Федеральное государственное автономное

Образовательное учреждение

Высшего образования

«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт Космических и информационных технологий

институт

Кафедра «Информатика»

кафедра

**ОТЧЕТ ПО ПРАКТИЧЕСКОЙ РАБОТЕ №4**

Усовершенствованные алгоритмы сортировки массивов

тема

Преподаватель \_\_\_\_\_\_\_\_\_ Р. Ю. Царев

подпись, дата инициалы, фамилия

Студент КИ16-16Б, №031620303 \_\_\_\_\_\_\_\_\_ В.В. Кокташев

Номер группы, зачетной книжки подпись, дата инициалы, фамилия

Красноярск 2018

# Цель работы

# Изучение усовершенствованных алгоритмов сортировки массивов и сравнения их эффективности по времени выполнения, числу перестановок и количеству сравнений.

# Задание

1. Реализовать в программе алгоритм шейкерной сортировки.
2. Реализовать в программе алгоритм быстрой сортировки.
3. Сравнить эффективность реализованных алгоритмов по времени выполнения**,** числу перестановок и количеству сравнений.

# Теория

**Шейкерная сортировка**

Сортировка перемешиванием, или Шейкерная сортировка, или двунаправленная — разновидность пузырьковой сортировки. Анализируя метод пузырьковой сортировки, можно отметить два обстоятельства.

Во-первых, если при движении по части массива перестановки не происходят, то эта часть массива уже отсортирована и, следовательно, её можно исключить из рассмотрения.

Во-вторых, при движении от конца массива к началу минимальный элемент «всплывает» на первую позицию, а максимальный элемент сдвигается только на одну позицию вправо.

Эти две идеи приводят к следующим модификациям в методе пузырьковой сортировки. Границы рабочей части массива (то есть части массива, где происходит движение) устанавливаются в месте последнего обмена на каждой итерации. Массив просматривается поочередно справа налево и слева направо.

**Анализ шейкерной сортировки.**Анализ  шейкерной сортировки довольно сложен. Стоит отметить, что усовершенствования не влияют на число перемещений, они лишь сокращают число излишних проверок. К несчастью, обмен местами двух элементов – чаще всего более дорогостоящая операция, чем их сравнение. Поэтому очевидные на первый взгляд улучшения дают не такой уж большой вы­игрыш, как ожидалось.

Шейкерная сортировка с успехом используется лишь в тех слу­чаях, когда известно, что элементы почти упорядо­чены, что на практике бывает весьма редко. Анализ показывает, что «обменная» сорти­ровка и ее усовершенствования фактически оказываются хуже сортировок с по­мощью включений и с помощью выбора.

**Быстрая сортировка**

Быстрая сортировка является существенно улучшенным вариантом алгоритма сортировки с помощью прямого обмена, известного, в том числе, своей низкой эффективностью. Принципиальное отличие состоит в том, что в первую очередь производятся перестановки на наибольшем возможном расстоянии и после каждого прохода элементы делятся на две независимые группы. Любопытный факт: улучшение самого неэффективного прямого метода сортировки дало в результате один из наиболее эффективных улучшенных методов.

Один из самых быстрых известных универсальных алгоритмов сортировки массивов: в среднем O (n\*log (n)) обменов при упорядочении n элементов; из-за наличия ряда недостатков на практике обычно используется с некоторыми доработками.

Общая идея алгоритма состоит в следующем:

Выбрать из массива элемент, называемый опорным. Это может быть любой из элементов массива. От выбора опорного элемента не зависит корректность алгоритма, но в отдельных случаях может сильно зависеть его эффективность (см.ниже).

Сравнить все остальные элементы с опорным и переставить их в массиве так, чтобы разбить массив на три непрерывных отрезка, следующие друг за другом: «меньшие опорного», «равные» и «большие».[1]

Для отрезков «меньших» и «больших» значений выполнить рекурсивно ту же последовательность операций, если длина отрезка больше единицы.

На практике массив обычно делят не на три, а на две части: например, «меньшие опорного» и «равные и большие»; такой подход в общем случае эффективнее, так как упрощает алгоритм разделения.

Быстрая сортировка относится к алгоритмам «разделяй и властвуй».

