating every node
    *sum = *sum + root->data;
//using global variable sum(updated and added to each node)
    root->data = *sum;
    replace(root->left, sum);
}

void replaceWithLargerNodesSum(BinaryTreeNode<int> *root) {
    int sum = 0;
    replace(root, &sum);
}
```

14. **Path Sum Root to Leaf:** For a given Binary Tree of type integer and a number K, print out all root-to-leaf paths where the sum of all the node data along the path is equal to K.



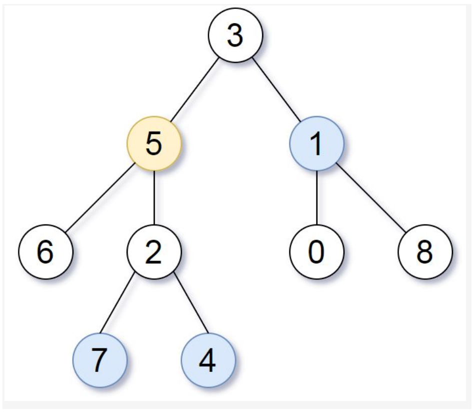
If you see in the above-depicted picture of Binary Tree, we see that there are a total of two paths, starting from the root and ending at the leaves which sum up to a value of K = 13.

The paths are:
a. 2 3 4 4
b. 2 3 8

One thing to note here is, there is another path in the right sub-tree in reference to the root, which sums up to 13 but since it doesn't end at the leaf, we discard it. The path is: 2 9 2(not a leaf)

```
void helper(BinaryTreeNode<int>* root, int k, vector<int> path) {
    if(root == NULL) return;
    k = k - root->data;      //updating data members
    path.push_back(root->data);

    if(k == 0 && root->left == NULL && root->right == NULL) {
        for(int i = 0; i < path.size(); i++) {    //printing path
            cout<<path[i]<<" ";                   //(leaf node necessary)
        }
        cout<<endl;
    }

    if(root->left != NULL) helper(root->left, k, path);   //recursive call
    if(root->right != NULL) helper(root->right, k, path);
}

void rootToLeafPathsSumToK(BinaryTreeNode<int> *root, int k) {
    vector<int> path;
    helper(root, k, path);
}
```

15. **Print nodes at distance k from node:** You are given a Binary Tree of type integer, a target node, and an integer value K.
    Print the data of all nodes that have a distance K from the target node. The order in which they would be printed will not matter.
    For a given input tree(refer to the image below):
    1. Target Node: 5
    2. K = 2



Starting from the target node 5, the nodes at distance K are 7 4 and 1.

```
void printNodeAtDepthK(BinaryTreeNode<int> *root,int k){
    if(root==NULL){
        return;
    }
    if(k==0){
      cout<<root->data<<endl;
    }
    printNodeAtDepthK(root->left,k-1);
    printNodeAtDepthK(root->right,k-1);
}

int nodesAtDistanceKHelper(BinaryTreeNode<int> *root,int element,int k){
    if(root==NULL){
        return -1;
    }
    if(root->data==element){
        printNodeAtDepthK(root,k);
        return 0;
    }

    int leftD=nodesAtDistanceKHelper(root->left,element,k);
    if(leftD!=-1){
        if(leftD +1 ==k){
            cout<<root->data<<endl;
        }
        else{
            printNodeAtDepthK(root->right,k-leftD-2);
            return leftD+1;
        }
    }

    int rightD=nodesAtDistanceKHelper(root->right,element,k);
    if(rightD!=-1){
        if(rightD +1 ==k){
            cout<<root->data<<endl;
        }
        else{
            printNodeAtDepthK(root->left,k-rightD-2);
            return rightD+1;
        }
    }
   return -1;
}

void nodesAtDistanceK(BinaryTreeNode<int> *root, int node, int k) {
     int ans=nodesAtDistanceKHelper(root,node,k);
}
```

16. **Pair sum in a BST:** Given a binary search tree and an integer S, find pair of nodes in the BST which sum to S. You can use extra space of the order of O(log n).
    Note:
    1. Assume BST contains all unique elements.
    2. In a pair, print the smaller element first.

```
void pairSum(int *array, int n, int sum)      //printing pairs
{
    int i = 0;
    int j = n - 1;
    while(i < j) {
        if(array[i] + array[j] < sum) i++;
        else if(array[i] + array[j] > sum) j--;
        else {
            cout<<array[i]<<" "<<array[j]<<endl;
            i++;
            j--;
        }
    }
}

int* array = new int[100000];
int index = 0;
void storeInOrder(BinaryTreeNode<int>* root) {    //forming array
      if (root == NULL) return;
      storeInOrder(root->left);
      array[index++] = root->data;
      storeInOrder(root->right);
}

void printNodesSumToS(BinaryTreeNode<int> *root, int s) {
    storeInOrder(root);
    pairSum(array, index, s);
}
```

Lecture 14 : Priority Queues

1. **Remove Min (Min Priority queue):** Implement the function RemoveMin for the min priority queue class.For a minimum priority queue, write the function for removing the minimum element present. Remove and return the minimum element.

```
#include <vector>

class PriorityQueue {
    vector<int> pq;

   public:
    bool isEmpty() {
        return pq.size() == 0;
    }

    int getSize() {
        return pq.size();
    }

    int getMin() {
        if (isEmpty()) {
            return 0;
        }
        return pq[0];
    }

    void insert(int element) {
        pq.push_back(element);

        int childIndex = pq.size() - 1;

        while (childIndex > 0) {
            int parentIndex = (childIndex - 1) / 2;

            if (pq[childIndex] < pq[parentIndex]) {
                int temp = pq[childIndex];
                pq[childIndex] = pq[parentIndex];
                pq[parentIndex] = temp;
            } else {
                break;
            }

            childIndex = parentIndex;
        }
    }

    int removeMin() {
        if(isEmpty()) return 0;      //Underflow
        int Ans = pq[0];             //Swapping first and last node
        pq[0] = pq[pq.size() - 1];
        pq.pop_back();              //remove min priority(at last node)

        int parentIndex = 0;        //Down-heapify
        int leftIndex = (2*parentIndex) + 1;
        int rightIndex = (2*parentIndex) + 2;
        while(leftIndex < pq.size()) {
            int minIndex = parentIndex;           //finding minimum index
            if(pq[leftIndex] < pq[minIndex]) minIndex = leftIndex;
            if(rightIndex < pq.size() && pq[rightIndex] < pq[minIndex]) minIndex
= rightIndex;
            if(minIndex == parentIndex) break;
                                  //no more interchanging left(final heap formed)
            int temp = pq[minIndex];
                                  //Swapping nodes(to follow heap order property)
            pq[minIndex] = pq[parentIndex];
            pq[parentIndex] = temp;
            parentIndex = minIndex;
            leftIndex = (2*parentIndex) + 1;
            rightIndex = (2*parentIndex) + 2;
        }
        return Ans;
    }
};
```

2. **Max Priority Queue:** Implement the class for Max Priority Queue which includes following functions -
   1. getSize - Return the size of priority queue i.e. number of elements present in the priority queue.
   2. isEmpty - Check if priority queue is empty or not. Return true or false accordingly.
   3. insert - Given an element, insert that element in the priority queue at the correct position.
   4. getMax - Return the maximum element present in the priority queue without deleting. Return -Infinity if the priority queue is empty.
   5. removeMax - Delete and return the maximum element present in the priority queue. Return -Infinity if the priority queue is empty.

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

class PriorityQueue {
    // Declare the data members here
      vector<int> pq;
    public:
    PriorityQueue() {
        // Implement the constructor here
    }

    /**************** Implement all the public functions here ***************/
    void insert(int element) {
        // Implement the insert() function here
        pq.push_back(element);
        int childIndex = pq.size() - 1;
        while(childIndex > 0) {
            int parentIndex = (childIndex - 1)/2;
            if(pq[parentIndex] < pq[childIndex]) {
                int temp = pq[parentIndex];
                pq[parentIndex] = pq[childIndex];
                pq[childIndex] = temp;
            }
            else break;
            childIndex = parentIndex;
        }
    }

    int getMax() {
        // Implement the getMax() function here
        if(isEmpty()) return 0;
        else return pq[0];
    }

    int removeMax() {
        // Implement the removeMax() function here
        if(isEmpty()) return 0;

        int Ans = pq[0];
        pq[0] = pq[pq.size() - 1];
        pq.pop_back();

        int parentIndex = 0;
        int leftIndex = 1;
        int rightIndex = 2;

        while(leftIndex < pq.size()) {
            int maxIndex = parentIndex;
            if(pq[maxIndex] < pq[leftIndex]) maxIndex = leftIndex;
            if(rightIndex < pq.size() && pq[maxIndex] < pq[rightIndex]) maxIndex
= rightIndex;
            if(parentIndex == maxIndex) break;
            int temp = pq[parentIndex];
            pq[parentIndex] = pq[maxIndex];
            pq[maxIndex] = temp;
            parentIndex = maxIndex;
            leftIndex = (2*parentIndex) + 1;
            rightIndex = (2*parentIndex) + 2;
        }
        return Ans;
    }
    int getSize() {
        // Implement the getSize() function here
        return pq.size();
    }

    bool isEmpty() {
        // Implement the isEmpty() function here
        return pq.size() == 0;
    }
};


int main() {
    PriorityQueue pq;
    int choice;
    cin >> choice;

    while (choice != -1) {
        switch (choice) {
            case 1:  // insert
                int element;
                cin >> element;
                pq.insert(element);
                break;
            case 2:  // getMax
                cout << pq.getMax() << "\n";
                break;
            case 3:  // removeMax
                cout << pq.removeMax() << "\n";
                break;
            case 4:  // size
                cout << pq.getSize() << "\n";
                break;
            case 5:  // isEmpty
                cout << (pq.isEmpty() ? "true\n" : "false\n");
            default:
                return 0;
        }

        cin >> choice;
    }
}
```

3. **In-place heap sort:** Given an integer array of size N. Sort this array (in decreasing order) using heap sort.
   Note: Space complexity should be O(1).

```
void heapSort(int arr[], int n) {
    for(int i = 0; i < n; i++) {         //forming heap
        int childIndex = i;
        while(childIndex > 0) {
            int parentIndex = (childIndex - 1)/2;
            if(arr[parentIndex] < arr[childIndex]) break;
            int temp = arr[parentIndex];
            arr[parentIndex] = arr[childIndex];
            arr[childIndex] = temp;
            childIndex = parentIndex;
        }
    }

    for(int i = n - 1; i >= 0; i--) {    //Down-heapify
        int temp = arr[i];
        arr[i] = arr[0];
        arr[0] = temp;

        int parentIndex = 0;
        int leftChildIndex = 1;
        int rightChildIndex = 2;

        while(leftChildIndex < i) {
            int minIndex = parentIndex;
            if(arr[minIndex] > arr[leftChildIndex]) minIndex = leftChildIndex;
            if(rightChildIndex < i && arr[minIndex] > arr[rightChildIndex])
minIndex = rightChildIndex;
            if(minIndex == parentIndex) break;

            int temp = arr[minIndex];
            arr[minIndex] = arr[parentIndex];
            arr[parentIndex] = temp;

            parentIndex = minIndex;
            leftChildIndex = (2 * parentIndex) + 1;
            rightChildIndex = (2 * parentIndex) + 2;
        }
    }
}
```

