МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«Сибирский государственный университет науки и технологий
имени академика М.Ф. Решетнева»**

Институт информатики и телекоммуникаций

Кафедра информатики и вычислительной техники

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ

Алгоритмы и структуры данных

Лабораторная 2 - Сравнительный анализ эффективности методов сортировки данных во внешней памяти

Руководитель _____ В. В. Тынченко
подпись, дата инициалы, фамилия

Обучающийся БПИ20-02, 201219047 _____ Р. А. Сухачев
номер группы, зачетной книжки подпись, дата инициалы, фамилия

Красноярск 2021 г.

ЦЕЛЬ РАБОТЫ

Изучение алгоритмов сортировки данных во внешней памяти, особенностей их программной реализации и эффективности работы на различных наборах исходных данных.

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Ознакомиться с общей постановкой задачи.

2. Ознакомится с вариантом задания – соответствует вашему номеру в списке группы.

3. Написать и отладить программу.

4. Подготовить отчет по лабораторной работе.

5. Защитить лабораторную работу перед преподавателем.

ПОСТАНОВКА ЗАДАЧИ

Разработать программу, предназначенную для исследования времени работы и поведения методов сортировки, указанных в варианте задания. Провести исследование параметров работы указанных в варианте задания методов сортировки на различных наборах данных. Представить отчет, содержащий результаты исследования и полученные выводы.

Оценить быстродействие указанных методов и степень естественности их поведения.

ХОД РАБОТЫ

Программа содержит меню, позволяющее выбирать один из двух режимов работы программы:

1) сортировка файла данных, сформированных случайным образом;
2) режим накопления статистических данных

В первом режиме пользователю доступны следующие возможности:

- задавать размер числовой последовательности, содержащейся в файле;
- указывать диапазон значений элементов последовательности;
- выбирать метод внешней сортировки.

Результаты работы программы в данном режиме:

- вывести на экран количество сравнений и перестановок элементов массива.

Во втором режиме пользователь имеет возможность:

- выбирать способ формирования элементов последовательности, содержащейся в файле (случайные значения, упорядоченная последовательность значений, значения расположены в обратном порядке);
- задавать диапазон и шаг изменения размера последовательности;
- выбирать метод внешней сортировки.
- указывать диапазон значений элементов массива;
- выбирать метод сортировки массива.

Результаты работы программы в данном режиме:

- Для каждого значения размера формируется набор данных и сортируется выбранным методом. В файл с указанным именем выводятся значения времени сортировки для каждого количества элементов в наборе.

