Team notebook

8 de noviembre de 2017

Índice

1. backtrack

1.1. aBogomolny

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

using namespace std;

void printArray(int perm[],int N){
  for(int i=0;i<N;i++)printf("%3d",perm[i]);
  printf("\n");
}

void AlexanderBogomolyn(int perm[],int N, int k){
  static int level= -1;
  level=level+1;
  perm[k]=level;
  if(level==N) printArray(perm,N);
  else{
    for(int i=0;i<N;i++){
      if(perm[i]==0){
        AlexanderBogomolyn(perm,N,i);
      }
    }
  }
  level--;
  perm[k]=0;
}

int main(){
    int N = 3;
    int perm[N];
    AlexanderBogomolyn(perm, N, 0);
    return 0;
}
```

1.2. hamiltonianCycle

```
#include <stdio.h>
#include <iostream>
#define V 5

using namespace std;

void printSolution(int path[]){
    printf ("Solution Exists:"
            " Following is one Hamiltonian Cycle \n");
    for (int i = 0; i < V; i++){
        printf(" %d ", path[i]);
    }

    // Let us print the first vertex again to show the complete cycle
    printf(" %d ", path[0]);
    printf("\n");
}

bool hamCycleUtil(bool graph[V][V],int arr[V],int x){
  if(x==V){
    return true;
  }

  for(int i=0;i<V;i++){
    if(graph[x][i]!=0&&i!=x){ //If it is not safe,
      arr[x]=i;
      if(hamCycleUtil(graph,arr,x+1)){
        return true;
      }
      arr[x]=-1;
    }
  }
  return false;
}

bool hamCycle(bool graph[V][V]){
  int arr[V];
  for(int i=0;i<V;i++){
    arr[i]=-1;
  }

  if(hamCycleUtil(graph,arr,0)){
    printSolution(arr);
```

```
  }else{
    printf("Has no solution\n");
  }

  return true;
}

int main(){
   /* Let us create the following graph
      (0)--(1)--(2)
       |   / \   |
       |  /   \  |
       | /     \ |
      (3)-------(4)   */
   bool graph1[V][V] = {{0, 1, 0, 1, 0},
                        {1, 0, 1, 1, 1},
                        {0, 1, 0, 0, 1},
                        {1, 1, 0, 0, 1},
                        {0, 1, 1, 1, 0},
                        };
    hamCycle(graph1); //Print the solution

   /* Let us create the following graph
      (0)--(1)--(2)
       |   / \   |
       |  /   \  |
       | /     \ |
      (3)       (4)    */
    bool graph2[V][V] = {{0, 1, 0, 1, 0},
                        {1, 0, 1, 1, 1},
                        {0, 1, 0, 0, 1},
                        {1, 1, 0, 0, 0},
                        {0, 1, 1, 0, 0},
                        };

    hamCycle(graph2); // Print the solution

    return 0;
}
```

1.3. knightTour

```
#include <stdio.h>
#include <iostream>
#define N 8

using namespace std;

void printSolution(int sol[N][N]){
    for (int x = 0; x < N; x++){
        for (int y = 0; y < N; y++){
            printf(" %2d ", sol[x][y]);
          }
        printf("\n");
    }
}

bool isSafe(int board[N][N],int x, int y){
  if(x>=N||x<0||y>=N||y<0){ //If it goes out of the board return false
    return false;
  }
  if(board[x][y]!= -1){ //If it is occupied return false
    return false;
  }
  return true;
}

bool solveKnightUtil(int board[N][N], int currX, int currY, int currMove,
     int x[N],int y[N]){
  if(currMove==N*N){ //If it overpasses the amount of moves, then return
      false
    return true;
  }

  int nextX;
  int nextY;

  for(int i=0;i<8;i++){ //Goes through all possibilities
    nextX=currX+x[i]; //Calculates the next x value
    nextY=currY+y[i]; //Calculates the next y value
    if(isSafe(board,nextX,nextY)){ //If the next value is safe
      board[nextX][nextY]=currMove; //Change the next values to the
          current Move
      if(solveKnightUtil(board,nextX,nextY,currMove+1,x,y)){ //If
          recursion takes to the end, it returns true
        return true;
      }else{
```

```
        board[nextX][nextY]= -1; //If it does not go to the end, it
            returns to -1
      }
    }
  }

  return false; //If it goes through all the possibilities and it still
      can not find, it's impossible, return false
  //Backtrack
}


bool solveKnight(){
  int board[N][N]; //Create a board with NxN

  for(int i=0;i<N;i++){
    for(int j=0;j<N;j++){
      board[i][j]= -1; //Assign -1 to the whole board
    }
  }

  board[0][0]=0; //Starts at board[0][0] therefore, its turn is 0

  // int x[8]={2,2,-2,-2,1,1,-1,-1}; //All possible moves that a piece
      can make
  // int y[8]={-1,1,-1,1,2,-2,2,-2};

  int x[8] = { 2, 1, -1, -2, -2, -1, 1, 2 };
  int y[8] = { 1, 2, 2,  1, -1, -2, -2, -1 };

  if(solveKnightUtil(board,0,0,1,x,y)){ //If it can solve itself with
      solveKnightUtil, then it prints the solution
    printSolution(board);
  }else{
    printf("Does not have any solutions\n"); //Else, prints it can not be
        accomplished
  }

  return true;
}


int main(){
    solveKnight();
    return 0;
```

```
}
```

1.4. knightTour2

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

using namespace std;

void printBoard(int **board,int N){
  for(int i=0;i<N;i++){
    for(int j=0;j<N;j++){
      printf(" %2d ",board[i][j]);
    }
    printf("\n");
  }
}

bool isSafe(int **board,int N,int x,int y){
  if(x>=N||x<0||y>=N||y<0) return false;
  if(board[x][y]!=-1) return false;
  return true;
}

bool solveKnightUtil(int **board,int N, int x[], int y[],int level,int
    currX, int currY){
  if(level==N*N) return true;

  for(int i=0;i<8;i++){
    int nextX=currX+x[i];
    int nextY=currY+y[i];
    if(isSafe(board,N,nextX,nextY)){
      board[nextX][nextY]=level;
      if(solveKnightUtil(board,N, x,y,level+1,nextX,nextY)) return true;
      board[nextX][nextY]=-1;
    }
  }
  return false;
}

void solveKnight(int N){
  int **board=new int*[N]; //Create the board
```

```
  for(int i=0;i<N;i++){
    board[i]=new int[N];
  }

  for(int i=0;i<N;i++){
    for(int j=0;j<N;j++){
      board[i][j]=-1;
    }
  }

  board[0][0]=0;

  int x[8] = { 2, 1, -1, -2, -2, -1, 1, 2 };
  int y[8] = { 1, 2, 2,  1, -1, -2, -2, -1 };

  if(solveKnightUtil(board,N,x,y,1,0,0)){
    printBoard(board,N);
  }else{
    printf("No Can Do\n");
  }
}

int main(){
    solveKnight(8);
    return 0;
}
```

1.5. nQueen

```
#include <iostream>
#include <vector>
#define N 8

using namespace std;

//n Queen Problem O(n!)
//RETURNS THE BOARD

void printSolution(int board[N][N]){
    for (int i = 0; i < N; i++){
        for (int j = 0; j < N; j++){
            printf(" %d ", board[i][j]);
        }
        printf("\n");
    }
}

bool isSafe(int board[N][N], int row, int col){
    int i, j;

    for (i = 0; i < col; i++){ //Check for each column until current
        column if the row has a 1
      if (board[row][i]){
        return false; //If found, then return false
      }
    }

    for (i=row, j=col; i>=0 && j>=0; i--, j--){ //Check for diagonally up
        towards the left see if a queen is there
      if (board[i][j]){
        return false; //If found, then return false
      }
    }

    for (i=row, j=col; j>=0 && i<N; i++, j--){ //Check for diagonally
        down towards the left see if a queen is there
      if (board[i][j]){
        return false; //If found, then return false
      }
    }

    return true; //Return true if you dont find anything
}

bool nQueenUtil(int board[N][N],int col){
  if(col>=N){
    return true; //If it reaches the end and finds a nQueenUtil in the end
  }
  for(int i=0;i<N;i++){ //For each index in the column
    if(isSafe(board,i,col)){ //If the index is safe
      board[i][col]=1; //Place the Queen in index i and column
      if(nQueenUtil(board, col+1)){ //If the next column can be
          accomodated too... Till the end
        return true;
      }
      board[i][col]=0; //We can not use this because it did not return
          true, return back to former
    }
```

```
  }
  return false; //If it goes through the whole column and can not locate,
      return false
}

bool solve(){
    int board[N][N];

    //Fill in the board with 0's
    for(int i=0;i<N;i++){
      for(int j=0;j<N;j++){
        board[i][j]=0;
      }
    }

    if (!nQueenUtil(board, 0)){ //It gives an empty board and starts from
        column 0
      printf("Solution does not exist\n"); //If it returns false, then it
          does not exist
      return false;
    }

    printSolution(board); //Else, it prints the board
    return true;
}

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

1.6. nQueen2

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

using namespace std;

void printBoard(int **board,int N){
  for(int i=0;i<N;i++){
    for(int j=0;j<N;j++){
      printf(" %2d",board[i][j]);
    }
    printf("\n");
  }
}

bool isSafe(int **board,int N,int row, int col){
  for(int i=col-1;i>=0;i--){
    if(board[row][i]) return false;
  }
  for(int i=row-1,j=col-1;i>=0&&j>=0;i--,j--){
    if(board[i][j]) return false;
  }
  for(int i=row+1,j=col-1;i<N&&j>=0;i++,j--){
    if(board[i][j]) return false;
  }
  return true;
}

bool nQueenUtil(int **board,int N, int col){
  if(N==col) return true;

  for(int row=0;row<N;row++){
    if(isSafe(board,N,row,col)){
      board[row][col]=1;
      if(nQueenUtil(board,N,col+1)) return true;
      board[row][col]=0;
    }
  }
  return false;
}

void solvenQueen(int N){
  int **board= new int*[N];
  for(int i=0;i<N;i++){
    board[i]=new int[N];
  }

  for(int i=0;i<N;i++){
    for(int j=0;j<N;j++){
      board[i][j]=0;
    }
  }

  if(nQueenUtil(board,N,0)){
    printBoard(board,N);
  }else{
```

```
    printf("No Can Do\n");
  }
}

int main(){
  solvenQueen(8);
  return 0;
}
```

1.7. nQueen3

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

using namespace std;

bool isSafe(int **board,int col, int row){
  for(int i=0;i<col;i++) if(board[row][i]==1) return false;
  for(int i=0;i<row;i++) if(board[i][col]==1) return false;
  for(int i=row-1,j=col-1;i>=0&&j>=0;i--,j--) if(board[i][j]==1) return
      false;
  for(int i=row+1,j=col-1;j>=0&&i<8;i++,j--) if(board[i][j]==1) return
      false;

  return true;
}

bool nQueenUtil(int **board,int col){

    if(col==8) return true;

    for(int i=0;i<8;i++){
      if(isSafe(board,col,i)){
        board[i][col]=1;
        if(nQueenUtil(board,col+1)) return true;
        board[i][col]=0;
      }
    }
    return false;
}

void nQueen(int **board){
  if(nQueenUtil(board,0)){
    for(int i=0;i<8;i++){
      for(int j=0;j<8;j++){
        printf("%3d",board[i][j]);
      }
      printf("\n");
    }
  }else{
    printf("No can do\n");
  }
}

int main(){
  int n=8;
  int **board=new int*[n];
  for(int i=0;i<8;i++) board[i]=new int[n]{0,0,0,0,0,0,0,0};

  nQueen(board);
}
```

1.8. ratMaze

```
#include <stdio.h>
#include <iostream>
#define N 4

using namespace std;

void printSolution(int sol[N][N]){
    for (int i = 0; i < N; i++){
        for (int j = 0; j < N; j++){
            printf(" %d ", sol[i][j]);
        }
        printf("\n");
    }
}

bool isSafe(int maze[N][N],int x, int y){
  if(x>=N||y>=N){ //If outside of the maze
    return false;
  }
  if(maze[x][y]==0){
    return false;
  }
```

