МИНОБРНАУКИ РОССИИ

Санкт-Петербургский государственный

электротехнический университет

«ЛЭТИ» им. В.И. Ульянова (Ленина)

Кафедра «Информационные системы»

**отчет**

по практической работе №2

по дисциплине «Программирование»

Тема: "Одномерные статистические массивы"

|  |  |  |
| --- | --- | --- |
| Студентка гр. 0323 |  | Рудницкая Т.С. |
| Преподаватель |  | Глущенко А.Г. |

Санкт-Петербург

2020

Цель работы.

Изучение структуры одномерных массивов, обработка данных одномерных массивов. Изучение различных видов сортировок. Проведение временной оценки различных действий с массивами.

Основные теоретические положения.

При использовании простых переменных каждой области памяти для хранения данных соответствует свое имя. Если с группой величин одинакового типа требуется выполнить однообразные действия, им дают одно имя, а различают по порядковому номеру (индексу). Это дает возможность компактно записать множество операций с использованием циклов. Массив представляет собой индексированную последовательность однотипных элементов с заранее определенным количеством элементов. Наглядно одномерный массив можно представить, как набор пронумерованных ячеек, в каждой из которых содержится определенное значение. Все массивы можно разделить на две группы: одномерные и многомерные. Описание массива в программе отличается от объявления обычной переменной наличием размерности массива, которая задается в квадратных скобках после имени.

Элементы массива нумеруются с нуля. При описании массива используются те же модификаторы (класс памяти, const и инициализатор), что и для простых переменных. Инициализировать массив можно и другим более простым способом: инициализирующие значения записываются в фигурных скобках. Значения элементам присваиваются по порядку. Если элементов в массиве больше, чем инициализаторов, элементы, для которых значения не указаны, обнуляются:

1. int arr[4] = {3, 2, 1}; // arr[0] = 3, arr[1] = 2, arr[2] = 1, arr[3] = 0

Размерность массива вместе с типом его элементов определяет объем памяти, необходимый для размещения массива, которое выполняется на этапе компиляции, 34 поэтому размерность должна быть задана целой положительной константой или константным выражением.

Если при описании массива не указана размерность, массив обязательно должен быть инициализирован. Компилятор сам определит размерность массива по количеству элементов.

Определить, сколько памяти выделено под массив, можно с помощью операции sizeof. Необходимо определить, сколько выделяется памяти на 1 элемент массива и умножить на количество элементов массива.

**Пузырьковая сортировка массива (bubble sort)**

Сортировка – процесс размещения элементов заданного множества объектов в определенном порядке. Когда элементы отсортированы, их проще найти, производить с ними различные операции. Сортировка напрямую влияет на скорость алгоритма, в котором нужно обратиться к определенному элементу массива. Простейшая из сортировок – сортировка обменом (пузырьковая сортировка). Вся суть метода заключается в сравнивании попарном сравнении элементов и последующем обмене. Таким образом, если следующий элемент меньше текущего, то они меняются местами, максимальный элемент массива постепенно смещается в конец массива, а минимальный – в начало. Один полный проход по массиву может гарантировать, что в конце массива находится максимальный элемент. Затем процесс повторяется до тех пор, пока вся последовательность не будет упорядочена. Важно заметить, что после первого прохода по массиву, уже имеется один упорядоченный элемент, он стоит на своем месте, и менять его не надо. Таким образом на следующем шаге будут сравниваться N-1 элемент.

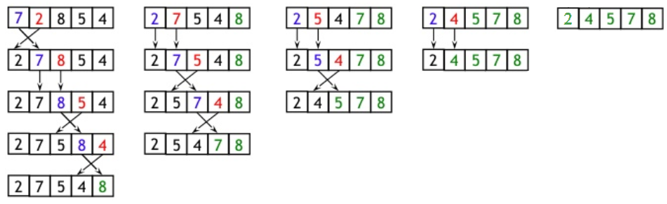


Рисунок 1 – пример работы пузырьковой сортировки

Очевидно, что хуже всего алгоритм будет работать, когда на вход подается массив, отсортированный в обратную сторону (от большего к меньшу). Быстрее же всего алгоритм работает с уже отсортированным массивом.

Но стандартный алгоритм пузырьковой сортировки предполагает полный циклический проход по массиву. Если изначально подается упорядоченная последовательность, то работа алгоритма все равно продолжиться. Исправить это можно, добавив условие проверки: если на текущей итерации ни один элемент не изменил свой индекс, то работа алгоритма прекращается.

**Шейкер-сортировка массива (shaker sort)**

Shaker sort – модификация пузырьковой сортировки. Принцип работы этой сортировки аналогичен bubble sort: попарное сравнение элементов и последующий обмен местами. Но имеется существенное отличие. Как только максимальный элемент становится на свое место, алгоритм не начинает новую итерацию с первого элемента, а запускает сортировку в обратную сторону. Алгоритм гарантирует, что после выполнения первой итерации, минимальный и максимальный элемент будут в начале и конце массива соответственно.

Затем процесс повторяется до тех пор, пока массив не будет отсортирован. За счет того, что сортировка работает в обе стороны, массив сортируется на порядок быстрее. Очевидным примером этого был бы случай, когда в начале массива стоит максимальный элемент, а в конце массива – минимальный. Shaker sort справится с этим за 1 итерацию, при условии, что другие элементы стоят на правильном месте.

Кажется, что bubble sort теряет свою эффективность по сравнению с shaker sort. Сортировка проходит в массиве в обоих направлениях, а не только от его начала к концу. Но в работе с большими массивами преимущество шейкер-сортировки уменьшается как раз из-за использования двух циклов.

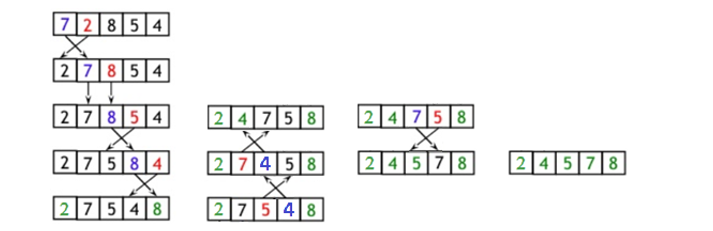


Рисунок 2 – пример работы Shaker sort

**Сортировка массива расчёской (comb sort)**

Очевидный недостаток bubble и shaker sort заключается в том, что элементы переставляются максимум на одну позицию.

Comb sort (сортировка расческой) – ещё одна модификация сортировки пузырьком. Алгоритм был разработан специально для случаев, когда минимальные элементы стоят слишком далеко, или максимальные – слишком близко к началу массива. В сортировке расческой переставляются элементы, стоящие на расстоянии.

Оптимально изначально взять расстояние равным длине массива , а далее уменьшать его на определенный коэффициент, который примерно равен 1.247. Когда расстояние станет равно 1, выполняется обычная сортировка пузырьком.

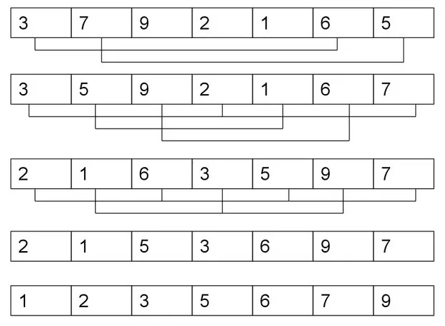


Рисунок 3 – пример работы Shaker sort

**Сортировка массива вставками (insert sort)**

Сортировка вставками (insert sort) – алгоритм сортировки, в котором элементы массива просматриваются по одному, и каждый новый элемент размещается в подходящее место среди ранее упорядоченных элементов.

Общая суть сортировки вставками такова:

1. Перебираются элементы в неотсортированной части массива.
2. Каждый элемент вставляется в отсортированную часть массива на то место, где он должен находится.

Сортировка вставками делит массив на 2 части – отсортированную и неотсортированную. С каждым новым элементом отсортированная часть будет увеличиваться, а неотсортированная уменьшаться. Причем найти нужное место для очередного элемента в отсортированном массиве достаточно легко.

Рассмотрим самый простой способ. Необходимо пройти массив слева направо и обработать каждый элемент. Слева будет наращиваться отсортированная часть массива, а справа – уменьшаться неотсортированная. В отсортированной части массива ищется точка вставки для очередного элемента. Сам элемент отправляется в буфер, что освобождает место в массиве и позволяет сдвинуть элементы и освободить точку вставки.

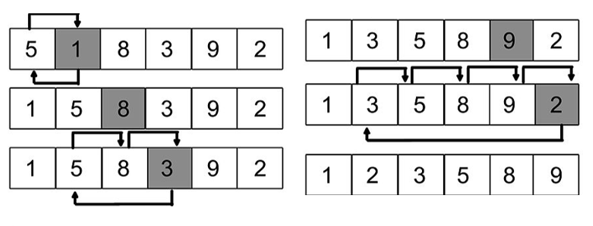


Рисунок 4 – пример работы insert sort

Существует множество модификаций сортировки вставками, некоторые из них затрагивают именно способ вставки элемента в отсортированную часть.

Лучше всего сортировка вставками работает при обработке почти отсортированных массивов. В таком случае insert sort работает быстрее других сортировок.

**Быстрая сортировка массива (quick sort)**

Быстрая сортировка (quick sort) – одна из самых быстрых сортировок. Эта сортировка по сути является существенно улучшенной версией алгоритма пузырьковой сортировки.

Общая идея алгоритма состоит в том, что сначала выбирается из массива элемент, который называется опорным. От выбора опорного элемента не зависит корректность алгоритма, но в отдельных случаях может сильно зависеть его эффективность. Затем необходимо сравнить все остальные элементы с опорным и переставить их в массиве так, чтобы разбить массив на три непрерывных отрезка, следующие друг за другом: меньше опорного, раны опорному и больше опорного. Для меньших и больших значений необходимо выполнить рекурсивно ту же последовательность операций, если длина отрезка больше единицы.

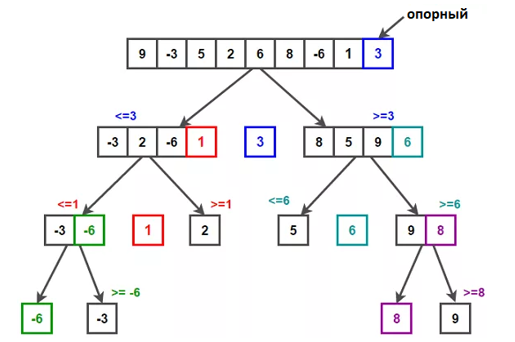


Рисунок 5 – пример работы quick sort

На практике массив обычно делят на две части: «меньше опорного» и «равные и большие» или «меньше опорного или равные» и «большие». Такой поход в общем случае эффективнее, ведь упрощается алгоритм разделения.

При том, что это один из самых быстродействующих из алгоритмов, данный алгоритм сортировки неустойчив, а прямая реализация в виде функции с двумя рекурсивными вызовами может привести к ошибке переполнения стека.

Постановка задачи.

Разработать алгоритм и написать программу, которая позволяет:

1. Создает целочисленный массив размерности *N* = 100. Элементы массивы должны принимать случайное значение в диапазоне от -99 до 99.
2. Отсортировать заданный в пункте 1 элементы массива сортировкой (от меньшего к большему). Определить время, затраченное на сортировку, используя библиотеку chrono.
3. Найти максимальный и минимальный элемент массива. Подсчитайте время поиска этих элементов в отсортированном массиве и неотсортированном, используя библиотеку chrono.
4. Выводит среднее значение (если необходимо, число нужно округлить) максимального и минимального значения. Выводит индексы всех элементов, которые равны этому значению, и их количество.
5. Выводит количество элементов в отсортированном массиве, которые меньше числа *a*, которое инициализируется пользователем.
6. Выводит количество элементов в отсортированном массиве, которые больше числа *b*, которое инициализируется пользователем.
7. Выводит информацию о том, есть ли введенное пользователем число в отсортированном массиве. Реализуйте алгоритм бинарного поиска. Сравните скорость его работы с обычным перебором. (\*)
8. Меняет местами элементы массива, индексы которых вводит пользователь. Выведите скорость обмена, используя библиотеку chrono.
9. Должна присутствовать возможность запуска каждого пункта многократно.

Выполнение работы.

Для упрощения работы с программой мной был разработан визуальный интерфейс на C++/CLI. Так как создание визуального интерфейса не входит в программу лабораторной работы, я не буду подробно останавливаться на этом, а код функций, используемых в работе программы, для простоты приводить на C++. Полный код, вместе с кодом функций, используемых для работы формы, таких как функции проверки введённых в форму значений, вывода массива, выделения элементов и др., размещён в приложении А.