C.cpp:

```
#include <iostream>
#include <string>
#include <math.h>
using namespace std;

void Distribution(ifstream &in1, ofstream &out2, ofstream &out3, string FileName1,
string FileName2, string FileName3, int n1, int n2, int s)
{
    int b, c, count = 0;

    in1.open(FileName1);
    out2.open(FileName2, ios::trunc);
    out3.open(FileName3, ios::trunc);

    for (size_t i = 0; i < n1; i++)
    {
```

```
        for (size_t j = 0; j < n2; j++)
        {
            if (i % 2 == 0)
            {
                in1 >> b;
                out2 << b << endl;
                count++;
            }
            else
            {
                in1 >> c;
                out3 << c << endl;
                count++;
            }

            if (count == s)
            {
                j = n2;
                i = n1;
            }
        }
    }

    in1.close();
    out2.close();
    out3.close();
}

void Direct_Merge(ifstream &in2, ifstream &in3, ofstream &out1, string FileName1, string
FileName2, string FileName3, int n2, int s, int k, int countComparison, int countTrans-
position, int flag)
{
    int b, c, countB = 0, countC = 0, nb = n2 / 2, nc = n2 / 2, cB = 0, cC = 0, maxcB =
0, maxcC = 0, sz = s, halfn2 = n2 / 2;
    double siz = s;

    in2.open(FileName2);
    in3.open(FileName3);
    out1.open(FileName1, ios::trunc);

    for (size_t i = 0; i < ceil(siz / halfn2); i++)
    {
        if ((sz - halfn2 > 0) && (i % 2 == 0))
            maxcB += halfn2;
        else if ((sz - halfn2 > 0) && (i % 2 == 1))
            maxcC += halfn2;
        else if ((sz - halfn2 <= 0) && (i % 2 == 0))
            maxcB += sz;
        else if ((sz - halfn2 <= 0) && (i % 2 == 1))
            maxcC += sz;
        sz -= halfn2;
    }
```

```
    for (size_t i = 0; i < ceil(siz / halfn2); i++)
    {
        if ((cC != maxcC) && (cB != maxcB))
        {
            in2 >> b;
            in3 >> c;
        }
        else if (cB != maxcB)
        {
            in2 >> b;
        }
        else if (cC != maxcC)
        {
            in3 >> c;
        }

        for (size_t j = 0; j < n2; j++)
        {
            if ((countB != nb) && (countC != nc) && (cC != maxcC) && (cB != maxcB))
            {
                countComparison++;

                if (b <= c)
                {
                    out1 << b << endl;
                    countTransposition++;

                    countB++;
                    cB++;

                    if (countB != nb)
                        in2 >> b;
                }
                else
                {
                    out1 << c << endl;
                    countTransposition++;

                    countC++;
                    cC++;

                    if (countC != nc)
                        in3 >> c;
                }
            }
            else if ((countB != nb) && (cB != maxcB))
            {
                out1 << b << endl;
                countTransposition++;

                countB++;
                cB++;
```

```
                if (countB != nb)
                    in2 >> b;
            }
            else if ((countC != nc) && (cC != maxcC))
            {
                out1 << c << endl;
                countTransposition++;

                countC++;
                cC++;

                if (countC != nc)
                    in3 >> c;
            }
        }
        countB = 0;
        countC = 0;
    }

    in2.close();
    in3.close();
    out1.close();

    if (flag == 1)
    {
        cout << "Iteration №" << k + 1 << endl;
        cout << "Count comparison: " << countComparison << endl;
        cout << "Count transposition: " << countTransposition << endl;
    }
}

void Arrays(int *Array1, int *Array2, int Fib1, int Fib2)
{
    for (size_t i = 0; i < Fib1; i++)
    {
        if (i < Fib2)
        {
            int j = i + Fib1;
            Array2[i] = Array1[i] + Array1[j];
        }
        else
            Array2[i] = Array1[i];
    }
}

void InitialDistribution(ifstream &in1, ofstream &out2, ofstream &out3, string File-
Name1, string FileName2, string FileName3, int size, int Fib[], int limit)
{
    int b, c, count = 0;

    in1.open(FileName1);
    out2.open(FileName2, ios::trunc);
    out3.open(FileName3, ios::trunc);
```

```
    if (Fib[0] < Fib[1])
    {
        for (size_t i = 0; i < Fib[1]; i++)
        {
            in1 >> b;
            out2 << b << "\n\n";
        }

        for (size_t i = 0; i < Fib[0]; i++)
        {
            if (size - Fib[1] > count)
            {
                in1 >> c;
                out3 << c << "\n\n";
                count++;
            }
            else
                out3 << limit << "\n\n";
        }
    }
    else
    {
        for (size_t i = 0; i < Fib[0]; i++)
        {
            in1 >> b;
            out2 << b << "\n\n";
        }

        for (size_t i = 0; i < Fib[1]; i++)
        {
            if (size - Fib[0] > count)
            {
                in1 >> c;
                out3 << c << "\n\n";
                count++;
            }
            else
                out3 << limit << "\n\n";
        }
    }

    in1.close();
    out2.close();
    out3.close();
}

void Multiphase_Merge(int SeriesB, int SeriesC, ifstream &in2, ifstream &in3, ofstream
&out1, ofstream &out3, string FileName2, string FileName3, string FileName1, int s, int
k, int *Array, int limit, int countComparison, int countTransposition, int flag)
{
    int b, c, countB = 0, countC = 0, maxcB, maxcC;
```

```
    if (k % 2 == 0)
    {
        out1.open(FileName1, ios::trunc);
        in2.open(FileName2);
    }
    else
    {
        out1.open(FileName2, ios::trunc);
        in2.open(FileName1);
    }

    in3.open(FileName3);

    for (size_t i = 0; i < SeriesC; i++)
    {
        maxcB = Array[i];
        maxcC = Array[i + SeriesB];

        in2 >> b;
        in3 >> c;

        for (size_t j = 0; j < (maxcB + maxcC); j++)
        {
            if ((b <= c) && (countB != maxcB))
            {
                countComparison++;

                out1 << b << endl;
                countTransposition++;

                countB++;

                if (countB != maxcB)
                    in2 >> b;
            }
            else if ((c < b) && (c != limit) && (countC != maxcC))
            {
                countComparison++;

                out1 << c << endl;
                countTransposition++;

                countC++;

                if (countC != maxcC)
                    in3 >> c;
            }
            else if (countB != maxcB)
            {
                out1 << b << endl;
                countTransposition++;

                countB++;
```

```
            if (countB != maxcB)
                in2 >> b;
        }
        else if ((c != limit) && (countC != maxcC))
        {
            out1 << c << endl;
            countTransposition++;

            countC++;

            if (countC != maxcC)
                in3 >> c;
        }

        if (j == (maxcB + maxcC - 1))
        {
            out1 << "\n\n";

            in2.get();
            in2.get();

            in3.get();
            in3.get();
        }
    }
    countB = 0;
    countC = 0;
}
in3.close();
out1.close();

out3.open(FileName3, ios::trunc);

for (size_t i = SeriesC; i < SeriesB; i++)
{
    maxcB = Array[i];

    in2 >> b;

    for (size_t j = 0; j < maxcB; j++)
    {
        out3 << b << endl;
        countTransposition++;

        countB++;

        if (countB != maxcB)
            in2 >> b;

        if (j == (maxcB - 1))
        {
            out3 << "\n\n";
```

```
                in2.get();
                in2.get();
            }
        }
        countB = 0;
    }

    out3.close();
    in2.close();

    if (flag == 1)
    {
        cout << "Iteration №" << k + 1 << endl;
        cout << "Count comparisons: " << countComparison << endl;
        cout << "Count transposition: " << countTransposition << endl;
    }
}
```