```
  return true;
}

bool solveMazeUtil(int maze[N][N],int finMaze[N][N],int x, int y){
  if(x==N-1&&y==N-1){ //Gets to destination
    finMaze[N-1][N-1]=1;
    return true;
  }


  if(isSafe(maze,x,y)){
    finMaze[x][y]=1;
    if(solveMazeUtil(maze,finMaze,x+1,y)){
      return true;
    }
    if(solveMazeUtil(maze,finMaze,x,y+1)){
      return true;
    }
    finMaze[x][y]=0;
    return false;
  }
  return false;
}

bool solveMaze(int maze[N][N]){
  int finMaze[N][N]; //Create final solution board
  for(int i=0;i<N;i++){ //Initialize the board with 0s
    for(int j=0;j<N;j++){
      finMaze[i][j]=0;
    }
  }

  if(solveMazeUtil(maze,finMaze,0,0)){
    printSolution(finMaze);
  }else{
    printf("Maze can not be solved");
  }
return true;
}

int main(){
    int maze[N][N] =
      { {1, 0, 0, 0},
        {1, 1, 0, 1},
        {0, 1, 0, 0},
        {1, 1, 1, 1}
    };
    solveMaze(maze);
    return 0;
}
```

1.9. stringPerm

```
#include <stdio.h>
#include <iostream>
#include <map>
#include <vector>
#include <string>

using namespace std;

void stringPermUtil(int n, char arr[],int arrCount[],string str,
    vector<string> &results,int level){
  if(level==str.length()){ //If our level is str.length which means that
      it exceeded its supposed level, we add
    results.push_back(str); //We add to results
    // cout<<str<<endl;
    return;
  }

  for(int i=0;i<n;i++){
    if(arrCount[i]==0){ //If we can not go on, just go to the enxt one
        which we can use
      continue;
    }
    str[level]=arr[i]; //We assign at the level index of our string the
        arr[i] char
    arrCount[i]--; //We reduce the count because we assigned it
    stringPermUtil(n,arr,arrCount,str,results,level+1); //We go down a
        level to find the next index of the string
    arrCount[i]++; //Once it comes back, we return the character count
  }
}

void permute(string str,vector<string> &results){
  map<char,int> mp; //Create a map to order keys and assign count to them
  for(int i=0;i<str.length();i++){
```

```
    if(mp.find(str[i])==mp.end()){ //Map.find() in c++ returns an
        iterator to last element if not found
      mp[str[i]]=1;
    }else{ //If found, we just add it to 1
      mp[str[i]]=mp[str[i]]+1;
    }
  }

  char arr[mp.size()]; //Create an array of individual chars
  int arrCount[mp.size()]; //Create an array of amount of char repetitions

  int i=0;
  for (map<char,int>::iterator it=mp.begin(); it!=mp.end(); it++){
    arr[i]= it->first; //first is the key
    arrCount[i]= it->second; //second is the value
    i++;
  }

  int n=sizeof(arr)/sizeof(arr[0]); //We need the total amount of unique
      characters
  stringPermUtil(n, arr,arrCount,str,results,0); //We start the backtrack
      function which will save everything to results
}

int main(){
  string str="ABCA";
  vector<string> results; //Create the vector results
  permute(str,results); //Permute finishes all operations and works with
      the vector results
  for(vector<string>::iterator
      it=results.begin();it!=results.end();it++){ //Use iterators
    cout<<*it<<endl; //Print all permutations
  }
  return 0;
}
```

1.10. stringPerm2

```
#include <string>
#include <iostream>
#include <stdio.h>
#include <vector>
#include <map>

using namespace std;

void permuteUtil(int n,string str, char c[], int count[],int level,
    vector<string> *results){

  if(level==str.length()){
    results->push_back(str);
    return;
  }

  for(int i=0;i<n;i++){
    if(count[i]==0) continue;
    count[i]--;
    str[level]=c[i];
    permuteUtil(n,str,c,count,level+1, results);
    count[i]++;
  }

}

void permute(string str,vector<string> *results){
  map<char,int> mp;
  for(int i=0;i<str.length();i++){
    if(mp.find(str[i])==mp.end()){
      mp[str[i]]=1;
    }else{
      mp[str[i]]++;
    }
  }
  char c[mp.size()];
  int count[mp.size()];

  int i=0;
  for(map<char,int>::iterator it=mp.begin();it!=mp.end();it++){
    c[i]=it->first;
    count[i]=it->second;
    i++;
  }

  permuteUtil(mp.size(),str,c,count,0,results);
}

int main(){
  string str="ABCA";
```

```
  vector<string> results; //Create the vector results
  permute(str,&results); //Permute finishes all operations and works with
      the vector results
  for(vector<string>::iterator
      it=results.begin();it!=results.end();it++){ //Use iterators
    cout<<*it<<endl; //Print all permutations
  }
  return 0;
}
```

1.11. stringPerm3

```
#include <stdio.h>
#include <iostream>
#include <vector>
#include <map>
#include <string>

using namespace std;

void permuteUtil(string str, vector<string> *answer, char c[],int
    count[],int n, int level){
  if(level==str.length()) answer->push_back(str);

  for(int i=0;i<n;i++){
    if(count[i]==0) continue;
    count[i]--;
    str[level]=c[i];
    permuteUtil(str,answer,c,count,n,level+1);
    count[i]++;
  }
}

void permute(string str, vector<string> *answer){
  map<char,unsigned int> mp;

  for(int i=0;i<str.length();i++){
    if(mp.find(str[i])==mp.end()) mp[str[i]]=1;
    else mp[str[i]]++;
  }

  char c[mp.size()];
  int count[mp.size()];
  int i=0;
  for(map<char,unsigned int>::iterator it=mp.begin();it!=mp.end();it++){
    c[i]=it->first;
    count[i]=it->second;
    i++;
  }

  permuteUtil(str, answer, c, count, mp.size(), 0);

}

int main(){
  string str="chan";
  vector<string> answer;
  permute(str,&answer);
  for(vector<string>::iterator it=answer.begin();it!=answer.end();it++)
      printf("%s\n",it->c_str());
}
```

1.12. sudoku

```
#include <stdio.h>
#include <iostream>
#define N 9

using namespace std;

bool isDone(int board[N][N],int &row,int &col){ //Go through the board to
    see if it is full
  for(row=0;row<N;row++){ //Row and Col saves itself while iterating to
      stop when it is 0
    for(col=0;col<N;col++){
      if(board[row][col]==0){
        return false; //Returns -1 if it finds a 0
      }
    }
  }
  return true; //Returns 1 if it is done (has no 0s)
}

bool isSafe(int board[N][N],int row, int col, int num){
  for(int i=0;i<N;i++){ //Check if num has been used in row or column
```

```
    if((board[i][col]==num)||(board[row][i]==num)){
      return false;
    }
  }
  row=row-row%3;
  col=col-col%3;
  for(int i=0;i<3;i++){ //Check if num is found in the box
    for(int j=0;j<3;j++){
      if(board[row+i][col+j]==num){
        return false;
      }
    }
  }

  return true;
}

bool solveSudoku(int board[N][N]){
  int row=0,col=0; //Initialize row and col to move it around
  if(isDone(board,row,col)){ //isDOne will change row and col to where
      board[][]==0
    return true;
  }

  for(int i=1;i<=9;i++){
    if(isSafe(board, row, col, i)){ //Send board, row, col, and number to
        see if number can fit there
      board[row][col]=i; //If it can fit there, save value there

      if(solveSudoku(board)){ //And finally, check if the board can be
          completed with further numbers
        return true;
      }

      board[row][col]=0; //If it can not be completed with guesses, then
          it saves it to 0 to try it with next number in for
    }
  }

  return false; //Returns false if it can not be completed to trigger
      backtrack
}

void printBoard(int board[N][N]){
    for (int row = 0; row < N; row++){
        for (int col = 0; col < N; col++){
              printf("%2d", board[row][col]);
         }
         printf("\n");
    }
}

int main(){
  int board[N][N] = {{3, 0, 6, 5, 0, 8, 4, 0, 0},
                     {5, 2, 0, 0, 0, 0, 0, 0, 0},
                     {0, 8, 7, 0, 0, 0, 0, 3, 1},
                     {0, 0, 3, 0, 1, 0, 0, 8, 0},
                     {9, 0, 0, 8, 6, 3, 0, 0, 5},
                     {0, 5, 0, 0, 9, 0, 6, 0, 0},
                     {1, 3, 0, 0, 0, 0, 2, 5, 0},
                     {0, 0, 0, 0, 0, 0, 0, 7, 4},
                     {0, 0, 5, 2, 0, 6, 3, 0, 0}};
  if (solveSudoku(board) == true){
        printBoard(board);
  }else{
        printf("No solution exists");
  }
  return 0;
}
```

1.13. sudoku2

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

using namespace std;

void printBoard(int **board, int N){
  for(int i=0;i<N;i++){
    for(int j=0;j<N;j++){
      printf(" %2d",board[i][j]);
    }
    printf("\n");
  }
}

bool foundEmpty(int **board,int N, int &x,int &y){
  for(x=0;x<N;x++){
```

```
    for(y=0;y<N;y++){
      if(board[x][y]==0){
        return false;
      }
    }
  }
  return true;
}

bool isSafe(int **board, int N,int x, int y, int i){
  for(int j=0;j<N;j++){
    if(board[j][y]==i||board[x][j]==i) return false;
  }

  x=x-x%3;
  y=y-y%3;

  for(int j=0;j<3;j++){
    for(int k=0;k<3;k++){
      if(board[j+x][k+y]==i) return false;
    }
  }
  return true;
}

bool sudokuUtil(int **board, int N){
  int x=0, y=0;
  if(foundEmpty(board,N,x,y)){
    return true;
  }

  for(int i=1;i<=9;i++){
    if(isSafe(board,N,x,y,i)){
      board[x][y]=i;
      if(sudokuUtil(board,N)) return true;
      board[x][y]=0;
    }
  }
  return false;
}

void solveSudoku(int **board, int N){
  if(sudokuUtil(board,N)){
    printBoard(board,N);
  }else{
    printf("No Can Do");
  }
}

int main(){
  int N=9;
  int **board= new int*[N];
  board[0]=new int[N]{3, 0, 6, 5, 0, 8, 4, 0, 0};
  board[1]=new int[N]{5, 2, 0, 0, 0, 0, 0, 0, 0};
  board[2]=new int[N]{0, 8, 7, 0, 0, 0, 0, 3, 1};
  board[3]=new int[N]{0, 0, 3, 0, 1, 0, 0, 8, 0};
  board[4]=new int[N]{9, 0, 0, 8, 6, 3, 0, 0, 5};
  board[5]=new int[N]{0, 5, 0, 0, 9, 0, 6, 0, 0};
  board[6]=new int[N]{1, 3, 0, 0, 0, 0, 2, 5, 0};
  board[7]=new int[N]{0, 0, 0, 0, 0, 0, 0, 7, 4};
  board[8]=new int[N]{0, 0, 5, 2, 0, 6, 3, 0, 0};
  solveSudoku(board,N);
  return 0;
}
```

1.14. sudoku3

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

using namespace std;

void printBoard(int **board){
  for(int i=0;i<9;i++){
    for(int j=0;j<9;j++){
      printf("%3d",board[i][j]);
    }
    printf("\n");
  }
}

bool isSolved(int **board,int &row,int &col){
  for(row=0;row<9;row++){
    for(col=0;col<9;col++){
      if(board[row][col]==0) return false;
    }
  }
  return true;
```

