Санкт-Петербургский Государственный Университет
Факультет Прикладной математики — процессов управления

Проектная работа

«Сравнительный анализ реализаций алгоритма быстрой сортировки»

Выполнил студент группы 19Б12-пу
Коломиец Александр Алексеевич

Санкт-Петербург 2020

Оглавление:

1. Описание и постановка задачи

2. План выполнения задачи
    - Выбор языков
    - Порядок работы

3. Определение терминов

4. Представление результатов

5. Отклонение от асимптотической оценки

6. Точное представление полученных результатов

7. Выводы

8. Практическое применение исследования

9. Список использованных источников

10. Приложение
    - Описание алгоритма быстрой сортировки
    - Программные коды
        - Реализации алгоритмов быстрой сортировки
        - Анализ полученных данных
        - Пример практического применения

Описание и постановка задачи:

Один из наиболее часто применяемых на практике алгоритмов сортировки — быстрая сортировка.
Ключевой шаг алгоритма заключается в выборе опорного значения, и в зависимости от идеи, заложенной в это действие, существуют различные реализации алгоритма.
Предполагается реализовать несколько вариантов алгоритма, а затем измерить время их работы на разном количестве входных элементов, и на основе полученных данных выявить наиболее эффективные реализации алгоритма в зависимости от количества входных элементов.
Предполагается изобразить результаты графически, посчитать отклонение от среднего ожидаемого времени, при помощи интерполяции аппроксимировать функцию и сравнить полученные результаты с теоретической оценкой сложности.
В качестве результата требуется представить вывод, на каком количестве входных данных какая из реализаций алгоритма показывает себя наиболее эффективной.

План выполнения задачи:

Выбор языков:

Для реализации алгоритмов предполагается использование языка Java SE 8.
Выбор языка обусловлен следующими факторами:
1. Популярность языка, а следовательно и практическая ценность полученных результатов работы.
2. Средние показатели по скорости и использованию памяти, что позволит ещё больше усреднить полученные результаты.
3. Использование виртуальной машины позволит максимально дистанцироваться от выбора конкретной машины для тестирования
4. Удобства, предоставляемые языком, такие как автоматический сборщик мусора, помогут сосредоточиться на поставленной задаче, а не тонкостях реализации алгоритма

Для проведения анализа полученных данных предполагается использование языка MatLab.
Выбор языка обусловлен следующими факторами:
1. MatLab является одной из старейших, тщательно проработанных и проверенных временем систем автоматизации математических расчетов
2. Эффективная и удобная реализация алгоритмов интерполяции
3. Высокая скорость численных вычислений
4. Наличие имеет библиотеки Image Processing Toolbox, которая обеспечивает широкий спектр функций, поддерживающих визуализацию проводимых вычислений непосредственно из среды MatLab
5. Наличие набора инструментов для создания математических моделей динамических систем, основанных на наблюдаемых входных/выходных данных.

Порядок работы:

В качестве вариантов выбора опорного элемента предполагаются следующие:
1. Произвольный элемент массива данных
2. Элемент на фиксированной позиции в массиве
3. Элемент, вычисляемый как медиана по трём элементам (первому, среднему и последнему)

Набор входных чисел предполагается случайно генерировать один раз перед проведением тестирования и затем использовать для всех реализаций.
Данные о времени тестов сохранить в файл.

1. При помощи языка MatLab требуется прочитать файл, и на основе полученных данных построить графики зависимости времени от количества входных данных для каждой из реализаций.
2. Изобразить графически теоретическое среднее время работы сравнить с полученными ранее графиками.
3. Посчитать максимальное и минимальное отклонения, среднее отклонение, худший и лучший случай в целом.
4. При помощи интерполяции построить функцию, задающую график зависимости времени от количества входных данных для каждой из реализаций, сравнить с функцией, описывающей теоретическое среднее время.