Main.cpp:

```
#include <fstream>
#include <iostream>
using namespace std;
#include "C.cpp"
#include <ctime>

// Вариант №20.

int main()
{
  int answer = 0;
  while (answer != 10)
  {
    cout << "\n1 - Sorting one random array.\n2 - Accumulation of statistical data.\n";
    cout << "10 - EXIT.\nEnter: ";
    cin >> answer;
    cout << endl;
    switch (answer)
    {
    case 1:
    {
      long size = 0, dBorder = 0, hBorder = 0;
      string FileName1 = "A.txt", FileName2 = "B.txt", FileName3 = "C.txt";
      ifstream in1, in2, in3;
      ofstream out1, out2, out3;

      out1.open(FileName1, ios::trunc);
      if (!out1)
      {
        cout << "\n No file";
```

```
        return 1;
      }

      cout << "Enter number of the Array: ";
      cin >> size;
      cout << "Specify the range of values of the array elements.\nEnter the smallest
number: ";
      cin >> dBorder;
      cout << "Enter the biggest number: ";
      cin >> hBorder;
      cout << endl;

      for (long i = 0; i < size; i++)
      {
        long o;
        o = rand() % (hBorder - dBorder + 1) + dBorder;
        out1 << o << endl;
      }
      cout << "\nDone.\n"
           << endl;

      out1.close();

      int answer2 = 0;
      while (answer2 != 10)
      {
        cout << "\n1 - Sorting an array by DirectMerge.\n2 - Sorting an array by POLY-
PHASE.\n";
        cout << "10 - BACK.\nEnter: ";
        cin >> answer2;
        cout << endl;
        switch (answer2)
        {
        case 1:
        {
          int n1, n2 = 2, degreeoftwo = 1, size_two, k = 0, countComparison = 0,
countTransposition = 0;

          while (pow(2, degreeoftwo) < size)
            degreeoftwo++;

          size_two = pow(2, degreeoftwo);
          n1 = size_two;

          for (n2 = 2; n2 <= size_two; n2 *= 2)
          {
            Distribution(in1, out2, out3, FileName1, FileName2, FileName3, n1, n2 / 2,
size);
            Direct_Merge(in2, in3, out1, FileName1, FileName2, FileName3, n2, size, k,
countComparison, countTransposition, 1);

            n1 /= 2;
```

```
            k++;
          }
        }
        break;

        case 2:
        {
          int k = 0, countComparison = 0, countTransposition = 0;

          int *Fib = new int[2];
          Fib[0] = 0;
          Fib[1] = 1;

          while (Fib[0] + Fib[1] < size)
            if (Fib[0] < Fib[1])
              Fib[0] += Fib[1];
            else
              Fib[1] += Fib[0];

          int *Array1 = new int[(Fib[0] + Fib[1])];
          int *Array2 = new int[Fib[1]];

          for (size_t i = 0; i < (Fib[0] + Fib[1]); i++)
          {
            Array1[i] = 1;

            if (i >= size)
              Array1[i] = 0;
          }

          InitialDistribution(in1, out2, out3, FileName1, FileName2, FileName3, size,
Fib, -1);

          while (Fib[0] + Fib[1] != 1)
          {
            if (k % 2 == 0)
            {
              delete[] Array2;

              if (Fib[0] < Fib[1])
              {
                Multiphase_Merge(Fib[1], Fib[0], in2, in3, out1, out3, FileName2, File-
Name3, FileName1, size, k, Array1, -1, countComparison, countTransposition, 1);

                Array2 = new int[Fib[1]];
                Arrays(Array1, Array2, Fib[1], Fib[0]);

                Fib[1] -= Fib[0];
              }
              else
              {
                Multiphase_Merge(Fib[0], Fib[1], in2, in3, out1, out3, FileName2, File-
Name3, FileName1, size, k, Array1, -1, countComparison, countTransposition, 1);
```

```
              Array2 = new int[Fib[0]];
              Arrays(Array1, Array2, Fib[0], Fib[1]);

              Fib[0] -= Fib[1];
            }
          }
          else
          {
            delete[] Array1;

            if (Fib[0] < Fib[1])
            {
              Multiphase_Merge(Fib[1], Fib[0], in2, in3, out1, out3, FileName2, File-
Name3, FileName1, size, k, Array1, -1, countComparison, countTransposition, 1);

              Array1 = new int[Fib[1]];
              Arrays(Array2, Array1, Fib[1], Fib[0]);

              Fib[1] -= Fib[0];
            }
            else
            {
              Multiphase_Merge(Fib[0], Fib[1], in2, in3, out1, out3, FileName2, File-
Name3, FileName1, size, k, Array1, -1, countComparison, countTransposition, 1);

              Array1 = new int[Fib[0]];
              Arrays(Array2, Array1, Fib[0], Fib[1]);

              Fib[0] -= Fib[1];
            }
          }
          k++;
        }
        delete[] Array1;
        delete[] Array2;
      }
      break;
      }
    }
    size = 0;
    out1.close();
  }
  break;

  case 2:
  {
    long size = 0, dBorder = 0, hBorder = 0, s = 0;