```
}

bool isSafe(int **board, int row, int col,int x){
  for(int i=0;i<9;i++) if(board[i][col]==x||board[row][i]==x) return
      false;

  row=row-row%3;
  col=col-col%3;
  for(int i=0;i<3;i++){
    for(int j=0;j<3;j++){
      if(board[i+row][j+col]==x) return false;
    }
  }
  return true;
}

bool sudokuUtil(int **board){
  int row,col;
  if(isSolved(board,row,col)) return true;

  for(int i=1;i<=9;i++){
    if(isSafe(board,row,col,i)){
      board[row][col]=i;
      if(sudokuUtil(board))return true;
      board[row][col]=0;
    }
  }
  return false;
}

void sudoku(int **board){
  if(sudokuUtil(board)) printBoard(board);
  else printf("No can do homey\n");
}

int main(){
  int N=9;
  int **board= new int*[N];
  board[0]=new int[N]{3, 0, 6, 5, 0, 8, 4, 0, 0};
  board[1]=new int[N]{5, 2, 0, 0, 0, 0, 0, 0, 0};
  board[2]=new int[N]{0, 8, 7, 0, 0, 0, 0, 3, 1};
  board[3]=new int[N]{0, 0, 3, 0, 1, 0, 0, 8, 0};
  board[4]=new int[N]{9, 0, 0, 8, 6, 3, 0, 0, 5};
  board[5]=new int[N]{0, 5, 0, 0, 9, 0, 6, 0, 0};
  board[6]=new int[N]{1, 3, 0, 0, 0, 0, 2, 5, 0};
  board[7]=new int[N]{0, 0, 0, 0, 0, 0, 0, 7, 4};
  board[8]=new int[N]{0, 0, 5, 2, 0, 6, 3, 0, 0};

  sudoku(board);


}
```

2. binaryHeap

```
#include <stdio.h>
#include <vector>
using namespace std;

struct MaxHeap {
    vector<int> container;

    MaxHeap() {
        container.push_back(0);
    }

    MaxHeap(vector<int> v) {
        container.push_back(0);
        vector<int>::iterator iter;
        for (iter = v.begin(); iter != v.end(); iter++) {
            container.push_back(*iter);
        }
        for (int i = parent(v.size()); i >= 1; i& 1 ? i-- : i = parent(i))
             {
            down_heap(i);
        }
    }

    int parent(int index) {
        return index >> 1;
    }

    int left(int index) {
        return index << 1;
    }

    int right(int index) {
```

```
        return (index << 1) + 1;
    }

    void push(int element) {
        container.push_back(element);
        up_heap(container.size() - 1);
    }

    void up_heap(int index) {
        if (parent(index) > 0 && container[parent(index)] <
            container[index]) {
            int tmp = container[index];
            container[index] = container[parent(index)];
            container[parent(index)] = tmp;
            up_heap(parent(index));
        }
    }

    void pop() {
        container[1] = container[container.size() - 1];
        container.pop_back();
        down_heap(1);
    }

    void down_heap(int index) {
        int max = index;
        int left = MaxHeap::left(index);
        int right = MaxHeap::right(index);
        if (left < container.size() && container[left] > container[max]) {
            max = left;
        }
        if (right < container.size() && container[right] > container[max])
            {
            max = right;
        }
        if (max != index) {
            int tmp = container[index];
            container[index] = container[max];
            container[max] = tmp;
            down_heap(max);
        }
    }
};

int main() {
    vector<int> a = {1, 2, 3, 4, 5, 6};
    MaxHeap m(a);
    for (auto n : m.container) {
        printf("%d ", n);
    }
    printf("\n");
    m.push(7);
    for (auto n : m.container) {
        printf("%d ", n);
    }
    printf("\n");
    m.push(5);
    for (auto n : m.container) {
        printf("%d ", n);
    }
    printf("\n");
    m.pop();
    for (auto n : m.container) {
        printf("%d ", n);
    }
}
```

3. dp

3.1. bitonic

```
#include <iostream>
#include <vector>

using namespace std;

//Longest Bitonic Subsequence O(n^2)
//RETURNS JUST THE LENGTH

int bitonic(int arr[],int n){

  int lis[n];
  int lds[n];
  //Base case for LIS and LDS, LDS starts at 0 because it adds up and
      from that start space, itself doesnt count.
  for(int i=0;i<n;i++) lis[i]=1;
  for(int i=0;i<n;i++) lds[i]=0;
```

```
  //Typical LIS
  for(int i=1;i<n;i++){
    for(int j=0;j<i;j++){
      if(arr[j]<=arr[i] && lis[j]+1>lis[i]){
        lis[i]=lis[j]+1;
      }
    }
  }

  //Typical LDS
  for (int i=n-2; i>=0; i--){
    for (int j=n-1; j>i; j--){
      if(arr[i] >arr[j] && lds[i] < lds[j] + 1){
        lds[i]=lds[j]+1;
      }
    }
  }

  //Find max after adding the two values
  int max=lis[0]+lds[0];
  for(int i=1;i<n;i++){
    if(lis[i]+lds[i]>max){
      max=lis[i]+lds[i];
    }
  }
  return max;
}

int main(){
  int arr[] = {0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15};
  int n = sizeof(arr)/sizeof(arr[0]);
  printf("Length of Longest Bitonic Subsequence is %d\n", bitonic( arr, n
      ) );
  return 0;
}
```

3.2. coverDistance

```
#include <iostream>
#include <vector>

using namespace std;

int coverDistance(int d){
  int count[d+1];
  count[0]=1; //Base case where to get to 0, it's only 1 option
  count[1]=1; //Base case where to get to step 1, you can only take the 1
      step option
  count[2]=2; //Base case where to get to step 2, you can either take 2
      1's or 1 2's.
  for(int i=3;i<=d;i++){
    //If you want to reach i, you can reach i from either i-1, i-2, and
        i-3 steps. Therefore you add all possibilities
    count[i]=count[i-1]+count[i-2]+count[i-3]; //Becuase you can only
        take 1, 2 or 3 steps.
  }
  return count[d];
}

int main(){
    int dist = 4;
    cout << coverDistance(dist)<<endl;
    return 0;
}
```

3.3. editDistance

```
#include <iostream>
#include <vector>
#include <string>

using namespace std;

//Edit Distance O(n*m)
//RETURNS OPERATION COUNT

int min(int a, int b,int c){
    return (a<=b&&a<=c)?a:(b<=c&&b<=a)?b:c;
}

int editDistance(string str1, string str2) {

    int m=str1.length();
    int n=str2.length();
    //Create the matrix with lengthxlength
```

```
    int dp[m+1][n+1];

    // Fill d[][] in bottom up manner
    for (int i=0; i<=m; i++){
        for (int j=0; j<=n; j++){
            if (i==0){// If first string is empty, only option is to
                insert all characters of second string
              dp[i][j] = j; // Min. operations = j (length string2 until
                  point j)
            }else if (j==0){ // If second string is empty, only option is
                to remove all characters of second string
              dp[i][j] = i; // Min. operations = i (length string1 until
                  point i)
            }else if (str1[i-1] == str2[j-1]){ // If last characters are
                same, ignore last char and recur for remaining string
              dp[i][j] = dp[i-1][j-1];
            }else{ // If last character are different, consider all
                possibilities and find minimum
              dp[i][j] = 1 + min(dp[i][j-1], // Insert
                                 dp[i-1][j], // Remove
                                 dp[i-1][j-1]); // Replace
            }
        }
    }
    return dp[m][n];
}

int main(){
    string str1 = "sunday";
    string str2 = "saturday";
    printf("%d\n",editDistance(str1, str2));
    return 0;
}
```

3.4. isKPalindrome

```
#include <stdio.h>
#include <iostream>
#include <algorithm>

using namespace std;

//A K-Palindrome String transforms into a Palindrome removing at most K
    Palindromes

//IF THE LONGEST LENGTH-LEN(LONGEST PALINDROMIC SUBSEQUENCE)>K, IT'S
    IMPOSSIBLE

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

bool isKPal(string str, int k){
  int n=str.length();
  string str2=str;
  reverse(str2.begin(),str2.end());
  int dp[n+1][n+1];

  for(int i=0;i<=n;i++){
    for(int j=0;j<=n;j++){
      if(i==0){
        dp[i][j]=j;
      }else if(j==0){
        dp[i][j]=i;
      }else if(str[i-1]==str2[j-1]){
        dp[i][j]=dp[i-1][j-1];
      }else{
        dp[i][j]=min(dp[i-1][j],dp[i][j-1]);
      }
    }
  }
  return dp[n][n]<k;
}

int main(){
    string str = "acdcb";
    int k = 2;
    isKPal(str, k)? cout << "Yes"<<endl : cout << "No"<<endl;
    return 0;
}
```

3.5. knapsack01

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

```
using namespace std;

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

int knapSack(int W, int wt[], int val[],int n){
  int dp[n+1][W+1];

  for(int i=0;i<n+1;i++){
    for(int j=0;j<W+1;j++){
      if(i==0||j==0){
        dp[i][j]=0;
      }else if(wt[i-1]<=j){ //When the corresponding weight is above the
          least amount, get the top value
        dp[i][j]=max(val[i-1]+dp[i-1][j-wt[i-1]],dp[i-1][j]);
      }else{
        dp[i][j]=dp[i-1][j];
      }
    }
  }
  return dp[n][W];
}

int main(){
    int val[] = {60, 100, 120};
    int wt[] = {10, 20, 30};
    int W = 50;
    int n = sizeof(val)/sizeof(val[0]);
    printf("%d\n", knapSack(W, wt, val, n));
    return 0;
}
```

3.6. largestSumContiguous

```
#include <iostream>
#include <vector>

using namespace std;

//Largest Sum Contiguous Kadane
int kadane( int arr[], int n){
  int total_max=arr[0],current_max=arr[0]; //total_max and current_max
      starts as the first index
  int start=0,fin_s=0,fin_end=0; //All indexes start at 1
  for(int i=1;i<n;i++){ //Starts at 1 because we already took into
      account arr[0]
    current_max+=arr[i]; //We update current_max to check
    if(current_max<arr[i]){ //If current_max<arr[i] all the previous
        becomes useless
      current_max=arr[i]; //Current_max becomes the new value since its
          bigger than previous sums
      start=i; //Since new block begins, start also begins
    }
    if(current_max>total_max){ //We check if the current block has the
        biggest sums
      total_max=current_max; //If it is, total sum becomes the current sum
      fin_s=start; //We save independently the start index and finish
          index
      fin_end=i;
    }
  }
  cout<<"Start index: "<<fin_s<<" \nFinish index: "<<fin_end<<endl; //We
      print the start index and finish index (inclusive)
  return total_max; //Returns the biggest sum
}

int main(){
    int a[] = {-2, -3, 4, -1, -2, 1, 5, -3};
    int n = sizeof(a)/sizeof(a[0]);
    int max_sum = kadane(a, n);
    cout << "Maximum contiguous sum is " << max_sum<<endl;
    return 0;
}
```

3.7. lcs

```
#include <iostream>
#include <vector>

using namespace std;

//Longest Common Subsequence
//RETURNS JUST THE LENGTH
```

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

int lcs( char *X, char *Y){
   int m = strlen(X);
   int n = strlen(Y);
   //m+1 and n+1 because we need base cases where we compare empty string
       with empty string
   int L[m+1][n+1];
   int i, j;

   //Base case where if given abcdaf and acbcf, we give value 0 to when
       strings are empty
   for(i=0;i<m;i++){
     L[i][0]=0;
   }

   for(i=0;i<n;i++){
     L[0][i]=0;
   }

   for (i=1; i<=m; i++){ //Goes from 1 to m because 0 is empty string
     for (j=1; j<=n; j++){ //Same as above
       if (X[i-1] == Y[j-1]){ //You check the value at index i-1 and j-1
           because 0 is empty string
         //If the values are the same, you add 1 of length to L[i-1][j-1]
         //since L[i-1][j-1] is the LCS of the 2 strings before the
             coinciding values
         L[i][j] = L[i-1][j-1] + 1;
       }else {
         //Else, you get the maximum of LCS of above or to the left
         L[i][j] = max(L[i-1][j], L[i][j-1]);
       }
     }
   }
   return L[m][n];
}

int main(){
  char X[] = "AGGTAB";
  char Y[] = "GXTXAYB";

  printf("Length of LCS is %d\n", lcs(X, Y) );
  return 0;
}
```

3.8. lisNN

```
#include <iostream>
#include <vector>

using namespace std;

//Longest Increasing Subsequence O(n^2)
//RETURNS JUST THE LENGTH

int lis(vector<int> &v){
  if (v.size() == 0) return 0;

  int temp[v.size()];

  //Give all values of temp to 1
  for(int i=0;i<v.size();i++) temp[i]=1;

  //Assign to temp[i] the maximum of any value of temp[j]+1 if v[j] is
      bigger
  for(int i=1;i<v.size();i++){
    for(int j=0;j<i;j++){
      if(v[j]<=v[i]&&temp[j]+1>temp[i]) temp[i]=temp[j]+1;
    }
  }

  //Find max, max=1 because default values of temp start at 1
  int max=1;
  for(int i=0;i<v.size();i++) if(temp[i]>max)max=temp[i];
  return max;
}

int main() {
    vector<int> v{ 2, 5, 3, 7, 11, 8, 10, 13, 6 };
    cout << "Length of Longest Increasing Subsequence is " <<
        lis(v)<<endl;
   return 0;
}
```