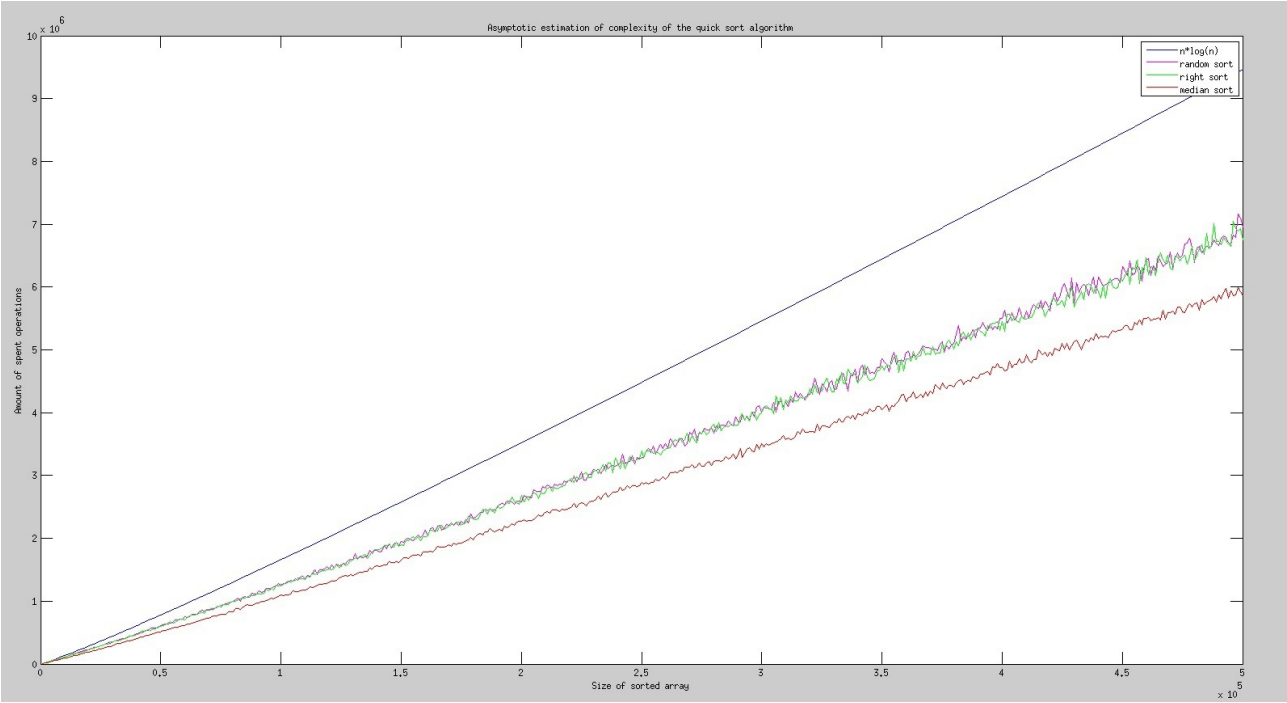
Определение терминов

Известно, что асимптотическая оценка сложности работы алгоритма быстрой сортировки составляет O(n*log(n)), где n – количество элементов для сортировки.
Требуется пояснить, что асимптотическая оценка не является оценкой времени выполнения, но связана с ней формулой T=KN, где K является коэффициентом, отвечающим скорости микропроцессора, эффективности сгенерированного компилятором кода и других технических факторов.
Асимптотическая оценка аппроксимирует количество логических операций, затрачиваемых алгоритмом для выполнения поставленной задачи. Логическая операция может не соответствовать одной машинной команде и описывает блок логически объединённых действий.
Следующим фактором, на который стоит обратить внимание, является выбранная модель памяти. Наиболее часто используемой моделью является так называемая RAM-модель, представляющая собой единый абстрактный блок памяти, где каждому элементу ставится в соответствие его адрес.
Учитывая выбранную модель памяти, предлагается использовать следующие логические операции для оценки сложности работы алгоритма:
1. Сравнение двух элементов массива
2. Копирование одного из элементов
3. Перемена двух элементов местами

Сбор данных

Следующим шагом после реализации трёх различных вариантов алгоритма быстрой сортировки является подготовка и сбор данных.
Сортировку предлагается проводить на различных массивах случайных целых чисел размером от 1000 до 500000 элементов с разницей между двумя соседними массивами в 1000 элементов.
Сортировки проверяются на одинаковых массивах для обеспечения наиболее объективных результатов.
Для большей точности полученных данных и избавления от случайных выбросов предлагается провести несколько тестирований, а затем усреднить полученные значения.
Так, на первом этапе выполнения проекта, было проведено 10 запусков программы с последующим подсчётом среднего арифметического значения полученных данных.
Посчитанные данные сохраняются в файл, а затем используются для дальнейшего анализа.