4. **K smallest Elements:** You are given with an integer k and an array of integers that contain numbers in random order. Write a program to find k smallest numbers from a given array. You need to save them in an array and return it.
   Time complexity should be O(n * logk) and space complexity should not be more than O(k).
   Note: Order of elements in the output is not important.

```
#include<queue>
vector<int> kSmallest(int arr[], int n, int k) {
    priority_queue<int> pq;
    vector<int> Ans;

    for(int i = 0; i < k; i++) {  //forming heap
        pq.push(arr[i]);
    }

    for(int i = k; i < n; i++) {  //Updating heap
        if(pq.top() > arr[i]) {
            pq.pop();
            pq.push(arr[i]);
        }
    }

    for(int i = 0; i < k; i++) {  //return Ans
        Ans.push_back(pq.top());
        pq.pop();
    }

    return Ans;
}
```

5. **K largest elements:** You are given with an integer k and an array of integers that contain numbers in random order. Write a program to find k largest numbers from given array. You need to save them in an array and return it.
   Time complexity should be O(nlogk) and space complexity should be not more than O(k).
   Order of elements in the output is not important.

```
#include <vector>
#include<queue>
vector<int> kLargest(int input[], int n, int k){
    priority_queue<int, vector<int>, greater<int> > pq;   //Min pq
    vector<int> Ans;

    for(int i = 0; i < k; i++) {  //forming heap
        pq.push(input[i]);
    }

    for(int i = k; i < n; i++) {  //Updating heap
        if(pq.top() < input[i]) {
            pq.pop();
            pq.push(input[i]);
        }
    }

    for(int i = 0; i < k; i++) {  //return Ans
        Ans.push_back(pq.top());
        pq.pop();
    }

    return Ans;
}
```

6. **Check Max-Heap:** Given an array of integers, check whether it represents max-heap or not. Return true if the given array represents max-heap, else return false.

```
bool helper(int arr[], int n, int k) {
    if(k > (n - 2)/2) return true;    //leaf node
    int leftIndex = (2*k) + 1;
    int rightIndex = (2*k) + 2;
    if(arr[k] >= arr[leftIndex] && arr[k] >= arr[rightIndex] && helper(arr, n,
leftIndex) && helper(arr, n, rightIndex)) return true;
    else return false;
}

bool isMaxHeap(int arr[], int n) {
    return helper(arr, n, 0);
}
```

7. **Kth largest element:** Given an array A of random integers and an integer k, find and return the kth largest element in the array.
   Note: Try to do this question in less than O(N * logN) time.

```
#include<queue>
int kthLargest(int* arr, int n, int k) {
    priority_queue<int, vector<int>, greater<int> > pq;   //Min pq

    for(int i = 0; i < k; i++) {    //forming heap
        pq.push(arr[i]);
    }

    for(int i = k; i < n; i++) {    //Updating heap
        if(pq.top() < arr[i]) {
            pq.pop();
            pq.push(arr[i]);
        }
    }

    return pq.top();     //kth largest element
}
```

8. **Merge K sorted arrays:** Given k different arrays, which are sorted individually (in ascending order). You need to merge all the given arrays such that output array should be sorted (in ascending order)

```
#include <queue>
vector<int> mergeKSortedArrays(vector<vector<int>*> input) {
    priority_queue<int, vector<int>, greater<int> > pq;
    for(int i = 0; i < input.size(); i++) {
        for(int j = 0; j < input[i]->size(); j++) {
            pq.push(input.at(i)->at(j));
        }
    }
    vector<int> Ans;
    int pqSize = pq.size();
    for(int k = 0; k < pqSize; k++) {
        Ans.push_back(pq.top());
        pq.pop();
    }
    return Ans;
}
```

9. **Running median:** You are given a stream of N integers. For every i-th integer added to the running list of integers, print the resulting median.
   Print only the integer part of the median.
   Ex : 6 4 2 2 3 4
   S = {6}, median = 6
   S = {6, 2} -> {2, 6}, median = 4
   S = {6, 2, 1} -> {1, 2, 6}, median = 2
   S = {6, 2, 1, 3} -> {1, 2, 3, 6}, median = 2
   S = {6, 2, 1, 3, 7} -> {1, 2, 3, 6, 7}, median = 3
   S = {6, 2, 1, 3, 7, 5} -> {1, 2, 3, 5, 6, 7}, median = 4

```
#include <queue>
void printRunningMedian(int *arr, int n) {
    if(n <= 0) return;

    priority_queue<int> maxPq;
    priority_queue<int, vector<int>, greater<int> > minPq;

    int median = arr[0];
    maxPq.push(arr[0]);
    cout<<maxPq.top()<<" ";

    for(int i=1;i<n;i++)
    {
        if(arr[i] <= median)
        {
            if(maxPq.size() == minPq.size())
            {
                maxPq.push(arr[i]);
                median = maxPq.top();
            }
            else if(maxPq.size() < minPq.size())
            {
                 maxPq.push(arr[i]);
                 median = ((maxPq.top()+minPq.top())/2);
            }
            else
            {
                minPq.push(maxPq.top());
                maxPq.pop();
                maxPq.push(arr[i]);
                median = ((maxPq.top()+minPq.top())/2);
            }
        }
        else
        {
            if(minPq.size() == maxPq.size())
            {
                minPq.push(arr[i]);
                median = minPq.top();
            }
            else if(minPq.size() < maxPq.size())
            {
                minPq.push(arr[i]);
                median =  ((maxPq.top()+minPq.top())/2);
            }
            else
            {
                maxPq.push(minPq.top());
                minPq.pop();
                minPq.push(arr[i]);
                median = ((maxPq.top()+minPq.top())/2);
            }
        }
        cout<<median<<" ";
    }
}
```

10. **Buy the ticket:** You want to buy a ticket for a well-known concert which is happening in your city. But the number of tickets available is limited. Hence the sponsors of the concert decided to sell tickets to customers based on some priority.
    A queue is maintained for buying the tickets and every person is attached with a priority (an integer, 1 being the lowest priority).

    The tickets are sold in the following manner -

    1. The first person (pi) in the queue requests for the ticket.

    2. If there is another person present in the queue who has higher priority than pi, then ask pi to move at end of the queue without giving him the ticket.

    3. Otherwise, give him the ticket (and don't make him stand in queue again).

    Giving a ticket to a person takes exactly 1 minute and it takes no time for removing and adding a person to the queue. And you can assume that no new person joins the queue. Given a list of priorities of N persons standing in the queue and the index of your priority (indexing starts from 0). Find and return the time it will take until you get the ticket.

```
#include <queue>
int buyTicket(int *arr, int n, int k) {
    int count = 0;
    queue<int> arrQueue;
```

```cpp
    priority_queue<int> arrPq;

    for(int i = 0; i < n; i++) {    //forming pq and queue
       arrQueue.push(arr[i]);
       arrPq.push(arr[i]);
    }

    while(!arrQueue.empty()) {
        if(arrPq.top() == arrQueue.front()) {
            count++;
            arrPq.pop();
            arrQueue.pop();
            if(k == 0) {     //Our term
                return count;
            }
            else {
                k = k - 1;
            }
        }
        else {
            int temp = arrQueue.front();    //Sending to back
            arrQueue.pop();
            arrQueue.push(temp);
            if(k == 0) {
                k = arrPq.size() - 1;
            }
            else k = k - 1;
        }
    }
    return count;
}
```

Lecture 15 : Hashmaps

1. **Maximum Frequency Number:** You are given an array of integers that contain numbers in random order. Write a program to find and return the number which occurs the maximum times in the given input.
   If two or more elements contend for the maximum frequency, return the element which occurs in the array first.

```
#include <unordered_map>
int highestFrequency(int arr[], int n) {
    unordered_map<int, int> ourmap;

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

    int max_count = 0, term = -1;
    for(int i = n - 1; i >= 0; i--) {
        if(ourmap[arr[i]] >= max_count) {
            max_count = ourmap[arr[i]];
            term = arr[i];
        }
    }
    return term;
}
```

2. **Print Intersection:** You have been given two integer arrays/list(ARR1 and ARR2) of size N and M, respectively. You need to print their intersection; An intersection for this problem can be defined when both the arrays/lists contain a particular value or to put it in other words, when there is a common value that exists in both the arrays/lists.
   Note : Input arrays/lists can contain duplicate elements. The intersection elements printed would be in the order they appear in the first sorted array/list(ARR1).

```
#include <unordered_map>
#include <vector>
void intersection(int *arr1, int *arr2, int n, int m)
{
    unordered_map<int, int> map1, map2;
    vector<int> Ans;

    for(int i = 0; i < n; i++){   //forming map for arr1
        map1[arr1[i]]++;
    }
    for(int i = 0; i < m; i++){   //forming map for arr2
        map2[arr2[i]]++;
    }

    for(int i = 0; i < m; i++) {      //forming output vector
        if(map1[arr2[i]] > 0 && map2[arr2[i]] > 0) {
            Ans.push_back(arr2[i]);
            map1[arr2[i]]--;   //Updating count
            map2[arr2[i]]--;
        }
    }

    sort(Ans.begin(), Ans.end());    //Sorting vector

    for(int i = 0; i < Ans.size(); i++) {    //output
        cout<<Ans[i]<<" ";
    }
}
```

3. **Pair Sum to 0:** Given a random integer array A of size N. Find and print the count of pair of elements in the array which sum up to 0.
   Note: Array A can contain duplicate elements as well.

```
#include <unordered_map>
int pairSum(int *arr, int n) {
    unordered_map<int, int> countmap;
    int count = 0;

      for(int i = 0; i < n; i++) {   //doing in 1 traversal. got TLE in 2
traversals
        count = count + countmap[-1*arr[i]];
        countmap[arr[i]]++;
      }
      return count;
}
```

4. **Extract Unique characters:** Given a string S, you need to remove all the duplicates. That means, the output string should contain each character only once. The respective order of characters should remain same, as in the input string.

```
#include <map>
string uniqueChar(string str) {
      map<char, int> countmap;
      string Ans;
      for(int i = 0; i < str.size(); i++) {
        countmap[str[i]]++;
        if(countmap[str[i]] == 1) Ans.push_back(str[i]);
      }
      return Ans;
}
```

5. **Longest Consecutive Sequence:** You are given an array of unique integers that contain numbers in random order. You have to find the longest possible sequence of consecutive numbers using the numbers from the given array. You need to return the output array which contains the starting and ending element. If the length of the longest possible sequence is one, then the output array must contain only a single element.