Алгоритм состоит из трёх шагов:

* Выбрать элемент из массива. Назовём его опорным.
* Разбиение: перераспределение элементов в массиве таким образом, что элементы меньше опорного помещаются перед ним, а больше или равные после.
* Рекурсивно применить первые два шага к двум подмассивам слева и справа от опорного элемента. Рекурсия не применяется к массиву, в котором только один или отсутствуют элементы.

# Ход работы

1. Было разработано настольное Windows-forms приложение, реализующее алгоритмы быстрой и шейкерной сортировки. Пользователь может не только добавлять и удалять элементы массива, но и изменять их (рисунки 1-5). Также, обеспечена устойчивость программы при любых воздействиях, задаваемых пользователем.

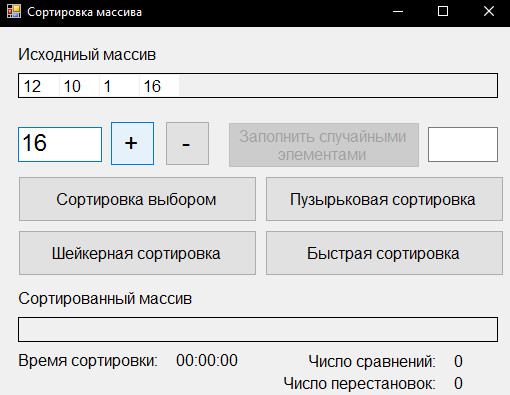


Рисунок 1 - Ввод элементов массива

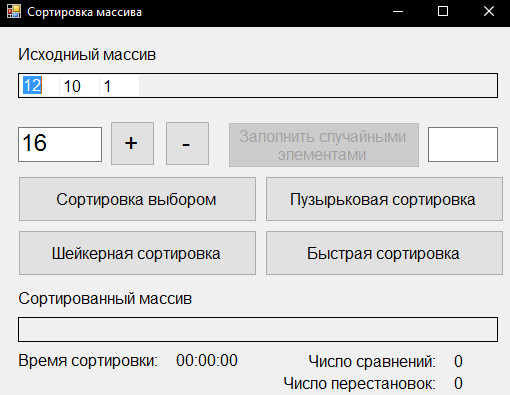


Рисунок 2 - Редактирование элементов массива

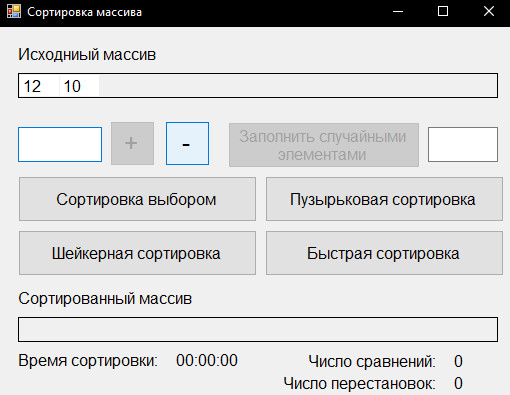


Рисунок 3 - Удаление элементов массива

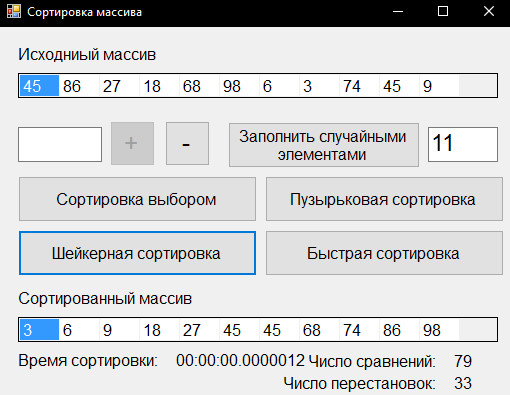


Рисунок 4 – Результат исполнения алгоритма шейкерной сортировки

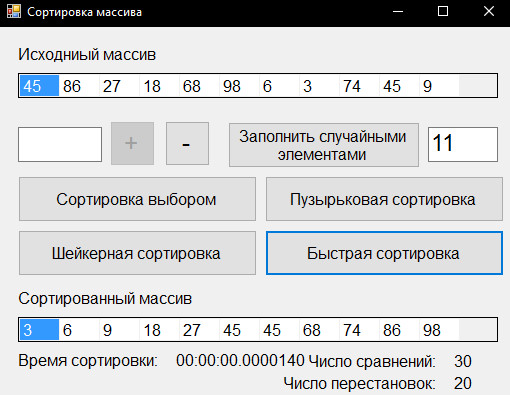


Рисунок 5 – Результат исполнения алгоритма быстрой сортировки

1. Проведено сравнение эффективности данных алгоритмов. Для каждого алгоритма в качестве примера использован массив, состоящий из 100 элементов. Каждый элемент массива имеет случайное значение (от 1 до 100) (рисунок 