Представление результатов

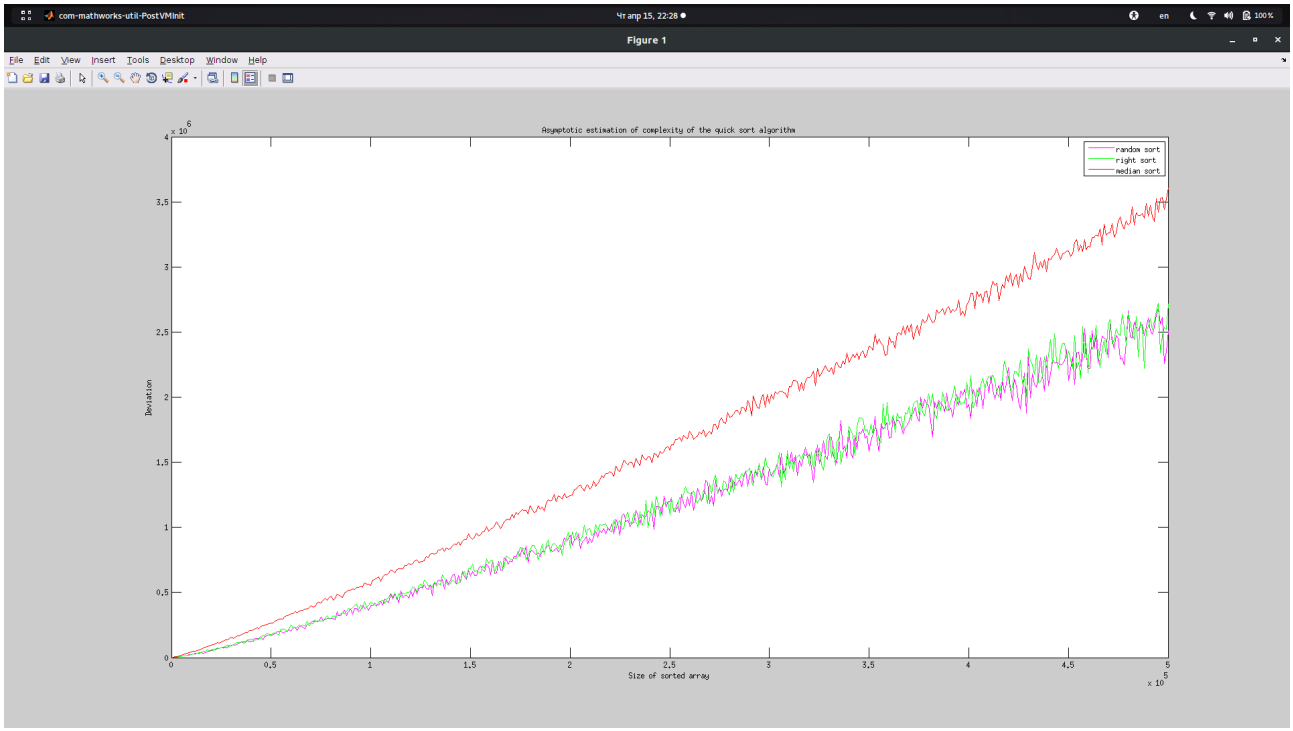


Полученные на предыдущем шаге данные были отображены в виде графиков для наглядности.
Синим представлена асимптотическая оценка O(n*log(n))
Фиолетовым реализация алгоритма с выбором в качестве опорного элемента случайного
Зелёным реализация алгоритма с выбором опорным крайнего правого элемента
Красным реализация алгоритма с выбором опорным элемента в качестве медианы.
По оси OX отображен размер массива, по оси OY количество затраченных на сортировку операций.
Несмотря на отсутствие на текущем этапе строгих математических вычислений и оценок, можно отметить очевидную тенденцию: реализация алгоритма быстрой отсортировки с выбором ведущего элемента в качестве медианы на больших объёмах данных показывает себя существенно эффективнее остальных реализаций.

Отклонение от асимптотической оценки:

Для более подробного анализа полученных данных предлагается провести оценку отклонения построенных графиков от искомой асимптотической оценки.
Для этого в каждой из точек графика, использованных при построении, следует посчитать разность между асимптотической оценкой и данными, полученными в ходе измерений.
Положительное значение говорит о том, что рассматриваемая реализация алгоритма работает более эффективно, чем теоретическая оценка, а отрицательное — менее эффективно.

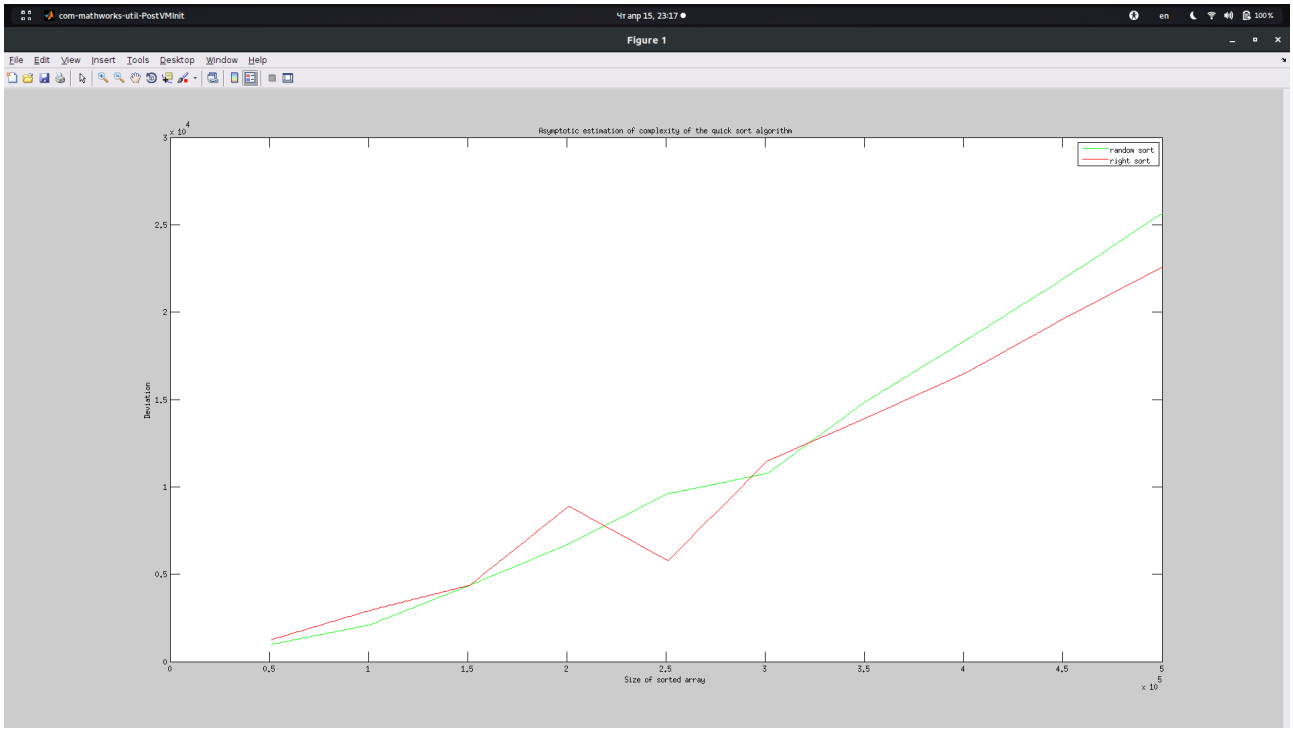
На всех графиках в дальнейшем розовым будет обозначаться сортировка с выбором опорного элемента случайным, зелёным — сортировка с выбором опорного элемента крайним правым, а красным - сортировка с выбором опорного элемента посредством подсчёта медианы.