Note:
1. Best solution takes O(n) time.
2. If two sequences are of equal length, then return the sequence starting with the number whose occurrence is earlier in the array.

```
#include <map>
vector<int> longestConsecutiveIncreasingSequence(int *arr, int n) {
    map<int, int> ourmap;
    for(int i = 0; i < n; i++) {   //Creating map
        ourmap[arr[i]] = 1;
    }

    for(int i = 0; i < n; i++) {   //Updating map for consecutive
        int next = arr[i] + 1;
        while(ourmap.count(next) > 0) {
            ourmap[arr[i]]++;
            next++;
        }
    }

    int max = arr[n - 1];          //Finding max consecutive term
    for(int i = n - 2; i >= 0; i--) {
        if(ourmap[arr[i]] >= ourmap[max]) max = arr[i];
    }

    vector<int> Ans;               //Returning Answer
    Ans.push_back(max);
    Ans.push_back(max + ourmap[max] - 1);
    return Ans;
}
```

6. **Pairs with difference K:** You are given with an array of integers and an integer K. You have to find and print the count of all such pairs which have difference K.
   Note: Take absolute difference between the elements of the array.

```
#include <unordered_map>
int getPairsWithDifferenceK(int *arr, int n, int k) {
       unordered_map<int, int> countmap;
       int count = 0;

       for(int i = 0; i < n; i++) {
         int term1 = arr[i] - k;
         int term2 = arr[i] + k;
         if(k == 0) count = count + countmap[arr[i]];
         else count = count + countmap[term1] + countmap[term2];
         countmap[arr[i]]++;
       }
       return count;
}
```

7. **Longest subset zero sum:** Given an array consisting of positive and negative integers, find the length of the longest subarray whose sum is zero.

```
//logic: if we get the same sum at i & j index then length of subset = i - j + 1

#include <map>
int lengthOfLongestSubsetWithZeroSum(int* arr, int n) {
    map<int, int> ourmap;      //<sum till that index, index>

    ourmap[0] = -1;    //taking -1th index sum as 0
    int sum = 0;
    int maxlength = 0;
    for(int i = 0; i < n; i++) {
        sum += arr[i];
        //if same sum key exits finding dist between both sum keys(diff of both
their indexes)
        if(ourmap.count(sum) > 0 && maxlength < i - ourmap[sum]) maxlength = i -
ourmap[sum];                                //Comparing that diff to find max
        else ourmap[sum] = i;
    }

    return maxlength;
}
```

Lecture 16 : Tries and Huffman Coding

1. **Search in Tries:** Implement the function SearchWord for the Trie class.
   For a Trie, write the function for searching a word. Return true if the word is found successfully, otherwise return false.

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

class TrieNode {
   public:
    char data;
    TrieNode **children;
    bool isTerminal;

    TrieNode(char data) {
        this->data = data;
        children = new TrieNode *[26];
        for (int i = 0; i < 26; i++) {
            children[i] = NULL;
        }
        isTerminal = false;
    }
};

class Trie {
    TrieNode *root;
    public:
    Trie() {
        root = new TrieNode('\0');
    }

    void insertWord(TrieNode *root, string word) {
        // Base case
        if (word.size() == 0) {
            root->isTerminal = true;
            return;
        }

        // Small Calculation
        int index = word[0] - 'a';
        TrieNode *child;
        if (root->children[index] != NULL) {
            child = root->children[index];
        } else {
            child = new TrieNode(word[0]);
            root->children[index] = child;
        }
        // Recursive call
        insertWord(child, word.substr(1));
    }

    void insertWord(string word) {
        insertWord(root, word);
    }

    bool search(TrieNode *root, string word) {
        if(word.length() == 0) {    //Base Case
            return false;
        }

        int index = word[0] - 'a';
        if(root->children[index] != NULL) {
            if(word.size() == 1 && root->children[index]->isTerminal == true)
return true;
            else return search(root->children[index], word.substr(1));
        }
        else return false;
    }

    bool search(string word) {
        return search(root, word);
    }
};

int main() {
    int choice;
    cin >> choice;
    Trie t;

    while (choice != -1) {
        string word;
        bool ans;
        switch (choice) {
            case 1:  // insert
                cin >> word;
                t.insertWord(word);
                break;
            case 2:  // search
                cin >> word;
                cout << (t.search(word) ? "true\n" : "false\n");
                break;
            default:
                return 0;
        }
        cin >> choice;
    }
    return 0;
}
```

2. **Pattern Matching:** Given a list of n words and a pattern p that we want to search. Check if the pattern p is present the given words or not. Return true if the pattern is present and false otherwise.

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

class TrieNode {
   public:
    char data;
    TrieNode **children;
    bool isTerminal;

    TrieNode(char data) {
        this->data = data;
        children = new TrieNode *[26];
        for (int i = 0; i < 26; i++) {
            children[i] = NULL;
        }
        isTerminal = false;
    }
};

class Trie {
    TrieNode *root;

   public:
    int count;

    Trie() {
        this->count = 0;
        root = new TrieNode('\0');
    }

    bool insertWord(TrieNode *root, string word) {
        // Base case
        if (word.size() == 0) {
            if (!root->isTerminal) {
                root->isTerminal = true;
                return true;
            } else {
                return false;
            }
        }

        // Small calculation
        int index = word[0] - 'a';
        TrieNode *child;
        if (root->children[index] != NULL) {
            child = root->children[index];
        } else {
            child = new TrieNode(word[0]);
            root->children[index] = child;
        }

        // Recursive call
        return insertWord(child, word.substr(1));
    }

    void insertWord(string word) {
        if (insertWord(root, word)) {
            this->count++;
        }
    }


////////////////////////////////////////////////////////////////////////////////
    //My Code:
    bool patternMatching(TrieNode* root, string pattern){
        if(pattern.size() == 0) {     //Base case
            return true;
        }

        int index = pattern[0] - 'a';
        if(root->children[index] != NULL) {        //Small Calculation
            return patternMatching(root->children[index], pattern.substr(1));
//Recursion
        }
        else return false;
    }

    bool patternMatching(vector<string> vect, string pattern) {
        for(int i = 0; i < vect.size(); i++) {          //forming suffix trie
            for(int j = 0; j < vect[i].size(); j++) {
                insertWord(vect[i].substr(j));
            }
        }
        return patternMatching(root, pattern);

////////////////////////////////////////////////////////////////////////////////
    }
};


int main() {
    Trie t;
    int n;
    cin >> n;
    string pattern;
    vector<string> vect;
    for (int i = 0; i < n; ++i) {
        string word;
        cin >> word;
        vect.push_back(word);
    }
    cin >> pattern;

    cout << (t.patternMatching(vect, pattern) ? "true" : "false");
}
```

3. **Palindrome Pair:** Given 'n' number of words, you need to find if there exist any two words which can be joined to make a palindrome or any word, which itself is a palindrome. The function should return either true or false. You don't have to print anything.

```
#include <bits/stdc++.h>
using namespace std;

class TrieNode {
   public:
    char data;
    TrieNode **children;
    bool isTerminal;
    int childCount;

    TrieNode(char data) {
        this->data = data;
        children = new TrieNode *[26];
        for (int i = 0; i < 26; i++) {
            children[i] = NULL;
        }
        isTerminal = false;
        childCount = 0;
    }
};

class Trie {
    TrieNode *root;

   public:
    int count;

    Trie() {
        this->count = 0;
        root = new TrieNode('\0');
    }

    bool add(TrieNode *root, string word) {
        // Base case
        if (word.size() == 0) {
            if (!root->isTerminal) {
                root->isTerminal = true;
                return true;
            } else {
                return false;
            }
        }

        // Small calculation
        int index = word[0] - 'a';
        TrieNode *child;
        if (root->children[index] != NULL) {
            child = root->children[index];
        } else {
            child = new TrieNode(word[0]);
            root->children[index] = child;
            root->childCount++;
        }

        // Recursive call
        return add(child, word.substr(1));
    }

    void add(string word) {
        if (add(root, word)) {
            this->count++;
        }
    }

    /*......................... Palindrome Pair......................... */
    //My code:
    bool isPalindrome(string word) {
        int l = 0;
        int h = word.size() - 1;

        while (h > l)
        {
            if (word[l++] != word[h--]) return false;
        }
        return true;
    }

    bool search(TrieNode *root, string word) {
        if(word.length() == 0) {    //Base Case
            return false;
        }

        int index = word[0] - 'a';
        if(root->children[index] != NULL) {
            if(word.size() == 1 && root->children[index]->isTerminal == true)
```

```
return true;
            else return search(root->children[index], word.substr(1));
        }
        else return false;
    }

    bool search(string word) {
        return search(root, word);
    }

    bool isPalindromePair(vector<string> words) {
        for(int i = 0; i < words.size(); i++) {    //forming tree
            add(words[i]);
        }
        for(int j = 0; j < words.size(); j++) {
            string temp = words[j];
            reverse(words[j].begin(), words[j].end());
                                          //too lazy to code reverse(thus using STL)
            int len = words[j].size();
            if (search(words[j]) == true || search(words[j].substr(1)) == true
|| search(words[j].substr(0, len - 1)) == true || isPalindrome(temp) == true)
return true;                                  //pair palindrome or itself palindrome
        }
//since we need to find if addition of 2 strings is palindrome thus we try
neglecting 1st/last term(which becomes middle) to check if thats a palindrome
        return false;
    }
};


int main() {
    Trie t;
    int n;
    cin >> n;
    vector<string> words(n);

    for (int i = 0; i < n; ++i) {
        cin >> words[i];
    }
    cout << (t.isPalindromePair(words) ? "true" : "false");
}
```

4. **Auto complete:** Given n number of words and an incomplete word w. You need to auto-complete that word w. That means, find and print all the possible words which can be formed using the incomplete word w.
   Note : Order of words does not matter.

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

class TrieNode {
   public:
    char data;
    TrieNode **children;
    bool isTerminal;

    TrieNode(char data) {
        this->data = data;
        children = new TrieNode *[26];
        for (int i = 0; i < 26; i++) {
            children[i] = NULL;
        }
        isTerminal = false;
    }
};

class Trie {
    TrieNode *root;

   public:
    int count;

    Trie() {
        this->count = 0;
        root = new TrieNode('\0');
    }

    bool insertWord(TrieNode *root, string word) {
        // Base case
        if (word.size() == 0) {
            if (!root->isTerminal) {
                root->isTerminal = true;
                return true;
            } else {
                return false;
            }
        }

        // Small calculation
        int index = word[0] - 'a';
        TrieNode *child;

        if (root->children[index] != NULL) {
            child = root->children[index];
        } else {
            child = new TrieNode(word[0]);
            root->children[index] = child;
        }

        // Recursive call
        return insertWord(child, word.substr(1));
    }

    void insertWord(string word) {
        if (insertWord(root, word)) {
            this->count++;
        }
    }

    /*........................... Auto complete ........................... */
    //My code:
    void print(TrieNode* node, string Ans, string pattern) {
        if(node == NULL) return;    //base case

        Ans = Ans + node->data;
        if(node->isTerminal == true) cout<<pattern.substr(0, pattern.size() -
1)<<Ans<<endl;    //pattern + Ans = output


        for(int i = 0; i < 26; i++) {
            print(node->children[i], Ans, pattern);
        }
    }

    void autoComplete(vector<string> input, string pattern) {
        for(int i = 0; i < input.size(); i++) {    //forming trie
            insertWord(input[i]);
        }

        TrieNode *temp = root;          //temp points to last node of pattern
        for(int i = 0; i < pattern.size(); i++) {
            int index = pattern[i] - 'a';
            temp = temp->children[index];
            if(temp == NULL) return;
        }

        string Ans;
        print(temp, Ans, pattern);
    }
};


int main() {
    Trie t;
    int n;
    cin >> n;
    vector<string> vect;

    for (int i = 0; i < n; ++i) {
        string word;
        cin >> word;
        vect.push_back(word);
    }

    string pattern;
    cin >> pattern;

    t.autoComplete(vect, pattern);
}
```

Lecture 17 : DP - 1

1. **Min Steps to 1:** Given a positive integer 'n', find and return the minimum number of steps that 'n' has to take to get reduced to 1.
   You can perform any one of the following 3 steps:
   1.) Subtract 1 from it. (n = n - 1) ,
   2.) If it's divisible by 2, divide by 2.(if n % 2 == 0, then n = n / 2) ,
   3.) If it's divisible by 3, divide by 3. (if n % 3 == 0, then n = n / 3).
   Write brute-force recursive solution for this.

