# Министерство образования и науки Российской Федерации

## Федеральное государственное бюджетное образовательное учреждение высшего образования

## «Новосибирский государственный технический университет»

NSTU_Logo_blue

## Кафедра параллельных вычислительных технологий

### Домашнее задание № 1 по дисциплине «Методы построения и анализа алгоритмов»

**Сортировки**

|  |  |  |
| --- | --- | --- |
|  | Факультет: | ФПМИ |
| Группа: | ПМИ-82 |
| Вариант: | 3 |
| Студент: | --- |
|  |  |
|  |  |
| Преподаватель: | Щукин Георгий Анатольевич |

Новосибирск

2019

1. **Условие задачи**

Требуется реализовать сортировку или поиск с использованием алгоритма/структуры данных, указанного в варианте. В качестве входных данных использовать множество записей о людях, каждая запись содержит такие обязательные поля, как имя, фамилия и год рождения. Входные данные генерировать самостоятельно (случайным образом, загружать из файлов). Алгоритм или структура данных должны поддерживать работу с произвольным числом записей.

*Вариант 3* : Пирамидальная сортировка.

1. **Описание алгоритма**

Пирамидальная сортировка:

В этом методе исходное множество элементов таблицы рассматривается как бинарное дерево, вершины которого располагаются в массиве следующим образом: если k – индекс вершины дерева, то ее потомки находятся в элементах массива с индексами 2k + 1 и 2(k + 1); корень дерева находится в элементе с индексом 0. Двоичное дерево называется пирамидой (максимизирующим деревом или минимизирующим деревом), если в каждой его вершине находится элемент, ключ которого больше или равен (меньше или равен) ключам его непосредственных потомков. Таким образом, элемент с максимальным значением ключа находится в корне дерева. Этот элемент в алгоритме сортировки выбором и является нужным, его обмениваем с элементом, занимающим не свое место в упорядоченной части множества – первый раз с последним элементом множества, затем с предпоследним и т. д. После обмена дерево перестает быть пирамидой. Для того чтобы восстановить пирамиду достаточно просмотреть не все элементы, а только то поддерево, в котором обнаружен очередной нужный (максимальный) элемент. Таким образом, пирамидальная сортировка состоит из двух частей:

– построение пирамиды из исходного множества;

– собственно сортировка выбором, при которой приходится восстанавливать пирамиду.

Пирамида (table \*a, int i, int j)

{ max =i;

do { t=max; lt = 2\*t + 1; rt = 2\*(t + 1);

if (lt <= j) if (ключ(a->elem [lt]) > ключ(a->elem [t])) max = lt;

if (rt <= j) if(ключ(a->elem [rt]) > ключ(a->elem [max])) max = rt;

x = a->elem [t];

a->elem [t] = a->elem [max]; a->elem [max] = x;

} while (t != max);

}

{ for (i = (T.n -1)/2; i>=0; i--)

Пирамида ( &T, i, T.n -1);

for( i =T. n -1; i>=1; i--)

{

x =T[0].elem;

T[0].elem = T[i].elem i;

T[i].elem = x;

Пирамида ( &T, 0, i - 1); } }

1. **Текст программы**

**“Sort.h”**

#include <stdio.h>

#include <stdlib.h>

#include <iostream>

#include <string>

#include <vector>

#include <fstream>

#include <ctime>

#include <algorithm>

using namespace std;

struct peoples {

string name, surname;

int DOB;

};

// Функция "просеивания" через кучу - формирование кучи для DOT

vector <peoples> siftDown\_DOT(vector <peoples> numbers, int root, int bottom)

{

int maxChild; // индекс максимального потомка

int done = 0; // флаг того, что куча сформирована

// Пока не дошли до последнего ряда

while ((root \* 2 <= bottom) && (!done))

{

if (root \* 2 == bottom) // если мы в последнем ряду,

maxChild = root \* 2; // запоминаем левый потомок

// иначе запоминаем больший потомок из двух

else if (numbers[root \* 2].DOB > numbers[root \* 2 + 1].DOB)

maxChild = root \* 2;

else

maxChild = root \* 2 + 1;

// если элемент вершины меньше максимального потомка

if (numbers[root].DOB < numbers[maxChild].DOB)

{

peoples a = numbers[root]; // меняем их местами

numbers[root] = numbers[maxChild];

numbers[maxChild] = a;

root = maxChild;

}

else // иначе

done = 1; // пирамида сформирована

}

return numbers;

}

// Функция сортировки на куче для DOT

vector <peoples> heapSort\_DOT(vector <peoples> numbers, int array\_size)

{

// Формируем нижний ряд пирамиды

for (int i = (array\_size / 2) - 1; i >= 0; i--)

numbers=siftDown\_DOT(numbers, i, array\_size - 1);