Как видно из представленного графика, все значения функций отклонения лежат в положительной полуплоскости, что говорит о более эффективной работе всех рассмотренных вариантов реализации алгоритма по сравнению с асимптотической оценкой.
Это в некотором роде подтверждает корректность собранных данных, так как функция $n * \log_2 n$ является верхней оценкой сложности, и следовательно ни одна из реализаций даже в теории не должна превышать это значение.

Следующий факт, который был заметен ещё при построении графиков сложности, заключается в существенном превосходстве реализации с использованием медианного значения при подсчёте опорного элемента над другими вариантами алгоритма.
В связи с этим более подробное сравнение этой реализации с какой-либо другой не является целесообразным — результаты будут весьма предсказуемы.
Однако важно понимать, что сортировка с выбором опорного элемента в качестве медианы, является также наиболее сложной из представленных реализаций, поэтому есть смысл рассмотреть оставшиеся два варианта и выбрать наиболее эффективный из них.

Аппроксимация значения рассматриваемых функций при помощи интерполяции даёт следующие результаты:

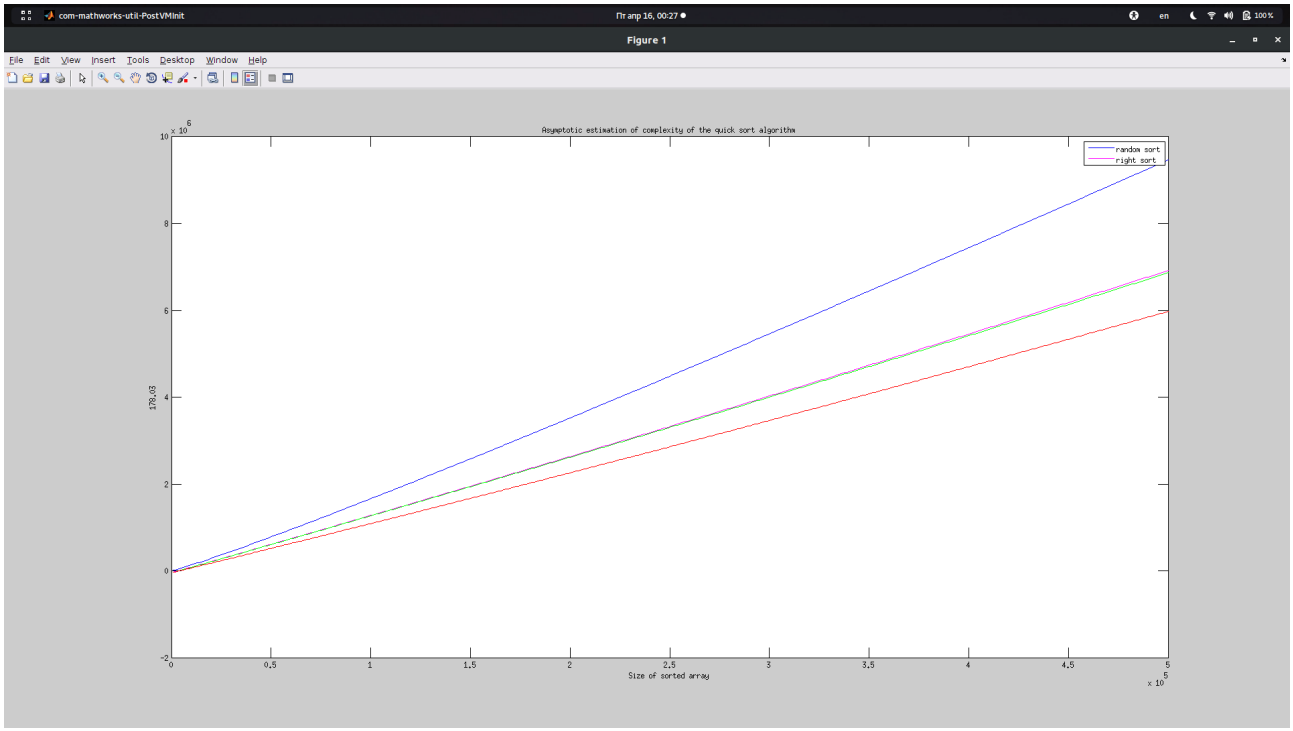


В зависимости от выбора точности результаты могут отличаться, однако общая тенденция может быть заметна. При размерах входного массива не более 250000 элементов лучше себя показывает сортировка с выбором опорным фиксированного правого элемента, в остальных случаях предпочтительнее оказывается использование сортировки с выбором опорным случайного элемента.

Точное представление полученных результатов

Финальным аспектом исследования является математически точное представление результатов тестирования. Для этого предлагается провести аппроксимацию полученных функций при помощи метода наименьших квадратов.

Полученные полиномы имеют первую степень и записываются в виде:
$0.0000017834 * n^2 + 13.0437 * n - 4.7662$ для сортировки с выбором случайного опорного элемента,
$0.0000016681 * n^2 + 12.9996 * n - 5.0532$ для сортировки с выбором фиксированного правого элемента в качестве опорного,
$0.0000017834 * n^2 + 11.183 * n - 4.7495$ для сортировки с выбором опорного элемента в качестве медианы
Построенные для этих функций графики будут иметь следующий вид:



Выводы

На основе проведённых исследований можно сделать следующие выводы:
1. Наиболее эффективной с точки зрения количества логических операций является реализация алгоритма быстрой сортировки, вычисляющая опорный элемент как медиану по трём значениям: первому, центральному и последнему. Данная реализация показывает себя наиболее эффективно вне зависимости от размера входных данных и в целом может быть рекомендована к использованию при прочих равных условиях.