```cpp
#include <bits/stdc++.h>
using namespace std;

int countMinStepsToOne(int n) {
	if(n == 1) return 0;
	else if(n < 1) return INT_MAX;

	int case1 = countMinStepsToOne(n - 1);
	int case2 = INT_MAX;
	if(n % 2 == 0) case2 = countMinStepsToOne(n / 2);
	int case3 = INT_MAX;
	if(n % 3 == 0) case3 = countMinStepsToOne(n / 3);
	return min(case1, min(case2, case3)) + 1;
}

int main()
{
	int n;
	cin >> n;
	cout << countMinStepsToOne(n);
}
```

2. **Min Steps to 1 using DP:**

```cpp
#include <bits/stdc++.h>
using namespace std;

int countHelper(int n, int *ans) {
    if(n == 1) return 0;
    if(ans[n] != -1) return ans[n];

    int case1, case2 = INT_MAX, case3 = INT_MAX;

    if(ans[n - 1] != -1) case1 = ans[n - 1];
    else case1 = countHelper(n - 1, ans);

    if(n % 2 == 0) {
        if(ans[n / 2] != -1) case2 = ans[n / 2];
        else case2 = countHelper(n / 2, ans);
    }
    if(n % 3 == 0) {
        if(ans[n / 3] != -1) case3 = ans[n / 3];
        else case3 = countHelper(n / 3, ans);
    }

    ans[n] = min(case1, min(case2, case3)) + 1;
    return ans[n];
}

int countStepsToOne(int n) {
       int *ans = new int[n + 1];
       for(int i = 0; i <= n; i++) {
        ans[i] = -1;
       }
    return countHelper(n, ans);
}

int main()
{
       int n;
       cin >> n;
       cout << countStepsToOne(n);
}
```

3. **Staircase using Dp:** A child runs up a staircase with 'n' steps and can hop either 1 step, 2 steps or 3 steps at a time. Implement a method to count and return all possible ways in which the child can run-up to the stairs.

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

long counthelper(int n, long *ans) {
    if(n == 0 || n == 1) return 1;    //Base case
    else if(n == 2) return 2;

    if(ans[n] != -1) return ans[n];   //return if already calculated

    long case1, case2, case3;         //recursion

    if(ans[n - 1] != -1) case1 = ans[n - 1];
    else case1 = counthelper(n - 1, ans);

    if(ans[n - 2] != -1) case2 = ans[n - 2];
    else case2 = counthelper(n - 2, ans);

    if(ans[n - 3] != -1) case3 = ans[n - 3];
    else case3 = counthelper(n - 3, ans);

    ans[n] = case1 + case2 + case3;   //Small calculation
    return ans[n];
}

long staircase(int n) {
	long *ans = new long[n + 1];
	for(int i = 0; i <= n; i++) {
	  ans[i] = -1;
	}
	return counthelper(n, ans);
}


int main()
{
	int n;
	cin >> n;
	cout << staircase(n);
}
```

4. **Minimum Count:** Given an integer N, find and return the count of minimum numbers required to represent N as a sum of squares.
   That is, if N is 4, then we can represent it as : {1^2 + 1^2 + 1^2 + 1^2} and {2^2}. The output will be 1, as 1 is the minimum count of numbers required to represent N as sum of squares.

```
#include <bits/stdc++.h>
using namespace std;

int counthelper(int n, int *ans) {
	if(n == 0) return 0;
	if(n < 0) return INT_MAX;

	if(ans[n] != -1) return ans[n];

	int size = sqrt(n);
	int mincount = INT_MAX;
	for(int i = 1; i <= size; i++) {
	  int next = n - (i * i);
	  int temp;
	  if(ans[next] != -1) {
	      temp = ans[next];
	  }
	  else {
	      temp = counthelper(next, ans);
	      ans[next] = temp;
	  }

	  if(ans[next] < mincount) mincount = ans[next];
	}
    return mincount + 1;
}
```

```
int minCount(int n) {
       int *ans = new int[n + 1];
       for(int i = 0; i <= n; i++) {
        ans[i] = -1;
       }
    return counthelper(n , ans);
}

int main()
{
       int n;
       cin >> n;
       cout << minCount(n);
}
```

5. **No. of balanced BTs:** Given an integer h, find the possible number of balanced binary trees of height h. You just need to return the count of possible binary trees which are balanced. This number can be huge, so, return output modulus 10^9 + 7.
   Write a simple recursive solution.

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

#include <cmath>
int balancedBTs(int n) {
    if(n == 1 || n == 0) return 1;

    int mod = (pow(10, 9)) + 7;
          //since we have int if a value goes out of range we take mod with this
    int x = balancedBTs(n - 1);
    int y = balancedBTs(n - 2);

    int temp1 = (int)(((long)(x)*x) % mod);       //typecasting to long
    int temp2 = (int)((2*(long)(x)*y) % mod);
                                             //long*int stores in temporary long container
    int Ans = (temp1 + temp2) % mod;              //thus typecasting back to int

    return Ans;
}

int main() {
    int n;
    cin >> n;
    cout << balancedBTs(n);
}
```

6. **No. of balanced BTs using DP:** Time complexity should be O(h).

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

#include <cmath>
int counthelper(int n, int *ans) {
    if(n == 1 || n == 0) return 1;
    if(ans[n] != -1) return ans[n];

    int mod = (pow(10, 9)) + 7;
//since we have int if a value goes out of range we take mod with this

    int x, y;
    if(ans[n - 1] != -1) x = ans[n - 1];
    else x = counthelper(n - 1, ans);
    if(ans[n - 2] != -1) y = ans[n - 2];
    else y = counthelper(n - 2, ans);

    int temp1 = (int)(((long)(x)*x) % mod);       //typecasting to long
    int temp2 = (int)((2*(long)(x)*y) % mod);
                                   //long*int stores in temporary long container
    ans[n] = (temp1 + temp2) % mod;               //thus typecasting back to int

    return ans[n];
}

int balancedBTs(int n) {
    int *ans = new int[n + 1];
    for(int i = 0; i <= n; i++) {
        ans[i] = -1;
    }
    return counthelper(n, ans);
}

int main() {
    int n;
    cin >> n;
    cout << balancedBTs(n);
}
```

Lecture 18 : DP - 2

1. **Min Cost Path:** An integer matrix of size (M x N) has been given. Find out the minimum cost to reach from the cell (0, 0) to (M - 1, N - 1).
   From a cell (i, j), you can move in three directions:
   1. ((i + 1), j) which is, "down"
   2. (i, (j + 1)) which is, "to the right"
   3. ((i+1), (j+1)) which is, "to the diagonal"

   The cost of a path is defined as the sum of each cell's values through which the route passes.

```cpp
#include <bits/stdc++.h>
using namespace std;

int minhelper(int **input, int m, int n, int i, int j) {
    if(i == m - 1 && j == n - 1) return input[i][j];    //Base case
    if(i >= m || j >= n) return INT_MAX;

    int case1 = minhelper(input, m, n, i + 1, j);      //Down
    int case2 = minhelper(input, m ,n, i, j + 1);      //right
    int case3 = minhelper(input, m, n, i + 1, j + 1);  //diagonal

    return min(case1, min(case2, case3)) + input[i][j];
}

int minCostPath(int **input, int m, int n) {
	return minhelper(input, m, n, 0, 0);
}

int main()
{
	int **arr, n, m;
	cin >> n >> m;
	arr = new int *[n];
	for (int i = 0; i < n; i++)
	{
		arr[i] = new int[m];
	}
	for (int i = 0; i < n; i++)
	{
		for (int j = 0; j < m; j++)
		{
			cin >> arr[i][j];
		}
	}
	cout << minCostPath(arr, n, m) << endl;
}
```

2. **Edit Distance:** Given two strings s and t of lengths m and n respectively, find the edit distance between the strings.

Edit Distance -> Edit Distance of two strings is the minimum number of operations required to make one string equal to another. In order to do so you can perform any of the following three operations only :
1. Delete a character
2. Replace a character with another one
3. Insert a character

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

int editDistance(string s1, string s2) {
       int m = s1.size();
       int n = s2.size();

       if(m == 0 || n == 0) {    //remaining terms must be deleted
         return max(m, n);        //Ex: goodbye & good
       }                           //-> bye must be deleted thus return 3

       if(s1[0] == s2[0]) {
         return editDistance(s1.substr(1), s2.substr(1));
       }
       else {
         int a = editDistance(s1.substr(1), s2);            //insert
         int b = editDistance(s1, s2.substr(1));            //delete
         int c = editDistance(s1.substr(1), s2.substr(1));  //replace
         int ans = min(a, min(b, c));
         return ans  + 1;
       }
}

int main() {
    string s1;
    string s2;
    cin >> s1;
    cin >> s2;
    cout << editDistance(s1, s2);
}
```

3. **Edit Distance (Memoization) :**

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

int editDistance(string s1, string s2, int **output) {
    int m = s1.size();
    int n = s2.size();
    if(m == 0 || n == 0) {
        return max(m, n);
    }
    if(output[m][n] != -1) {
        return output[m][n];
    }

    int ans;
    if(s1[0] == s2[0]) {
        ans = editDistance(s1.substr(1), s2.substr(1), output);
    }
    else {
        int a = editDistance(s1.substr(1), s2, output);       //insert
        int b = editDistance(s1, s2.substr(1), output);       //delete
        int c = editDistance(s1.substr(1), s2.substr(1), output);   //replace
        ans = min(a, min(b, c)) + 1;
    }
    output[m][n] = ans;
    return ans;
}

int editDistance(string s1, string s2) {
    int m = s1.size();
      int n = s2.size();

      int **output = new int*[m + 1];     //forming output 2d array
      for(int i = 0; i <= m; i++) {
        output[i] = new int[n + 1];
        for(int j = 0; j <= n; j++) {
            output[i][j] = -1;          //initializing to -1
        }
      }

      return editDistance(s1, s2, output);
}

int main()
{
      string s1;
      string s2;
      cin >> s1;
      cin >> s2;
      cout << editDistance(s1, s2) << endl;
}
```

4. **Knapsack (Brute force) :** A thief robbing a store can carry a maximal weight of W into his knapsack. There are N items, and i-th item weigh 'Wi' and the value being 'Vi.' What would be the maximum value V, that the thief can steal?