Везде по ходу программы использовался одномерный статический массив.

1. Генерация массива.

По заданию нужен массив размерности N = 100, а его элементы должны принимать случайное значение в диапазоне от -99 до 99. Для дальнейшей работы с этим массивом я ввела несколько глобальных переменных:

1. //количество строк и столбцов для более удобного вывода массива
2. int cols = 10, rows = 10;
3. //количество элементов
4. //в случае, если надо будет переделать работу так
5. //чтобы размерность массива задавалась пользователем,
6. //нужно будет поменять только эти три переменные
7. const int arrayWidth = 100,
8. //диапазон значений массива
9. min = -99,
10. max = 99;
11. //флаг, показывающий что массив отсортирован
12. bool isSorted = false;
13. //массив, заданной ширины
14. int Arr[arrayWidth];

Для генерации псевдослучайных чисел я использовала библиотеку <ctime>. Проблема генерации случайных чисел состоит в отсутствии случайности алгоритмов генерации. Для того, чтобы внести элемент случайности традиционно используется опорный элемент, в качестве которого чаще всего принимают значение системного времени в текущий момент.

Функция getRandomNumber принимает в качестве аргументов минимальное и максимальное значения диапазона.

1. int getRandomNumber(int min, int max) {
2. return (rand() % (max - min + 1) + min);
3. }

Для задания диапазона я выбрала способ с остатком от деления случайного значения функции rand(). Остаток от деления задаст ширину диапазона, а прибавление минимального значения сдвинет диапазон от нуля на минимальное значение.

Функция генерации массива:

1. void generateArray(int arrayWidth, int min, int max, int \*Arr){
2. // устанавливаем опорное значение с помощью значения системного времени
3. srand(time(0));
4. for (int i = 0; i < arrayWidth; ++i)
5. Arr[i] = getRandomNumber(min, max);
6. }

После того, как массив сгенерирован, его нужно вывести. Функция showArray для вывода массива в консоль:

1. void showArray(int arrayWidth, int \*Arr) {
2. for (int i = 0; i < arrayWidth; i++)
3. {
4. std::cout << Arr[i] << "\t";
5. if ((i + 1) % rows == 0)
6. std::cout << "\n";
7. }
8. }

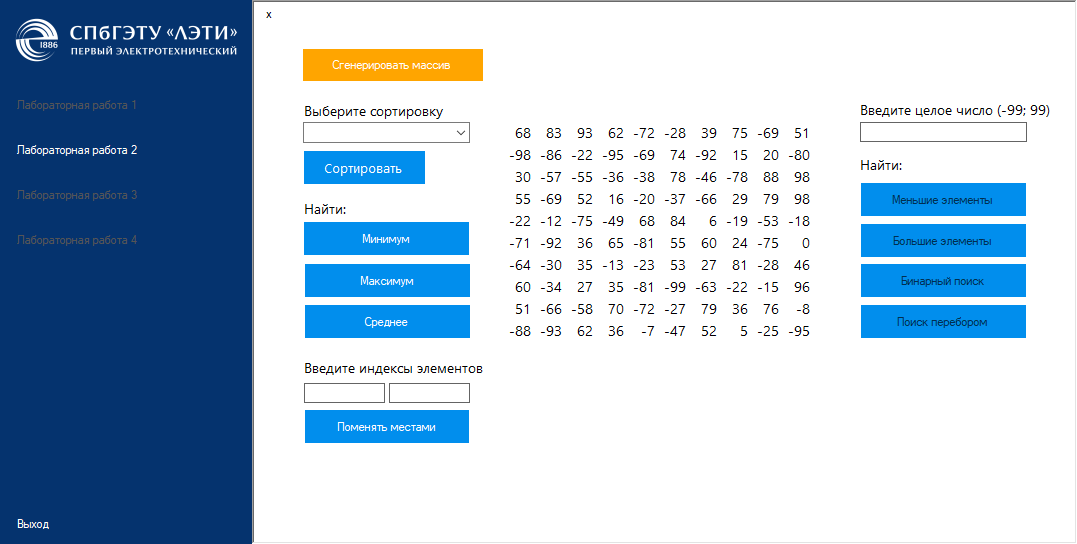


Рисунок 6 — пример вывода сгенерированного массива для работы

1. Сортировка элементов. В своей работе я предоставляю пользователю возможность выбрать любую и пяти вариантов сортировки: пузырьковая, сортировка «расчёской», комбинированная, сортировка вставками и быстрая сортировка. Сравнительные характеристики разных видов сортировок представлены в приложении Б.

Функция bubble sort с комментариями:

1. void bubbleSort(int arrayWidth, int\* Arr) {
2. for (int i = 0; i < arrayWidth; i++) {
3. // устанавливаем флаг, показывающий, что массив уже отсортирован
4. bool isSort = true;
5. for (int j = 0; j < arrayWidth - (i + 1); j++) {
6. if (Arr[j] > Arr[j + 1]) {
7. // если по дороге попадается неотсортированный элемент,
8. // присваиваем флагу значение false и меняем элементы местами
9. isSort = false;
10. std::swap(Arr[j], Arr[j + 1]);
11. }
12. }
13. // если за весь проход массива ни разу не попались неотсортированные элементы,
14. // то прерываем цикл
15. if (isSort) break;
16. }
17. }

Я использую модицифированную версию пузырьковой сортировки, где для ускорения работы сортировка прерывается, если за целый проход по массиву не было перемещено ни одного элемента.

Для замера времени выполнения функций я использую библиотеку <chrono>. Функция замера принимает в качестве коллбэка функцию, время исполнения которой нужно замерить. К сожалению, в C++ очень сложно сделать действительно универсальный коллбэк с любой сигнатурой и чтение дополнительной литературы мне пока не помогло разобраться. Поэтому в работе используются несколько идентичных функций, различающихся сигнатурой коллбэк функции. Пример одной из таких функций:

1. int benchSort(int arrayWidth, int\* Arr, void (\*funcP)(int arrayWidth, int\* Arr)) {
2. using Time = time\_point<steady\_clock>;
3. using Diff = microseconds;
4. // Запоминаем значение системного времени до начала выполнения функции
5. // используем монотонные часы steady\_clock
6. Time start = steady\_clock::now();
7. // вызываем коллбэк функцию
8. funcP(arrayWidth, Arr);
9. // Запоминаем значение системного времени после выполнения функции
10. Time end = steady\_clock::now();
11. // Определяем тип объекта интервала и вычисляем его значение
12. Diff diff = duration\_cast<Diff>(end - start);
13. // вычисляем количество тактов в интервале
14. // и возвращаем итог
15. return diff.count();
16. }

В ней я выставляю размерность для вывода результата (в данном случае микросекунды). Для замера использую монотонные часы steady\_clock. Замеряется системное время перед выполнением функции и после выполнения. Функция возвращает разницу между вторым и первым замером (количество тиков), сконвертированное в заданный формат.

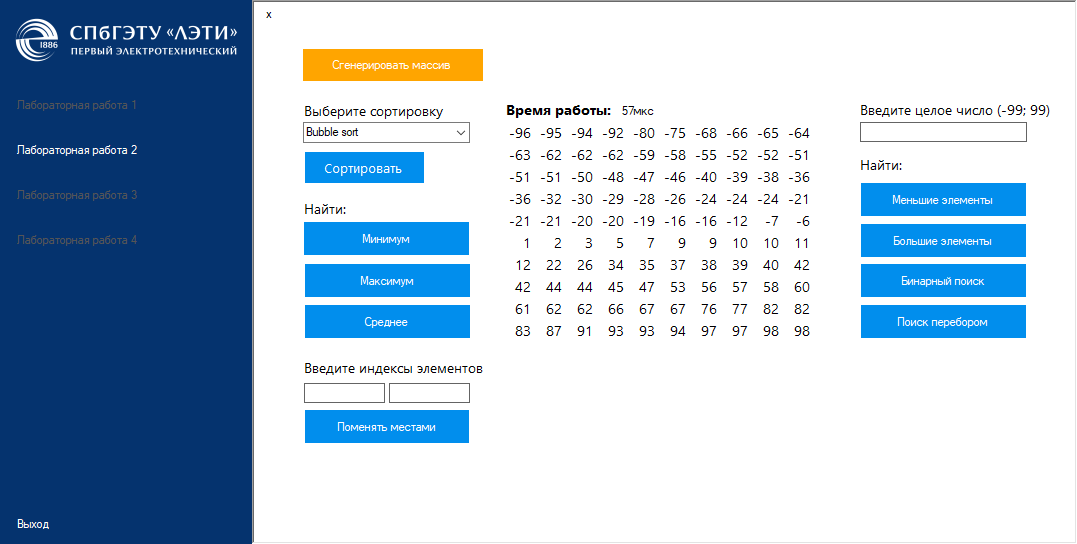


Рисунок 7 — пример работы пузырьковой сортировки с таймером.

Функция шейкер сортировки с комментариями:

1. void shakerSort(int arrayWidth, int\* Arr)
2. {
3. for (int i = 0; i < arrayWidth - i; i++) {
4. bool isSort = true;
5. // проход в из начала в конец
6. // "выталкивание" наибольшего элемента в конец
7. for (int j = 0; j < arrayWidth - (i + 1); j++) {
8. if (Arr[j] > Arr[j + 1]) {
9. isSort = false;
10. std::swap(Arr[j], Arr[j + 1]);
11. }
12. }
13. // проход в из конца в начало
14. // "выталкиваение" наименьшего элемента в начало
15. for (int j = arrayWidth - (i + 1); j >= i + 1; j--) {
16. if (Arr[j] < Arr[j - 1]) {
17. isSort = false;
18. std::swap(Arr[j], Arr[j - 1]);
19. }
20. }
21. if (isSort) break;
22. }
23. }

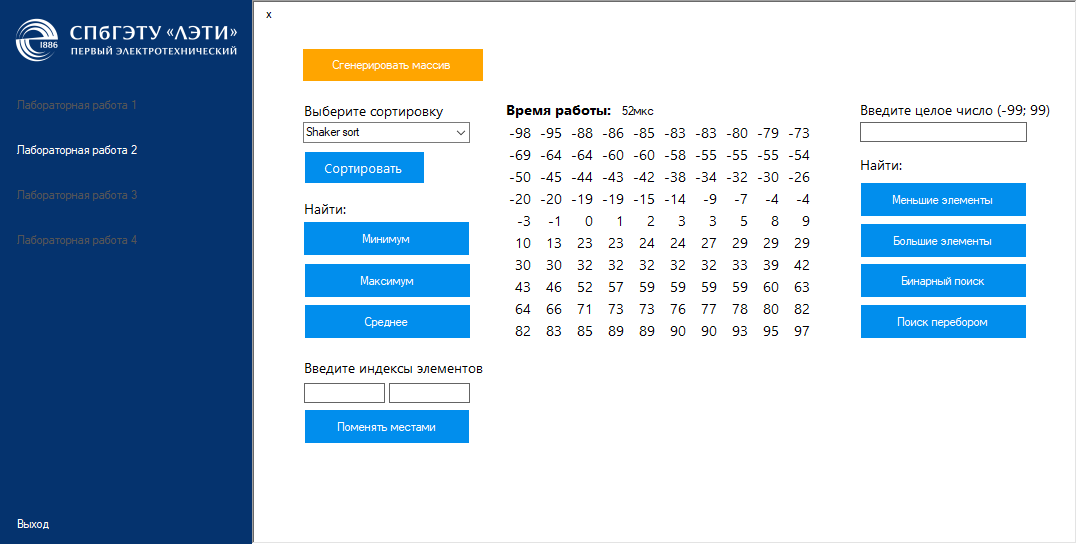


Рисунок 8 — пример работы шейкер сортировки с таймером.