2. При сортировке сравнительно небольших массивов, размер которых не превышает 250000 элементов, большую эффективность показывает реализация с выбором в качестве опорного фиксированного крайнего правого элемента, тогда как при больших размерах входного массива оптимальной является реализация с выбором случайного элемента.

Практическое применение исследования

Полученные знания позволяют более эффективно применять алгоритм быстрой сортировки на практике. В качестве примера предлагается рассмотреть следующую задачу:
Имеется .csv файл, содержащий данные о ежемесячных изменениях цен 22 продуктов, с 1960 до 2017 года, таким образом, суммарно файл содержит 15730 записей.
Требуется проверить наличие определённой стоимости определённого продукта. Очевидно, что линейный перебор потребует $O(n)$, то есть 15730 операций в худшем случае. Для уменьшения времени работы программы предлагается воспользоваться бинарным поиском, обеспечивающим сложность работы $O(\log_2 n)$. Однако, алгоритм бинарного поиска требует сортировки предоставленных данных по возрастанию, и для этой цели можно воспользоваться алгоритмом быстрой сортировки, который обеспечит суммарную сложность выполнения $O(n*\log_2^2 n)$. Очевидно, что для выполнения операции поиска один раз

применение данного алгоритма не является целесообразным, однако уже для $\log_2(15370) = 14$ операций данный алгоритм покажет более оптимальное время работы.

Список использованных источников

1. Robert Lafore: Data structures & Algorithms in Java second edition
2. Курс лекций АиСД year2019 s1
3. Веб-сайт docs.exponenta.ru
4. Веб-сайт uk.mathworks.com
5. Веб-сайт cyberforum.ru

Приложение

Описание работы алгоритма быстрой сортировки

Алгоритм быстрой сортировки является рекурсивным алгоритмом, основанном на так называемом принципе «разделяй и властвуй». В основе этого принципа лежит идея разбиения одной большой и сложной задачи на несколько более мелких и простых в решении.

Так на каждом шаге алгоритма быстрой сортировки переданный массив разбивается на два необязательно равных массива с возможно нулевой длиной. Для выбора разбиения используется так называемый опорный элемент — некоторый элемент x массива, относительно которого будут распределены остальные элементы. Так, в первый подмассив попадут все элементы меньше (меньше или равные x), а во второй подмассив все элементы больше (больше или равные) x. Операция повторяется рекурсивно до тех пор, пока все элементы не окажутся на своих позициях.