3.9. lisNlogN

```
#include <iostream>
#include <vector>

using namespace std;

//Longest Increasing Subsequence O(nLogn)
//RETURNS THE JUST LENGTH
//I DONT UNDERSTAND THIS YET GODDAMMIT I HATE MYSELF
//WUBBA LUBBA DUB DUB


int CeilIndex(vector<int> &v, int l, int r, int key) {
    while (r-l > 1) {
    int m = l + (r-l)/2;
    if (v[m] >= key) r = m;
    else l = m;
    }
    return r;
}

int LongestIncreasingSubsequenceLength(vector<int> &v) {
    if (v.size() == 0) return 0;

    vector<int> tail(v.size(), 0);
    int length = 1; // always points empty slot in tail

    tail[0] = v[0];
    for (size_t i = 1; i < v.size(); i++) {
        if (v[i] < tail[0])
            // new smallest value
            tail[0] = v[i];
        else if (v[i] > tail[length-1])
            // v[i] extends largest subsequence
            tail[length++] = v[i];
        else
            // v[i] will become end candidate of an existing subsequence or
            // Throw away larger elements in all LIS, to make room for
                upcoming grater elements than v[i]
            // (and also, v[i] would have already appeared in one of LIS,
                identify the location and replace it)
            tail[CeilIndex(tail, -1, length-1, v[i])] = v[i];
    }

    return length;
}

int main() {
    vector<int> v{ 2, 5, 3, 7, 11, 8, 10, 13, 6 };
    cout << "Length of Longest Increasing Subsequence is " <<
        LongestIncreasingSubsequenceLength(v)<<endl;
   return 0;
}
```

3.10. lps

```
#include <iostream>
#include <vector>
#include <string>

using namespace std;

//Longest Palindromic Subsequence O(n^2)
//RETURNS JUST THE LENGTH

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

int lps(char *s) {

    //Create the matrix with lengthxlength
    int arr[strlen(s)][strlen(s)];

    //Case where we contemplate string 0-0, 1-1... etc, always palindrome
        because 1 string.
    for(int i=0;i<strlen(s);i++){
        arr[i][i]=1;
    }

    for(int l=2;l<=strlen(s);l++){ //Length of the strings go up, start
        with 2: 0-2, 1-3, 2-4, 3-5... etc
        for(int i=0;i<strlen(s)-1;i++){ //Start value of 0-2, 1-3... etc
            int j=i+l-1; //Calculate i to j for 0-2, 1-3... etc
            if (s[i] == s[j] && l == 2){ //If length is 2 and same
                characters, ONLY INCREMNET BY 1
              arr[i][j] = 2;
```

```
            }if(s[i]==s[j]){ //Else, ALWAYS INCREMENT BY 2 with the value
                diagonal, down to the left
                 arr[i][j]=arr[i+1][j-1]+2;
            }else{ //Else, MAX OF BELOW AND TO THE LEFT
               arr[i][j]=max(arr[i][j-1],arr[i+1][j]);
            }
         }
      }
   return arr[0][strlen(s)-1]; //Return the end of the first row which
       is the max palindromic subsequence length
}

int main(){
   // char seq[] = "alaaabaaabbbbaaaa";
   char seq[] = "agbdba";
   int n = strlen(seq);
   printf ("The length of the LPS is %d\n", lps(seq));
   return 0;
}
```

3.11. lrs

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

using namespace std;

//Longest Repeating Subsequence
//A simple variation from Longest Common Subsequence (LCS)

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

string lrs(string str){
  int n=str.length();
  int dp[n+1][n+1];

  for(int i=0;i<=n;i++){
    dp[0][i]=0;
    dp[i][0]=0;
  }
  for(int i=1;i<=n;i++){
    for(int j=1;j<=n;j++){
      if(str[i-1]==str[j-1]&& i!=j){
        dp[i][j]=dp[i-1][j-1]+1;
      }else{
        dp[i][j]=max(dp[i-1][j],dp[i][j-1]);
      }
    }
  }
  return dp[n][n];
}

int main(){
   string str = "AABEBCDD";
   cout << lrs(str)<<endl;
   return 0;
}
```

3.12. maxSumNonAdjSubseq

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

using namespace std;

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

int FindMaxSum(int arr[],int n){
  int exclusive=0; //Exclusive starts at 0 because at arr[1], we musn't
      take into account arr[0]
  int inclusive=arr[0]; //Inclusive starts at arr[0]
  int newex=0; //Just a placeholder
  for(int i=1;i<n;i++){
    newex=inclusive; //Save the old inclusive for exclusive
    inclusive=max(inclusive,exclusive+arr[i]); //New inclusive will be
        the max of current high, and to see if new high
    exclusive=newex; //Exclusive will be the past high which is high
        until i-1
  }
```

```
  return inclusive; //Return inclusive because it is the highest at point
      N-1
}

int main(){
  int arr[] = {5, 5, 10, 100, 10, 5};
  int n = sizeof(arr) / sizeof(arr[0]);
  printf("%d\n", FindMaxSum(arr, n));
  return 0;
}
```

3.13. minimumJumps

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

using namespace std;

int minJumps(int arr[],int n){
  int dp[n];
  dp[0]=0;

  for(int i=1;i<n;i++){ //Iterate through all, to check with each place
    dp[i]=INT_MAX-1; //Start with INT_MAX in case you can not reach it
    for(int j=0;j<i;j++){ //Go through each element to see if you can
        reach arr[i] from it
      if(arr[j]+j>=i&&dp[j]+1<dp[i]){ //If you can reach and if dp[j]+1
          is smaller, we change it
        dp[i]=dp[j]+1;
      }
    }
  }
  return dp[n-1]==INT_MAX-1?-1:dp[n-1]; //If the end is INT_MAX-1, it
      means you can not reach it
}

int main(){
  int arr[] = {1, 3, 6, 3, 2, 3, 6, 8, 9, 5};
  int n = sizeof(arr)/sizeof(arr[0]);
  printf("Minimum number of jumps to reach end is %d\n", minJumps(arr,n));
  return 0;
}
```

3.14. minimumJumpsn

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

using namespace std;

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

int minJumpsn(int arr[],int n){

  if(n<=1) return 0; //If array is empty, return 0
  if(arr[0]==0) return -1; //If it can not get out of first index, return
      -1

  int maxReach=arr[0]; //MaxReach starts as first available option
  int step=arr[0]; //Amount of steps you may take is first available
      option
  int jumps=1; //The jump it takes to get to MaxReach

  for(int i=1;i<n;i++){ //Goes through the array
    if(i==n-1) return jumps; //Returns the least jumps to get to n-1

    maxReach=max(maxReach,i+arr[i]); //Check if the new arr[] index
        reaches a new max

    step--; //You use a step to get to this index;

    if(step==0){ //If you reach the end of your steps with with and you
        have no steps left
      jump++; //Increase jump because a new jump is needed

      if(i >= maxReach){ //If you exceed your maxReach, it is because you
          can not reach it
        return -1; //Return -1
      }
      step = maxReach - i; //
    }
  }

}

int main(){
```

```
  int arr[] = {1, 3, 6, 3, 2, 3, 6, 8, 9, 5};
  int n = sizeof(arr)/sizeof(arr[0]);
  printf("Minimum number of jumps to reach end is %d\n",
      minJumpsn(arr,n));
  return 0;
}
```

3.15. nonconsecutive1

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

using namespace std;

int countStrings(int n){
    int a[n], b[n];
    a[0] = b[0] = 1;
    for (int i = 1; i < n; i++){
        a[i] = a[i-1] + b[i-1];
        b[i] = a[i-1];
    }
    return a[n-1] + b[n-1];
}

int main(){
  for(int i=0;i<15;i++)
    cout << countStrings(i) << endl;
  return 0;
}
```

3.16. printFuncLCS

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

using namespace std;

int main(){
  //GIVEN THAT L is DP already done
  char lcs[index+1];
  lcs[index] = '\0'; // Set the terminating character

  // Start from the right-most-bottom-most corner and
  // one by one store characters in lcs[]
  int i = m, j = n;
  while (i > 0 && j > 0){
     // If current character in X[] and Y are same, then
     // current character is part of LCS
     if (X[i-1] == Y[j-1]){
         lcs[index-1] = X[i-1]; // Put current character in result
         i--; j--; index--;   // reduce values of i, j and index
     }

     // If not same, then find the larger of two and
     // go in the direction of larger value
     else if (L[i-1][j] > L[i][j-1]){
       i--;
     }
     else{
        j--;
     }
  }
  // Print the lcs
  cout << "LCS of " << X << " and " << Y << " is " << lcs;
}
```

4. fun

4.1. 8puzzle

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

using namespace std;

bool isSolvable(int puzzle[]){
  int inv=0;
  for(int i=0;i<9;i++){
    for(int j=i+1;j<9;j++){
      if((puzzle[i]&&puzzle[j])&&(puzzle[i]>puzzle[j])) inv++;
    }
  }
  if(inv%2) return false;
```

```
  return true;
}

int main(){

  int **puzzle= new int*[3];
  puzzle[0]=new int[3]{1,8,2};
  puzzle[1]=new int[3]{0,4,3};
  puzzle[2]=new int[3]{7,6,5};

  isSolvable((int *)puzzle)? cout << "Solvable"<<endl:
                    cout << "Not Solvable"<<endl;
  return 0;
}
```

5. graph

5.1. Connectivity

5.1.1. biconnected

```
// A C++ program to find if a given undirected graph is
// biconnected
#include<iostream>
#include <list>
#define NIL -1
using namespace std;

// A class that represents an undirected graph
class Graph
{
    int V;   // No. of vertices
    list<int> *adj;  // A dynamic array of adjacency lists
    bool isBCUtil(int v, bool visited[], int disc[], int low[],
            int parent[]);
public:
    Graph(int V); // Constructor
    void addEdge(int v, int w); // to add an edge to graph
    bool isBC();  // returns true if graph is Biconnected
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w);
    adj[w].push_back(v); // Note: the graph is undirected
}

// A recursive function that returns true if there is an articulation
// point in given graph, otherwise returns false.
// This function is almost same as isAPUtil() here ( http://goo.gl/Me9Fw )
// u --> The vertex to be visited next
// visited[] --> keeps tract of visited vertices
// disc[] --> Stores discovery times of visited vertices
// parent[] --> Stores parent vertices in DFS tree
bool Graph::isBCUtil(int u, bool visited[], int disc[],int low[],int
     parent[])
{
    // A static variable is used for simplicity, we can avoid use of
        static
    // variable by passing a pointer.
    static int time = 0;

    // Count of children in DFS Tree
    int children = 0;

    // Mark the current node as visited
    visited[u] = true;

    // Initialize discovery time and low value
    disc[u] = low[u] = ++time;

    // Go through all vertices aadjacent to this
    list<int>::iterator i;
    for (i = adj[u].begin(); i != adj[u].end(); ++i)
    {
        int v = *i; // v is current adjacent of u

        // If v is not visited yet, then make it a child of u
        // in DFS tree and recur for it
        if (!visited[v])
        {
            children++;
```

```
            parent[v] = u;

            // check if subgraph rooted with v has an articulation point
            if (isBCUtil(v, visited, disc, low, parent))
               return true;

            // Check if the subtree rooted with v has a connection to
            // one of the ancestors of u
            low[u] = min(low[u], low[v]);

            // u is an articulation point in following cases

            // (1) u is root of DFS tree and has two or more chilren.
            if (parent[u] == NIL && children > 1)
               return true;

            // (2) If u is not root and low value of one of its child is
            // more than discovery value of u.
            if (parent[u] != NIL && low[v] >= disc[u])
               return true;
        }

        // Update low value of u for parent function calls.
        else if (v != parent[u])
            low[u] = min(low[u], disc[v]);
    }
    return false;
}