6). Для более точных результатов выполнение каждого алгоритма было рассмотрено на разных вариациях массивов, отличающихся как размерами, так и степенью отсортированности. Для каждого алгоритма программа высчитывает количество выполненных сравнений и перестановок, а также время его исполнения. На рисунках 7-8 можно проследить разницу в выполнении алгоритмов программой.

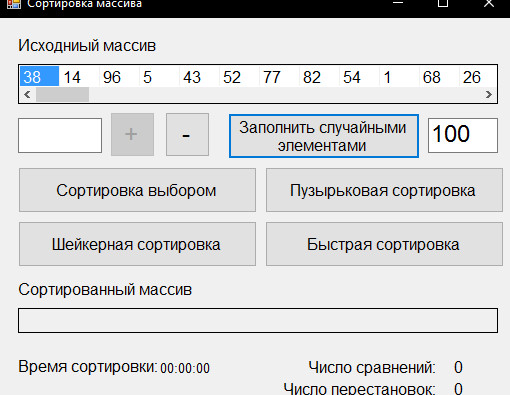


Рисунок 6 – Заполнение массива 100 случайными элементами

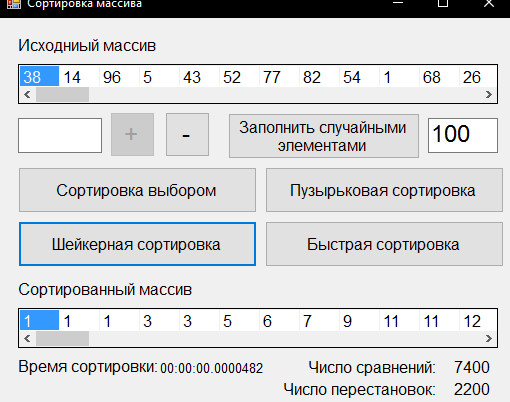


Рисунок 7 - Число сравнений и перестановок алгоритма шейкерной сортировки

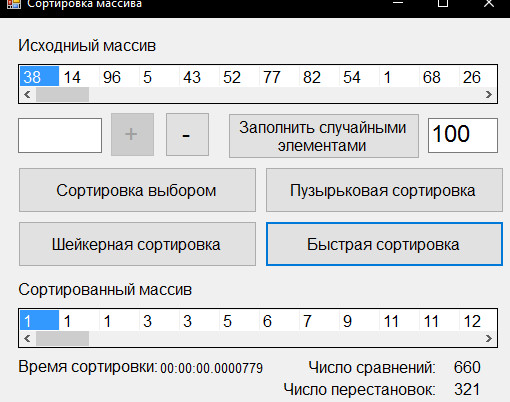


Рисунок 8 - Число сравнений и перестановок алгоритма быстрой сортировки

1. В результате, число сравнений и количество перестановок практически в ста процентах случаев случаев оказывается большим у шейкерной сортировки, по сравнению с быстрой. Однако время выполнения алгоритмов достаточно сильно варьируется, поэтому, по результатам проведённых экспериментов, нельзя с увереностью сказать выполнение какого алгоритма занимает меньшее количество времени. Время выполнения алгоритма быстрой сортировки сильно зависит от ***случайно*** выбираемых опорных элементов. Взяв во внимание все данные результаты сравнений, можно с точность заявить, что алгоритм быстрой сортировки является более эффективным алгоритмом сортировки, относительно алгоритма шейкерной сортировки.