Реализации алгоритмов быстрой сортировки на языке Java SE8

Класс, предназначенный для подготовки тестовых массивов, сбора данных и загрузки результатов в файл

```
public class Measurement {
  private static final int RESEARCH_AMOUNT = 500;
  private static final int RESEARCHES = 10;
  public static void main(String[] args) throws FileNotFoundException {
    long[] randomResults = new long[RESEARCH_AMOUNT];
    long[] rightElementResults = new long[RESEARCH_AMOUNT];
    long[] medianResults = new long[RESEARCH_AMOUNT];
    for (int i = 0; i < RESEARCHES; i++) {
```

```
            System.out.println("Испытание " + (i + 1));
            int[][] data = prepareData();
            System.out.println("Копирование данных");
            int[][] randomData = copyData(data);
            int[][] rightElementData = copyData(data);
            int[][] medianData = copyData(data);
            System.out.println("Сортировка со случайным выбором опорного элемента");
            long[] curRandomResults = testSort(new RandomQuickSort(), randomData);
            System.out.println("Сортировка с выбором опорного элемента крайним правым");
            long[] curRightElementResults = testSort(new RightElementQuickSort(), rightElementData);
            System.out.println("Сортировка с выбором опорного элемента по медиане");
            long[] curMedianResults = testSort(new MedianQuickSort(), medianData);
            System.out.println("Суммирование");
            for (int j = 0; j < RESEARCH_AMOUNT; j++) {
                randomResults[j] += curRandomResults[j];
                rightElementResults[j] += curRightElementResults[j];
                medianResults[j] += curMedianResults[j];
            }
        }
        System.out.println("Подсчёт среднего арифметического");
        for (int i = 0; i < RESEARCH_AMOUNT; i++) {
            randomResults[i] /= RESEARCHES;
            rightElementResults[i] /= RESEARCHES;
            medianResults[i] /= RESEARCHES;
        }
        System.out.println("Сохранение результатов");
        writeToFile(randomResults, "results/random.txt");
        writeToFile(rightElementResults, "results/right.txt");
        writeToFile(medianResults, "results/median.txt");
    }
    /**
     * Функция, измеряющая время анализа сортировки набора данных одной из реализаций
быстрой сортировки
     */
    private static long[] testSort(QuickSort quickSort, int[][] data) {
        long[] results = new long[RESEARCH_AMOUNT];
        for (int i = 0; i < RESEARCH_AMOUNT; i++) {
            quickSort.sort(data[i]);
            results[i] = quickSort.getActions();
            quickSort.setActions(0);
        }
        return results;
    }
    /**
     * Функция, генерирующая массивы данных для дальнейшего анализа
     */
    private static int[][] prepareData() {
        int[][] data = new int[RESEARCH_AMOUNT][];
        int size = 1000;
        for (int i = 0; i < RESEARCH_AMOUNT; i++) {
            data[i] = generateArray(size);
            size += 1000;
```

```
        }
        return data;
    }
    /**
     * Функция для выводя массива на экран
     */
    private static void print(int[] array) {
        for (int j : array) {
            System.out.print(j + " ");
        }
        System.out.println();
    }
    /**
     * Функция для записи полученных данных в файл
     */
    private static void writeToFile(long[] results, String filename) throws FileNotFoundException {
        PrintWriter writer = new PrintWriter(
                new FileOutputStream(
                        new File("src/CalculationPackages/" + filename), false));
        for (long result : results) {
            writer.write(result + " ");
        }
        writer.write("\n");
        writer.close();
    }
    private static int[][] copyData(int[][] data) {
        int[][] result = new int[RESEARCH_AMOUNT][];
        int size = 1000;
        for (int i = 0; i < RESEARCH_AMOUNT; i++) {
            result[i] = generateArray(size);
            size += 1000;
        }
        for (int i = 0; i < RESEARCH_AMOUNT; i++) {
            System.arraycopy(data[i], 0, result[i], 0, RESEARCH_AMOUNT);
        }
        return result;
    }
    /**
     * Функция для создания массива случайных чисел указанной длины
     */
    private static int[] generateArray(int size) {
        Random random = new Random();
        int[] array = new int[size];
        for (int i = 0; i < array.length; i++) {
            array[i] = random.nextInt() % 100;
        }
        return array;
    }
}
```

Класс, представляющий абстрактную реализацию алгоритма быстрой сортировки. Все приведённые далее классы являются его наследниками.

```
public interface QuickSort {
  void sort(int[] array);
  int getActions();
  void setActions(int actions);
}
```

Класс, предоставляющий реализацию алгоритма быстрой сортировки с выбором случайного опорного элемента

```
public class RandomQuickSort implements QuickSort {
  private final Random random = new Random();
  private int actions;
  public int getActions() {
    return actions;
  }
  public void setActions(int actions) {
    this.actions = actions;
  }
  /**
   * Функция быстрой сортировки
   */
  public void sort(int[] array) {
    recQuickSort(array, 0, array.length - 1);
  }
  /**
   * Рекурсивная функция сортировки подмассивов
   */
  private void recQuickSort(int[] array, int l, int r) {
    int i = l;
    int j = r;
    int x = array[l + Math.abs(random.nextInt()) % (r - l + 1)];
    do {
      while (array[i] < x) {
        actions += 1;
        i++;
      }
      while (x < array[j]) {
        actions += 1;
        j--;
      }
      if (i <= j) {
        swap(array, i, j);
        i++;
        j--;
      }
    } while (i <= j);
    if (l < j) {
      recQuickSort(array, l, j);
    }
    if (r > i) {
      recQuickSort(array, i, r);
```

```
    }
  }
  /**
   * Функция для перемены местами двух элементов в массиве
   */
  private void swap(int[] array, int i, int j) {
    actions += 1;
    int temp = array[i];
    array[i] = array[j];
    array[j] = temp;
  }
}
```

