S. A.

#include <iostream>

#include <vector>

#include <random>

#include <algorithm>

//Se dau niste numere complexe, sa se sorteze numerele dupa parteaReala+parteImaginara

// 3+8i (3,8)

struct Complex

{

        int parteReala;

        int parteImaginara;

        bool operator<(Complex obj2)

        {

                return (parteReala + parteImaginara < obj2.parteReala + obj2.parteImaginara);

        }

        void writeComplex()

        {

                std::cout << parteReala << " " << parteImaginara << " i" << std::endl;

        }

        //Complex adunare(Complex obj);

};

void readComplex(Complex &obj)

{

        std::cin >> obj.parteReala >> obj.parteImaginara;

}

void egalComplex(Complex &obj1, Complex obj2) //!! obj2 se poate face const&

{

        obj1.parteReala = obj2.parteReala;

        obj1.parteImaginara = obj2.parteImaginara;

}

bool compareComplex(Complex obj1, Complex obj2)

{

        return (obj1.parteReala + obj1.parteImaginara < obj2.parteReala + obj2.parteImaginara);

}

void QuickSort(Complex arr[], int left, int right)

{

        int i = left, j = right;

        Complex pivot;

        egalComplex(pivot, arr[(left + right) / 2]);

        /\*Partitionare \*/

        while (i <= j) {

                while (compareComplex(arr[i],pivot)) //comparare complex

                        i++;

                while (compareComplex(pivot,arr[j]))

                        j--;

                if (i <= j) {

                        Complex tmp;

                        egalComplex(tmp,arr[i]);

                        egalComplex(arr[i], arr[j]);

                        egalComplex(arr[j], tmp);

                        i++;

                        j--;

                }

        }

        /\* Recursivitate \*/

        if (left < j)

                QuickSort(arr, left, j);

        if (i < right)

                QuickSort(arr, i, right);

}

void printArray(int arr[], int n)

{

        for (int i = 0; i < n; i++)

                std::cout << arr[i] << " ";

        std::cout << "\n";

}

int main()

{

        int n;

        std::cin >> n;

        Complex array[100];

        for (int index = 0; index < n; ++index)

                readComplex(array[index]);

        for (int index = 0; index < n; ++index)

                array[index].writeComplex();

        //printArray(array, n);

        QuickSort(array, 0, n - 1);

        for (int index = 0; index < n; ++index)

                array[index].writeComplex();

        //printArray(array, n);

        //std::sort(array, array + n, compareComplex);

        std::sort(array, array + n);

        system("pause");

        return(0);

}

1. a[i]>a[i-1] && a[i]>a[i+1] -> gasit

* a[i]>a[i-1] && a[i]<a[i+1] -> caut dreapta
* a[i]<a[i-1] && a[i]>a[i+1] -> caut stanga
* +caz particular primul si ultimul element
* #include <iostream>
* using namespace std;
* void Citire(int& dim, int tablou[])
* {
* cin >> dim;
* for (int i = 0; i < dim; i++)
* {
* cin >> tablou[i];
* }
* }
* int Binar(int tablou[], int minim, int maxim)
* {
* if (minim > maxim)
* {
* return -1;
* }
* else
* {
* int mijloc = (minim + maxim) / 2;
* if (tablou[mijloc] > tablou[mijloc - 1] && tablou[mijloc] > tablou[mijloc + 1])
* {
* return mijloc;
* }
* if (tablou[mijloc-1] > tablou[mijloc])
* {
* return Binar(tablou, minim, mijloc - 1);
* }
* if(tablou[mijloc+1]>tablou[mijloc])
* {
* return Binar(tablou, mijloc + 1, maxim);
* }
* }
* }
* int main()
* {
* int dim, tablou[100];
* Citire(dim, tablou);
* cout << tablou[Binar(tablou, 0, dim - 1)];
* }
* // 2
* #include <iostream>
* void ReadArray(int matrix[][100], int size)
* {
* for (int row = 0; row < size; row++)
* {
* for (int col = 0; col < size; col++)
* {
* std::cin >> matrix[row][col];
* }
* }
* }
* int BinarySearch(int matrix[][100], int size, int left, int right, int element)
* {
* // 1 2
* // 3 4
* // 1 2 3 4
* if (left <= right)
* {
* int midArray = (left + right) / 2;
* int midElement = matrix[midArray / size][midArray % size];
* if (element == midElement)
* {
* return midArray;
* }
* if (element < midElement)
* {
* return BinarySearch(matrix, size, left, midArray - 1, element);
* }
* return BinarySearch(matrix, size, midArray + 1, right, element);
* }
* return -1;
* }
* int main()
* {
* int size, element;
* std::cout << "n=";
* std::cin >> size;
* std::cout << "element:";
* std::cin >> element;
* int matrix[100][100];
* ReadArray(matrix, size);
* int unidimIndex = BinarySearch(matrix, size, 0, size \* size - 1, element);
* std::cout << unidimIndex / size << ", " << unidimIndex % size;
* }

3. sort cu cautare binara per fiecare elemnet 0 -> n-1

Varianta 2: Cand citesc un numar 'nr', atribuiesc vectorului a[nr] = nr, astfel incat vectorul va fi sortat direct din citire

4. verificam daca elementul median are propr cerura, daca da, returnam

daca nu, daca elementul median este mai mare ca indicele sau, atunci cautam pe ramura dreapta.

daca nu, daca elementul median este mai mic decat indicele sau, atunci cautare pe ramura stanga