// The main function that returns true if graph is Biconnected,
// otherwise false. It uses recursive function isBCUtil()
bool Graph::isBC()
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    int *disc = new int[V];
    int *low = new int[V];
    int *parent = new int[V];

    // Initialize parent and visited, and ap(articulation point)
    //  arrays
    for (int i = 0; i < V; i++)
    {
        parent[i] = NIL;
        visited[i] = false;
    }

    // Call the recursive helper function to find if there is an
       articulation
    // point in given graph. We do DFS traversal starring from vertex 0
    if (isBCUtil(0, visited, disc, low, parent) == true)
        return false;

    // Now check whether the given graph is connected or not. An
       undirected
    // graph is connected if all vertices are reachable from any starting
    // point (we have taken 0 as starting point)
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            return false;

    return true;
}

// Driver program to test above function
int main()
{
    // Create graphs given in above diagrams
    Graph g1(2);
    g1.addEdge(0, 1);
    g1.isBC()? cout << "Yes\n" : cout << "No\n";

    Graph g2(5);
    g2.addEdge(1, 0);
    g2.addEdge(0, 2);
    g2.addEdge(2, 1);
    g2.addEdge(0, 3);
    g2.addEdge(3, 4);
    g2.addEdge(2, 4);
    g2.isBC()? cout << "Yes\n" : cout << "No\n";

    Graph g3(3);
    g3.addEdge(0, 1);
    g3.addEdge(1, 2);
    g3.isBC()? cout << "Yes\n" : cout << "No\n";

    Graph g4(5);
    g4.addEdge(1, 0);
    g4.addEdge(0, 2);
    g4.addEdge(2, 1);
```

```
    g4.addEdge(0, 3);
    g4.addEdge(3, 4);
    g4.isBC()? cout << "Yes\n" : cout << "No\n";

    Graph g5(3);
    g5.addEdge(0, 1);
    g5.addEdge(1, 2);
    g5.addEdge(2, 0);
    g5.isBC()? cout << "Yes\n" : cout << "No\n";

    return 0;
}
```

5.1.2. bridges

```
// A C++ program to find bridges in a given undirected graph
#include<iostream>
#include <list>
#define NIL -1
using namespace std;

// A class that represents an undirected graph
class Graph
{
    int V;   // No. of vertices
    list<int> *adj;  // A dynamic array of adjacency lists
    void bridgeUtil(int v, bool visited[], int disc[], int low[],
                    int parent[]);
public:
    Graph(int V); // Constructor
    void addEdge(int v, int w); // to add an edge to graph
    void bridge();  // prints all bridges
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w);
    adj[w].push_back(v); // Note: the graph is undirected
}

// A recursive function that finds and prints bridges using
// DFS traversal
// u --> The vertex to be visited next
// visited[] --> keeps tract of visited vertices
// disc[] --> Stores discovery times of visited vertices
// parent[] --> Stores parent vertices in DFS tree
void Graph::bridgeUtil(int u, bool visited[], int disc[],
                                  int low[], int parent[])
{
    // A static variable is used for simplicity, we can
    // avoid use of static variable by passing a pointer.
    static int time = 0;

    // Mark the current node as visited
    visited[u] = true;

    // Initialize discovery time and low value
    disc[u] = low[u] = ++time;

    // Go through all vertices aadjacent to this
    list<int>::iterator i;
    for (i = adj[u].begin(); i != adj[u].end(); ++i)
    {
        int v = *i; // v is current adjacent of u

        // If v is not visited yet, then recur for it
        if (!visited[v])
        {
            parent[v] = u;
            bridgeUtil(v, visited, disc, low, parent);

            // Check if the subtree rooted with v has a
            // connection to one of the ancestors of u
            low[u] = min(low[u], low[v]);

            // If the lowest vertex reachable from subtree
            // under v is below u in DFS tree, then u-v
            // is a bridge
            if (low[v] > disc[u])
              cout << u <<" " << v << endl;
        }
```

```
        // Update low value of u for parent function calls.
        else if (v != parent[u])
            low[u] = min(low[u], disc[v]);
    }
}

// DFS based function to find all bridges. It uses recursive
// function bridgeUtil()
void Graph::bridge()
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    int *disc = new int[V];
    int *low = new int[V];
    int *parent = new int[V];

    // Initialize parent and visited arrays
    for (int i = 0; i < V; i++)
    {
        parent[i] = NIL;
        visited[i] = false;
    }

    // Call the recursive helper function to find Bridges
    // in DFS tree rooted with vertex 'i'
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            bridgeUtil(i, visited, disc, low, parent);
}

// Driver program to test above function
int main()
{
    // Create graphs given in above diagrams
    cout << "\nBridges in first graph \n";
    Graph g1(5);
    g1.addEdge(1, 0);
    g1.addEdge(0, 2);
    g1.addEdge(2, 1);
    g1.addEdge(0, 3);
    g1.addEdge(3, 4);
    g1.bridge();

    cout << "\nBridges in second graph \n";
    Graph g2(4);
    g2.addEdge(0, 1);
    g2.addEdge(1, 2);
    g2.addEdge(2, 3);
    g2.bridge();

    cout << "\nBridges in third graph \n";
    Graph g3(7);
    g3.addEdge(0, 1);
    g3.addEdge(1, 2);
    g3.addEdge(2, 0);
    g3.addEdge(1, 3);
    g3.addEdge(1, 4);
    g3.addEdge(1, 6);
    g3.addEdge(3, 5);
    g3.addEdge(4, 5);
    g3.bridge();

    return 0;
}
```

5.1.3. isConnected

```
// C++ program to check if there is exist a path between two vertices
// of a graph.
#include<iostream>
#include <list>
using namespace std;

// This class represents a directed graph using adjacency list
// representation
class Graph
{
    int V;    // No. of vertices
    list<int> *adj;  // Pointer to an array containing adjacency lists
public:
    Graph(int V); // Constructor
    void addEdge(int v, int w); // function to add an edge to graph
    bool isReachable(int s, int d);
};

Graph::Graph(int V)
{
    this->V = V;
```

```
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to vs list.
}

// A BFS based function to check whether d is reachable from s.
bool Graph::isReachable(int s, int d)
{
    // Base case
    if (s == d)
      return true;

    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Create a queue for BFS
    list<int> queue;

    // Mark the current node as visited and enqueue it
    visited[s] = true;
    queue.push_back(s);

    // it will be used to get all adjacent vertices of a vertex
    list<int>::iterator i;

    while (!queue.empty())
    {
        // Dequeue a vertex from queue and print it
        s = queue.front();
        queue.pop_front();

        // Get all adjacent vertices of the dequeued vertex s
        // If a adjacent has not been visited, then mark it visited
        // and enqueue it
        for (i = adj[s].begin(); i != adj[s].end(); ++i)
        {
            // If this adjacent node is the destination node, then
            // return true
            if (*i == d)
                return true;

            // Else, continue to do BFS
            if (!visited[*i])
            {
                visited[*i] = true;
                queue.push_back(*i);
            }
        }
    }

    // If BFS is complete without visiting d
    return false;
}

// Driver program to test methods of graph class
int main()
{
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    int u = 1, v = 3;
    if(g.isReachable(u, v))
        cout<< "\n There is a path from " << u << " to " << v;
    else
        cout<< "\n There is no path from " << u << " to " << v;

    u = 3, v = 1;
    if(g.isReachable(u, v))
        cout<< "\n There is a path from " << u << " to " << v;
    else
        cout<< "\n There is no path from " << u << " to " << v;

    return 0;
}
```

5.1.4. isStronglyConnected

```cpp
// C++ program to check if a given directed graph is strongly
// connected or not
#include <iostream>
#include <list>
#include <stack>
using namespace std;

class Graph
{
    int V;   // No. of vertices
    list<int> *adj;  // An array of adjacency lists

    // A recursive function to print DFS starting from v
    void DFSUtil(int v, bool visited[]);
public:
    // Constructor and Destructor
    Graph(int V) { this->V = V; adj = new list<int>[V];}
    ~Graph() { delete [] adj; }

    // Method to add an edge
    void addEdge(int v, int w);

    // The main function that returns true if the graph is strongly
    // connected, otherwise false
    bool isSC();

    // Function that returns reverse (or transpose) of this graph
    Graph getTranspose();
};

// A recursive function to print DFS starting from v
void Graph::DFSUtil(int v, bool visited[])
{
    // Mark the current node as visited and print it
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFSUtil(*i, visited);
}

// Function that returns reverse (or transpose) of this graph
Graph Graph::getTranspose()
{
    Graph g(V);
    for (int v = 0; v < V; v++)
    {
        // Recur for all the vertices adjacent to this vertex
        list<int>::iterator i;
        for(i = adj[v].begin(); i != adj[v].end(); ++i)
        {
            g.adj[*i].push_back(v);
        }
    }
    return g;
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to vs list.
}

// The main function that returns true if graph is strongly connected
bool Graph::isSC()
{
    // St1p 1: Mark all the vertices as not visited (For first DFS)
    bool visited[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Step 2: Do DFS traversal starting from first vertex.
    DFSUtil(0, visited);

     // If DFS traversal doesnt visit all vertices, then return false.
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
             return false;

    // Step 3: Create a reversed graph
    Graph gr = getTranspose();

    // Step 4: Mark all the vertices as not visited (For second DFS)
    for(int i = 0; i < V; i++)
        visited[i] = false;

    // Step 5: Do DFS for reversed graph starting from first vertex.
    // Staring Vertex must be same starting point of first DFS
    gr.DFSUtil(0, visited);
```

```
    // If all vertices are not visited in second DFS, then
    // return false
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
             return false;

    return true;
}

// Driver program to test above functions
int main()
{
    // Create graphs given in the above diagrams
    Graph g1(5);
    g1.addEdge(0, 1);
    g1.addEdge(1, 2);
    g1.addEdge(2, 3);
    g1.addEdge(3, 0);
    g1.addEdge(2, 4);
    g1.addEdge(4, 2);
    g1.isSC()? cout << "Yes\n" : cout << "No\n";

    Graph g2(4);
    g2.addEdge(0, 1);
    g2.addEdge(1, 2);
    g2.addEdge(2, 3);
    g2.isSC()? cout << "Yes\n" : cout << "No\n";

    return 0;
}
```

5.1.5. nIslands

```
// Program to count islands in boolean 2D matrix
#include <stdio.h>
#include <string.h>
#include <stdbool.h>

#define ROW 5
#define COL 5

// A function to check if a given cell (row, col) can be included in DFS
int isSafe(int M[][COL], int row, int col, bool visited[][COL])
{
    // row number is in range, column number is in range and value is 1
    // and not yet visited
    return (row >= 0) && (row < ROW) &&
           (col >= 0) && (col < COL) &&
           (M[row][col] && !visited[row][col]);
}

// A utility function to do DFS for a 2D boolean matrix. It only considers
// the 8 neighbours as adjacent vertices
void DFS(int M[][COL], int row, int col, bool visited[][COL])
{
    // These arrays are used to get row and column numbers of 8 neighbours
    // of a given cell
    static int rowNbr[] = {-1, -1, -1, 0, 0, 1, 1, 1};
    static int colNbr[] = {-1, 0, 1, -1, 1, -1, 0, 1};

    // Mark this cell as visited
    visited[row][col] = true;

    // Recur for all connected neighbours
    for (int k = 0; k < 8; ++k)
        if (isSafe(M, row + rowNbr[k], col + colNbr[k], visited) )
            DFS(M, row + rowNbr[k], col + colNbr[k], visited);
}

// The main function that returns count of islands in a given boolean
// 2D matrix
int countIslands(int M[][COL])
{
    // Make a bool array to mark visited cells.
    // Initially all cells are unvisited
    bool visited[ROW][COL];
    memset(visited, 0, sizeof(visited));

    // Initialize count as 0 and travese through the all cells of
    // given matrix
    int count = 0;
    for (int i = 0; i < ROW; ++i)
        for (int j = 0; j < COL; ++j)
            if (M[i][j] && !visited[i][j]) // If a cell with value 1 is not
            {                              // visited yet, then new island found
                DFS(M, i, j, visited);     // Visit all cells in this island.
                ++count;                   // and increment island count
```