// Просеиваем через пирамиду остальные элементы

for (int i = array\_size - 1; i >= 1; i--)

{

peoples a = numbers[0];

numbers[0] = numbers[i];

numbers[i] = a;

numbers=siftDown\_DOT(numbers, 0, i - 1);

}

return numbers;

}

/\*

//cортировка фамилии

void siftDown\_SURNAME(peoples\* numbers, int root, int bottom)

{

int maxChild; // индекс максимального потомка

int done = 0; // флаг того, что куча сформирована

// Пока не дошли до последнего ряда

while ((root \* 2 <= bottom) && (!done))

{

if (root \* 2 == bottom) // если мы в последнем ряду,

maxChild = root \* 2; // запоминаем левый потомок

// иначе запоминаем больший потомок из двух

else if (numbers[root \* 2].surname > numbers[root \* 2 + 1].surname)

maxChild = root \* 2;

else

maxChild = root \* 2 + 1;

// если элемент вершины меньше максимального потомка

if (numbers[root].surname < numbers[maxChild].surname)

{

peoples a = numbers[root]; // меняем их местами

numbers[root] = numbers[maxChild];

numbers[maxChild] = a;

root = maxChild;

}

else // иначе

done = 1; // пирамида сформирована

}

}

void heapSort\_SURNAME(peoples\* numbers, int array\_size)

{

// Формируем нижний ряд пирамиды

for (int i = (array\_size / 2) - 1; i >= 0; i--)

siftDown\_SURNAME(numbers, i, array\_size - 1);

// Просеиваем через пирамиду остальные элементы

for (int i = array\_size - 1; i >= 1; i--)

{

peoples a = numbers[0];

numbers[0] = numbers[i];

numbers[i] = a;

siftDown\_SURNAME(numbers, 0, i - 1);

}

}

// сортировка по имени

void siftDown\_NAME(peoples\* numbers, int root, int bottom)

{

int maxChild; // индекс максимального потомка

int done = 0; // флаг того, что куча сформирована

// Пока не дошли до последнего ряда

while ((root \* 2 <= bottom) && (!done))

{

if (root \* 2 == bottom) // если мы в последнем ряду,

maxChild = root \* 2; // запоминаем левый потомок

// иначе запоминаем больший потомок из двух

else if (numbers[root \* 2].name > numbers[root \* 2 + 1].name)

maxChild = root \* 2;

else

maxChild = root \* 2 + 1;

// если элемент вершины меньше максимального потомка

if (numbers[root].name < numbers[maxChild].name)

{

peoples a = numbers[root]; // меняем их местами

numbers[root] = numbers[maxChild];

numbers[maxChild] = a;

root = maxChild;

}

else // иначе

done = 1; // пирамида сформирована

}

}

void heapSort\_NAME(peoples\* numbers, int array\_size)

{

// Формируем нижний ряд пирамиды

for (int i = (array\_size / 2) - 1; i >= 0; i--)

siftDown\_NAME(numbers, i, array\_size - 1);

// Просеиваем через пирамиду остальные элементы

for (int i = array\_size - 1; i >= 1; i--)

{

peoples a = numbers[0];

numbers[0] = numbers[i];

numbers[i] = a;

siftDown\_NAME(numbers, 0, i - 1);

}

}

\*/

int get\_length(string s)

{

int length = 0;

string str;

ifstream fin;

fin.open(s);

while (!fin.eof())

{

getline(fin, str);

length++;

}

fin.close();

return length;

};

//чтение из массива

vector <peoples> fread(vector <peoples> arr)

{

int i = 0, m;

string str;

ifstream fin;

fin.open("Text.txt");

while (!fin.eof())

{

fin >> str;

arr[i].name = str;

fin >> str;

arr[i].surname = str;

fin >> m;

arr[i].DOB = m;

//cout << arr[i].name << endl << arr[i].surname << endl << arr[i].DOB << endl;

i++;

}

fin.close();

return arr;

}

// запись отсортированного в новый текстовый файл

void frecord(vector <peoples> arr, int length, string s)

{

ofstream fout;

fout.open(s);

for (int i = 0; i < length; i++)

{

fout << arr[i].name << " " << arr[i].surname << " " << arr[i].DOB << endl;

}

fout.close();

}

**“Sort\_only\_names.h”**

#pragma once

// сортировка по имени

void siftDown\_only\_NAME(vector <peoples> &numbers, int root, int bottom)

{

int maxChild; // индекс максимального потомка

int done = 0; // флаг того, что куча сформирована

// Пока не дошли до последнего ряда

while ((root \* 2 <= bottom) && (!done))

{

if (root \* 2 == bottom) // если мы в последнем ряду,

maxChild = root \* 2; // запоминаем левый потомок

// иначе запоминаем больший потомок из двух

else if (numbers[root \* 2].name > numbers[root \* 2 + 1].name)

maxChild = root \* 2;

else

maxChild = root \* 2 + 1;