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

int knapsack(int *weights, int *values, int n, int maxWeight) {
	if (n == 0 || maxWeight == 0) return 0;
	if(weights[0] > maxWeight) {
	  return knapsack(weights + 1, values + 1, n - 1, maxWeight);
	}
	int case1 = knapsack(weights + 1, values + 1, n - 1, maxWeight -
weights[0]) + values[0];                                        //include
	int case2 = knapsack(weights + 1, values + 1, n - 1, maxWeight);
                                                                //not include
	return max(case1, case2);
}

int main()
{
	int n;
	cin >> n;
	int *weights = new int[n];
	int *values = new int[n];
	for (int i = 0; i < n; i++)
	{
		cin >> weights[i];
	}
	for (int i = 0; i < n; i++)
	{
		cin >> values[i];
	}
	int maxWeight;
	cin >> maxWeight;
	cout << knapsack(weights, values, n, maxWeight) << endl;
	delete[] weights;
	delete[] values;
}
```

5. **Knapsack (Memoization and DP):**

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

//Memoization
int knapsack(int* weight, int* value, int v, int w, int **output) {
    if(v < 0) return 0;
    if(output[v][w] != -1) return output[v][w];

    int ans;
    if(weight[v] > w) {
        ans = knapsack(weight, value, v - 1, w, output);
    }
    else {
        int a = knapsack(weight, value, v - 1, w - weight[v], output) +
value[v];                                                           //include
        int b = knapsack(weight, value, v - 1, w, output);          //not include
        ans = max(a, b);
    }

    output[v][w] = ans;
    return ans;
}

int knapsack(int* weight, int* value, int n, int maxWeight) {
       int **output = new int*[n + 1];
       for(int i = 0; i <= n; i++) {
        output[i] = new int[maxWeight + 1];
        for(int j = 0; j <= maxWeight; j++) {
            output[i][j] = -1;
        }
       }
       return knapsack(weight, value, n, maxWeight, output);
}

int main() {
    int n;
    cin >> n;
    int* wt = new int[n];
    int* val = new int[n];
    for (int i = 0; i < n; i++) {
        cin >> wt[i];
    }
    for (int j = 0; j < n; j++) {
        cin >> val[j];
    }
    int w;
    cin >> w;
    cout << knapsack(wt, val, n, w) << "\n";
    delete[] wt;
    delete[] val;
}
```

6. **Loot Houses:** A thief wants to loot houses. He knows the amount of money in each house. He cannot loot two consecutive houses. Find the maximum amount of money he can loot.

   Memoization:

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

int maxMoneyLooted(int *arr, int n, int *output) {
    if(n <= 0) return 0;                        //base case
    if(output[0] != -1) return output[0];  //return if answer already calculated

    int case1 = maxMoneyLooted(arr + 2, n - 2, output + 2) + arr[0];
                                                    //looting and skipping next house
    int case2 = maxMoneyLooted(arr + 1, n - 1, output + 1);
                                               //skipping this house and looting next
    output[0] = max(case1, case2);             //save for future use
    return output[0];
}

int maxMoneyLooted(int *arr, int n) {
      int *output = new int[n + 1];
      for(int i = 0; i <= n; i++) {
        output[i] = -1;
      }
      return maxMoneyLooted(arr, n, output);
}

int main()
{
      int n;
      cin >> n;
      int *arr = new int[n];
      for (int i = 0; i < n; i++)
      {
            cin >> arr[i];
      }
      cout << maxMoneyLooted(arr, n);
      delete[] arr;
}
```

DP:

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

int maxMoneyLooted(int *arr, int n) {
      int *output = new int[n + 2];
      output[n + 1] = 0;
      output[n] = 0;

      for(int i = n - 1; i >= 0; i--) {
        int a = output[i + 2] + arr[i];
        int b = output[i + 1];
        output[i] = max(a, b);
      }

      return output[0];
}
```

```
int main()
{
	int n;
	cin >> n;
	int *arr = new int[n];
	for (int i = 0; i < n; i++)
	{
		cin >> arr[i];
	}
	cout << maxMoneyLooted(arr, n);
	delete[] arr;
}
```

7. **Longest Increasing Subsequence:** Given an array with N elements, you need to find the length of the longest subsequence in the given array such that all elements of the subsequence are sorted in strictly increasing order.

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

int longestIncreasingSubsequence(int* arr, int n) {
   int *output = new int[n];
   int max = 0;
   for(int i = 0; i <n; i++) {
      output[i] = 1;
      for(int j = i - 1; j >=0; j--) {
        if(arr[j] < arr[i] && output[j] >= output[i]) output[i] = output[j] + 1;
//checking max subsequence of element less than our element and then adding it
to the back of that subsequence
        if(output[i] > max) max = output[i];
      }
   }
   return max;
}

int main() {
    int n;
    cin >> n;
    int* arr = new int[n];

    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }
    cout << longestIncreasingSubsequence(arr, n);
}
```

8. **All possible ways:** Given two integers a and b. You need to find and return the count of possible ways in which we can represent the number a as the sum of unique integers raise to the power b.
   Ex: 100 2
   Following are the three ways:
   1. 100 = 10^2
   2. 100 = 8^2 + 6^2
   3. 100 = 7^2+5^2+4^2+3^2+1^2 ; Ans is 3

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

//solution uses backtracking
#include<cmath>
int getAllWays(int a, int b, int limit) {
	if(a == 0) return 1;
	if(a < 0 || limit < 1) return 0;

	int case1 = getAllWays(a - pow(limit, b), b, limit - 1);
	int case2 = getAllWays(a, b, limit - 1);
	return case1 + case2;
}

int getAllWays(int a, int b) {
	int limit = pow(a, (1.0/b));
    return getAllWays(a, b, limit);
}

int main()
{
    int a, b;
    cin >> a >> b;
    cout << getAllWays(a, b);
}
```

9. **Ways To Make Coin Change:** For the given infinite supply of coins of each denomination, D = {D0, D1, D2, D3, ... Dn-1}. You need to figure out the total number of ways W, in which you can make the change for Value V using coins of denominations D.
   Return 0 if the change isn't possible.
   Ex: 1,2,3 (3 coins) -> Number of ways are - 4 total i.e. (1,1,1,1), (1,1, 2), (1, 3) and (2, 2).

```
//https://www.youtube.com/watch?v=DJ4a7cmjZY0&ab_channel=BackToBackSWE

#include <iostream>
using namespace std;

int countWaysToMakeChange(int denominations[], int numDenominations, int value){
	int **output = new int*[numDenominations + 1];
	for(int i = 0; i <= numDenominations; i++) {
         output[i] = new int[value + 1];
         for(int j = 1; j <= value; j++) {
             output[i][j] = -1;
         }
        }

        for(int i = 0; i <= numDenominations; i++) {    //1st column = 1
         output[i][0] = 1;
        }

        for(int j = 1; j <= value; j++) {               //1st row = 0
         output[0][j] = 0;
        }

     for(int i = 1; i <= numDenominations; i++) {    //filling rest cells
         for(int j = 1; j <= value; j++) {
             int case1 = output[i - 1][j];     //not including coin
             int case2 = 0;                    //including coin
             if(j >= denominations[i - 1]) {
                 case2 = output[i][j - denominations[i - 1]];
             }
             output[i][j] = case1 + case2;
         }
        }
        return output[numDenominations][value];
}

int main()
{

        int numDenominations;
        cin >> numDenominations;

        int *denominations = new int[numDenominations];

        for (int i = 0; i < numDenominations; i++)
        {
               cin >> denominations[i];
        }

        int value;
        cin >> value;

        cout << countWaysToMakeChange(denominations, numDenominations, value);

        delete[] denominations;
}
```

10. **Matrix Chain Multiplication:** Given a chain of matrices A1, A2, A3,.....An, you have to figure out the most efficient way to multiply these matrices. In other words, determine where to place parentheses to minimize the number of multiplications.

You will be given an array p[] of size n + 1. Dimension of matrix Ai is p[i - 1]*p[i]. You need to find the minimum number of multiplications needed to multiply the chain.

Brute force (recursion) :

```
#include <iostream>
#include <limits.h>
using namespace std;

int matrixChainMultiplication(int* arr, int s, int e) {
    if(s == e || s == e - 1) return 0;   //no matrix/single matrix

    int mini = INT_MAX;
    for(int k = s + 1; k <= e - 1; k++) {
        int smallAns = matrixChainMultiplication(arr, s, k) +
matrixChainMultiplication(arr, k, e) + arr[s]*arr[k]*arr[e];
        if(smallAns < mini) mini= smallAns;
    }
    return mini;
}

int matrixChainMultiplication(int* arr, int n) {
    return matrixChainMultiplication(arr, 0, n);
}

int main() {
    int n;
    cin >> n;
    int* arr = new int[n];
    for (int i = 0; i <= n; i++) {
        cin >> arr[i];
    }
    cout << matrixChainMultiplication(arr, n);
    delete[] arr;
}
```

Memoization:

```
#include <iostream>
#include <limits.h>
using namespace std;

int matrixChainMultiplication(int* arr, int s, int e, int **output) {
    if(s == e || s == e - 1) return 0;   //no matrix/single matrix
    if(output[s][e] != -1) return output[s][e];

    int mini = INT_MAX;
    for(int k = s + 1; k <= e - 1; k++) {
        int smallAns = matrixChainMultiplication(arr, s, k, output) +
matrixChainMultiplication(arr, k, e, output) + arr[s]*arr[k]*arr[e];
```

```
        if(smallAns < mini) mini = smallAns;
    }
    output[s][e] = mini;
    return output[s][e];
}

int matrixChainMultiplication(int* arr, int n) {
    int **output = new int*[n + 1];
    for(int i = 0; i <= n ; i++) {
        output[i] = new int[n + 1];
        for(int j = 0; j <= n; j++) {
            output[i][j] = -1;
        }
    }
    return matrixChainMultiplication(arr, 0, n, output);
}

int main() {
    int n;
    cin >> n;
    int* arr = new int[n];
    for (int i = 0; i <= n; i++) {
        cin >> arr[i];
    }
    cout << matrixChainMultiplication(arr, n);
    delete[] arr;
}
```

11. **Coin Tower:** Whis and Beerus are playing a new game today. They form a tower of N coins and make a move in alternate turns. Beerus plays first. In one step, the player can remove either 1, X, or Y coins from the tower. The person to make the last move wins the game. Can you find out who wins the game?