    string FileName1 = "A.txt", FileName2 = "B.txt", FileName3 = "C.txt", FileNameTime
= "Time.txt";
    ifstream in1, in2, in3;
    ofstream out, out1, out2, out3;
    int lim;
```

```
      cout << "Enter number of the Array: ";
      cin >> size;
      cout << "Specify the range of values of the array elements.\nEnter the smallest
number: ";
      cin >> dBorder;
      cout << "Enter the biggest number: ";
      cin >> hBorder;
      cout << "Enter the step of the creation an array: ";
      cin >> s;
      cout << endl;

      int answer2 = 0;
      while (answer2 != 10)
      {
        cout << "\n1 - Create a random array.";
        cout << "\n2 - Create a straight order array.";
        cout << "\n3 - Create a reversed order array.";
        cout << "\n10 - BACK.\nEnter: ";
        cin >> answer2;
        cout << endl;
        switch (answer2)
        {
        case 1:
        {
          out1.open(FileName1, ios::trunc);

          for (long i = 0; i < size; i++)
          {
            long o;
            o = rand() % (hBorder - dBorder + 1) + dBorder;
            out1 << o << endl;
          }
          cout << "\nDone.\n"
               << endl;

          out1.close();

          int answer3 = 0;
          cout << "\n1 - DirectMerge.\n2 - MultiPhaseMerge.\nEnter: ";
          cin >> answer3;
          cout << endl;
          switch (answer3)
          {
          case 1:
          {
            out.open(FileNameTime, ios::trunc);

            for (long c = s; c <= size; c += s)
            {
              int n1, n2 = 2, degreeoftwo = 1, size_two, k = 0, countComparison = 0,
countTransposition = 0;
```

```
              while (pow(2, degreeoftwo) < c)
                degreeoftwo++;

              size_two = pow(2, degreeoftwo);
              n1 = size_two;

              clock_t t0 = clock();
              for (n2 = 2; n2 <= size_two; n2 *= 2)
              {
                Distribution(in1, out2, out3, FileName1, FileName2, FileName3, n1, n2 /
2, c);
                Direct_Merge(in2, in3, out1, FileName1, FileName2, FileName3, n2, c, k,
countComparison, countTransposition, 2);

                n1 /= 2;

                k++;
              }
              clock_t t1 = clock();

              out << "Sorting by DirectMerge" << endl
                  << "Elements in array: " << c << endl
                  << "Time: " << (long double)(t1 - t0) / CLOCKS_PER_SEC << endl
                  << endl;
            }
          }
          break;

          case 2:
          {
            out.open(FileNameTime, ios::trunc);

            int k = 0, countComparison = 0, countTransposition = 0;

            int *Fib = new int[2];
            Fib[0] = 0;
            Fib[1] = 1;

            for (long c = s; c <= size; c += s)
            {
              while (Fib[0] + Fib[1] < c)
                if (Fib[0] < Fib[1])
                  Fib[0] += Fib[1];
                else
                  Fib[1] += Fib[0];

              int *Array1 = new int[(Fib[0] + Fib[1])];
              int *Array2 = new int[Fib[1]];

              for (size_t i = 0; i < (Fib[0] + Fib[1]); i++)
              {
                Array1[i] = 1;
```

```
            if (i >= c)
              Array1[i] = 0;
          }

          InitialDistribution(in1, out2, out3, FileName1, FileName2, FileName3, c,
Fib, -1);

          clock_t t0 = clock();
          while (Fib[0] + Fib[1] != 1)
          {
            if (k % 2 == 0)
            {
              delete[] Array2;

              if (Fib[0] < Fib[1])
              {
                Multiphase_Merge(Fib[1], Fib[0], in2, in3, out1, out3, FileName2,
FileName3, FileName1, c, k, Array1, -1, countComparison, countTransposition, 1);

                Array2 = new int[Fib[1]];
                Arrays(Array1, Array2, Fib[1], Fib[0]);

                Fib[1] -= Fib[0];
              }
              else
              {
                Multiphase_Merge(Fib[0], Fib[1], in2, in3, out1, out3, FileName2,
FileName3, FileName1, c, k, Array1, -1, countComparison, countTransposition, 1);

                Array2 = new int[Fib[0]];
                Arrays(Array1, Array2, Fib[0], Fib[1]);

                Fib[0] -= Fib[1];
              }
            }
            else
            {
              delete[] Array1;

              if (Fib[0] < Fib[1])
              {
                Multiphase_Merge(Fib[1], Fib[0], in2, in3, out1, out3, FileName2,
FileName3, FileName1, c, k, Array1, -1, countComparison, countTransposition, 1);

                Array1 = new int[Fib[1]];
                Arrays(Array2, Array1, Fib[1], Fib[0]);