Функция комбинированной сортировки с комментариями:

1. void combSort(int arrayWidth, int\* Arr){
2. // коэффициент, на который будет уменьшаться шаг "расчёски"
3. float k = 1.28f;
4. // ширина проверяемого диапазона
5. // на начальном этапе равна ширине массива
6. // после будет уменьшаться в k раз
7. int S = arrayWidth;
8. //пока диапазон не схлопнется до к элементов
9. while (k < S) {
10. //сравниваем элементы из начала и конца диапазона
11. //и сдвигаем "расчёску", пока не упрёмся правым концом в конец массива
12. for (int i = 0; i <= arrayWidth - S; i++) {
13. if (Arr[i] > Arr[i + S - 1]) {
14. std::swap(Arr[i], Arr[i + S - 1]);
15. }
16. }
17. //уменьшаем диапазон
18. S /= k;
19. }
20. //после того, как диапазон стал слишком маленьким
21. //остаток "причёсываем" пузырьком
22. bubbleSort(arrayWidth, Arr);
23. }

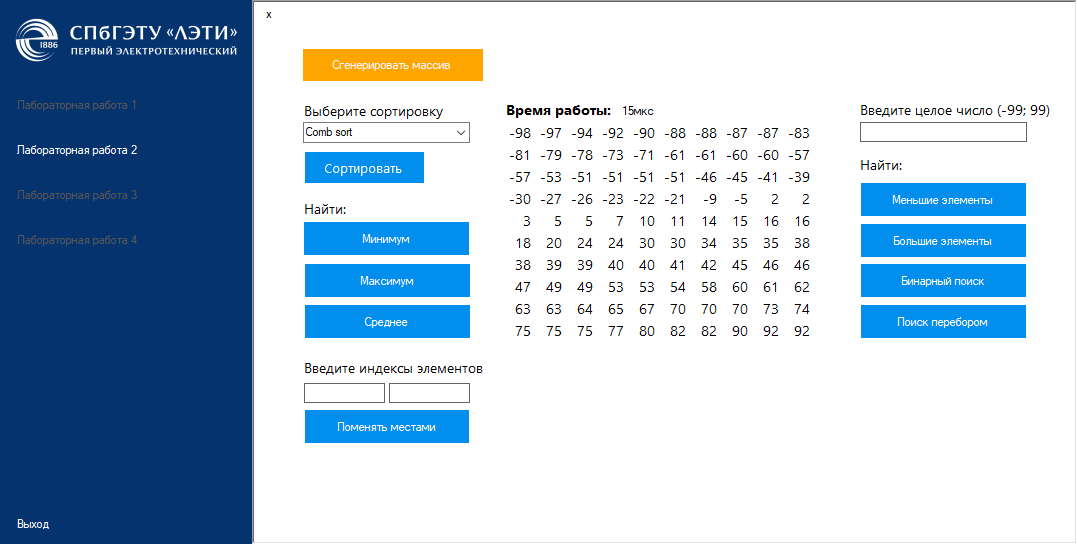


Рисунок 9 — пример работы комбинированной сортировки с таймером.

Функция сортировки вставками с комментариями:

1. void insertSort(int arrayWidth, int\* Arr){
2. for (int i = 1; i < arrayWidth; i++){
3. // установка элемента на своё место
4. // в отсортированной части массива
5. for (int j = i; j > 0 && Arr[j - 1] > Arr[j]; j--)
6. std::swap(Arr[j - 1], Arr[j]);
7. }
8. }

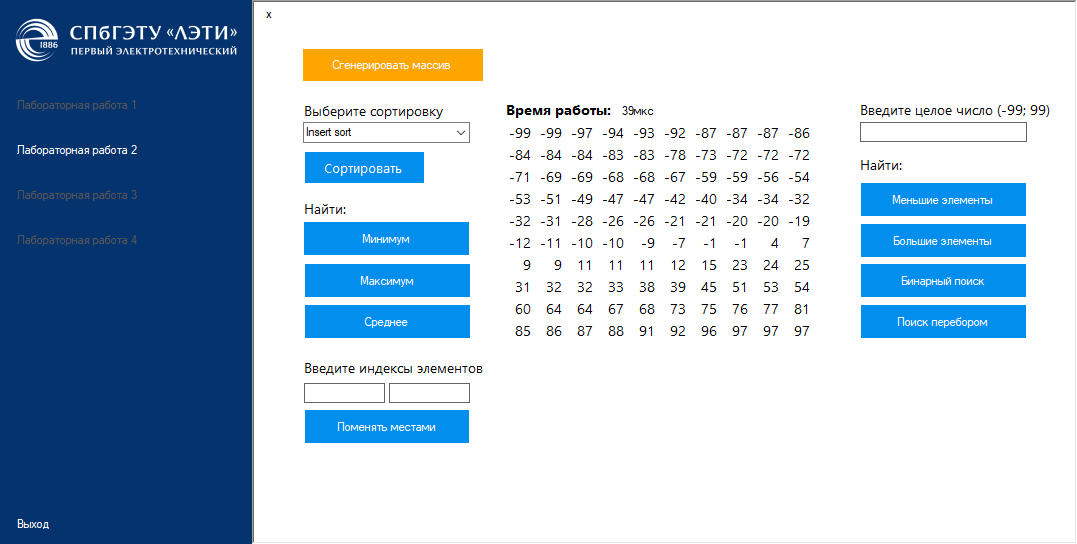
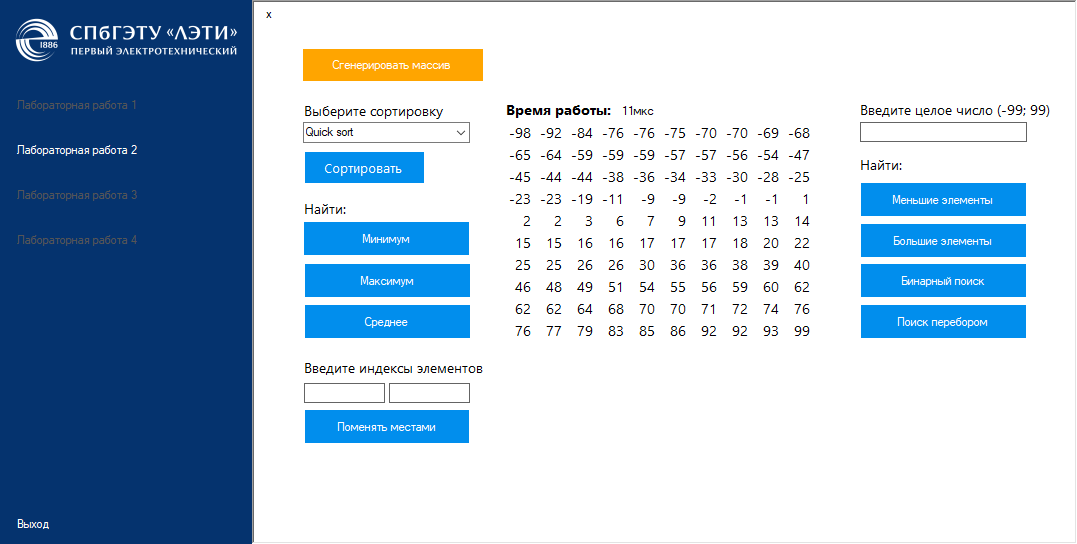


Рисунок 10 — пример работы сортировки вставками с таймером.

Функция быстрой сортировки с комментариями.

1. void quickSort(int arrayWidth, int\* Arr){
2. //устанавливаем левый (i) и правый (j) маркеры
3. int left = 0;
4. int right = arrayWidth - 1;
5. //находим опорный элемент в центре массива
6. int mid = Arr[arrayWidth / 2];
7. //продолжаем сводить маркеры, пока они не пересекутся
8. while (left <= right){
9. //В левой части массива пропускаем элементы, которые меньше опорного
10. while (Arr[left] < mid) {
11. left++;
12. }
13. //В правой части пропускаем элементы, которые больше опорного
14. while (Arr[right] > mid) {
15. right--;
16. }
17. //Меняем элементы местами
18. if (left <= right) {
19. std::swap(Arr[left], Arr[right]);
20. left++;
21. right--;
22. }
23. }
24. //рекурсивно вызываем функцию сортировки, если остались неотсортированные элементы
25. if (right > 0) {
26. //для левой половины от места встречи маркеров
27. quickSort(right + 1, Arr);
28. }
29. if (left < arrayWidth) {
30. //для правой половины от места встречи маркеров
31. //чтобы не вводить отдельные переменные,
32. //обращаемся к i-тому элементу массива по ссылке
33. quickSort(arrayWidth - left, &Arr[left]);
34. }
35. }

В случае, если левый или правый маркер достигнут опорного элемента быстрее другого, они встанут на место опорного, и будут при необходимости двигаться дальше, продолжая сортировать элементы. При рекурсивном вызове текущие положения маркеров задаются в качестве границ массивов для дальнейшей сортировки.

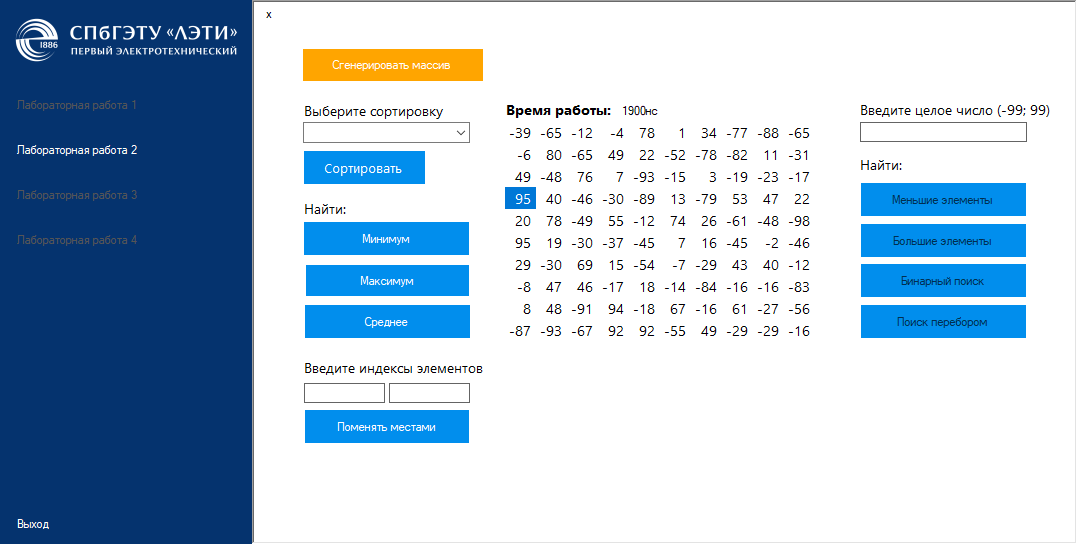
Рисунок 11 — пример работы быстрой сортировки с таймером.

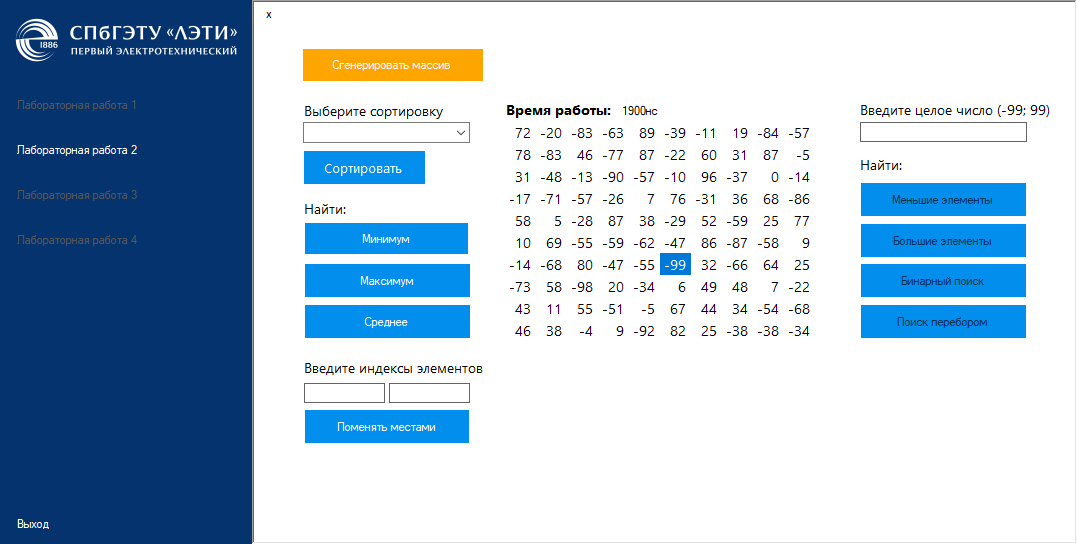
1. Поиск максимального и минимального элементов массива. Сравнение времени поиска в отсортированном и неотсортированном массивах.