    Memoization:

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

int findWinner(int n, int x, int y, int *output) {
    if(n == 0) return 0;
    if(n < 0) return 1;

    if(output[n] != -1) return output[n];

    int case1 = findWinner(n - 1, x, y, output);
    int case2 = findWinner(n - x, x, y, output);
    int case3 = findWinner(n - y, x, y, output);

    if(case1 == 1 && case2 == 1 && case3 == 1) output[n] = 0;
```

```
                                    //if all possibilities are win then you will lose
    else output[n] = 1;
                         //if any 1 possibility is lose for next player, you will win
    return output[n];
}

string findWinner(int n, int x, int y) {
       int *output = new int[n + 1];
       for(int i = 0; i <= n; i++) {
        output[i] = -1;
       }

       int ans = findWinner(n , x, y, output);

       if(ans == 1) return "Beerus";
       else if(ans == 0) return "Whis";
}

int main()
{
       int n, x, y;
       cin >> n >> x >> y;
       cout << findWinner(n, x, y);
}
```

DP:

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

string findWinner(int n, int x, int y) {
       int *output = new int[n + 1];
    output[0] = 0;
    for(int i = 1; i <= n; i++) {
        if(output[i - 1] == 0) output[i] = 1;
        else if(i - x >= 0 && output[i - x] == 0) output[i] = 1;
        else if(i - y >= 0 && output[i - y] == 0) output[i] = 1;
        else output[i] = 0;
    }
    if(output[n] == 1) return "Beerus";
    else if(output[n] == 0) return "Whis";
}

int main()
{
       int n, x, y;
       cin >> n >> x >> y;
       cout << findWinner(n, x, y);
}
```

12. **Maximum Square Matrix With All Zeros:** Given an NxM matrix that contains only 0s and 1s, find out the size of the maximum square sub-matrix with all 0s. You need to return the size of the square matrix with all 0s.

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

int findMaxSquareWithAllZeros(int **arr, int n, int m){
       int **output = new int*[n];
       for(int i = 0; i < n; i++) {
         output[i] = new int[m];
       }

      int maxi = 0;
       for(int i = 0; i < n; i++) {             //filling 1st column
         if(arr[i][0] == 0) {
              output[i][0] = 1;
              if(maxi == 0) maxi = 1;
         }
         else output[i][0] = 0;
       }
       for(int j = 1; j < m; j++) {             //filling 1st row
         if(arr[0][j] == 0) {
              output[0][j] = 1;
              if(maxi == 0) maxi = 1;
         }
         else output[0][j] = 0;
       }

       for(int i = 1; i < n; i++) {             //filling the rest
         for(int j = 1; j < m; j++) {
              if(arr[i][j] == 1) output[i][j] = 0;
              else {
                  int a = output[i][j - 1];        //left
                  int b = output[i - 1][j];        //up
                  int c = output[i - 1][j - 1];    //diagonal
                  output[i][j] = min(a, min(b, c)) + 1;
                  if (maxi < output[i][j]) maxi = output[i][j];
              }
         }
       }
       return maxi;
}

int main()
{
       int **arr, n, m, i, j;
       cin >> n >> m;
       arr = new int *[n];
       for (i = 0; i < n; i++)
       {
```

```
                arr[i] = new int[m];
        }
        for (i = 0; i < n; i++)
        {
                for (j = 0; j < m; j++)
                {
                        cin >> arr[i][j];
                }
        }
        cout << findMaxSquareWithAllZeros(arr, n, m) << endl;
        delete[] arr;
        return 0;
}
```

13. **Shortest Subsequence:** Gary has two strings S and V. Now, Gary wants to know the length shortest subsequence in S, which is not a subsequence in V.
    Note: The input data will be such that there will always be a solution.

    Brute force (Recursion) :

```
#include <iostream>
#include <string>
#include<limits.h>
using namespace std;

int solve(string s, string v) {
    if(s.size() == 0) return INT_MAX - 1;
                                    //(INT_MAX + 1 below[IN B] was going out of range)
    if(v.size() <= 0) return 1;

    int a = solve(s.substr(1), v);                    //Not including 1st term of s

    int i;
    for(i = 0; i < v.size(); i++) {
        if(s[0] == v[i]) {
            break;
        }
    }
    if(i == v.size()) return 1;

    int b = solve(s.substr(1), v.substr(i + 1)) + 1;    //including 1st term of s
    return min(a, b);
}

int main() {
    string s, v;
    cin >> s >> v;
    cout << solve(s, v);
}
```

Memoization:

```
#include <iostream>
#include <string>
#include<limits.h>
using namespace std;

int solve(string s, string v, int **output) {
    int n = s.size();
    int m = v.size();

    if(n == 0) return INT_MAX - 1;
    if(m <= 0) return 1;

    if(output[n][m] != -1) return output[n][m];

    int a = solve(s.substr(1), v, output);         //Not including 1st term of s

    int i;
    for(i = 0; i < v.size(); i++) {
        if(s[0] == v[i]) {
            break;
        }
    }
    if(i == v.size()) return 1;

    int b = solve(s.substr(1), v.substr(i + 1), output) + 1;
                                                       //including 1st term of s
    output[n][m] = min(a, b);
    return output[n][m];
}

int solve(string s, string v) {
    int n = s.size();
    int m = v.size();
    int **output = new int*[n + 1];
    for(int i = 0; i <= n; i++) {
        output[i] = new int[m + 1];
        for(int j = 0; j <= m; j++) {
            output[i][j] = -1;
        }
    }
    return solve(s, v, output);
}

int main() {
    string s, v;
    cin >> s >> v;
    cout << solve(s, v);
}
```

Lecture 19 : Graphs 1

1. **BFS Traversal:** Given an undirected and disconnected graph G(V, E), print its BFS traversal.
   Note:
   1. Here you need to consider that you need to print BFS path starting from vertex 0 only.
   2. V is the number of vertices present in graph G and vertices are numbered from 0 to V-1.
   3. E is the number of edges present in graph G.
   4. Take graph input in the adjacency matrix.
   5. Handle for Disconnected Graphs as well

A) **For connected graphs:**

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

int main() {
    int v, e;
    cin>>v>>e;
    if(v == 0) return 0;

    int **edges = new int*[v];
    for(int i = 0; i < v; i++) {
        edges[i] = new int[v];
        for(int j = 0; j < v; j++) {
            edges[i][j] = 0;
        }
    }

    for(int i = 0; i < e; i++) {
        int j,s;
        cin>>j>>s;
        edges[j][s] = 1;
        edges[s][j] = 1;
    }

    bool *visited = new bool[v];
    for(int i = 0; i < v; i++) {
        visited[i] = false;
    }

    queue<int> remain;
    remain.push(0);
    visited[0] = true;
    while(remain.size() != 0) {
        int i = remain.front();
        remain.pop();
        cout<<i<<" ";
        for(int j = 0; j < v; j++) {
            if(i == j) continue;
            if(edges[i][j] == 1 && !visited[j]) {
                remain.push(j);
                visited[j] = true;
            }
        }
    }

    delete [] visited;
    for(int i = 0; i < v; i++) {
        delete edges[i];
    }
    delete [] edges;
}
```

B) **For disconnected graph:**

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

//printing a connected graph
void printBFS(int **edges, int n, int sv, bool *visited) {
    queue<int> pending;
    pending.push(sv);       //(sv : starting index)
    visited[sv] = true;

    while(!pending.empty()) {
        int i = pending.front();
        pending.pop();
        cout<<i<<" ";
        for(int j = 0; j < n; j++) {
            if(j == i) continue;
            if(edges[i][j] == 1 && !visited[j]) {
                pending.push(j);
                visited[j] = true;
            }
        }
    }
}

void BFS(int **edges, int n) {       //sending starting vertices of a single
connected graph to PrintBFS
    bool *visited = new bool[n];
    for(int i = 0; i < n; i++) {
        visited[i] = false;
    }
    for(int i = 0; i < n; i++) {     //starting from 0th vertex
        if(!visited[i]) {
    //if vertex not printed yet then its entire connected graph will be printed
```

```
            printBFS(edges, n, i, visited);
        }
    }
    delete [] visited;
}

int main() {
    int v, e;
    cin>>v>>e;
    if(v == 0) return 0;

    int **edges = new int*[v];
    for(int i = 0; i < v; i++) {
        edges[i] = new int[v];
        for(int j = 0; j < v; j++) {
            edges[i][j] = 0;
        }
    }

    for(int i = 0; i < e; i++) {
        int j,s;
        cin>>j>>s;
        edges[j][s] = 1;
        edges[s][j] = 1;
    }

    BFS(edges, v);
}
```

2. Has Path: Given an undirected graph G(V, E) and two vertices v1 and v2 (as integers), check if there exists any path between them or not. Print true if the path exists and false otherwise.
Note:
1. V is the number of vertices present in graph G and vertices are numbered from 0 to V-1.
2. E is the number of edges present in graph G.

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

bool check(int **edges, int v, int a, int b, bool *visited) {
    visited[a] = true;
    for(int i = 0; i < v; i++) {
        if(i == a) continue;   //no vertex checking with itself required
        if(edges[a][i] == 1 && !visited[i]) {
            if(i == b) return true;
            else if(check(edges, v, i, b, visited)) return true;
        }
    }
    return false;    //if vertex has no adjacent vertices return false
}
```

```
int main() {
    int v, e;    //vertex/edges
    cin>>v>>e;

    if(v == 0 || e == 0) return 0;

    int **edges = new int*[v];
    for(int i = 0; i < v; i++) {
        edges[i] = new int[v];
        for(int j = 0; j < v; j++) {
            edges[i][j] = 0;
        }
    }

    for(int i = 0; i < e; i++) {
        int j,s;
        cin>>j>>s;
        edges[j][s] = 1;
        edges[s][j] = 1;
    }

    int a,b;     //vertices to be checked
    cin>>a>>b;

    bool *visited = new bool[v];
    for(int i = 0; i < v; i++) {
        visited[i] = false;
    }

    if(check(edges, v, a, b, visited)) cout<<"true";
    else cout<<"false";
}
```

3. Get Path - DFS: Given an undirected graph G(V, E) and two vertices v1 and v2(as integers), find and print the path from v1 to v2 (if exists). Print nothing if there is no path between v1 and v2.
Find the path using DFS and print the first path that you encountered.

Note:

1. V is the number of vertices present in graph G and vertices are numbered from 0 to V-1.
2. E is the number of edges present in graph G.
3. Print the path in reverse order. i.e., print v2 first, then intermediate vertices and v1 at last.
4. Save the input graph in Adjacency Matrix.

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

```
vector<int> check(int **edges, int v, int a, int b, bool *visited) {
    visited[a] = 1;
    vector<int> path;
    if(a == b) {
        path.push_back(b);
        return path;
    }
    for(int i = 0; i < v; i++) {
        if(a == i) continue;
        if(edges[a][i] == 1 && !visited[i]) {
            vector<int> SmallAns = check(edges, v, i, b, visited);
            if(SmallAns.size() != 0) {
                SmallAns.push_back(a);
                return SmallAns;
            }
        }
    }
    return path;
}

int main() {
    int v, e;    //vertex/edges
    cin>>v>>e;


    int **edges = new int*[v];
    for(int i = 0; i < v; i++) {
        edges[i] = new int[v];
        for(int j = 0; j < v; j++) {
            edges[i][j] = 0;
        }
    }

    for(int i = 0; i < e; i++) {
        int j,s;
        cin>>j>>s;
        edges[j][s] = 1;
        edges[s][j] = 1;
    }

    int a,b;     //vertices to be checked
    cin>>a>>b;
    bool *visited = new bool[v];
    for(int i = 0; i < v; i++) {
        visited[i] = false;
    }
    vector<int> path = check(edges, v, a, b, visited);
    for(int i = 0; i < path.size(); i++) {
        cout<<path[i]<<" ";
    }
}
```

4. **Get Path - BFS:** Find the path using BFS and print the shortest path available.