```
        }

    return count;
}

// Driver program to test above function
int main()
{
    int M[][COL]= { {1, 1, 0, 0, 0},
        {0, 1, 0, 0, 1},
        {1, 0, 0, 1, 1},
        {0, 0, 0, 0, 0},
        {1, 0, 1, 0, 1}
    };

    printf("Number of islands is: %d\n", countIslands(M));

    return 0;
}
```

5.1.6. tarjan

```
// A C++ program to find strongly connected components in a given
// directed graph using Tarjan's algorithm (single DFS)
#include<iostream>
#include <list>
#include <stack>
#define NIL -1
using namespace std;

// A class that represents an directed graph
class Graph
{
    int V;   // No. of vertices
    list<int> *adj;  // A dynamic array of adjacency lists

    // A Recursive DFS based function used by SCC()
    void SCCUtil(int u, int disc[], int low[],
                 stack<int> *st, bool stackMember[]);
public:
    Graph(int V); // Constructor
    void addEdge(int v, int w); // function to add an edge to graph
    void SCC();  // prints strongly connected components
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w);
}

// A recursive function that finds and prints strongly connected
// components using DFS traversal
// u --> The vertex to be visited next
// disc[] --> Stores discovery times of visited vertices
// low[] -- >> earliest visited vertex (the vertex with minimum
//             discovery time) that can be reached from subtree
//             rooted with current vertex
// *st -- >> To store all the connected ancestors (could be part
//           of SCC)
// stackMember[] --> bit/index array for faster check whether
//                   a node is in stack
void Graph::SCCUtil(int u, int disc[], int low[], stack<int> *st,
                    bool stackMember[])
{
    // A static variable is used for simplicity, we can avoid use
    // of static variable by passing a pointer.
    static int time = 0;

    // Initialize discovery time and low value
    disc[u] = low[u] = ++time;
    st->push(u);
    stackMember[u] = true;

    // Go through all vertices adjacent to this
    list<int>::iterator i;
    for (i = adj[u].begin(); i != adj[u].end(); ++i)
    {
        int v = *i; // v is current adjacent of 'u'

        // If v is not visited yet, then recur for it
        if (disc[v] == -1)
        {
```

```
            SCCUtil(v, disc, low, st, stackMember);

            // Check if the subtree rooted with 'v' has a
            // connection to one of the ancestors of 'u'
            // Case 1 (per above discussion on Disc and Low value)
            low[u] = min(low[u], low[v]);
        }

        // Update low value of 'u' only of 'v' is still in stack
        // (i.e. it's a back edge, not cross edge).
        // Case 2 (per above discussion on Disc and Low value)
        else if (stackMember[v] == true)
            low[u] = min(low[u], disc[v]);
    }

    // head node found, pop the stack and print an SCC
    int w = 0; // To store stack extracted vertices
    if (low[u] == disc[u])
    {
        while (st->top() != u)
        {
            w = (int) st->top();
            cout << w << " ";
            stackMember[w] = false;
            st->pop();
        }
        w = (int) st->top();
        cout << w << "n";
        stackMember[w] = false;
        st->pop();
    }
}

// The function to do DFS traversal. It uses SCCUtil()
void Graph::SCC()
{
    int *disc = new int[V];
    int *low = new int[V];
    bool *stackMember = new bool[V];
    stack<int> *st = new stack<int>();

    // Initialize disc and low, and stackMember arrays
    for (int i = 0; i < V; i++)
    {
        disc[i] = NIL;
        low[i] = NIL;
        stackMember[i] = false;
    }

    // Call the recursive helper function to find strongly
    // connected components in DFS tree with vertex 'i'
    for (int i = 0; i < V; i++)
        if (disc[i] == NIL)
            SCCUtil(i, disc, low, st, stackMember);
}

// Driver program to test above function
int main()
{
    cout << "nSCCs in first graph n";
    Graph g1(5);
    g1.addEdge(1, 0);
    g1.addEdge(0, 2);
    g1.addEdge(2, 1);
    g1.addEdge(0, 3);
    g1.addEdge(3, 4);
    g1.SCC();

    cout << "nSCCs in second graph n";
    Graph g2(4);
    g2.addEdge(0, 1);
    g2.addEdge(1, 2);
    g2.addEdge(2, 3);
    g2.SCC();

    cout << "nSCCs in third graph n";
    Graph g3(7);
    g3.addEdge(0, 1);
    g3.addEdge(1, 2);
    g3.addEdge(2, 0);
    g3.addEdge(1, 3);
    g3.addEdge(1, 4);
    g3.addEdge(1, 6);
    g3.addEdge(3, 5);
    g3.addEdge(4, 5);
    g3.SCC();

    cout << "nSCCs in fourth graph n";
    Graph g4(11);
    g4.addEdge(0,1);g4.addEdge(0,3);
```

```
    g4.addEdge(1,2);g4.addEdge(1,4);
    g4.addEdge(2,0);g4.addEdge(2,6);
    g4.addEdge(3,2);
    g4.addEdge(4,5);g4.addEdge(4,6);
    g4.addEdge(5,6);g4.addEdge(5,7);g4.addEdge(5,8);g4.addEdge(5,9);
    g4.addEdge(6,4);
    g4.addEdge(7,9);
    g4.addEdge(8,9);
    g4.addEdge(9,8);
    g4.SCC();

    cout << "nSCCs in fifth graph n";
    Graph g5(5);
    g5.addEdge(0,1);
    g5.addEdge(1,2);
    g5.addEdge(2,3);
    g5.addEdge(2,4);
    g5.addEdge(3,0);
    g5.addEdge(4,2);
    g5.SCC();

    return 0;
}
```

5.2. MinimumSpanningTree

5.2.1. kruskal

```
// C++ program for Kruskal's algorithm to find Minimum Spanning Tree
// of a given connected, undirected and weighted graph
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// a structure to represent a weighted edge in graph
struct Edge
{
    int src, dest, weight;
};

// a structure to represent a connected, undirected
// and weighted graph
struct Graph
{
    // V-> Number of vertices, E-> Number of edges
    int V, E;

    // graph is represented as an array of edges.
    // Since the graph is undirected, the edge
    // from src to dest is also edge from dest
    // to src. Both are counted as 1 edge here.
    struct Edge* edge;
};

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
    struct Graph* graph = new Graph;
    graph->V = V;
    graph->E = E;

    graph->edge = new Edge[E];

    return graph;
}

// A structure to represent a subset for union-find
struct subset
{
    int parent;
    int rank;
};

// A utility function to find set of an element i
// (uses path compression technique)
int find(struct subset subsets[], int i)
{
    // find root and make root as parent of i
    // (path compression)
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);

    return subsets[i].parent;
}

// A function that does union of two sets of x and y
// (uses union by rank)
void Union(struct subset subsets[], int x, int y)
```

```
{
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

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

    // If ranks are same, then make one as root and
    // increment its rank by one
    else
    {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

// Compare two edges according to their weights.
// Used in qsort() for sorting an array of edges
int myComp(const void* a, const void* b)
{
    struct Edge* a1 = (struct Edge*)a;
    struct Edge* b1 = (struct Edge*)b;
    return a1->weight > b1->weight;
}

// The main function to construct MST using Kruskal's algorithm
void KruskalMST(struct Graph* graph)
{
    int V = graph->V;
    struct Edge result[V]; // Tnis will store the resultant MST
    int e = 0; // An index variable, used for result[]
    int i = 0; // An index variable, used for sorted edges

    // Step 1: Sort all the edges in non-decreasing
    // order of their weight. If we are not allowed to
    // change the given graph, we can create a copy of
    // array of edges
    qsort(graph->edge, graph->E, sizeof(graph->edge[0]), myComp);

    // Allocate memory for creating V ssubsets
    struct subset *subsets =
        (struct subset*) malloc( V * sizeof(struct subset) );

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

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

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

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

    // print the contents of result[] to display the
    // built MST
    printf("Following are the edges in the constructed MST\n");
    for (i = 0; i < e; ++i)
        printf("%d -- %d == %d\n", result[i].src, result[i].dest,
                                                result[i].weight);
    return;
}

// Driver program to test above functions
int main()
{
    /* Let us create following weighted graph
            10
        0--------1
```

```
      |  \     |
     6|   5\   |15
      |      \ |
      2--------3
          4       */
    int V = 4; // Number of vertices in graph
    int E = 5; // Number of edges in graph
    struct Graph* graph = createGraph(V, E);


    // add edge 0-1
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;
    graph->edge[0].weight = 10;

    // add edge 0-2
    graph->edge[1].src = 0;
    graph->edge[1].dest = 2;
    graph->edge[1].weight = 6;

    // add edge 0-3
    graph->edge[2].src = 0;
    graph->edge[2].dest = 3;
    graph->edge[2].weight = 5;

    // add edge 1-3
    graph->edge[3].src = 1;
    graph->edge[3].dest = 3;
    graph->edge[3].weight = 15;

    // add edge 2-3
    graph->edge[4].src = 2;
    graph->edge[4].dest = 3;
    graph->edge[4].weight = 4;

    KruskalMST(graph);

    return 0;
}
```

5.2.2. prim

```
// Like  Kruskals  algorithm,  Prims  algorithm is also a Greedy algorithm.
// It starts with an empty spanning tree. The idea is to maintain two
    sets of vertices.
// The first set contains the vertices already included in the MST,
// the other set contains the vertices not yet included. At every step,
// it considers all the edges that connect the two sets, and picks the
    minimum weight edge from these edges.
// After picking the edge, it moves the other endpoint of the edge to the
    set containing MST.


// How does  Prims  Algorithm Work?
// The idea behind Prims algorithm is simple, a spanning tree means all
    vertices must be connected.
// So the two disjoint subsets (discussed above) of vertices must be
    connected to make a Spanning Tree.
// And they must be connected with the minimum weight edge to make it a
    Minimum Spanning Tree.

// A C / C++ program for Prim's Minimum Spanning Tree (MST) algorithm.
// The program is for adjacency matrix representation of the graph

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

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

// A utility function to find the vertex with minimum key value, from
// the set of vertices not yet included in MST
int minKey(int key[], bool mstSet[])
{
   // Initialize min value
   int min = INT_MAX, min_index;

   for (int v = 0; v < V; v++)
     if (mstSet[v] == false && key[v] < min)
         min = key[v], min_index = v;

   return min_index;
}

// A utility function to print the constructed MST stored in parent[]
int printMST(int parent[], int n, int graph[V][V])
{
   printf("Edge  Weight\n");
```

```
    for (int i = 1; i < V; i++)
       printf("%d - %d  %d \n", parent[i], i, graph[i][parent[i]]);
}

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

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

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

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

       // Add the picked vertex to the MST Set
       mstSet[u] = true;

       // Update key value and parent index of the adjacent vertices of
       // the picked vertex. Consider only those vertices which are not
          yet
       // included in MST
       for (int v = 0; v < V; v++)

          // graph[u][v] is non zero only for adjacent vertices of m
          // mstSet[v] is false for vertices not yet included in MST
          // Update the key only if graph[u][v] is smaller than key[v]
         if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
            parent[v] = u, key[v] = graph[u][v];
    }

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


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

    // Print the solution
    primMST(graph);

    return 0;
}
```

5.3. ShortestPath

5.3.1. acyclicGraph

```
// C++ program to find single source shortest paths for Directed Acyclic
    Graphs
#include<iostream>
#include <list>
#include <stack>
#include <limits.h>
#define INF INT_MAX
using namespace std;
```

```cpp
// Graph is represented using adjacency list. Every node of adjacency list
// contains vertex number of the vertex to which edge connects. It also
// contains weight of the edge
class AdjListNode
{
    int v;
    int weight;
public:
    AdjListNode(int _v, int _w) { v = _v; weight = _w;}
    int getV()     {  return v; }
    int getWeight() { return weight; }
};

// Class to represent a graph using adjacency list representation
class Graph
{
    int V;   // No. of vertices'

    // Pointer to an array containing adjacency lists
    list<AdjListNode> *adj;

    // A function used by shortestPath
    void topologicalSortUtil(int v, bool visited[], stack<int> &Stack);
public:
    Graph(int V); // Constructor

    // function to add an edge to graph
    void addEdge(int u, int v, int weight);

    // Finds shortest paths from given source vertex
    void shortestPath(int s);
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<AdjListNode>[V];
}

void Graph::addEdge(int u, int v, int weight)
{
    AdjListNode node(v, weight);
    adj[u].push_back(node); // Add v to u's list
}
```