Функции поиска максимального и минимального элементов:

1. int searchMax(int arrayWidth, int\* Arr) {
2. int max = Arr[0];
3. for (int i = 1; i < arrayWidth; i++)
4. if (Arr[i] > max)
5. max = Arr[i];
6. return max;
7. }
8. int searchMin(int arrayWidth, int\* Arr) {
9. int min = Arr[0];
10. for (int i = 1; i < arrayWidth; i++)
11. if (Arr[i] < min)
12. min = Arr[i];
13. return min;
14. }

Среднее время их работы в неотсортированном массиве — 1900нс. Среднее время работы функции поиска максимального элемента в отсортированном массиве доходит до 2400нс. Остаётся таким же и составляет порядка 1900нс.





Рисунки 12,13 — примеры поиска максимального и минимального элементов неотсортированного массива

1. Вывести среднее из максимального и минимального элементов. Вывести все элементы, которые равны этому значению, и их количество.

Для поиска среднего использую описанные выше функции поиска максимального и минимального значений.

Код функции поиска:

1. int searchMiddle(int arrayWidth, int\* Arr) {
2. //вызов функции поиска минимального элемента
3. int minEl = Arr[searchMin(arrayWidth, Arr)];
4. //вызов функции поиска максимального элемента
5. int maxEl = Arr[searchMax(arrayWidth, Arr)];
6. int middleEl = round((minEl + maxEl) / 2);
7. return middleEl;
8. }

Для округления использую метод round() библиотеки <cmath>. Этот метод округления по математическим правилам. В случае, если ни одного элемента с таким значением не найдено, выводится сообщение.

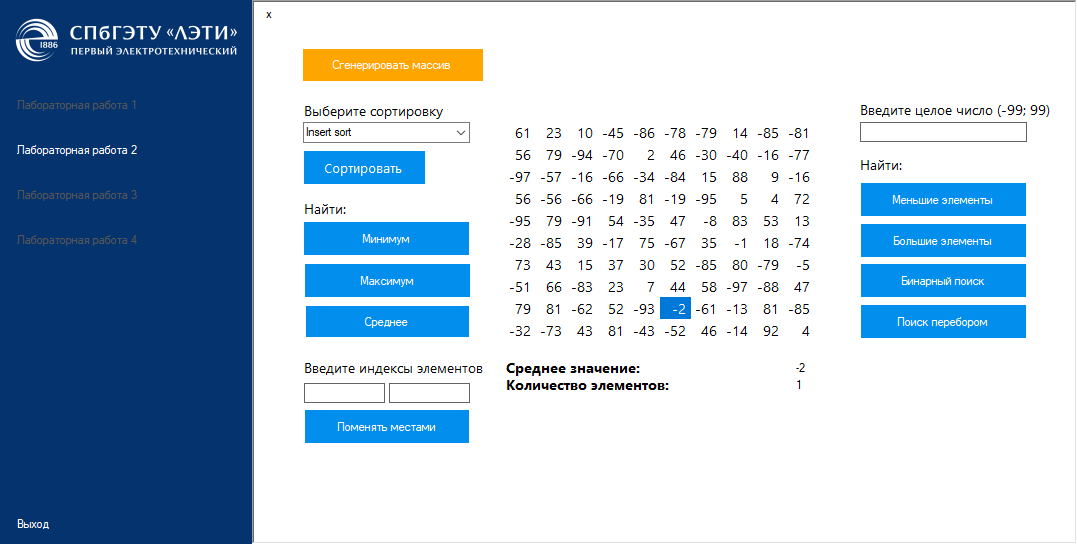


Рисунок 14 — пример поиска среднего элемента массива

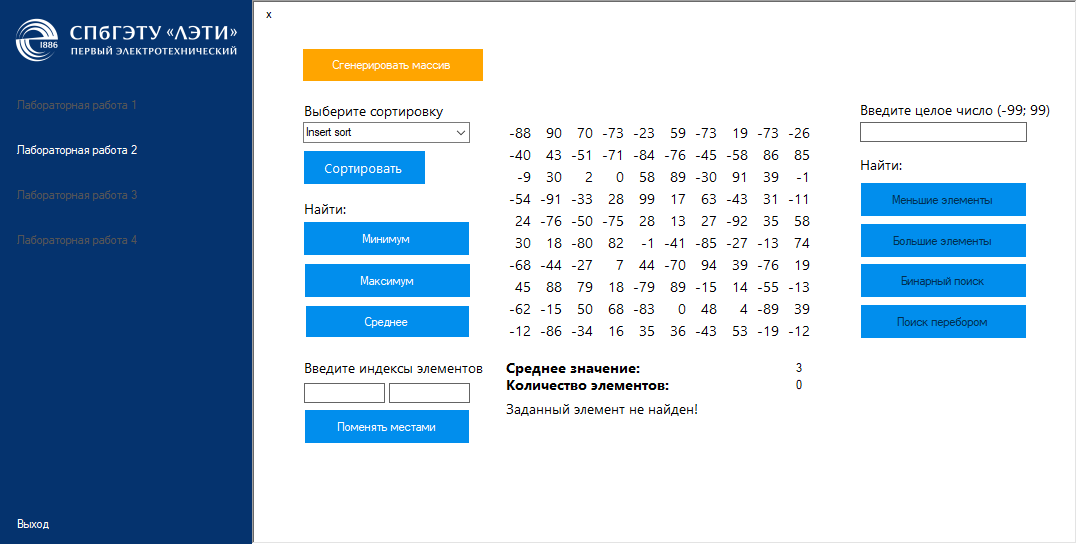


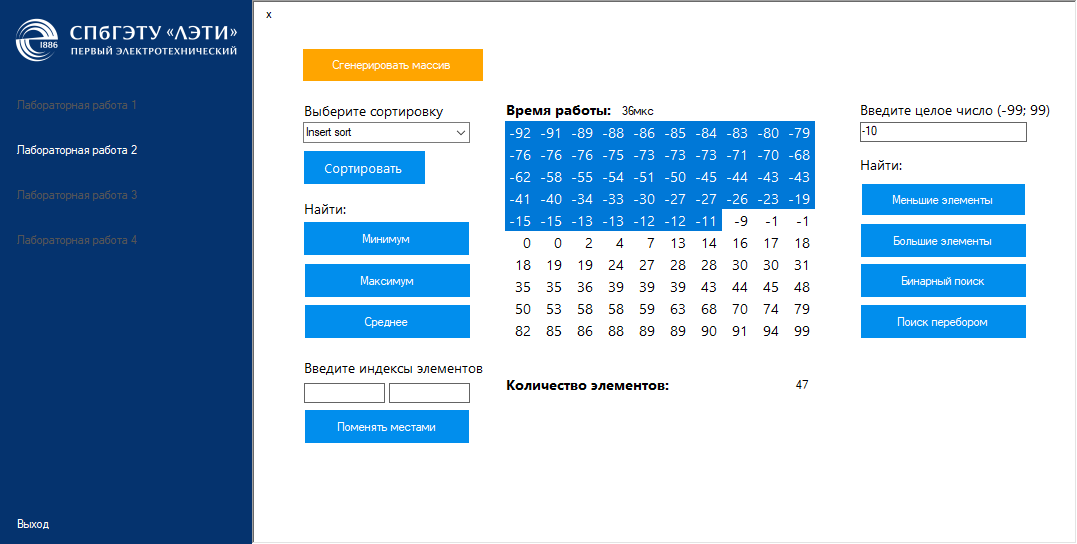
Рисунок 15 — элемент со средним значением не найден

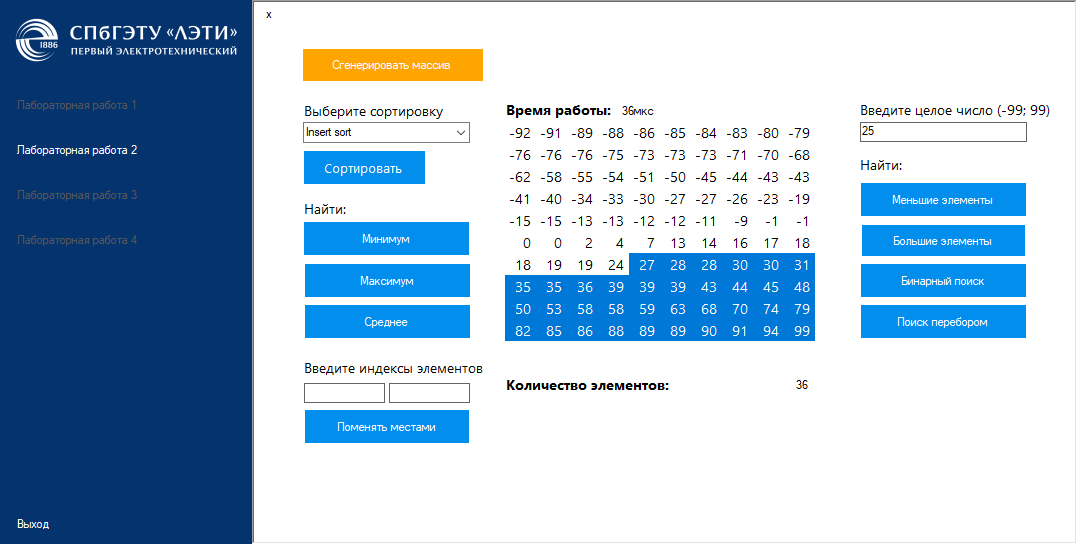
1. Вывод количества элементов, меньших числа a, заданного пользователем и их количества
2. Вывод количества элементов, больших числа b, заданного пользователем и их количества

Функции поиска элементов, меньших или больших числа заданного пользователем:

1. int searchLowerA(int\* Arr, int a)
2. {
3. int lessNum = 0;
4. while (Arr[lessNum] < a)
5. lessNum++;
6. return lessNum;
7. }
8. int searchMoreB(int arrayWidth, int\* Arr, int b)
9. {
10. int i = arrayWidth - 1,
11. moreNum = 0;
12. while (Arr[i] > b) {
13. i--;
14. moreNum++;
15. }
16. return moreNum;
17. }

Так как массив отсортирован, то для поиска элементов достаточно перебрать только ту часть массива, в которой находятся элементы меньшие (большие) числа, заданного пользователем.





Рисунки 16,173 — примеры поиска элементов массива меньших и больших числа, введённого пользователем

Примечание: в моём GUI во избежание возможных ошибок эти функции недоступны, пока массив не отсортирован.

1. Вывод информации о том, есть ли в массиве число, заданное пользователем.

В работе я реализовала два варианта поиска: бинарный и поиск перебором.

Бинарный поиск — более сложный, но более быстрый (на больших объёмах данных) механизм. В массиве размерности 100 разницей времени выполнения можно пренебречь, а если искомый элемент находится в начале массива, то бинарный поиск может быть даже медленнее перебора.

Функция бинарного поиска:

1. int binarySearch(int\* Arr, int neededNum, int startPos, int endPos) {
2. //определяем ширину отрезка массива, в котором производим поиск
3. int arrWidth = endPos - startPos + 1;
4. //устанавливаем условие выхода из рекурсии
5. if (arrWidth > 1) {
6. //ставим указатель в центр отрезка
7. int i = arrWidth / 2 + startPos;
8. //если число меньше, чем нужное, то рекурсивно
9. //вызываем функцию снова на отрезке справа от указателя
10. if (Arr[i] < neededNum)
11. return binarySearch(Arr, neededNum, i + 1, endPos);
12. //если число больше, чем нужное, то рекурсивно
13. //вызываем функцию снова на отрезке слева от указателя
14. else if (Arr[i] > neededNum)
15. return binarySearch(Arr, neededNum, startPos, i - 1);
16. //или возвращаем позицию найденного числа
17. else
18. return i;
19. }
20. //последний рассматриваемый элемент может быть искомым
21. //проверим это
22. else if (Arr[startPos] == neededNum)
23. return startPos;
24. //Если искомое число и теперь не наншлось, то это значит,
25. //что в массиве его нет, возвращаем -1
26. else
27. return -1;
28. }