                Fib[1] -= Fib[0];
              }
              else
              {
                Multiphase_Merge(Fib[0], Fib[1], in2, in3, out1, out3, FileName2,
FileName3, FileName1, c, k, Array1, -1, countComparison, countTransposition, 1);
```

```
                    Array1 = new int[Fib[0]];
                    Arrays(Array2, Array1, Fib[0], Fib[1]);

                    Fib[0] -= Fib[1];
                  }
                }
                k++;
              }
              clock_t t1 = clock();

              delete[] Array1;
              delete[] Array2;

              out << "Sorting by MultiPhaseMerge" << endl
                  << "Elements in array: " << c << endl
                  << "Time: " << (long double)(t1 - t0) / CLOCKS_PER_SEC << endl
                  << endl;
            }
          }
          break;
          }
        }
        break;

        case 2:
        {
          out1.open(FileName1, ios::trunc);

          while (dBorder < hBorder)
          {
            out1 << dBorder << endl;
            dBorder++;
          }
          cout << "\nDone.\n"
               << endl;

          out1.close();

          int answer3 = 0;
          cout << "\n1 - DirectMerge.\n2 - MultiPhaseMerge.\nEnter: ";
          cin >> answer3;
          cout << endl;
          switch (answer3)
          {
          case 1:
          {
            out.open(FileNameTime, ios::trunc);

            for (long c = s; c <= size; c += s)
            {
              int n1, n2 = 2, degreeoftwo = 1, size_two, k = 0, countComparison = 0,
countTransposition = 0;
```

```
            while (pow(2, degreeoftwo) < c)
              degreeoftwo++;

            size_two = pow(2, degreeoftwo);
            n1 = size_two;

            clock_t t0 = clock();
            for (n2 = 2; n2 <= size_two; n2 *= 2)
            {
              Distribution(in1, out2, out3, FileName1, FileName2, FileName3, n1, n2 /
2, c);
              Direct_Merge(in2, in3, out1, FileName1, FileName2, FileName3, n2, c, k,
countComparison, countTransposition, 2);

              n1 /= 2;

              k++;
            }
            clock_t t1 = clock();

            out << "Sorting by DirectMerge" << endl
                << "Elements in array: " << c << endl
                << "Time: " << (long double)(t1 - t0) / CLOCKS_PER_SEC << endl
                << endl;
          }
        }
        break;

        case 2:
        {
          out.open(FileNameTime, ios::trunc);

          int k = 0, countComparison = 0, countTransposition = 0;

          int *Fib = new int[2];
          Fib[0] = 0;
          Fib[1] = 1;

          for (long c = s; c <= size; c += s)
          {
            while (Fib[0] + Fib[1] < c)
              if (Fib[0] < Fib[1])
                Fib[0] += Fib[1];
              else
                Fib[1] += Fib[0];

            int *Array1 = new int[(Fib[0] + Fib[1])];
            int *Array2 = new int[Fib[1]];

            for (size_t i = 0; i < (Fib[0] + Fib[1]); i++)
            {
              Array1[i] = 1;
```

```
              if (i >= c)
                Array1[i] = 0;
            }

            InitialDistribution(in1, out2, out3, FileName1, FileName2, FileName3, c,
Fib, -1);

            clock_t t0 = clock();
            while (Fib[0] + Fib[1] != 1)
            {
              if (k % 2 == 0)
              {
                delete[] Array2;

                if (Fib[0] < Fib[1])
                {
                  Multiphase_Merge(Fib[1], Fib[0], in2, in3, out1, out3, FileName2,
FileName3, FileName1, c, k, Array1, -1, countComparison, countTransposition, 1);

                  Array2 = new int[Fib[1]];
                  Arrays(Array1, Array2, Fib[1], Fib[0]);

                  Fib[1] -= Fib[0];
                }
                else
                {
                  Multiphase_Merge(Fib[0], Fib[1], in2, in3, out1, out3, FileName2,
FileName3, FileName1, c, k, Array1, -1, countComparison, countTransposition, 1);

                  Array2 = new int[Fib[0]];
                  Arrays(Array1, Array2, Fib[0], Fib[1]);

                  Fib[0] -= Fib[1];
                }
              }
              else
              {
                delete[] Array1;

                if (Fib[0] < Fib[1])
                {
                  Multiphase_Merge(Fib[1], Fib[0], in2, in3, out1, out3, FileName2,
FileName3, FileName1, c, k, Array1, -1, countComparison, countTransposition, 1);

                  Array1 = new int[Fib[1]];
                  Arrays(Array2, Array1, Fib[1], Fib[0]);

                  Fib[1] -= Fib[0];
                }
                else
                {
```