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

int main() {
    int v, e;
    cin>>v>>e;
    if(v == 0) return 0;

    int **edges = new int*[v];
    for(int i = 0; i < v; i++) {
        edges[i] = new int[v];
        for(int j = 0; j < v; j++) {
            edges[i][j] = 0;
        }
    }

    for(int i = 0; i < e; i++) {
        int j,s;
        cin>>j>>s;
        edges[j][s] = 1;
        edges[s][j] = 1;
    }

    int a, b;      //vertex for which path needs to be found
    cin>>a>>b;

    bool *visited = new bool[v];
    for(int i = 0; i < v; i++) {
        visited[i] = false;
    }

    unordered_map<int, int> ourmap;
    queue<int> remain;

    remain.push(a);
    visited[a] = true;
    while(remain.size() != 0) {
        int i = remain.front();
        remain.pop();
        //cout<<i<<" ";
        for(int j = 0; j < v; j++) {
            if(i == j) continue;
            if(edges[i][j] == 1 && !visited[j]) {
                ourmap[j] = i;
                remain.push(j);
                visited[j] = true;
                if(j == b) break;
            }
        }
    }

    int flag = 0;
    int k = b;
    while(ourmap.count(k) > 0) {
        flag = 1;
        cout<<k<<" ";
        k = ourmap[k];
    } if (flag == 1) cout<<k;

    delete [] visited;
    for(int i = 0; i < v; i++) {
        delete edges[i];
    }
    delete [] edges;
}
```

5. Is Connected ?: Given an undirected graph G(V,E), check if the graph G is connected graph or not.
Note:
1. V is the number of vertices present in graph G and vertices are numbered from 0 to V-1.
2. E is the number of edges present in graph G.

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

void check(int **edges, int v, int sv, bool *visited) {    //DFS traversal
    visited[sv] = 1;
    for(int i = 0; i < v; i++) {
        if(sv == i) continue;
        if(edges[sv][i] == 1 && !visited[i]) {
            check(edges, v, i, visited);
        }
    }
}

int main() {
    int v, e;    //vertex/edges
    cin>>v>>e;

    int **edges = new int*[v];
    for(int i = 0; i < v; i++) {
        edges[i] = new int[v];
        for(int j = 0; j < v; j++) {
            edges[i][j] = 0;
        }
    }
    for(int i = 0; i < e; i++) {
        int j,s;
        cin>>j>>s;
        edges[j][s] = 1;
        edges[s][j] = 1;
    }

    bool *visited = new bool[v];
    for(int i = 0; i < v; i++) {
        visited[i] = false;
    }

    check(edges, v, 0, visited); //calling function to check path
                                 //letting it start from vertex 0(not necessary)
    int flag = 1;
    for(int i = 0; i < v; i++) {
        if(visited[i] == 0) flag = 0;
                                  //if a vertex isn't visited then dis-connected
    }

    if(flag == 1) cout<<"true";
    else cout<<"false";
}
```

6. All connected components: Given an undirected graph G(V,E), find and print all the connected components of the given graph G.
Note:
1. V is the number of vertices present in graph G and vertices are numbered from 0 to V-1.
2. E is the number of edges present in graph G.
3. You need to take input in main and create a function which should return all the connected components. And then print them in the main, not inside function.
Print different components in a new line and each component should be printed in increasing order (separated by space). Order of different components doesn't matter.

```
#include <iostream>
#include <algorithm>    //for sort
#include <vector>
using namespace std;

//forming vectors
void DFStraverse(int **edges, int v, int i, bool *visited, vector<int> &vec) {
    vec.push_back(i);            //passed vector by reference(to alter original)
    visited[i] = true;
    for(int j = 0; j < v; j++) {
        if(i == j) continue;
        if(edges[i][j] == 1 && !visited[j]) {
            DFStraverse(edges, v, j, visited, vec);
        }
    }
}

int main() {
    int v, e;
    cin>>v>>e;
    if(v == 0) return 0;

    int **edges = new int*[v];
    for(int i = 0; i < v; i++) {
        edges[i] = new int[v];
        for(int j = 0; j < v; j++) {
            edges[i][j] = 0;
        }
    }

    for(int i = 0; i < e; i++) {
        int j,s;
        cin>>j>>s;
        edges[j][s] = 1;
        edges[s][j] = 1;
    }

    bool *visited = new bool[v];
    for(int i = 0; i < v; i++) {
        visited[i] = false;
    }

    vector<vector<int>> Ans;                //forming vector array of vectors
    for(int i = 0; i < v; i++) {
        if(!visited[i]) {
            vector<int> SmallAns;
            DFStraverse(edges, v, i, visited, SmallAns);
            Ans.push_back(SmallAns);
        }
    }

    for(int i = 0; i < Ans.size(); i++) {    //printing all disconnected graphs
        sort(Ans[i].begin(), Ans[i].end());
        for(int j = 0; j < Ans[i].size(); j++) {
            cout<<Ans[i][j]<<" ";
        }
        cout<<endl;
    }
}
```

7. Islands: An island is a small piece of land surrounded by water . A group of islands is said to be connected if we can reach from any given island to any other island in the same group . Given V islands (numbered from 1 to V) and E connections or edges between islands. Can you count the number of connected groups of islands?

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

void DFStraverse(int **edges, int v, int i, bool *visited) {
    visited[i] = true;
    for(int j = 0; j < v; j++) {
        if(i == j) continue;
        if(edges[i][j] == 1 && !visited[j]) {
            DFStraverse(edges, v, j, visited);
        }
    }
}

int main() {
    int v, e;
    cin>>v>>e;

    int **edges = new int*[v];
    for(int i = 0; i < v; i++) {
        edges[i] = new int[v];
        for(int j = 0; j < v; j++) {
            edges[i][j] = 0;
        }
    }

    for(int i = 0; i < e; i++) {
        int j,s;
        cin>>j>>s;
        edges[j][s] = 1;
        edges[s][j] = 1;
    }

    bool *visited = new bool[v];
    for(int i = 0; i < v; i++) {
        visited[i] = false;
    }

    int count = 0;
    for(int i = 0; i < v; i++) {
        if(!visited[i]) {          //every time an unvisited vertex is found
            count++;               //it is part of new island
            DFStraverse(edges, v, i, visited);
        }
    }
    cout<<count;
}
```

8. Coding Ninjas: Given a NxM matrix containing Uppercase English Alphabets only. Your task is to tell if there is a path in the given matrix which makes the sentence "CODINGNINJA". There is a path from any cell to all its neighbouring cells. For a particular cell, neighbouring cells are those cells that share an edge or a corner with the cell.

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

bool dfs(vector<vector<char>> &board, int n, int m, int i, int j, bool
**visited, string c) {
    if(c.size() == 0) return true;
    visited[i][j] = true;
    bool ans = false;

    if(i-1>=0 && j-1>=0 && board[i-1][j-1]==c[0] && !visited[i-1][j-1])
        if(dfs(board,n,m,i-1,j-1,visited,c.substr(1))) ans = true;

    if(i-1>=0 && board[i-1][j]==c[0] && !visited[i-1][j])
        if(dfs(board,n,m,i-1,j,visited,c.substr(1))) ans = true;;

    if(i-1>=0 && j+1<m && board[i-1][j+1]==c[0] && !visited[i-1][j+1])
        if(dfs(board,n,m,i-1,j+1,visited,c.substr(1))) ans = true;

    if(j+1<m && board[i][j+1]==c[0] && !visited[i][j+1])
        if(dfs(board,n,m,i,j+1,visited,c.substr(1))) ans = true;

    if(i+1<n && j+1<m && board[i+1][j+1]==c[0] && !visited[i+1][j+1])
        if(dfs(board,n,m,i+1,j+1,visited,c.substr(1))) ans = true;

    if(i+1<n && board[i+1][j]==c[0] && !visited[i+1][j])
        if(dfs(board,n,m,i+1,j,visited,c.substr(1))) ans = true;

    if(i+1<n && j-1>=0 && board[i+1][j-1]==c[0] && !visited[i+1][j-1])
        if(dfs(board,n,m,i+1,j-1,visited,c.substr(1))) ans = true;

    if(j-1>=0 && board[i][j-1]==c[0] && !visited[i][j-1])
        if(dfs(board,n,m,i,j-1,visited,c.substr(1))) ans = true;

    visited[i][j] = false;   //since we will backtrack now, wrong path should be
    return ans;          //changed back to false or it wont traverse on next run
}

bool hasPath(vector<vector<char>> &board, int n, int m) {
    bool **visited = new bool*[n];
    for(int i = 0 ; i < n; i++) {
        visited[i] = new bool[m];
        for(int j = 0; j < m; j++) {
            visited[i][j] = false;
        }
    }
    string str = "CODINGNINJA";   //string to be checked

    for(int i = 0; i < n; i++) {
        for(int j = 0; j < m; j++) {
            if(!visited[i][j] && board[i][j] == str[0]) {
                if(dfs(board, n, m, i, j, visited, str.substr(1))) return true;
            }
        }
    }
    return false;
}

int main() {
    int n, m;
    cin >> n >> m;

    vector<vector<char>> board(n, vector<char>(m));

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m; ++j) {
            cin >> board[i][j];
        }
    }

    cout << (hasPath(board, n, m) ? 1 : 0);
}
```

9. **Connecting Dots:** Gary has a board of size NxM. Each cell in the board is a coloured dot. There exist only 26 colours denoted by uppercase Latin characters (i.e. A,B,...,Z). Now Gary is getting bored and wants to play a game. The key of this game is to find a cycle that contains dots of the same colour. Formally, we call a sequence of dots d1, d2, ..., dk a cycle if and only if it meets the following condition:

1. These k dots are different: if i ≠ j then di is different from dj.
2. k is at least 4.
3. All dots belong to the same colour.
4. For all 1 ≤ i ≤ k - 1: di and di + 1 are adjacent. Also, dk and d1 should also be adjacent. Cells x and y are called adjacent if they share an edge.