// если элемент вершины меньше максимального потомка

if (numbers[root].name < numbers[maxChild].name)

{

string temp = numbers[root].name; // меняем их местами

numbers[root].name = numbers[maxChild].name;

numbers[maxChild].name = temp;

root = maxChild;

}

else // иначе

done = 1; // пирамида сформирована

}

}

void heapSort\_only\_NAME(vector <peoples> &numbers, int array\_size)

{

// Формируем нижний ряд пирамиды

for (int i = (array\_size / 2) - 1; i >= 0; i--)

siftDown\_only\_NAME(numbers, i, array\_size - 1);

// Просеиваем через пирамиду остальные элементы

for (int i = array\_size - 1; i >= 1; i--)

{

string temp = numbers[0].name;

numbers[0].name = numbers[i].name;

numbers[i].name = temp;

siftDown\_only\_NAME(numbers, 0, i - 1);

}

}

void fread\_name(vector <peoples> &arr,string s)

{

int i = 0, m;

string str;

ifstream fin;

fin.open(s);

while (!fin.eof())

{

fin >> str;

arr[i].name = str;

i++;

}

fin.close();

}

void frecord\_names(vector <peoples> arr, int length, string s)

{

ofstream fout;

fout.open(s);

for (int i = 0; i < length; i++)

{

fout << arr[i].name << endl;

}

fout.close();

}

**“Source.cpp”**

#include "Sort.h" // библеотека с функциями сортировки

#include "Sort\_only\_names.h" // только для имен

#include <chrono>

int main()

{

srand(time(NULL));

setlocale(LC\_ALL, "ru");

int length = get\_length("Text.txt");

vector <peoples> v(length);

v=fread(v);

v = heapSort\_DOT(v, length);

frecord(v,length, "Sort\_DOT.txt");

/////////////////////////////////////////////////////////////////

length = get\_length("Names.txt");

vector <peoples> v2(length);

vector <string> v3(length);

fread\_name(v2, "Names.txt");

auto t1 = chrono::high\_resolution\_clock::now();

heapSort\_only\_NAME(v2, length);

auto t2 = chrono::high\_resolution\_clock::now();

double time = chrono::duration<double>(t2 - t1).count();

for (int i = 0; i < length; i++)

{

v3[i] = v2[i].name;

}

auto t3 = chrono::high\_resolution\_clock::now();

sort(v3.begin(), v3.end());

auto t4 = chrono::high\_resolution\_clock::now();

double time2 = chrono::duration<double>(t4 - t3).count();

cout << "время выполнения сортировки массива размером :" << length << " = " << time<<endl;

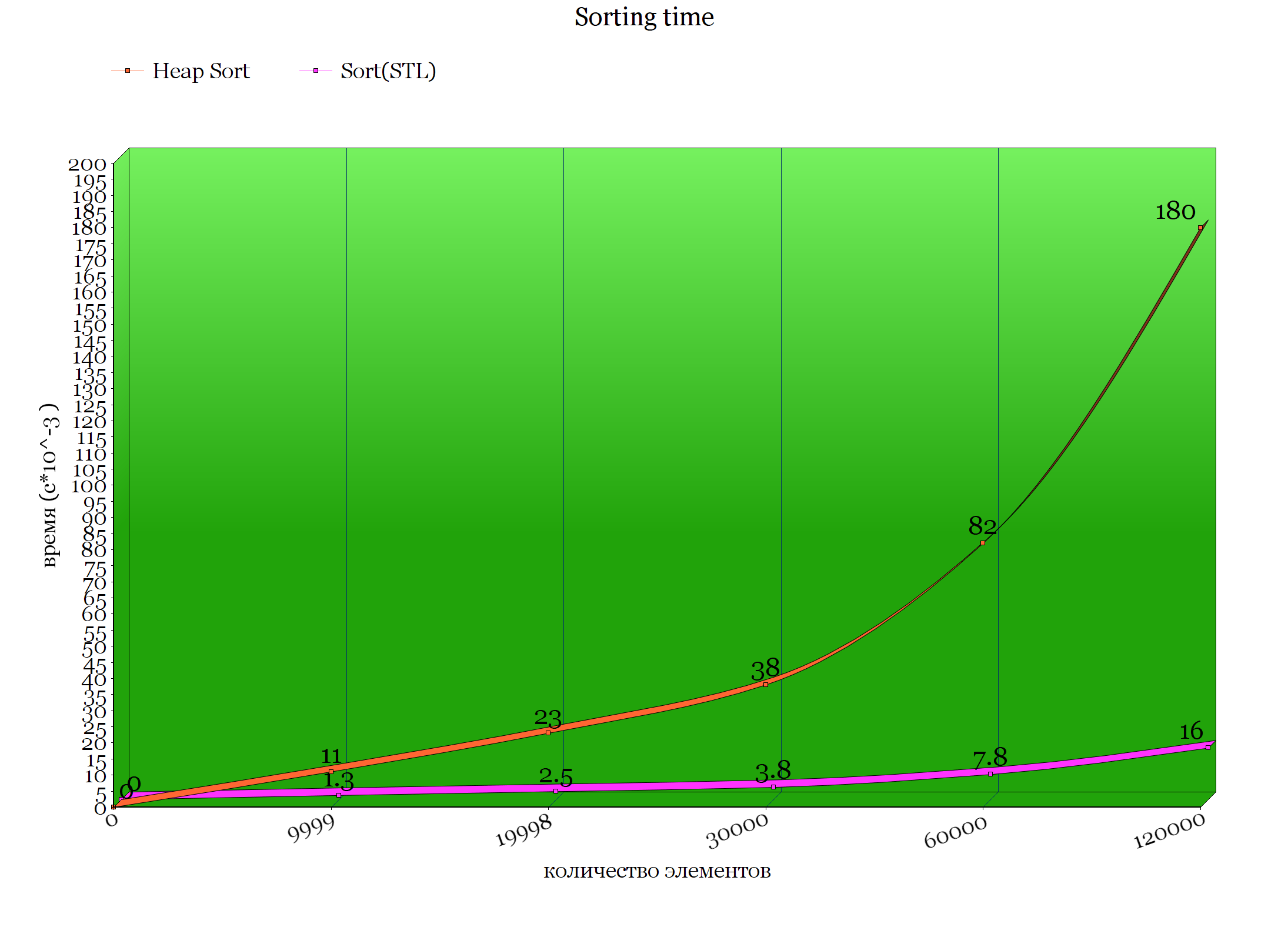
cout << "время STL сортировки того же массива = " << time2 << endl;