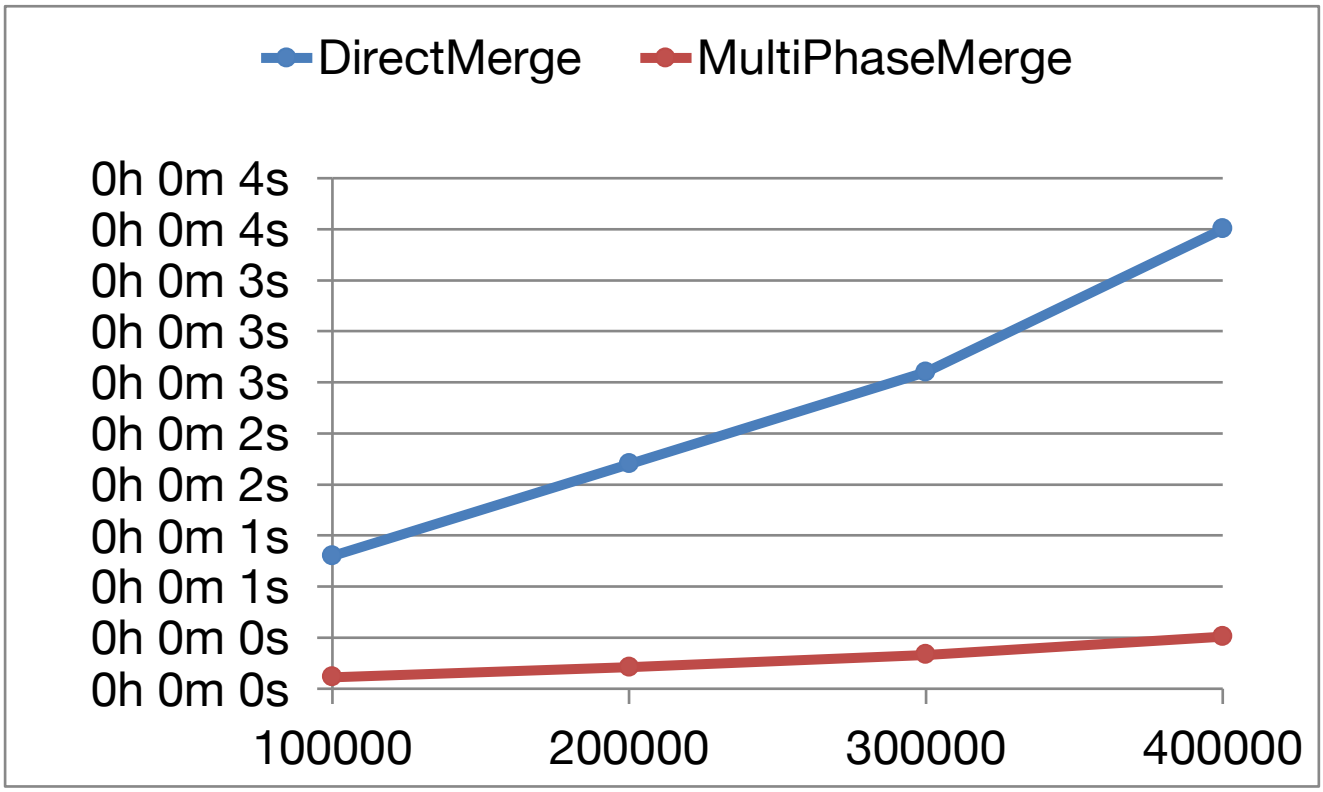
```
                    Multiphase_Merge(Fib[0], Fib[1], in2, in3, out1, out3, FileName2,
FileName3, FileName1, c, k, Array1, -1, countComparison, countTransposition, 1);

                    Array1 = new int[Fib[0]];
                    Arrays(Array2, Array1, Fib[0], Fib[1]);

                    Fib[0] -= Fib[1];
                  }
                }
                k++;
              }
              clock_t t1 = clock();

              delete[] Array1;
              delete[] Array2;

              out << "Sorting by MultiPhaseMerge" << endl
                  << "Elements in array: " << c << endl
                  << "Time: " << (long double)(t1 - t0) / CLOCKS_PER_SEC << endl
                  << endl;
            }
          }
          break;
          }
        }
        break;

        case 3:
        {
          out.open(FileNameTime, ios::trunc);

          while (hBorder > dBorder)
          {
            out1 << hBorder << endl;
            hBorder--;
          }
          cout << "\nDone.\n"
               << endl;

          out1.close();

          int answer3 = 0;
          cout << "\n1 - DirectMerge.\n2 - MultiPhaseMerge.\nEnter: ";
          cin >> answer3;
          cout << endl;
          switch (answer3)
          {
          case 1:
          {
            out.open(FileNameTime, ios::trunc);

            for (long c = s; c <= size; c += s)
            {
```

```
              int n1, n2 = 2, degreeoftwo = 1, size_two, k = 0, countComparison = 0,
countTransposition = 0;

              while (pow(2, degreeoftwo) < c)
                degreeoftwo++;

              size_two = pow(2, degreeoftwo);
              n1 = size_two;

              clock_t t0 = clock();
              for (n2 = 2; n2 <= size_two; n2 *= 2)
              {
                Distribution(in1, out2, out3, FileName1, FileName2, FileName3, n1, n2 /
2, c);
                Direct_Merge(in2, in3, out1, FileName1, FileName2, FileName3, n2, c, k,
countComparison, countTransposition, 2);

                n1 /= 2;

                k++;
              }
              clock_t t1 = clock();

              out << "Sorting by DirectMerge" << endl
                  << "Elements in array: " << c << endl
                  << "Time: " << (long double)(t1 - t0) / CLOCKS_PER_SEC << endl
                  << endl;
            }
          }
          break;

          case 2:
          {
            out.open(FileNameTime, ios::trunc);

            int k = 0, countComparison = 0, countTransposition = 0;

            int *Fib = new int[2];
            Fib[0] = 0;
            Fib[1] = 1;

            for (long c = s; c <= size; c += s)
            {
              while (Fib[0] + Fib[1] < c)
                if (Fib[0] < Fib[1])
                  Fib[0] += Fib[1];
                else
                  Fib[1] += Fib[0];

              int *Array1 = new int[(Fib[0] + Fib[1])];
              int *Array2 = new int[Fib[1]];

              for (size_t i = 0; i < (Fib[0] + Fib[1]); i++)
```