Класс, предоставляющий реализацию алгоритма быстрой сортировки с выбором фиксированного крайнего правого элемента в качестве опорного

```
public class RightElementQuickSort implements QuickSort {
  private int actions;
  public int getActions() {
    return actions;
  }
  public void setActions(int actions) {
    this.actions = actions;
  }
  /**
   * Функция быстрой сортировки
   */
  public void sort(int[] array) {
    recQuickSort(array, 0, array.length - 1);
  }
  /**
   * Рекурсивная функция сортировки подмассивов
   */
  private void recQuickSort(int[] array, int left, int right) {
    if (right - left <= 0)
      return;
    else {
      int pivot = array[right];
      int partition = partition(array, left, right, pivot);
      recQuickSort(array, left, partition - 1);
      recQuickSort(array, partition + 1, right);
    }
  }
  /**
   * Функция для разбиения относительно данного элемента
   */
  private int partition(int[] array, int left, int right, long pivot) {
    int leftPtr = left - 1;
    int rightPtr = right;
    while (true) {
      while (array[++leftPtr] < pivot) {
        actions += 1;
```

```
        }
        while (rightPtr > 0 && array[--rightPtr] > pivot) {
          actions += 1;
        }
        if (leftPtr >= rightPtr){
          break;
        } else{
          swap(array, leftPtr, rightPtr);
        }
      }
      swap(array, leftPtr, right);
      return leftPtr;
    }
    /**
     * Функция для перемены местами двух элементов в массиве
     */
    private void swap(int[] array, int i, int j) {
      actions += 1;
      int temp = array[i];
      array[i] = array[j];
      array[j] = temp;
    }
}
```

Класс, предоставляющий реализацию алгоритма быстрой сортировки с выбором опорного элемента, как медианного значения первого, центрального и последнего элементов

```
public class MedianQuickSort implements QuickSort {
    private int actions;
    public int getActions() {
      return actions;
    }
    public void setActions(int actions) {
      this.actions = actions;
    }
    /**
     * Функция быстрой сортировки
     */
    public void sort(int[] array) {
      recQuickSort(array, 0, array.length - 1);
    }
    /**
     * Рекурсивная функция сортировки подмассивов
     */
    private void recQuickSort(int[] array, int left, int right) {
      int size = right - left + 1;
      if (size <= 3) {
        manualSort(array, left, right);
      } else {
        int pivot = median(array, left, right);
        int partition = partition(array, left, right, pivot);
```

```
        recQuickSort(array, left, partition - 1);
        recQuickSort(array, partition + 1, right);
      }
    }
    /**
     * Функция для вычисления медианы по трём элементам - первому, центральному и последнему
     */
    private int median(int[] array, int left, int right) {
      int center = (left + right) / 2;
      if (array[left] > array[center]) {
        actions += 1;
        swap(array, left, center);
      }
      if (array[left] > array[right]) {
        actions += 1;
        swap(array, left, right);
      }
      if (array[center] > array[right]) {
        actions += 1;
        swap(array, center, right);
      }
      swap(array, center, right - 1);
      return array[right - 1];
    }
    /**
     * Функция для разбиения относительно данного элемента
     */
    private int partition(int[] array, int left, int right, long pivot) {
      int leftPtr = left;
      int rightPtr = right - 1;
      while (true) {
        while (array[++leftPtr] < pivot) {
          actions += 1;
        }
        while (array[--rightPtr] > pivot) {
          actions += 1;
        }
        if (leftPtr >= rightPtr) {
          break;
        } else {
          swap(array, leftPtr, rightPtr);
        }
      }
      swap(array, leftPtr, right - 1);
      return leftPtr;
    }
    /**
     * Функция для ручной сортировки массивов малой длины, так как сортировка с
          вычислением медианы по трём элементам
     * требуется наличия хотя бы четырёх элементов для корректной работы
     */
```

```
  private void manualSort(int[] array, int left, int right) {
    int size = right - left + 1;
    if (size <= 1) {
      return;
    }
    if (size == 2) {
      if (array[left] > array[right]) {
        actions += 1;
        swap(array, left, right);
      }
      return;
    } else {
      if (array[left] > array[right - 1]) {
        actions += 1;
        swap(array, left, right - 1);
      }
      if (array[left] > array[right]) {
        actions += 1;
        swap(array, left, right);
      }
      if (array[right - 1] > array[right]) {
        actions += 1;
        swap(array, right - 1, right);
      }
    }
  }
  /**
   * Функция для перемены местами двух элементов в массиве
   */
  private void swap(int[] array, int i, int j) {
    actions += 1;
    int temp = array[i];
    array[i] = array[j];
    array[j] = temp;
  }
}
```