# Вывод

В ходе практической работы были разработаны и отлажены алгоритмы быстрой и шейкерной. Результаты сравнения эффективности двух алгоритмов совпали с теоретическими ожиданиями. Алгоритм быстрой сортировки превзошел алгоритм сортировки.

# Исходный код программы

Листинг 1 - файл SortForm.cs

|  |
| --- |
| using System;  using System.Windows.Forms;  namespace AlgorithmsLabs  {  public partial class SortForm : Form  {  SortController Controller = new SortController();  public SortForm()  {  InitializeComponent();  }  private void btnAdd\_Click(object sender, EventArgs e)  {  Controller.AddElement(dgvArray, tbNumber.Text);  btnRemove.Enabled = true;  }  private void btnRemove\_Click(object sender, EventArgs e)  {  Controller.RemoveElement(dgvArray);  if (dgvArray.ColumnCount < 1)  btnRemove.Enabled = false;  }  private void btnRandom\_Click(object sender, EventArgs e)  {  int n = Convert.ToInt32(tbRandom.Text);  Controller.FillRandom(dgvArray, n);  if (n != 0)  btnRemove.Enabled = true;  else btnRemove.Enabled = false;  }  private void btnSort\_Click(object sender, EventArgs e)  {  Controller.SortArray(dgvArray, dgvSortedArray, (sender as Button).Name);  lblCompare.Text = Sort.GetCompareCount().ToString();  lblSwap.Text = Sort.GetSwapCount().ToString();  }  private void tbNumber\_TextChanged(object sender, EventArgs e)  {  btnAdd.Enabled = !String.IsNullOrEmpty(tbNumber.Text);  }  private void tbRandom\_TextChanged(object sender, EventArgs e)  {  btnRandom.Enabled = !String.IsNullOrEmpty(tbRandom.Text);  }  private void dgvArray\_CellValidating(object sender, DataGridViewCellValidatingEventArgs e)  {  int value;  string cellValue = e.FormattedValue.ToString();  if (!Int32.TryParse(cellValue, out value))  {  MessageBox.Show("Некорректное значение элемента массива!", "Ошибка", MessageBoxButtons.OK, MessageBoxIcon.Error);  e.Cancel = true;  }  }  private void NumberValidating(object sender, KeyPressEventArgs e)  {  char l = e.KeyChar;  if ((l < '0' || l > '9') && (l != '\b'))  e.Handled = true;  }  private void SortForm\_FormClosed(object sender, FormClosedEventArgs e)  {  Application.Exit();  }  }  } |