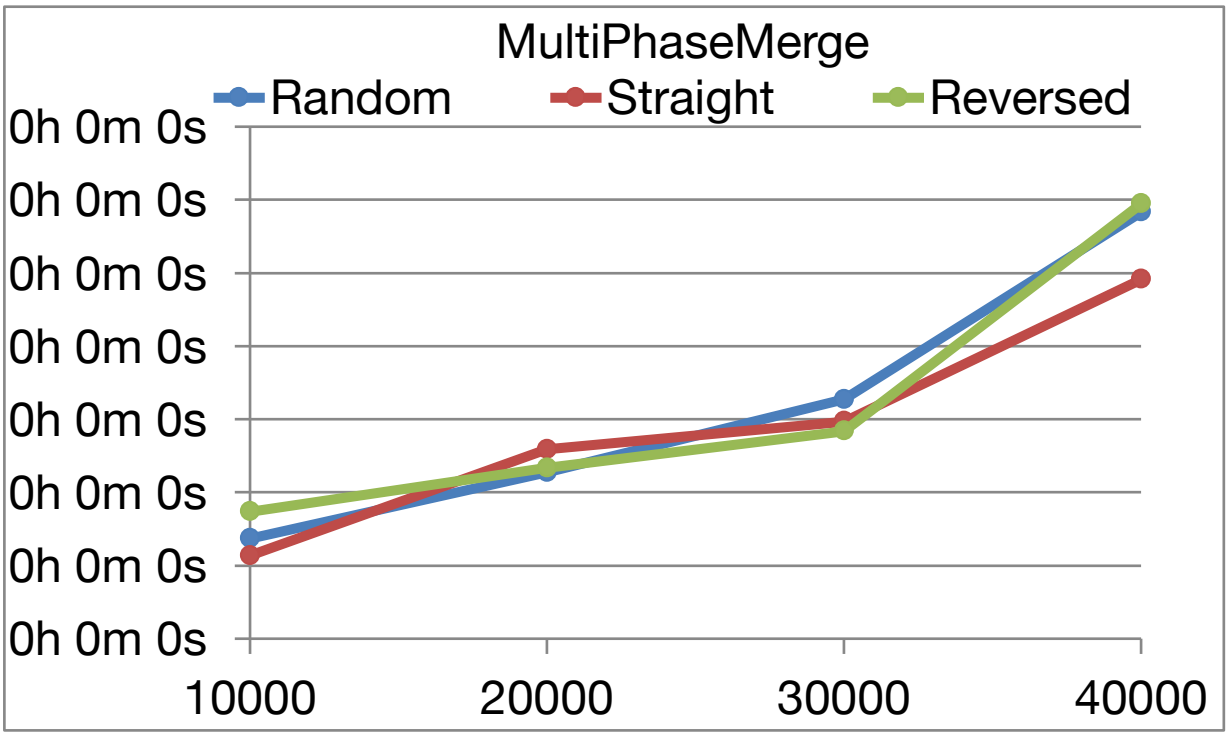
```
            {
              Array1[i] = 1;

              if (i >= c)
                Array1[i] = 0;
            }

            InitialDistribution(in1, out2, out3, FileName1, FileName2, FileName3, c,
Fib, -1);

            clock_t t0 = clock();
            while (Fib[0] + Fib[1] != 1)
            {
              if (k % 2 == 0)
              {
                delete[] Array2;

                if (Fib[0] < Fib[1])
                {
                  Multiphase_Merge(Fib[1], Fib[0], in2, in3, out1, out3, FileName2,
FileName3, FileName1, c, k, Array1, -1, countComparison, countTransposition, 1);

                  Array2 = new int[Fib[1]];
                  Arrays(Array1, Array2, Fib[1], Fib[0]);

                  Fib[1] -= Fib[0];
                }
                else
                {
                  Multiphase_Merge(Fib[0], Fib[1], in2, in3, out1, out3, FileName2,
FileName3, FileName1, c, k, Array1, -1, countComparison, countTransposition, 1);

                  Array2 = new int[Fib[0]];
                  Arrays(Array1, Array2, Fib[0], Fib[1]);

                  Fib[0] -= Fib[1];
                }
              }
              else
              {
                delete[] Array1;

                if (Fib[0] < Fib[1])
                {
                  Multiphase_Merge(Fib[1], Fib[0], in2, in3, out1, out3, FileName2,
FileName3, FileName1, c, k, Array1, -1, countComparison, countTransposition, 1);

                  Array1 = new int[Fib[1]];
                  Arrays(Array2, Array1, Fib[1], Fib[0]);

                  Fib[1] -= Fib[0];
                }
                else
```