```cpp
// A recursive function used by shortestPath. See below link for details
// http://www.geeksforgeeks.org/topological-sorting/
void Graph::topologicalSortUtil(int v, bool visited[], stack<int> &Stack)
{
    // Mark the current node as visited
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    list<AdjListNode>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
    {
        AdjListNode node = *i;
        if (!visited[node.getV()])
            topologicalSortUtil(node.getV(), visited, Stack);
    }

    // Push current vertex to stack which stores topological sort
    Stack.push(v);
}

// The function to find shortest paths from given vertex. It uses
    recursive
// topologicalSortUtil() to get topological sorting of given graph.
void Graph::shortestPath(int s)
{
    stack<int> Stack;
    int dist[V];

    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function to store Topological Sort
    // starting from all vertices one by one
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            topologicalSortUtil(i, visited, Stack);

    // Initialize distances to all vertices as infinite and distance
    // to source as 0
    for (int i = 0; i < V; i++)
        dist[i] = INF;
    dist[s] = 0;
```

```
    // Process vertices in topological order
    while (Stack.empty() == false)
    {
        // Get the next vertex from topological order
        int u = Stack.top();
        Stack.pop();

        // Update distances of all adjacent vertices
        list<AdjListNode>::iterator i;
        if (dist[u] != INF)
        {
          for (i = adj[u].begin(); i != adj[u].end(); ++i)
             if (dist[i->getV()] > dist[u] + i->getWeight())
                dist[i->getV()] = dist[u] + i->getWeight();
        }
    }

    // Print the calculated shortest distances
    for (int i = 0; i < V; i++)
        (dist[i] == INF)? cout << "INF ": cout << dist[i] << " ";
}

// Driver program to test above functions
int main()
{
    // Create a graph given in the above diagram. Here vertex numbers are
    // 0, 1, 2, 3, 4, 5 with following mappings:
    // 0=r, 1=s, 2=t, 3=x, 4=y, 5=z
    Graph g(6);
    g.addEdge(0, 1, 5);
    g.addEdge(0, 2, 3);
    g.addEdge(1, 3, 6);
    g.addEdge(1, 2, 2);
    g.addEdge(2, 4, 4);
    g.addEdge(2, 5, 2);
    g.addEdge(2, 3, 7);
    g.addEdge(3, 4, -1);
    g.addEdge(4, 5, -2);

    int s = 1;
    cout << "Following are shortest distances from source " << s <<" n";
    g.shortestPath(s);

    return 0;
}
```

5.3.2. bellmanFord

```
// A C++ program for Bellman-Ford's single source
// shortest path algorithm.
#include <bits/stdc++.h>

// a structure to represent a weighted edge in graph
struct Edge
{
    int src, dest, weight;
};

// a structure to represent a connected, directed and
// weighted graph
struct Graph
{
    // V-> Number of vertices, E-> Number of edges
    int V, E;

    // graph is represented as an array of edges.
    struct Edge* edge;
};

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
    struct Graph* graph = new Graph;
    graph->V = V;
    graph->E = E;
    graph->edge = new Edge[E];
    return graph;
}

// A utility function used to print the solution
void printArr(int dist[], int n)
{
    printf("Vertex Distance from Source\n");
    for (int i = 0; i < n; ++i)
        printf("%d \t\t %d\n", i, dist[i]);
}

// The main function that finds shortest distances from src to
// all other vertices using Bellman-Ford algorithm. The function
// also detects negative weight cycle
void BellmanFord(struct Graph* graph, int src)
```

```
{
    int V = graph->V;
    int E = graph->E;
    int dist[V];

    // Step 1: Initialize distances from src to all other vertices
    // as INFINITE
    for (int i = 0; i < V; i++)
        dist[i]  = INT_MAX;
    dist[src] = 0;

    // Step 2: Relax all edges |V| - 1 times. A simple shortest
    // path from src to any other vertex can have at-most |V| - 1
    // edges
    for (int i = 1; i <= V-1; i++)
    {
        for (int j = 0; j < E; j++)
        {
            int u = graph->edge[j].src;
            int v = graph->edge[j].dest;
            int weight = graph->edge[j].weight;
            if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
                dist[v] = dist[u] + weight;
        }
    }

    // Step 3: check for negative-weight cycles. The above step
    // guarantees shortest distances if graph doesn't contain
    // negative weight cycle. If we get a shorter path, then there
    // is a cycle.
    for (int i = 0; i < E; i++)
    {
        int u = graph->edge[i].src;
        int v = graph->edge[i].dest;
        int weight = graph->edge[i].weight;
        if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
            printf("Graph contains negative weight cycle");
    }

    printArr(dist, V);

    return;
}

// Driver program to test above functions
int main()
{
    /* Let us create the graph given in above example */
    int V = 5; // Number of vertices in graph
    int E = 8; // Number of edges in graph
    struct Graph* graph = createGraph(V, E);

    // add edge 0-1 (or A-B in above figure)
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;
    graph->edge[0].weight = -1;

    // add edge 0-2 (or A-C in above figure)
    graph->edge[1].src = 0;
    graph->edge[1].dest = 2;
    graph->edge[1].weight = 4;

    // add edge 1-2 (or B-C in above figure)
    graph->edge[2].src = 1;
    graph->edge[2].dest = 2;
    graph->edge[2].weight = 3;

    // add edge 1-3 (or B-D in above figure)
    graph->edge[3].src = 1;
    graph->edge[3].dest = 3;
    graph->edge[3].weight = 2;

    // add edge 1-4 (or A-E in above figure)
    graph->edge[4].src = 1;
    graph->edge[4].dest = 4;
    graph->edge[4].weight = 2;

    // add edge 3-2 (or D-C in above figure)
    graph->edge[5].src = 3;
    graph->edge[5].dest = 2;
    graph->edge[5].weight = 5;

    // add edge 3-1 (or D-B in above figure)
    graph->edge[6].src = 3;
    graph->edge[6].dest = 1;
    graph->edge[6].weight = 1;

    // add edge 4-3 (or E-D in above figure)
    graph->edge[7].src = 4;
    graph->edge[7].dest = 3;
```

```
    graph->edge[7].weight = -3;

    BellmanFord(graph, 0);

    return 0;
}
```

5.3.3. dijkstraAdjList

```
// C / C++ program for Dijkstra's shortest path algorithm for adjacency
// list representation of graph

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

// A structure to represent a node in adjacency list
struct AdjListNode
{
    int dest;
    int weight;
    struct AdjListNode* next;
};

// A structure to represent an adjacency liat
struct AdjList
{
    struct AdjListNode *head; // pointer to head node of list
};

// A structure to represent a graph. A graph is an array of adjacency
     lists.
// Size of array will be V (number of vertices in graph)
struct Graph
{
    int V;
    struct AdjList* array;
};

// A utility function to create a new adjacency list node
struct AdjListNode* newAdjListNode(int dest, int weight)
{
    struct AdjListNode* newNode =
            (struct AdjListNode*) malloc(sizeof(struct AdjListNode));
    newNode->dest = dest;
    newNode->weight = weight;
    newNode->next = NULL;
    return newNode;
}

// A utility function that creates a graph of V vertices
struct Graph* createGraph(int V)
{
    struct Graph* graph = (struct Graph*) malloc(sizeof(struct Graph));
    graph->V = V;

    // Create an array of adjacency lists. Size of array will be V
    graph->array = (struct AdjList*) malloc(V * sizeof(struct AdjList));

     // Initialize each adjacency list as empty by making head as NULL
    for (int i = 0; i < V; ++i)
        graph->array[i].head = NULL;

    return graph;
}

// Adds an edge to an undirected graph
void addEdge(struct Graph* graph, int src, int dest, int weight)
{
    // Add an edge from src to dest. A new node is added to the adjacency
    // list of src. The node is added at the begining
    struct AdjListNode* newNode = newAdjListNode(dest, weight);
    newNode->next = graph->array[src].head;
    graph->array[src].head = newNode;

    // Since graph is undirected, add an edge from dest to src also
    newNode = newAdjListNode(src, weight);
    newNode->next = graph->array[dest].head;
    graph->array[dest].head = newNode;
}

// Structure to represent a min heap node
struct MinHeapNode
{
    int  v;
    int dist;
};
```

```c
// Structure to represent a min heap
struct MinHeap
{
    int size;     // Number of heap nodes present currently
    int capacity; // Capacity of min heap
    int *pos;    // This is needed for decreaseKey()
    struct MinHeapNode **array;
};

// A utility function to create a new Min Heap Node
struct MinHeapNode* newMinHeapNode(int v, int dist)
{
    struct MinHeapNode* minHeapNode =
           (struct MinHeapNode*) malloc(sizeof(struct MinHeapNode));
    minHeapNode->v = v;
    minHeapNode->dist = dist;
    return minHeapNode;
}

// A utility function to create a Min Heap
struct MinHeap* createMinHeap(int capacity)
{
    struct MinHeap* minHeap =
         (struct MinHeap*) malloc(sizeof(struct MinHeap));
    minHeap->pos = (int *)malloc(capacity * sizeof(int));
    minHeap->size = 0;
    minHeap->capacity = capacity;
    minHeap->array =
         (struct MinHeapNode**) malloc(capacity * sizeof(struct
             MinHeapNode*));
    return minHeap;
}

// A utility function to swap two nodes of min heap. Needed for min
    heapify
void swapMinHeapNode(struct MinHeapNode** a, struct MinHeapNode** b)
{
    struct MinHeapNode* t = *a;
    *a = *b;
    *b = t;
}

// A standard function to heapify at given idx
// This function also updates position of nodes when they are swapped.
// Position is needed for decreaseKey()
void minHeapify(struct MinHeap* minHeap, int idx)
{
    int smallest, left, right;
    smallest = idx;
    left = 2 * idx + 1;
    right = 2 * idx + 2;

    if (left < minHeap->size &&
        minHeap->array[left]->dist < minHeap->array[smallest]->dist )
      smallest = left;

    if (right < minHeap->size &&
        minHeap->array[right]->dist < minHeap->array[smallest]->dist )
      smallest = right;

    if (smallest != idx)
    {
        // The nodes to be swapped in min heap
        MinHeapNode *smallestNode = minHeap->array[smallest];
        MinHeapNode *idxNode = minHeap->array[idx];

        // Swap positions
        minHeap->pos[smallestNode->v] = idx;
        minHeap->pos[idxNode->v] = smallest;

        // Swap nodes
        swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);

        minHeapify(minHeap, smallest);
    }
}

// A utility function to check if the given minHeap is ampty or not
int isEmpty(struct MinHeap* minHeap)
{
    return minHeap->size == 0;
}

// Standard function to extract minimum node from heap
struct MinHeapNode* extractMin(struct MinHeap* minHeap)
{
    if (isEmpty(minHeap))
        return NULL;

    // Store the root node
```

```c
    struct MinHeapNode* root = minHeap->array[0];

    // Replace root node with last node
    struct MinHeapNode* lastNode = minHeap->array[minHeap->size - 1];
    minHeap->array[0] = lastNode;

    // Update position of last node
    minHeap->pos[root->v] = minHeap->size-1;
    minHeap->pos[lastNode->v] = 0;

    // Reduce heap size and heapify root
    --minHeap->size;
    minHeapify(minHeap, 0);

    return root;
}

// Function to decreasy dist value of a given vertex v. This function
// uses pos[] of min heap to get the current index of node in min heap
void decreaseKey(struct MinHeap* minHeap, int v, int dist)
{
    // Get the index of v in heap array
    int i = minHeap->pos[v];

    // Get the node and update its dist value
    minHeap->array[i]->dist = dist;

    // Travel up while the complete tree is not hepified.
    // This is a O(Logn) loop
    while (i && minHeap->array[i]->dist < minHeap->array[(i - 1) /
        2]->dist)
    {
        // Swap this node with its parent
        minHeap->pos[minHeap->array[i]->v] = (i-1)/2;
        minHeap->pos[minHeap->array[(i-1)/2]->v] = i;
        swapMinHeapNode(&minHeap->array[i], &minHeap->array[(i - 1) / 2]);

        // move to parent index
        i = (i - 1) / 2;
    }
}

// A utility function to check if a given vertex
// 'v' is in min heap or not
bool isInMinHeap(struct MinHeap *minHeap, int v)
{
   if (minHeap->pos[v] < minHeap->size)
     return true;
   return false;
}