Листинг 2 - файл SortController.cs

|  |
| --- |
| using System;  using System.Drawing;  using System.Windows.Forms;  using System.Diagnostics;  namespace AlgorithmsLabs  {  class SortController  {  public void AddElement(DataGridView dgvArray, string Element)  {  int n = dgvArray.ColumnCount;  DataGridViewTextBoxColumn newCol = new DataGridViewTextBoxColumn();  newCol.Width = 40;  dgvArray.Columns.Add(newCol);  if (n > 10)  dgvArray.Size = new Size(dgvArray.Size.Width, 40);  if (n == 0)  dgvArray.Rows.Add(Element);  else dgvArray.Rows[0].Cells[n].Value = Element;  }  public void RemoveElement(DataGridView dgvArray)  {  int n = dgvArray.ColumnCount;  if (n > 0)  dgvArray.Columns.RemoveAt(n - 1);  if (n < 13)  dgvArray.Size = new Size(dgvArray.Size.Width, 25);  }  public void FillRandom(DataGridView dgvArray, int n)  {  Random rand = new Random();  if (dgvArray.ColumnCount != 0)  dgvArray.Columns.Clear();  for (int i = 0; i < n; i++)  {  DataGridViewTextBoxColumn newCol = new DataGridViewTextBoxColumn();  newCol.Width = 40;  dgvArray.Columns.Add(newCol);  if (i == 0)  dgvArray.Rows.Add(rand.Next(1, 100).ToString());  else dgvArray.Rows[0].Cells[i].Value = rand.Next(1, 100);  }  if (n > 11)  dgvArray.Size = new Size(dgvArray.Size.Width, 40);  else dgvArray.Size = new Size(dgvArray.Size.Width, 25);  }  public string SortArray(DataGridView dgvArray, DataGridView dgvSortedArray, string btnSort)  {  Sort.ResetCounters();  int[] Array = GetArray(dgvArray);  Stopwatch timer = new Stopwatch();  timer.Start();  CallSort(Array, btnSort);  timer.Stop();  SetSortedArray(dgvSortedArray, Array);  return timer.Elapsed.ToString();  }  private void CallSort(int[] Array, string SortName)  {  if (SortName == "btnSelectionSort")  Sort.SelectionSort(Array);  if (SortName == "btnBubbleSort")  Sort.BubbleSort(Array);  if (SortName == "btnShakerSort")  Sort.ShakerSort(Array);  if (SortName == "btnQuickSort")  Sort.QuickSort(Array, 0, Array.Length - 1);  }  private int[] GetArray(DataGridView dgvArray)  {  int n = dgvArray.ColumnCount;  int[] Array = new int[n];  for (int i = 0; i < n; i++)  Array[i] = Convert.ToInt32(dgvArray.Rows[0].Cells[i].Value);  return Array;  }  private void SetSortedArray(DataGridView dgvSortedArray, int[] Array)  {  if (dgvSortedArray.ColumnCount != 0)  dgvSortedArray.Columns.Clear();  for (int i = 0; i < Array.Length; i++)  {  DataGridViewTextBoxColumn newCol = new DataGridViewTextBoxColumn();  newCol.Width = 40;  dgvSortedArray.Columns.Add(newCol);  if (i == 0)  dgvSortedArray.Rows.Add(Array[i].ToString());  else dgvSortedArray.Rows[0].Cells[i].Value = Array[i];  }  if (Array.Length > 11)  dgvSortedArray.Size = new Size(dgvSortedArray.Size.Width, 40);  else dgvSortedArray.Size = new Size(dgvSortedArray.Size.Width, 25);  }  }  } |

Листинг 3 - файл Sort.cs

|  |
| --- |
| using System;  namespace AlgorithmsLabs  {  static class Sort  {  static private int CompareCount;  static private int SwapCount;  static public void ResetCounters()  {  CompareCount = 0;  SwapCount = 0;  }  static public int GetCompareCount()  {  return CompareCount;  }  static public int GetSwapCount()  {  return SwapCount;  }  static public void SelectionSort(int[] Array)  {  for (int i = 0; i < Array.Length - 1; i++)  {  int min = i;  for (int j = i + 1; j < Array.Length; j++, CompareCount++)  if (Array[j] < Array[min])  min = j;  Swap(ref Array[i], ref Array[min]);  }  }  static public void BubbleSort(int[] Array)  {  for (int i = 0; i < Array.Length - 1; i++)  for (int j = 0; j < Array.Length - i - 1; j++, CompareCount++)  if (Array[j] > Array[j + 1])  Swap(ref Array[j], ref Array[j + 1]);  }  static public void ShakerSort(int[] Array)  {  int Left = 0;  int Right = Array.Length - 1;  while (Left <= Right)  {  for (int i = Left; i < Right; i++, CompareCount++)  if (Array[i] > Array[i + 1])  Swap(ref Array[i], ref Array[i + 1]);  Right--;  CompareCount += Right - Left;  for (int i = Right; i > Left; i--, CompareCount++)  if (Array[i - 1] > Array[i])  Swap(ref Array[i - 1], ref Array[i]);  Left++;  }  }  static public void QuickSort(int[] Array, int Start, int End)  {  if (Start >= End)  return;  int Border = Start;  int Pivot = (new Random()).Next(Start, End + 1);  for (int i = Start; i <= End; i++, CompareCount++)  if (Array[i] < Array[Pivot])  {  Swap(ref Array[i], ref Array[Border]);  if (Border == Pivot)  Pivot = i;  Border++;  }  Swap(ref Array[Pivot], ref Array[Border]);  QuickSort(Array, Start, Border - 1);  QuickSort(Array, Border + 1, End);  }  static private void Swap(ref int a, ref int b)  {  int tmp = a;  a = b;  b = tmp;  SwapCount++;  }  }  } |