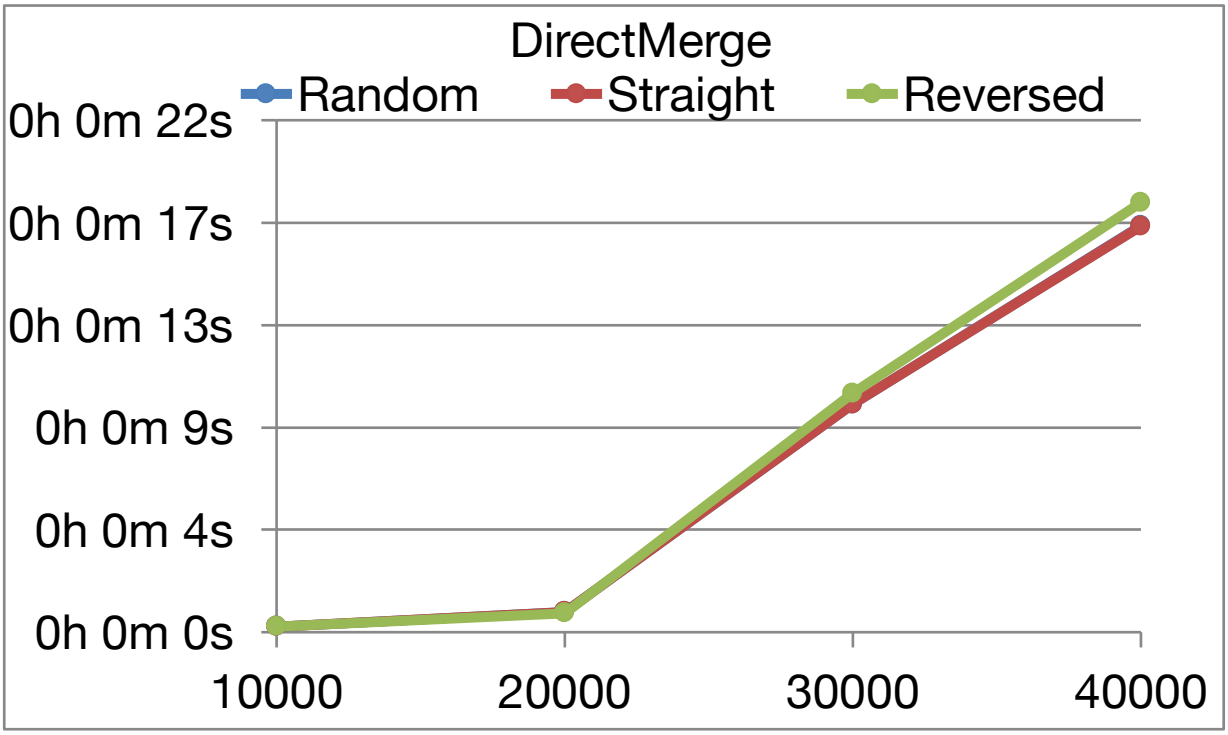
```
                {
                  Multiphase_Merge(Fib[0], Fib[1], in2, in3, out1, out3, FileName2,
FileName3, FileName1, c, k, Array1, -1, countComparison, countTransposition, 1);

                  Array1 = new int[Fib[0]];
                  Arrays(Array2, Array1, Fib[0], Fib[1]);

                  Fib[0] -= Fib[1];
                }
              }
              k++;
            }
            clock_t t1 = clock();

            delete[] Array1;
            delete[] Array2;

            out << "Sorting by MultiPhaseMerge" << endl
                << "Elements in array: " << c << endl
                << "Time: " << (long double)(t1 - t0) / CLOCKS_PER_SEC << endl
                << endl;
          }
        }
        break;
        }
      }
      break;
      }
    }
    break;
    }
  }
  return 0;
}
```

Вывод: сортировка прямым слиянием работает немного хуже многофазной сортировки с увеличением кол-ва элементов массива.



2. Для каждого алгоритма сортировки из варианта построить по три графика зависимости времени сортировки от количества элементов в массиве, формируя массив тремя различными способами: случайные значения, упорядоченная последовательность значений, значения расположены в обратном порядке. Три графика для одного алгоритма располагаются на одной координатной плоскости. На основании полученных данных сделать вывод о степени естественности поведения каждого из алгоритмов сортировки.





Вывод: сортировка прямым слиянием ведет себя естественно. Многофазная сортировка тоже ведет себя естественно.

ВЫВОДЫ

Изучил алгоритмы сортировки данных во внешней памяти, особенности их программной реализации и эффективности работы на различных наборах исходных данных.