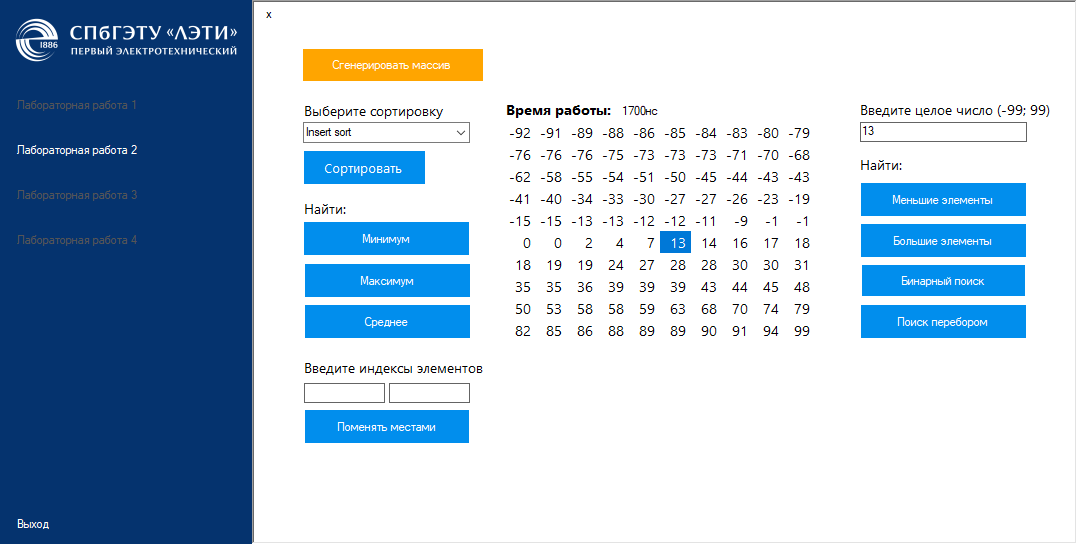


Рисунок 18 — пример работы алгоритма бинарного поиска

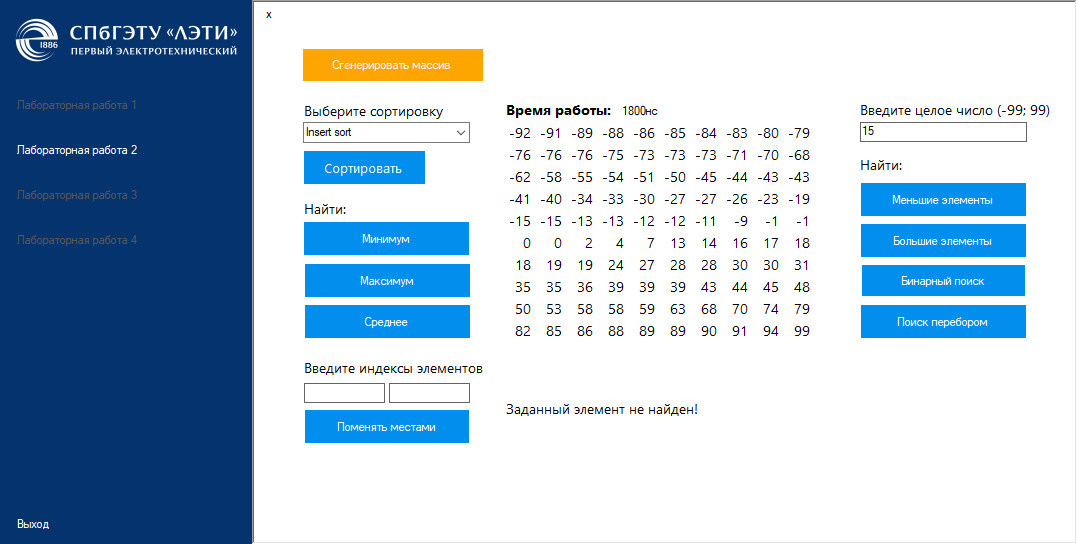


Рисунок 19 — в результате бинарного поиска элемент не был найден

Функция поиска простым перебором числа, заданного пользователем:

1. int bruteforce(int arrWidth, int\* Arr, int neededNum) {
2. int i = 0;
3. //перебираем массив до тех пор, пока элемент под указателем
4. //не становится больше искомого
5. while (Arr[i] <= neededNum) {
6. if (Arr[i] == neededNum) {
7. return i;
8. }
9. else i++;
10. }
11. //Если искомого числа в массиве нет, возвращаем -1
12. return -1;
13. }

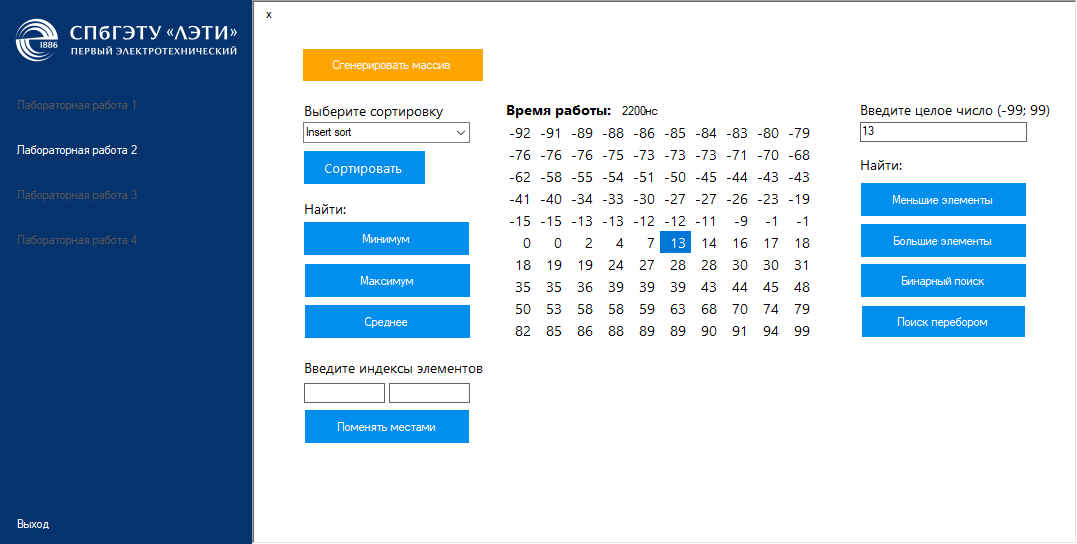


Рисунок 20 — пример работы алгоритма поиска перебором

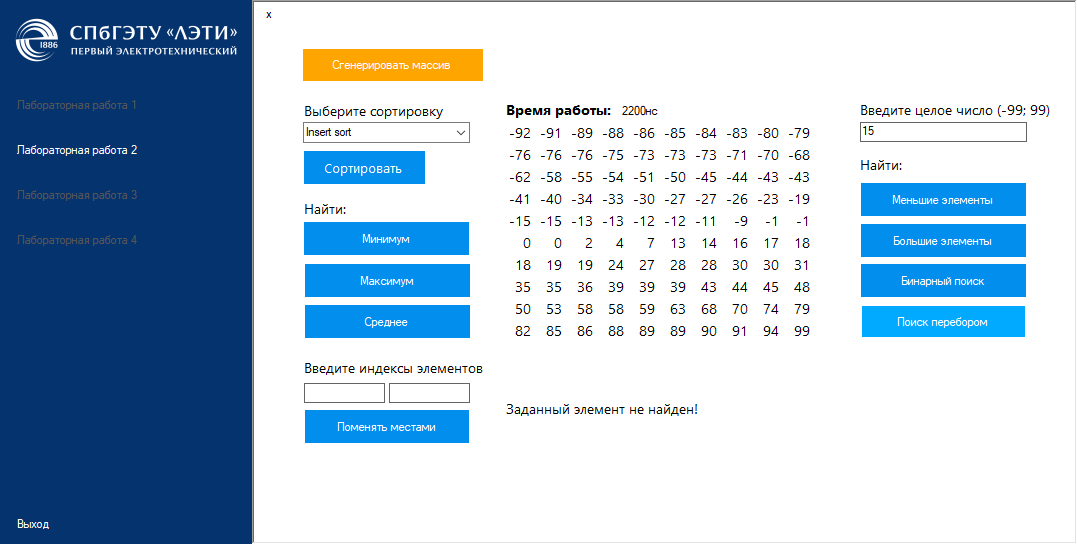


Рисунок 21— в результате поиска перебором элемент не был найден

1. Обмен элементов массива с индексами, введёнными пользователем.

Код функции обмена двух элементов:

1. std::swap(Arr[a], Arr[b]);

При работе в графическом интерфейсе моей программы предусмотрена обработка ошибок, допущенных пользователем. Примеры работы с ошибками размещены в приложении В.

Выводы.

После выполнения работы и чтения дополнительных материалов у меня остался единственный вывод: работа со статическими одномерными массивами себя не оправдывает. Гораздо удобнее для решения реальных, а не учебных задач использовать векторы из библиотеки <Vector>. Так же я смогла посмотреть на практике на работу разных алгоритмов сортировки.

Дополнительно к этому я освоила создание графических интерфейсов на C++\CLI и убедилась в том, что это не лучший вариант. Впрочем, графические интерфейсы для C++ неудобны все, так как язык изначально создавался не для этого.

Приложение А

Полный код программы

Заголовочные файлы:

1. service.h (служебные функции, которые будут использоваться и в других лабораторных работах, а также в этом файле подключены используемые библиотеки)
2. #ifndef Service\_H
3. #define Service\_H
4. #include <iostream>
5. #include <ctime>
6. #include <chrono>
7. #include <cmath>
8. #include <string>
9. using namespace std::chrono;
10. int getRandomNumber(int min, int max);
11. /\*\*
12. \* генерация случайного числа
13. \* @param {min}, {max} - диапазон значений
14. \* @retutn int - возвращает полученное число
15. \*/
16. void showArray(int arrayWidth, int\* Arr);
17. /\*\*
18. \* вывод массива в консоль
19. \* @param {arrayWidth} - количество элементов
20. \* @param {\*Arr} - указатель на массив, который нужно вывести
21. \*/
22. void generateArray(int arrayWidth, int min, int max, int\* Arr);
23. /\*\*
24. \* генерация массива - количество элементов
25. \* @param {arrayWidth} - количество элементов
26. \* @param {min}, {max} - диапазон значений
27. \* @param {\*Arr} - указатель на массив, в который будут записываться значения
28. \*/
29. #endif Service\_H;
30. lab2-functions.h (заголовки функций, перенесённых в готовое приложение из консольного варианта)
31. #ifndef Lab2\_H
32. #define Lab2\_H
33. /\*\*
34. \* генерация массива
35. \* @param {arrayWidth} - количество элементов
36. \* @param {min}, {max} - диапазон значений
37. \* @param {\*Arr} - указатель на массив, в который будут записываться значения
38. \*/
39. void generateArray(int arrayWidth, int min, int max, int\* Arr);
40. /\*\*
41. \* пузырьковая сортировка заданного массива
42. \* @param {arrayWidth} - количество элементов
43. \* @param {\*Arr} - указатель на массив, в который будут записываться значения
44. \*/
45. void bubbleSort(int arrayWidth, int\* Arr);
46. /\*\*
47. \* шейкер-сортировка заданного массива
48. \* @param {arrayWidth} - количество элементов
49. \* @param {\*Arr} - указатель на массив, в который будут записываться значения
50. \*/
51. void shakerSort(int arrayWidth, int\* Arr);
52. /\*\*
53. \* комбинированная сортировка (сортировка "расчёской") заданного массива
54. \* @param {arrayWidth} - количество элементов
55. \* @param {\*Arr} - указатель на массив, в который будут записываться значения
56. \*/
57. void combSort(int arrayWidth, int\* Arr);
58. /\*\*
59. \* сортировка вставками заданного массива
60. \* @param {arrayWidth} - количество элементов
61. \* @param {\*Arr} - указатель на массив, в который будут записываться значения
62. \*/
63. void insertSort(int arrayWidth, int\* Arr);
64. /\*\*
65. \* быстрая сортировка заданного массива
66. \* @param {arrayWidth} - количество элементов
67. \* @param {\*Arr} - указатель на массив, в который будут записываться значения
68. \*/
69. void quickSort(int arrayWidth, int\* Arr);
70. /\*\*
71. \* замер времени для разных функций сортировки
72. \* к сожалению, я пока не смогла найти способ сделать универсальный коллбэк,
73. \* поэтому кое-где в проекте код таймера будет дублироваться
74. \* @param {arrayWidth}, @param {\*Arr} - количество элементов и указатель на сортируемый массив
75. \* @param {\*funcP} - указатель на одну из функций сортировки
76. \* @returns возвращает время в мс
77. \*/
78. int benchFunc(int arrayWidth, int\* Arr, void(\*funcP)(int arrayWidth, int\* Arr));
79. /\*\*
80. \* замер времени для разных функций сортировки (со схожими сигнатурами)
81. \* дублирование функции benchFunc(...) для коллбэков с другой сигнатурой
82. int benchFunc2(int arrayWidth, int\* Arr, int(\*funcP)(int arrayWidth, int\* Arr));
83. int benchFunc3(int\* Arr, int neededNum, int startPos, int endPos, int(\*funcP)(int\* Arr, int neededNum, int startPos, int endPos));
84. int benchFunc4(int arrWidth, int\* Arr, int neededNum, int(\*funcP)(int arrWidth, int\* Arr, int neededNum));
85. /\*\*
86. \* замер времени исполнения разных функций
87. \* @param {arrayWidth}, @param {\*Arr} - количество элементов и ссылка на массив
88. \* @param {\*funcP} - указатель на коллбэк функцию
89. \* @returns возвращает время в мс
90. \*/
91. int searchMax(int arrayWidth, int\* Arr);
92. /\*\*
93. \* поиск минимального элемента массива
94. \* @param {arrayWidth}, @param {\*Arr} - количество элементов и указатель на массив, в котором проводится поиск
95. \* @returns возвращает минимальный элемент
96. \*/
97. int searchMin(int arrayWidth, int\* Arr);
98. /\*\*
99. \* поиск среднего элемента массива
100. \* @param {arrayWidth}, @param {\*Arr} - количество элементов и указатель на массив, в котором проводится поиск
101. \* @param {&middleNum} - ссылка на переменную, в которой хранится количество элементов, равных среднему
102. \* @returns возвращает минимальный элемент
103. \*/
104. int searchMiddle(int arrayWidth, int\* Arr);
105. /\*\*
106. \* подсчёт количества элементов, меньших а, вводимого пользователем
107. \* @param {\*Arr} - указатель на массив, в котором проводится поиск
108. \* @param {a} - заданное пользователем число
109. \* @returns возвращает искомое количество
110. \*/
111. int searchLowerA(int\* Arr, int a);
112. /\*\*
113. \* подсчёт количества элементов, больших b, вводимого пользователем
114. \* @param {arrayWidth}, @param {\*Arr} - количество элементов и указатель на массив, в котором проводится поиск
115. \* @param {b} - заданное пользователем число
116. \* @returns возвращает искомое количество
117. \*/
118. int searchMoreB(int arrayWidth, int\* Arr, int b);/\*\*
119. \* поиск в массиве позиции числа, введённого пользователем
120. \* @param {\*Arr} -указатель на массив, в котором проводится поиск
121. \* @param {neededNum} - заданное пользователем число
122. \* @param {startPos}, @param {endPos} - начальная и конечная позиции
123. \* @returns возвращает искомое число или -1, если элемент не найден
124. \*/
125. int binarySearch(int\* Arr, int neededNum, int startPos, int endPos);
126. /\*\*
127. \* поиск в массиве позиции числа, введённого пользователем
128. \* @param {arrayWidth}, @param {\*Arr} - количество элементов и указатель на массив, в котором проводится поиск
129. \* @param {neededNum} - заданное пользователем число
130. \* @param {position} - стартовая позиция, по умолчанию поиск начинается с нулевого элемента
131. \* @returns возвращает число или -1, если элемент не найден\*/
132. int bruteforce(int arrWidth, int\* Arr, int neededNum);
133. #endif
134. lab2.h (файл формы. Настройки элементов формы не нужны в данной работе, так что для упрощения чтения я их опущу)
135. //Первичная настройка свойств таблицы
136. private: void formTableSetup();
137. //Очистка выделения в таблице
138. private: void formTableClean();
139. //вывод массива в форму
140. private: void showArrayInForm();
141. //вывод времени выполнения функции
142. private: int intTypeCheck(String^ text);
143. //вывод времени выполнения функции
144. private: int indexTypeCheck(String^ text, String^ errorText);
145. //вывод времени выполнения функции
146. private: void showTime(int time, String^ unit);
147. //вывод количества найденных элементов
148. private: void showNum(int num);
149. //вывод ошибки
150. private: void showError(String^ text);
151. //вывод среднего значения в массиве
152. private: void showMiddle(int middle);
153. //скрытие времени выполнения функции
154. private: void hideTime();
155. //скрытие количества найденных элементов
156. private: void hideNum();
157. //скрытие среднего значения
158. private: void hideMiddle();
159. //скрытие всех информационных элементов
160. private: void hideInfo();
161. //показ найденного элемента
162. private: void showEl(int index);
163. //дизаблим кнопки, которые не должны работать, пока массив не отсортирован
164. private: void btnsDisable();
166. //а теперь их энаблим
167. private: void btnsEnable();
168. //КОНТРОЛЫ:
169. //Закрытие формы
170. private: System::Void btnCloseForm\_Click(System::Object^ sender, System::EventArgs^ e);
171. //Обработка клика по кнопке "Генерация массива"
172. private: System::Void btnGetArray\_Click(System::Object^ sender, System::EventArgs^ e);
173. //Обработка клика по кнопке "Сортировать"
174. private: System::Void btnSort\_Click(System::Object^ sender, System::EventArgs^ e);
175. //Обработка клика по кнопке "Минимальное значение"
176. private: System::Void btnSearchMin\_Click(System::Object^ sender, System::EventArgs^ e);
177. //Обработка клика по кнопке "Максимальное значение"
178. private: System::Void btnSearchMax\_Click(System::Object^ sender, System::EventArgs^ e);
179. //Обработка клика по кнопке "Среднее значение"
180. private: System::Void btnSearchMiddle\_Click(System::Object^ sender, System::EventArgs^ e);
181. //Обработка клика по кнопке "Поменять местами"
182. private: System::Void btnSwap\_Click(System::Object^ sender, System::EventArgs^ e);
183. //Обработка клика по кнопке "Меньшие элементы"
184. private: System::Void btnSearchLess\_Click(System::Object^ sender, System::EventArgs^ e);
185. //Обработка клика по кнопке "Большие элементы"
186. private: System::Void btnSearchMore\_Click(System::Object^ sender, System::EventArgs^ e);
187. //Обработка клика по кнопке "Бинарный поиск"
188. private: System::Void btnBinarySearch\_Click(System::Object^ sender, System::EventArgs^ e);
189. //Обработка клика по кнопке "Поиск перебором"
190. private: System::Void btnBruteforceSearch\_Click(System::Object^ sender, System::EventArgs^ e);

Основные файлы:

1. service.cpp (в этом файле размещены функции, которые будут использоваться и в других лабораторных работах)
2. #include "Service.h"
3. int getRandomNumber(int min, int max) {
4. return (rand() % (max - min + 1) + min);
5. }
6. void showArray(int arrayWidth, int \*Arr) {
7. for (int i = 0; i < arrayWidth; i++)
8. {
9. std::cout << Arr[i] << "\t";
10. if ((i + 1) % 5 == 0)
11. std::cout << "\n";
12. }
13. }
14. void generateArray(int arrayWidth, int min, int max, int\* Arr) {
15. // устанавливаем опорное значение с помощью значения системного времени
16. srand(time(0));
17. int el;
18. for (int i = 0; i < arrayWidth; ++i)
19. Arr[i] = getRandomNumber(-99, 99);
20. }
21. lab2-functions.cpp (в этом файле размещены почти все функции, упоминаемые ранее в отчёте)
22. #include "Service.h"
23. #include "Lab2-functions.h"
24. void bubbleSort(int arrayWidth, int\* Arr) {
25. for (int i = 0; i < arrayWidth; i++) {
26. // устанавливаем флаг, показывающий, что массив уже отсортирован
27. bool isSort = true;
28. for (int j = 0; j < arrayWidth - (i + 1); j++) {
29. if (Arr[j] > Arr[j + 1]) {
30. // если по дороге попадается неотсортированный элемент,
31. // присваиваем флагу значение false и меняем элементы местами
32. isSort = false;
33. std::swap(Arr[j], Arr[j + 1]);
34. }
35. }
36. // если за весь проход массива ни разу не попались неотсортированные элементы,
37. // то прерываем цикл
38. if (isSort) break;
39. }
40. }
41. void shakerSort(int arrayWidth, int\* Arr)
42. {
43. for (int i = 0; i < arrayWidth - i; i++) {
44. bool isSort = true;
45. // проход в из начала в конец
46. // "выталкивание" наибольшего элемента в конец
47. for (int j = 0; j < arrayWidth - (i + 1); j++) {
48. if (Arr[j] > Arr[j + 1]) {
49. isSort = false;
50. std::swap(Arr[j], Arr[j + 1]);
51. }
52. }
53. // проход в из конца в начало
54. // "выталкиваение" наименьшего элемента в начало
55. for (int j = arrayWidth - (i + 1); j >= i + 1; j--) {
56. if (Arr[j] < Arr[j - 1]) {
57. isSort = false;
58. std::swap(Arr[j], Arr[j - 1]);
59. }
60. }
61. if (isSort) break;
62. }
63. }
64. void combSort(int arrayWidth, int\* Arr){
65. // коэффициент, на который будет уменьшаться шаг "расчёски"
66. float k = 1.28;
67. // ширина проверяемого диапазона
68. // на начальном этапе равна ширине массива
69. // после будет уменьшаться в k раз
70. int S = arrayWidth;
71. //пока диапазон не схлопнется до к элементов
72. while (k < S) {
73. //сравниваем элементы из начала и конца диапазона
74. //и сдвигаем "расчёску", пока не упрёмся правым концом в конец массива
75. for (int i = 0; i <= arrayWidth - S; i++) {
76. if (Arr[i] > Arr[i + S - 1]) {
77. std::swap(Arr[i], Arr[i + S - 1]);
78. }
79. }
80. //уменьшаем диапазон
81. S /= k;
82. }
83. //после того, как диапазон стал слишком маленьким
84. //остаток "причёсываем" пузырьком
85. bubbleSort(arrayWidth, Arr);
86. }
87. void insertSort(int arrayWidth, int\* Arr){
88. for (int i = 1; i < arrayWidth; i++){
89. // установка элемента на своё место
90. // в отсортированной части массива
91. for (int j = i; j > 0 && Arr[j - 1] > Arr[j]; j--)
92. std::swap(Arr[j - 1], Arr[j]);
93. }
94. }
95. void quickSort(int arrayWidth, int\* Arr) {
96. //устанавливаем левый (i) и правый (j) маркеры
97. int left = 0;
98. int right = arrayWidth - 1;
99. //находим опорный элемент в центре массива
100. int mid = Arr[arrayWidth / 2];
101. //продолжаем сводить маркеры, пока они не пересекутся
102. while (left <= right) {
103. //В левой части массива пропускаем элементы, которые меньше опорного
104. while (Arr[left] < mid) {
105. left++;
106. }
107. //В правой части пропускаем элементы, которые больше опорного
108. while (Arr[right] > mid) {
109. right--;
110. }
111. //Меняем элементы местами
112. if (left <= right) {
113. std::swap(Arr[left], Arr[right]);
114. left++;
115. right--;
116. }
117. }
118. //рекурсивно вызываем функцию сортировки, если остались неотсортированные элементы
119. if (right > 0) {
120. //для левой половины от места встречи маркеров
121. quickSort(right + 1, Arr);
122. }
123. if (left < arrayWidth) {
124. //для правой половины от места встречи маркеров
125. //чтобы не вводить отдельные переменные,
126. //обращаемся к i-тому элементу массива по ссылке
127. quickSort(arrayWidth - left, &Arr[left]);
128. }
129. }
130. int benchFunc(int arrayWidth, int\* Arr, void(\*funcP)(int arrayWidth, int\* Arr)) {
131. using Time = time\_point<steady\_clock>;
132. using Diff = microseconds;
133. // Запоминаем значение системного времени до начала выполнения функции
134. // используем монотонные часы steady\_clock
135. Time start = steady\_clock::now();
136. // вызываем коллбэк функцию
137. funcP(arrayWidth, Arr);
138. // Запоминаем значение системного времени после выполнения функции
139. Time end = steady\_clock::now();
140. // Определяем тип объекта интервала и вычисляем его значение
141. Diff diff = duration\_cast<Diff>(end - start);
142. // вычисляем количество тактов в интервале
143. // и возвращаем итог
144. return diff.count();
145. }
146. int benchFunc2(int arrayWidth, int\* Arr, int(\*funcP)(int arrayWidth, int\* Arr))
147. {
148. using Time = time\_point<steady\_clock>;
149. using Diff = nanoseconds;
150. // Запоминаем значение системного времени до начала выполнения функции
151. // используем монотонные часы steady\_clock
152. Time start = steady\_clock::now();
153. // вызываем коллбэк функцию
154. funcP(arrayWidth, Arr);
155. // Запоминаем значение системного времени после выполнения функции
156. Time end = steady\_clock::now();
157. // Определяем тип объекта интервала и вычисляем его значение
158. Diff diff = duration\_cast<Diff>(end - start);
159. // вычисляем количество тактов в интервале
160. // и возвращаем итог
161. return diff.count();
162. }
163. int benchFunc3(int\* Arr, int neededNum, int startPos, int endPos, int(\*funcP)(int\* Arr, int neededNum, int startPos, int endPos))
164. {
165. using Time = time\_point<steady\_clock>;
166. using Diff = nanoseconds;
167. // Запоминаем значение системного времени до начала выполнения функции
168. // используем монотонные часы steady\_clock
169. Time start = steady\_clock::now();
170. // вызываем коллбэк функцию
171. funcP(Arr, neededNum, startPos, endPos);
172. // Запоминаем значение системного времени после выполнения функции
173. Time end = steady\_clock::now();
174. // Определяем тип объекта интервала и вычисляем его значение
175. Diff diff = duration\_cast<Diff>(end - start);
176. // вычисляем количество тактов в интервале
177. // и возвращаем итог
178. return diff.count();
179. }
180. int benchFunc4(int arrWidth, int\* Arr, int neededNum, int(\*funcP)(int arrWidth, int\* Arr, int neededNum))
181. {
182. using Time = time\_point<steady\_clock>;
183. using Diff = nanoseconds;
184. // Запоминаем значение системного времени до начала выполнения функции
185. // используем монотонные часы steady\_clock
186. Time start = steady\_clock::now();
187. // вызываем коллбэк функцию
188. funcP(arrWidth, Arr, neededNum);
189. // Запоминаем значение системного времени после выполнения функции
190. Time end = steady\_clock::now();
191. // Определяем тип объекта интервала и вычисляем его значение
192. Diff diff = duration\_cast<Diff>(end - start);
193. // вычисляем количество тактов в интервале
194. // и возвращаем итог
195. return diff.count();
196. }
197. int searchMax(int arrayWidth, int\* Arr) {
198. int index = 0;
199. int max = Arr[index];
200. for (int i = 1; i < arrayWidth; i++)
201. if (Arr[i] > max) {
202. max = Arr[i];
203. index = i;
204. }
205. return index;
206. }
207. int searchMin(int arrayWidth, int\* Arr) {
208. int index = 0;
209. int min = Arr[index];
210. for (int i = 1; i < arrayWidth; i++)
211. {
212. if (Arr[i] < min) {
213. min = Arr[i];
214. index = i;
215. }
216. }
217. return index;
218. }
219. int searchMiddle(int arrayWidth, int\* Arr) {
220. //вызов функции поиска минимального элемента
221. int minEl = Arr[searchMin(arrayWidth, Arr)];
222. //вызов функции поиска максимального элемента
223. int maxEl = Arr[searchMax(arrayWidth, Arr)];
224. int middleEl = round((minEl + maxEl) / 2);
225. return middleEl;
226. }
227. int searchLowerA(int\* Arr, int a)
228. {
229. int lessNum = 0;
230. while (Arr[lessNum] < a)
231. lessNum++;
232. return lessNum;
233. }
234. int searchMoreB(int arrayWidth, int\* Arr, int b)
235. {
236. int i = arrayWidth - 1,
237. moreNum = 0;
238. while (Arr[i] > b) {
239. i--;
240. moreNum++;
241. }
242. return moreNum;
243. }
244. int binarySearch(int\* Arr, int neededNum, int startPos, int endPos) {
245. //определяем ширину отрезка массива, в котором производим поиск
246. int arrWidth = endPos - startPos + 1;
247. //устанавливаем условие выхода из рекурсии
248. if (arrWidth > 1) {
249. //ставим указатель в центр отрезка
250. int i = arrWidth / 2 + startPos;
251. //если число меньше, чем нужное, то рекурсивно
252. //вызываем функцию снова на отрезке справа от указателя
253. if (Arr[i] < neededNum)
254. return binarySearch(Arr, neededNum, i + 1, endPos);
255. //если число больше, чем нужное, то рекурсивно
256. //вызываем функцию снова на отрезке слева от указателя
257. else if (Arr[i] > neededNum)
258. return binarySearch(Arr, neededNum, startPos, i - 1);
259. //или возвращаем позицию найденного числа
260. else
261. return i;
262. }
263. //последний рассматриваемый элемент может быть искомым
264. //проверим это
265. else if (Arr[startPos] == neededNum)
266. return startPos;
267. //Если искомое число и теперь не наншлось, то это значит,
268. //что в массиве его нет, возвращаем -1
269. else
270. return -1;
271. }
272. int bruteforce(int arrWidth, int\* Arr, int neededNum) {
273. int i = 0;
274. //перебираем массив до тех пор, пока элемент под указателем
275. //не становится больше искомого
276. while (Arr[i] <= neededNum) {
277. if (Arr[i] == neededNum) {
278. return i;
279. }
280. else i++;
281. }
282. //Если искомого числа в массиве нет, возвращаем -1
283. return -1;
284. }
285. lab2.cpp (основной файл для работы интерфейса лабораторной работы №2. В нём заданы глобальные переменные для работы с массивом, а так же обработчики поведения элементов управления формы.
286. #include "Lab2.h"
287. #include "../Sources/Service.h"
288. //количество строк и столбцов для более удобного вывода массива
289. int cols = 10, rows = 10;
290. //количество элементов
291. //в случае, если надо будет переделать работу так
292. //чтобы размерность массива задавалась пользователем,
293. //нужно будет поменять только эти три переменные
294. const int arrayWidth = 100,
295. //диапазон значений массива
296. min = -99,
297. max = 99;
298. //флаг, показывающий что массив отсортирован
299. bool isSorted = false;
300. //массив, заданной ширины
301. int Arr[arrayWidth];
302. void Coursach::Lab2::formTableSetup()
303. {
304. formTable->RowCount = rows;
305. formTable->ColumnCount = cols;
306. formTableClean();
307. }
308. void Coursach::Lab2::formTableClean()
309. {
310. for (int i = 0; i < cols; i++)
311. for (int j = 0; j < rows; j++)
312. formTable->Rows[i]->Cells[j]->Selected = false;
313. }
314. void Coursach::Lab2::showArrayInForm()
315. {
316. int counter = 0;
317. while (counter < arrayWidth) {
318. for (int i = 0; i < cols; i++)
319. for (int j = 0; j < rows; j++)
320. {
321. formTable->Rows[i]->Cells[j]->Value = Arr[counter];
322. counter++;
323. }
324. }
325. }
326. int Coursach::Lab2::intTypeCheck(String^ text)
327. {
328. Int16 num;
329. if (Int16::TryParse(System::Convert::ToString(text), num)) {
330. if (num >= min && num <= max)
331. return num;
332. else {
333. showError("Число " + num + " не входит в заданный диапазон");
334. return min - 100;
335. }
336. }
337. else {
338. showError("Проверьте правильность ввода");
339. return min - 100;
340. }
341. }
342. int Coursach::Lab2::indexTypeCheck(String^ text, String^ errorText)
343. {
344. Int16 num;
345. if (Int16::TryParse(System::Convert::ToString(text), num)) {
346. if (num >= 0 && num < arrayWidth)
347. return num;
348. else {
349. showError("Введите правильный индекс от 0 до 99 в " + errorText);
350. return -1;
351. }
352. }
353. else {
354. showError("Введите оба индекса");
355. return -1;
356. }
357. }
358. void Coursach::Lab2::showTime(int time, String^ unit)
359. {
360. infoLabel1->Text = time.ToString() + unit;
361. if (infoLabel1->Visible == false) infoLabel1->Visible = true;
362. if (infoLabelHeader->Visible == false) infoLabelHeader->Visible = true;
363. }
364. void Coursach::Lab2::showNum(int num)
365. {
366. infoLabel2->Text = num.ToString();
367. if (infoLabel2->Visible == false) infoLabel2->Visible = true;
368. if (infoLabelHeader2->Visible == false) infoLabelHeader2->Visible = true;
369. }
370. void Coursach::Lab2::showError(String^ text)
371. {
372. errorLabel1->Text = text;
373. errorLabel1->Visible = true;
374. }
375. void Coursach::Lab2::showMiddle(int middle)
376. {
377. infoLabel4->Text = middle.ToString();
378. if (infoLabel4->Visible == false) infoLabel4->Visible = true;
379. if (infoLabelHeader4->Visible == false) infoLabelHeader4->Visible = true;
380. }
381. void Coursach::Lab2::hideTime()
382. {
383. if (infoLabel1->Visible == true) infoLabel1->Visible = false;
384. if (infoLabelHeader->Visible == true) infoLabelHeader->Visible = false;
385. }
386. void Coursach::Lab2::hideNum()
387. {
388. if (infoLabel2->Visible == true) infoLabel2->Visible = false;
389. if (infoLabelHeader2->Visible == true) infoLabelHeader2->Visible = false;
390. }
391. void Coursach::Lab2::hideMiddle()
392. {
393. if (infoLabel4->Visible == true) infoLabel4->Visible = false;
394. if (infoLabelHeader4->Visible == true) infoLabelHeader4->Visible = false;
395. }
396. void Coursach::Lab2::btnsDisable()
397. {
398. if (isSorted == false) {
399. btnBinarySearch->Enabled = false;
400. btnSearchLess->Enabled = false;
401. btnSearchMore->Enabled = false;
402. btnBruteforceSearch->Enabled = false;
403. }
404. }
405. void Coursach::Lab2::btnsEnable()
406. {
407. if (isSorted == true) {
408. btnBinarySearch->Enabled = true;
409. btnSearchLess->Enabled = true;
410. btnSearchMore->Enabled = true;
411. btnBruteforceSearch->Enabled = true;
412. }
413. }
414. void Coursach::Lab2::hideInfo()
415. {
416. formTableClean();
417. hideTime();
418. hideNum();
419. hideMiddle();
420. infoLabel3->Visible = false;
421. }
422. void Coursach::Lab2::showEl(int index)
423. {
424. int i = index / 10;
425. int j = index % 10;
426. formTable->Rows[i]->Cells[j]->Selected = true;
427. }
428. System::Void Coursach::Lab2::btnCloseForm\_Click(System::Object^ sender, System::EventArgs^ e)
429. {
430. this->Close();
431. }
432. System::Void Coursach::Lab2::btnGetArray\_Click(System::Object^ sender, System::EventArgs^ e)
433. {
434. formTableSetup();
435. btnsDisable();
436. if (arrayControlsPanel->Visible == false)
437. arrayControlsPanel->Visible = true;
438. generateArray(arrayWidth, min, max, Arr);
439. isSorted = false;
440. hideInfo();
441. showArrayInForm();
442. }
443. System::Void Coursach::Lab2::btnSort\_Click(System::Object^ sender, System::EventArgs^ e)
444. {
445. hideInfo();
446. int time;
447. if (errorLabel1->Visible == true) errorLabel1->Visible = false;
448. switch (sortsBox->SelectedIndex) {
449. case 0:
450. time = benchFunc(arrayWidth, Arr, bubbleSort);
451. break;
452. case 1:
453. time = benchFunc(arrayWidth, Arr, shakerSort);
454. break;
455. case 2:
456. time = benchFunc(arrayWidth, Arr, combSort);
457. break;
458. case 3:
459. time = benchFunc(arrayWidth, Arr, insertSort);
460. break;
461. case 4:
462. time = benchFunc(arrayWidth, Arr, quickSort);
463. break;
464. default:
465. String^ errText = "Вы не выбрали тип сортировки!";
466. showError(errText);
467. return;
468. }
469. isSorted = true;
470. btnsEnable();
471. showArrayInForm();
472. showTime(time, "мкс");
473. }
474. System::Void Coursach::Lab2::btnSearchMin\_Click(System::Object^ sender, System::EventArgs^ e)
475. {
476. hideInfo();
477. int time = benchFunc2(arrayWidth, Arr, searchMin);
478. int index = searchMin(arrayWidth, Arr);
479. showTime(time, "нс");
480. showEl(index);
481. }
482. System::Void Coursach::Lab2::btnSearchMax\_Click(System::Object^ sender, System::EventArgs^ e)
483. {
484. hideInfo();
485. int time = benchFunc2(arrayWidth, Arr, searchMax);
486. int index = searchMax(arrayWidth, Arr);
487. showTime(time, "нс");
488. showEl(index);
489. }
490. System::Void Coursach::Lab2::btnSearchMiddle\_Click(System::Object^ sender, System::EventArgs^ e)
491. {
492. hideInfo();
493. int middleNum = 0;
494. int middleEl = searchMiddle(arrayWidth, Arr);
495. for (int i = 0; i < arrayWidth; i++) {
496. if (Arr[i] == middleEl) {
497. showEl(i);
498. middleNum++;
499. }
501. }
502. showNum(middleNum);
503. showMiddle(middleEl);
504. if (middleNum == 0)
505. infoLabel3->Visible = true;
506. }
507. System::Void Coursach::Lab2::btnSearchLess\_Click(System::Object^ sender, System::EventArgs^ e)
508. {
509. formTableClean();
510. int a = intTypeCheck(textBoxNum->Text);
511. if (a < min)
512. return;
513. else {
514. int lessNum = searchLowerA(Arr, a);
515. for (int i = 0; i < lessNum; i++)
516. showEl(i);
517. showNum(lessNum);
518. }
519. }
520. System::Void Coursach::Lab2::btnSearchMore\_Click(System::Object^ sender, System::EventArgs^ e)
521. {
522. formTableClean();
523. int b = intTypeCheck(textBoxNum->Text);
524. if (b < min)
525. return;
526. else {
527. int moreNum = searchMoreB(arrayWidth, Arr, b);
528. for (int i = arrayWidth - 1; i > (arrayWidth - moreNum - 1); i--)
529. showEl(i);
530. showNum(moreNum);
531. }
532. }
533. System::Void Coursach::Lab2::btnBinarySearch\_Click(System::Object^ sender, System::EventArgs^ e)
534. {
535. hideInfo();
536. int neededNum = intTypeCheck(textBoxNum->Text);
537. if(neededNum < min)
538. return;
539. else {
540. int time = benchFunc3(Arr, neededNum, 0, arrayWidth - 1, binarySearch);
541. showTime(time, "нс");
542. int pos = binarySearch(Arr, neededNum, 0, arrayWidth - 1);
543. if (pos != -1)
544. showEl(pos);
545. else
546. infoLabel3->Visible = true;
547. }
548. }
549. System::Void Coursach::Lab2::btnBruteforceSearch\_Click(System::Object^ sender, System::EventArgs^ e)
550. {
551. hideInfo();
552. int neededNum = intTypeCheck(textBoxNum->Text);
553. if (neededNum < min)
554. return;
555. else {
556. int time = benchFunc4(arrayWidth, Arr, neededNum, bruteforce);
557. showTime(time, "нс");
558. int pos = bruteforce(arrayWidth, Arr, neededNum);
559. if (pos != -1)
560. showEl(pos);
561. else
562. infoLabel3->Visible = true;
563. }
564. }
565. System::Void Coursach::Lab2::btnSwap\_Click(System::Object^ sender, System::EventArgs^ e)
566. {
567. int a = indexTypeCheck(textBoxIndexA->Text, "поле 1");
568. int b = indexTypeCheck(textBoxIndexB->Text, "поле 2");
569. if (a < 0 || b < 0)
570. return;
571. else {
572. std::swap(Arr[a], Arr[b]);
573. showArrayInForm();
574. showEl(a);
575. showEl(b);
576. }
577. }