// A utility function used to print the solution
void printArr(int dist[], int n)
{
    printf("Vertex Distance from Source\n");
    for (int i = 0; i < n; ++i)
        printf("%d \t\t %d\n", i, dist[i]);
}

// The main function that calulates distances of shortest paths from src
    to all
// vertices. It is a O(ELogV) function
void dijkstra(struct Graph* graph, int src)
{
    int V = graph->V;// Get the number of vertices in graph
    int dist[V];    // dist values used to pick minimum weight edge in cut

    // minHeap represents set E
    struct MinHeap* minHeap = createMinHeap(V);

    // Initialize min heap with all vertices. dist value of all vertices
    for (int v = 0; v < V; ++v)
    {
        dist[v] = INT_MAX;
        minHeap->array[v] = newMinHeapNode(v, dist[v]);
        minHeap->pos[v] = v;
    }

    // Make dist value of src vertex as 0 so that it is extracted first
    minHeap->array[src] = newMinHeapNode(src, dist[src]);
    minHeap->pos[src] = src;
    dist[src] = 0;
    decreaseKey(minHeap, src, dist[src]);

    // Initially size of min heap is equal to V
    minHeap->size = V;

    // In the followin loop, min heap contains all nodes
    // whose shortest distance is not yet finalized.
    while (!isEmpty(minHeap))
```

```
    {
        // Extract the vertex with minimum distance value
        struct MinHeapNode* minHeapNode = extractMin(minHeap);
        int u = minHeapNode->v; // Store the extracted vertex number

        // Traverse through all adjacent vertices of u (the extracted
        // vertex) and update their distance values
        struct AdjListNode* pCrawl = graph->array[u].head;
        while (pCrawl != NULL)
        {
            int v = pCrawl->dest;

            // If shortest distance to v is not finalized yet, and
                distance to v
            // through u is less than its previously calculated distance
            if (isInMinHeap(minHeap, v) && dist[u] != INT_MAX &&
                                          pCrawl->weight + dist[u] < dist[v])
            {
                dist[v] = dist[u] + pCrawl->weight;

                // update distance value in min heap also
                decreaseKey(minHeap, v, dist[v]);
            }
            pCrawl = pCrawl->next;
        }
    }

    // print the calculated shortest distances
    printArr(dist, V);
}


// Driver program to test above functions
int main()
{
    // create the graph given in above fugure
    int V = 9;
    struct Graph* graph = createGraph(V);
    addEdge(graph, 0, 1, 4);
    addEdge(graph, 0, 7, 8);
    addEdge(graph, 1, 2, 8);
    addEdge(graph, 1, 7, 11);
    addEdge(graph, 2, 3, 7);
    addEdge(graph, 2, 8, 2);
    addEdge(graph, 2, 5, 4);
    addEdge(graph, 3, 4, 9);
    addEdge(graph, 3, 5, 14);
    addEdge(graph, 4, 5, 10);
    addEdge(graph, 5, 6, 2);
    addEdge(graph, 6, 7, 1);
    addEdge(graph, 6, 8, 6);
    addEdge(graph, 7, 8, 7);

    dijkstra(graph, 0);

    return 0;
}
```

5.3.4. floydWarshall

```
// The Floyd Warshall Algorithm is for solving the All Pairs Shortest
    Path problem.
// The problem is to find shortest distances between every pair of
    vertices in a
// given edge weighted directed Graph.

// C Program for Floyd Warshall Algorithm
#include<stdio.h>

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

/* Define Infinite as a large enough value. This value will be used
  for vertices not connected to each other */
#define INF 99999

// A function to print the solution matrix
void printSolution(int dist[][V]);

// Solves the all-pairs shortest path problem using Floyd Warshall
    algorithm
void floydWarshall (int graph[][V])
{
    /* dist[][] will be the output matrix that will finally have the
        shortest
      distances between every pair of vertices */
    int dist[V][V], i, j, k;
```

```
    /* Initialize the solution matrix same as input graph matrix. Or
       we can say the initial values of shortest distances are based
       on shortest paths considering no intermediate vertex. */
    for (i = 0; i < V; i++)
        for (j = 0; j < V; j++)
            dist[i][j] = graph[i][j];

    /* Add all vertices one by one to the set of intermediate vertices.
      ---> Before start of a iteration, we have shortest distances
           between all
      pairs of vertices such that the shortest distances consider only the
      vertices in set {0, 1, 2, .. k-1} as intermediate vertices.
      ----> After the end of a iteration, vertex no. k is added to the
           set of
      intermediate vertices and the set becomes {0, 1, 2, .. k} */
    for (k = 0; k < V; k++)
    {
        // Pick all vertices as source one by one
        for (i = 0; i < V; i++)
        {
            // Pick all vertices as destination for the
            // above picked source
            for (j = 0; j < V; j++)
            {
                // If vertex k is on the shortest path from
                // i to j, then update the value of dist[i][j]
                if (dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }

    // Print the shortest distance matrix
    printSolution(dist);
}

/* A utility function to print solution */
void printSolution(int dist[][V])
{
    printf ("Following matrix shows the shortest distances"
            " between every pair of vertices \n");
    for (int i = 0; i < V; i++)
    {
        for (int j = 0; j < V; j++)
        {
            if (dist[i][j] == INF)
                printf("%7s", "INF");
            else
                printf ("%7d", dist[i][j]);
        }
        printf("\n");
    }
}

// driver program to test above function
int main()
{
    /* Let us create the following weighted graph
            10
       (0)------->(3)
        |         /|\
      5 |          |
        |          | 1
       \|/         |
       (1)------->(2)
            3           */
    int graph[V][V] = { {0, 5, INF, 10},
                        {INF, 0,  3, INF},
                        {INF, INF, 0,  1},
                        {INF, INF, INF, 0}
                      };

    // Print the solution
    floydWarshall(graph);
    return 0;
}
```

6. math

6.1. divisibility

6.1.1. divisible11

```
#include <iostream>
#include <stdio.h>
#include <string>
//A number is multiple of 11 if the difference of sum of odd places and
    sum of even places is divisible by 11
```

```
using namespace std;

bool check11(string str){
  int n=str.length();
  int odd=0,even=0;
  for(int i=0;i<n;i++){
    if(i%2){
      even+=(str[i]-'0');
    }else{
      odd+=(str[i]-'0');
    }
  }
  return (odd-even)%11==0;
}

int main(){
    string str = "1024";
    check11(str)? cout << "Yes"<<endl : cout << "No "<<endl;
    return 0;
}
```

6.1.2. divisible3

```
#include <stdio.h>
#include <iostream>
#include <string>
//A number is divisible by 3 if the sum of all digits is divisble by 3

using namespace std;

bool check3(string str){
  int n=str.length();
  int sum=0;
  for(int i=0;i<n;i++){
    sum+=(str[i]-'0');
  }
  return (sum%3==0);
}

int main(){
    string str = "1332";
    check3(str)? cout << "Yes"<<endl : cout << "No "<<endl;
    return 0;
}
```

6.1.3. divisible4

```
#include <iostream>
#include <stdio.h>
#include <string>
//Divisible if last 2 numbers are divisible by 4
//Divisible if If the tens digit is even, the ones digit must be 0, 4, or
    8. If the tens digit is odd, the ones digit must be 2 or 6.
//Twice the tens digit, plus the ones digit is divisible by 4

using namespace std;

bool check4(string str){
  int n=str.length();
  if(n<2) return (str[0]-'0')%4==0;
  return (((str[n-2]-'0')*10)+(str[n-1]-'0'))%4==0;
}

int main(){
    string str = "9";
    check4(str)? cout << "Yes"<<endl : cout << "No "<<endl;
    return 0;
}
```

6.1.4. divisible7

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

using namespace std;

bool check7(string){

}

int main(){
    string str = "9";
```

```
    check7(str)? cout << "Yes"<<endl : cout << "No "<<endl;
    return 0;
}
```

7. primes

7.1. KnownNumber

```
#include <iostream>
using namespace std;
/*
        Eratosthenes' sieve that collects primes on an array
        use when you need a O(n) list of ordered primes and
        you know how many there will be

        Pros:
                - Faster because it doesn't rellocate
        Cons:
                - You have to know the number of primes in range

        Parameters:
                -n: the number to which you want to generate primes
                -a: an array of the correct size to store the resulting
                    primes
 */
#include <string.h>
typedef unsigned long long ll;

void genPrimes(ll n, ll a[]) {
    bool prime[n + 1];
    memset(prime, true, sizeof(prime));

    for (ll p = 2; p * p <= n; p++) {
        if (prime[p]) {
            for (ll i = p * p; i <= n; i += p) {
                prime[i] = false;
            }
        }
    }
    ll c = 0;
    for (ll p = 2; p <= n; p++) {
        if (prime[p]) {
            a[c++] = p;
        }
    }
}

int main() {
    ll a[6];
    genPrimes(13, a);
    for (auto p : a) {
        cout << p << '\n';
    }
}
```

7.2. SmallestPrimeFactor

```
#include <iostream>
using namespace std;
/*
        Eratosthenes' sieve that stores in an array from 0 to n the
        smallest prime factor of the number in its index

        Parameters:
                -n: the number to which you want to generate SPF
                -spf: an array of n+1 elements to store the factors
 */
#include <string.h>
typedef unsigned long long ll;

void getSPF(ll n, ll spf[]) {
    spf[0] = 0; // for consistency and easier debug
    for (ll i = 1; i <= n; i++) {
        spf[i] = (i & 1) ? i : 2;
    }

    for (ll p = 3; p * p <= n; p++) {
        if (spf[p] == p) {
            for (ll i = p * p; i <= n; i += p) {
                if (spf[i] == i) {
                    spf[i] = p;
                }
            }
        }
    }
```

```
}

int main() {
    ll a[14];
    getSPF(13, a);
    for (auto p : a) {
        cout << p << '\n';
    }
}
```

7.3. UnknownNumber

```
#include <iostream>
/*
       Eratosthenes' sieve that collects primes on a vector
       use when you need a O(n) list of ordered primes but
       you don't know how many primes will result from the sieve

       Pros:
              - Don't need to know number of primes
       Cons:
              - Multiple relocations will probably make it slower

       Parameters:
              -n: the number to which you want to generate primes
              -v: a vector where you want to collect them
 */
#include <string.h>
#include <vector>
typedef unsigned long long ll;
using namespace std;

void genPrimes(ll n, vector<ll>& v) {
    bool prime[n + 1];
    memset(prime, true, sizeof(prime));

    for (ll p = 2; p * p <= n; p++) {
        if (prime[p]) {
            for (ll i = p * p; i <= n; i += p) {
                prime[i] = false;
            }
        }
    }
    for (ll p = 2; p <= n; p++) {
        if (prime[p]) {
            v.push_back(p);
        }
    }
}

int main() {
    vector<ll> v;
    genPrimes(13, v);
    for (auto p : v) {
        cout << p << '\n';
    }
}
```

8. slidingWindowMinMax

```
// sliding window min and max O(n)

#include <queue>

int arr[]; // arreglo de valores
int n;     // tamao del arreglo
int w;     // tamao de la ventana

// G.front = max, S.front = min
deque<int> S(w), G(w);

// Process first window of size w
int i;
for (i = 0; i < w; i++) {
    while ((!S.empty()) && arr[S.back()] >= arr[i]) {
        S.pop_back();
    }

    while ((!G.empty()) && arr[G.back()] <= arr[i]) {
        G.pop_back();
    }

    G.push_back(i);
    S.push_back(i);
}
```

```
// max and min of first window
int max = arr[G.front()];
int min = arr[S.front()];

// Process rest of the Array elements
for (; i < n; i++) {
    while (!S.empty() && S.front() <= i - w) {
        S.pop_front();
    }
    while (!G.empty() && G.front() <= i - w) {
        G.pop_front();
    }

    while ((!S.empty()) && arr[S.back()] >= arr[i]) {
        S.pop_back();
    }
    while ((!G.empty()) && arr[G.back()] <= arr[i]) {
        G.pop_back();
    }

    G.push_back(i);
    S.push_back(i);

    int max = arr[G.front()];
    int min = arr[S.front()];
}
```