frecord\_names(v2, length,"Sort\_only\_names.txt");

system("pause");

}

**3.График времени**



**4. Юнит-тесты**

#define CATCH\_CONFIG\_MAIN // This tells Catch to provide a main() - only do this in one cpp file

#include "catch.hpp"

#include "Sort.h"

bool are\_equal(const std::vector<peoples> &a, const std::vector<peoples> &b) {

int temp = 0;

for (int i = 0; i < a.size(); i++)

{

if (a[i].DOB == b[i].DOB && a[i].name == b[i].name && a[i].surname == a[i].surname)

temp++;

}

if (temp == a.size())

return true;

else

return false;

}

TEST\_CASE( "Empty array", "[count\_sort]" ) {

vector <peoples> v;

REQUIRE( are\_equal(heapSort\_DOT(v), v) );

}

TEST\_CASE( "One element array", "[count\_sort]" ) {

vector <peoples> v(1);

v[0].name = "Sergey";

v[0].surname = "Sazonov";

v[0].DOB = 2000;

REQUIRE( are\_equal(heapSort\_DOT(v), v ) );

}

TEST\_CASE("Unsorted array", "[count\_sort]") {

vector <peoples> v(6),v2(6);

v=fread(v,"Unsorted.txt");

v2 = fread(v2,"Sorted.txt");

REQUIRE(are\_equal(heapSort\_DOT(v), v2));

}

TEST\_CASE("Sorted array", "[count\_sort]") {

vector <peoples> v(6), v2(6);

v = fread(v, "Sorted.txt");

v2 = fread(v2, "Sorted.txt");

REQUIRE(are\_equal(heapSort\_DOT(v), v2));

}

TEST\_CASE("Sorted array with duplicates", "[count\_sort]") {

vector <peoples> v(12), v2(12);

v = fread(v, "Sorted\_duplicates.txt");

v2 = fread(v2, "Sorted\_duplicates.txt");

REQUIRE(are\_equal(heapSort\_DOT(v), v2));

}

TEST\_CASE("Unsorted array with duplicates", "[count\_sort]") {

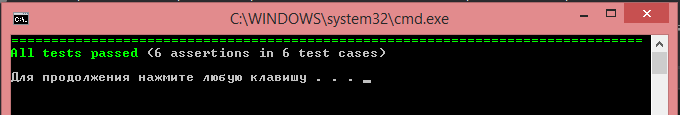
vector <peoples> v(12), v2(12);

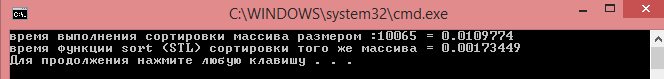
v = fread(v, "Unsorted\_duplicates.txt");

v2 = fread(v2, "Sorted\_duplicates.txt");

REQUIRE(are\_equal(heapSort\_DOT(v), v2));

}

****

****