Приложение Б

Сравнение скорости работы разных методов сортировки

Для сравнения времени работы разных алгоритмов сортировки я сделала по десять измерений для каждой из них. Получившийся средний результат для наглядности я вывела в виде столбчатой диаграммы.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Bubble** | **Shaker** | **Comb** | **Insert** | **Quick** |
| 55 | 53 | 11 | 35 | 11 |
| 54 | 48 | 11 | 50 | 12 |
| 63 | 45 | 11 | 40 | 12 |
| 57 | 50 | 12 | 39 | 11 |
| 52 | 50 | 11 | 33 | 11 |
| 53 | 49 | 15 | 37 | 12 |
| 54 | 49 | 11 | 36 | 11 |
| 58 | 54 | 12 | 37 | 12 |
| 56 | 47 | 11 | 34 | 11 |
| 57 | 41 | 11 | 30 | 11 |

Таблица 1 — измерения времени выполнения разных алгоритмов сортировок в микросекундах.

Рисунок 24 — сравнение скорости работы разных видов сортировки

**ПРИЛОЖЕНИЕ В**

**ДЕМОНСТРАЦИЯ ОБРАБОТКИ ПОЛЬЗОВАТЕЛЬСКИХ ОШИБОК**

Под пользовательскими ошибками в данном случае я понимаю те, которые происходят из-за того, что пользователь забыл ввести значение в отведённое для этого поле или ввёл неправильное значение (вне требуемого диапазона или неправильного типа данных). Случай, когда алгоритм поиска не находит введённое пользователем значение, ошибкой не является.

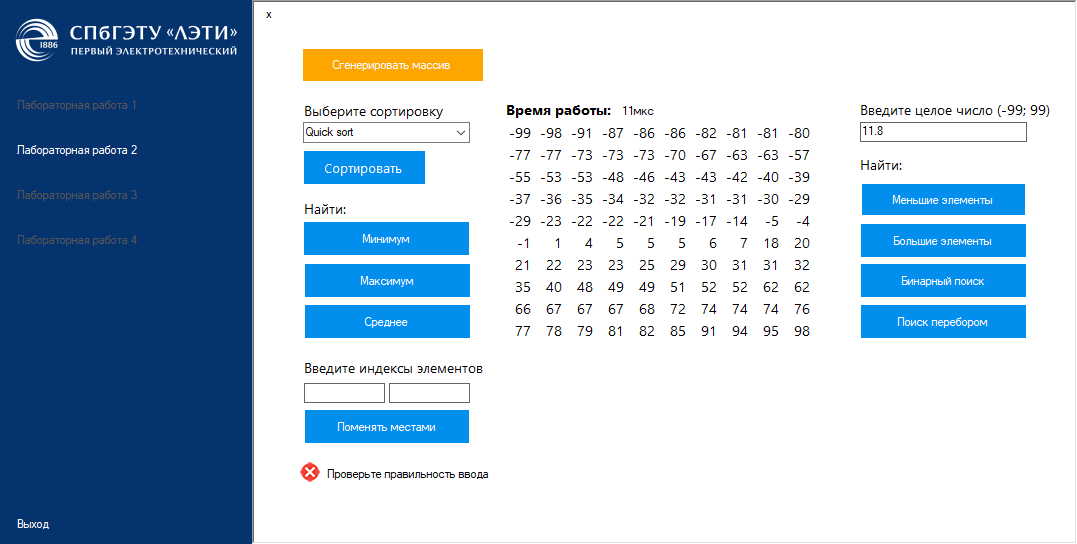


Рисунок 25 — попытка ввести в поле ввода число с плавающей точкой

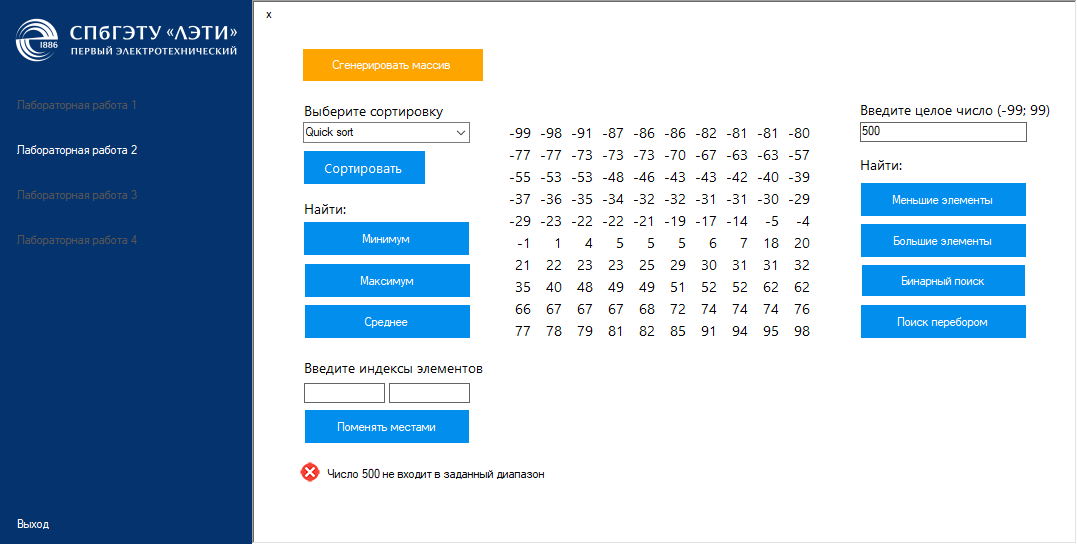


Рисунок 26 — попытка ввести в поле ввода число вне заданного диапазона

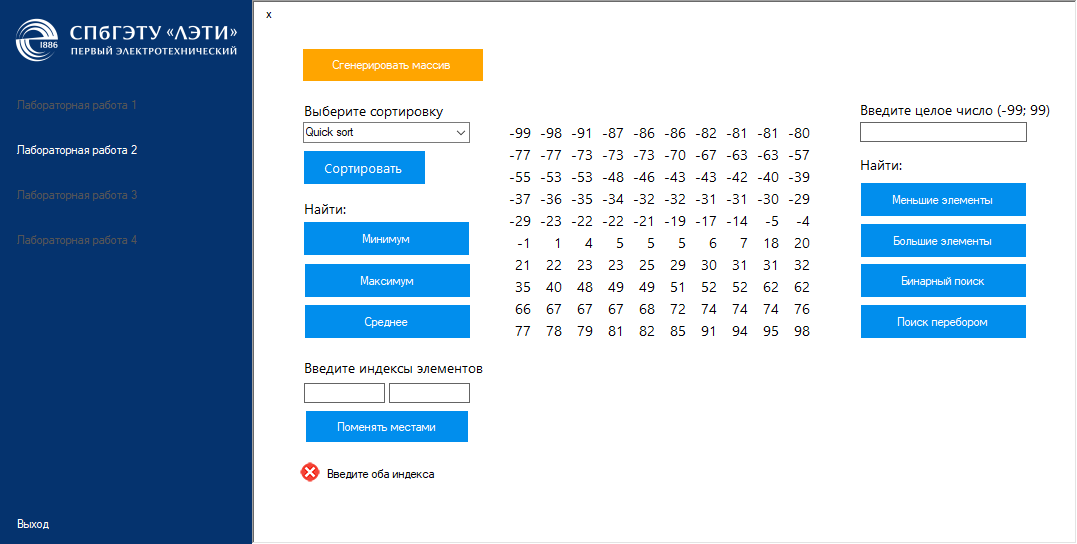


Рисунок 27 — В одно или оба поля не введён индекс для обмена элементов

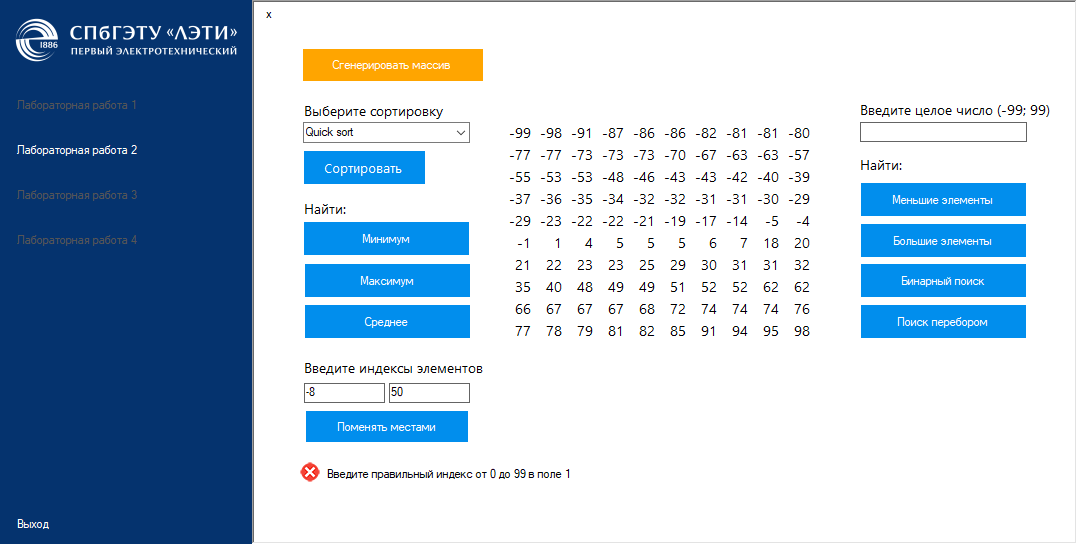


Рисунок 28 — В одно из полей введено значение вне заданного диапазона