Since Gary is colour blind, he wants your help. Your task is to determine if there exists a cycle on the board.

```
//Small bug in code, (Maybe) since count is sent by reference, after
backtracking count isn't reset.
#include <iostream>
#include <vector>
using namespace std;

int counter(vector<vector<char>> &board, int n, int m, int i, int j, int ii, int
jj, bool **visited, char curr, int &count) {
                                //ii and jj to compare last vertex with first vertex
    visited[i][j] = true;
    count++;
    if(count >= 4 && ((i == ii - 1 && j == jj) || (i == ii && j == jj - 1) || (i
== ii + 1 && j == jj) || (i == ii && j == jj + 1))) return true;

    //down
    if(i - 1 >= 0 && board[i - 1][j] == curr && !visited[i - 1][j])
if(counter(board, n, m, i - 1, j, ii, jj, visited, curr, count)) return 1;
    //left
    if(j - 1 >= 0 && board[i][j - 1] == curr && !visited[i][j - 1])
if(counter(board, n, m, i, j - 1, ii, jj, visited, curr, count)) return 1;
    //up
    if(i + 1 < n  && board[i + 1][j] == curr && !visited[i + 1][j])
if(counter(board, n, m, i + 1, j, ii, jj, visited, curr, count)) return 1;
    //right
    if(j + 1 < m  && board[i][j + 1] == curr && !visited[i][j + 1])
if(counter(board, n, m, i, j + 1, ii, jj, visited, curr, count)) return 1;

    return false;  //if no condition returns true
}

bool hasCycle(vector<vector<char>> &board, int n, int m) {
    bool **visited = new bool*[n];
    for(int i = 0 ; i < n; i++) {
        visited[i] = new bool[m];
        for(int j = 0; j < m; j++) {
            visited[i][j] = false;
        }
    }

    int count = 0;
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < m; j++) {
            if(!visited[i][j]) {
                int count = 0;
                char curr = board[i][j];
                if(counter(board, n, m, i, j, i, j, visited, curr, count))
return true;
            }
        }
    }
    return false;
}

int main() {
    int n, m;
    cin >> n >> m;

    vector<vector<char>> board(n, vector<char>(m));
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m; ++j) {
            cin >> board[i][j];
        }
    }

    cout << (hasCycle(board, n, m) ? "true" : "false");
}
```

10. **Largest Piece:** It's Gary's birthday today and he has ordered his favourite square cake consisting of '0's and '1's . But Gary wants the biggest piece of '1's and no '0's . A piece of cake is defined as a part which consists of only '1's, and all '1's share an edge with each other on the cake. Given the size of cake N and the cake, can you find the count of '1's in the biggest piece of '1's for Gary ?

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

int counter(vector<vector<int>> &cake, int n, int i, int j, bool **visited, int
&count) {    //sending count by reference
    count++;
    visited[i][j] = true;

    //down
    if(i - 1 >= 0 && cake[i - 1][j] == 1 && !visited[i - 1][j])
        counter(cake, n, i - 1, j, visited, count);
    //left
    if(j - 1 >= 0 && cake[i][j - 1] == 1 && !visited[i][j - 1])
        counter(cake, n, i, j - 1, visited, count);
    //up
    if(i + 1 < n  && cake[i + 1][j] == 1 && !visited[i + 1][j])
        counter(cake, n, i + 1, j, visited, count);
    //right
    if(j + 1 < n  && cake[i][j + 1] == 1 && !visited[i][j + 1])
        counter(cake, n, i, j + 1, visited, count);
}

int getBiggestPieceSize(vector<vector<int>> &cake, int n) {
    bool **visited = new bool*[n];
    for(int i = 0; i < n; i++) {
        visited[i] = new bool[n];
        for(int j = 0; j < n; j++) {
            visited[i][j] = false;
        }
    }

    int max = 0;
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            if(cake[i][j] == 1 && !visited[i][j]) {
                int count = 0;
                counter(cake, n, i, j, visited, count);
                if(count > max) max = count;
            }
        }
    }
    return max;
}

int main() {
    int n;
    cin >> n;

    vector<vector<int>> cake(n, vector<int>(n));

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            cin >> cake[i][j];
        }
    }

    cout << getBiggestPieceSize(cake, n);
}
```

11. 3 Cycle: Given a graph with N vertices (numbered from 0 to N-1) and M undirected edges, then count the distinct 3-cycles in the graph. A 3-cycle PQR is a cycle in which (P,Q), (Q,R) and (R,P) are connected by an edge.

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

int main() {
    int v, e;
    cin>>v>>e;

    int **edges = new int*[v];
    for(int i = 0; i < v; i++) {
        edges[i] = new int[v];
        for(int j = 0; j < v; j++) {
            edges[i][j] = 0;
        }
    }

    for(int i = 0; i < e; i++) {
        int j,s;
        cin>>j>>s;
        edges[j][s] = 1;
```

```
            edges[s][j] = 1;
        }

        int count = 0;
        for(int i = 0; i < v; i++) {
            for(int j = 0; j < v; j++) {
                if(i != j && edges[i][j] == 1) {
                    for(int k = 0; k < v; k++) {
                        if(k != i && k != j && edges[j][k] == 1) {
                            if(edges[k][i] == 1) count++;
                        }
                    }
                }
            }
        }

        cout<<count/6;
    }
```

Lecture 20 : Graphs 2

1. **Kruskal's Algorithm:** Given an undirected, connected and weighted graph G(V, E) with V number of vertices (which are numbered from 0 to V-1) and E number of edges. Find and print the Minimum Spanning Tree (MST) using Kruskal's algorithm.
   For printing MST follow the steps -

   1. In one line, print an edge which is part of MST in the format -

   v1 v2 w

   where, v1 and v2 are the vertices of the edge which is included in MST and whose weight is w. And v1 <= v2 i.e. print the smaller vertex first while printing an edge.

   2. Print V-1 edges in above format in different lines.

   Note : Order of different edges doesn't matter.

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

class edge {
    public:
        int source;
        int destination;
        int weight;
};

int findParent(int v, int *parent) {
    if(parent[v] == v) return v;                          //topmost parent found
    return findParent(parent[v], parent);
                                    //recursion till index and value become same
}

bool sortByWeight(const edge &lhs, const edge &rhs) {
    return lhs.weight < rhs.weight;
}

void kruskals(edge *input, int n, int e) {
    sort(input, input + e, sortByWeight);             //sorting input by weight

    edge *output = new edge[n - 1];

    //Union-find algorithm to check for cycles

    int *parent = new int[n];
    for(int i = 0; i < n; i++) {
        parent[i] = i;      //initializing with itself(to signify different set)
    }

    int count = 0;
    for(int i = 0; i < e; i++) {
        if(count == n - 1) break;                      //output MST array formed

        edge currentEdge = input[i];
        int sourceParent = findParent(currentEdge.source, parent);
        int destParent = findParent(currentEdge.destination, parent);

        if(sourceParent != destParent) {
            output[count] = currentEdge;
            count++;
            parent[sourceParent] = destParent;
                                            //updating parents(making them same)
        }
    }

    for(int i = 0; i < n - 1; i++) {
        if(output[i].source < output[i].destination) {
            cout<<output[i].source<<" "<<output[i].destination<<"
"<<output[i].weight<<endl;
        }
        else {
            cout<<output[i].destination<<" "<<output[i].source<<"
"<<output[i].weight<<endl;
        }
    }
}

int main() {
    int n, e;
    cin>>n>>e;

    edge *input = new edge[e];

    for(int i = 0; i < e; i++) {                  //taking input for input array
        cin>>input[i].source>>input[i].destination>>input[i].weight;
    }

    kruskals(input, n, e);
}
```

2. **Prim's Algorithm:** Given an undirected, connected and weighted graph G(V, E) with V number of vertices (which are numbered from 0 to V-1) and E number of edges. Find and print the Minimum Spanning Tree (MST) using Kruskal's algorithm.
   For printing MST follow the steps -

   1. In one line, print an edge which is part of MST in the format -

   v1 v2 w

   where, v1 and v2 are the vertices of the edge which is included in MST and whose weight is w. And v1 <= v2 i.e. print the smaller vertex first while printing an edge.

   2. Print V-1 edges in above format in different lines.

   Note : Order of different edges doesn't matter.

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

int findMinVertex(int *weight, bool *visited, int v) {
    int minVertex = -1;
    for(int i = 0; i < v; i++) {
        if(!visited[i] && (minVertex == -1 || weight[i] < weight[minVertex])) {
            minVertex = i;
        }
    }
    return minVertex;
}

void prims(int **edges, int v) {
    bool *visited = new bool[v];
    int *parent = new int[v];
    int *weight = new int[v];

    for(int i = 0; i < v; i++) {
        visited[i] = false;
        weight[i] = INT_MAX;
    }
    parent[0] = -1;
    weight[0] = 0;

    for(int i = 0; i < v - 1; i++) {
        //finding minVertex
        int minVertex = findMinVertex(weight, visited, v);
        visited[minVertex] = true;

        //explore unvisited Neighbors
        for(int j = 0; j < v; j++) {
            if(minVertex == j) continue;
            if(edges[minVertex][j] != 0 && !visited[j]) {
                if(weight[j] > edges[minVertex][j]) {
                    weight[j] = edges[minVertex][j];
                    parent[j] = minVertex;
                }
            }
        }
    }

    for(int i = 0; i < v; i++) {
        if(parent[i] == -1) continue;
        if(i < parent[i]) cout<<i<<" "<<parent[i]<<" "<<weight[i]<<endl;
        else cout<<parent[i]<<" "<<i<<" "<<weight[i]<<endl;
    }
}

int main() {
    int v, e;
    cin>>v>>e;

    int **edges = new int*[v];
    for(int i = 0; i < v; i++) {
        edges[i] = new int[v];
        for(int j = 0; j < v; j++) {
            edges[i][j] = 0;
        }
    }

    for(int i = 0; i < e; i++) {
        int j,s,w;
        cin>>j>>s>>w;
        edges[j][s] = w;
        edges[s][j] = w;
    }

    prims(edges, v);

    for(int i = 0; i < v; i++) {
        delete [] edges[i];
    }
    delete [] edges;
}
```

3. **Dijkstra's Algorithm:** Given an undirected, connected and weighted graph G(V, E) with V number of vertices (which are numbered from 0 to V-1) and E number of edges. Find and print the shortest distance from the source vertex (i.e. Vertex 0) to all other vertices (including source vertex also) using Dijkstra's Algorithm.

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

int findMinVertex(int *distances, bool *visited, int v) {
    int minVertex = -1;
    for(int i = 0; i < v; i++) {
        if(!visited[i] && (minVertex == -1 || distances[i] <
distances[minVertex])) {
            minVertex = i;
        }
    }
    return minVertex;
}

void djikstras(int **edges, int v) {
    bool *visited = new bool[v];
    int *distances = new int[v];

    for(int i = 0; i < v; i++) {
        visited[i] = false;
        distances[i] = INT_MAX;
    }
    distances[0] = 0;

    for(int i = 0; i < v; i++) {
        //finding minDistVertex
        int minVertex = findMinVertex(distances, visited, v);
        visited[minVertex] = true;

        //explore unvisited Neighbors
        for(int j = 0; j < v; j++) {
            if(minVertex == j) continue;
            if(edges[minVertex][j] != 0 && !visited[j]) {
                int c_dist = distances[minVertex] + edges[minVertex][j];
                if(distances[j] > c_dist) {
                    distances[j] = c_dist;
                }
            }
        }
    }

    for(int i = 0; i < v; i++) {
        cout<<i<<" "<<distances[i]<<endl;
    }
    delete [] visited;
      delete [] distances;
}

int main() {
    int v, e;
    cin>>v>>e;

    int **edges = new int*[v];
    for(int i = 0; i < v; i++) {
        edges[i] = new int[v];
        for(int j = 0; j < v; j++) {
            edges[i][j] = 0;
        }
    }

    for(int i = 0; i < e; i++) {
        int j,s,w;
        cin>>j>>s>>w;
        edges[j][s] = w;
        edges[s][j] = w;
    }

    djikstras(edges, v);

    for(int i = 0; i < v; i++) {
        delete [] edges[i];
    }
    delete [] edges;
}
```