Программа для анализа полученных данных на языке MatLab

```
% Выгрузка данных из файлов
randomId = fopen('random.txt','r');
random = transpose(fscanf(randomId, '%f'));

rightId = fopen('right.txt','r');
right = transpose(fscanf(rightId, '%f'));

medianId = fopen('median.txt','r');
median = transpose(fscanf(medianId, '%f'));

size = 1000:1000:500000;

idx = 1000;
```

```
n = 0;
for i = 1:500
    n(1,i) = idx * log2(idx);
    idx = idx + 1000;
end

% Построение графиков по результатам выборки
plot(size, n, 'b');
hold on;
plot(size, random, 'm');
hold on;
plot(size, right, 'g');
hold on;
plot(size, median, 'r');
hold off;

xlabel('Size of sorted array');
ylabel('Amount of spent operations');

title('Asymptotic estimation of complexity of the quick sort algorithm');
legend('n*log(n)', 'random sort', 'right sort', 'median sort');

% Подсчёт отклонений от асимптотической оценки
diff1 = 0;
diff2 = 0;
diff3 = 0;
for i = 1:500
    diff1(1,i) = n(1,i) -random(1,i);
    diff2(1,i) = n(1,i) - right(1,i);
    diff3(1,i) = n(1,i) - median(1,i);
end

% Построение графиков отклонений
plot(size, diff1, 'm');
hold on;
plot(size, diff2, 'g');
hold on;
plot(size, diff3, 'r');
hold off;

xlabel('Size of sorted array');
ylabel('Amount of spent operations');

title('Asymptotic estimation of complexity of the quick sort algorithm');
legend('n*log(n)', 'random sort', 'right sort', 'median sort');
x = (1:500);
xi = 0;
xiidx = 0;
for i = 1:500
    xi(1,i) = xiidx;
    xiidx = xiidx + 0.015;
end
```

```
% Интерполирование и построение графиков отклонений
interpol1 = interp1(x, diff1, xi, 'linear');
plot(size, interpol1, 'g');
hold on;
interpol2 = interp1(x, diff2, xi, 'linear');
plot(size, interpol2, 'r');
hold off;

xlabel('Size of sorted array');
ylabel('Deviation');

title('Asymptotic estimation of complexity of the quick sort algorithm');
legend('random sort', 'right sort');

% Аппроксимация результатов
p1 = polyfit(size, random, 2);
p2 = polyfit(size, right, 2);
p3 = polyfit(size, median, 2);

appr1 = 0;
appr2 = 0;
appr3 = 0;
idx = 1000;
for i = 1:500
    appr1(1,i) = p1(1,1) * idx * idx + p1(1,2) * idx + p1(1,3);
    appr2(1,i) = p2(1,1) * idx * idx + p2(1,2) * idx + p2(1,3);
    appr3(1,i) = p3(1,1) * idx * idx + p3(1,2) * idx + p3(1,3);
    idx = idx + 1000;
end

% Построение графиков аппроксимированных функций
plot(size, n, 'b');
hold on;
plot(size, appr1, 'm');
hold on;
plot(size, appr2, 'g');
hold on;
plot(size, appr3, 'r');
hold off;

xlabel('Size of sorted array');
ylabel(xlabel('Size of sorted array'));

title('Asymptotic estimation of complexity of the quick sort algorithm');
legend('random sort', 'right sort');

% Закрытие файлов
fclose(randomId);
fclose(rightId);
fclose(medianId);
```

Программа практического применения алгоритма быстрой сортировки на языке Java

```
/**
 * Класс для демонстрации практического применения алгоритма быстрой сортировки с подсчётом опорного элемента по медиане.
 */
public class PracticalApplication {
  private static final String DATA = "src/CalculationPackages/Practice/data.csv";
  /**
   * Номер столбца в исходном файле
   * Каждому столбцу соответствуют данные об одном конкретном продукте
   */
  private static final int COLUMN_NUMBER = 1;
  public static void main(String[] args) {
    List<String> data = getData(COLUMN_NUMBER);
    MedianQuickSort medianQuickSort = new MedianQuickSort();
    int[] ints = data.stream().mapToInt(x -> (int) (Double.parseDouble(x) * 100)).toArray();
    medianQuickSort.sort(ints);
    double[] doubles = Arrays.stream(ints).mapToDouble(x -> x / 100.).toArray();
    // Бинарный поиск элемента с заданным значением
    System.out.println(Arrays.binarySearch(doubles, 1.21));
  }
  /**
   * Выгрузка данных из файла
   */
  private static ArrayList<String> getData(int column) {
    ArrayList<String> data = new ArrayList<>();
    try (LineNumberReader reader = new LineNumberReader(new BufferedReader(new
FileReader(DATA)))) {
      // Чтение строки с заголовками
      String[] headers = reader.readLine().split(",");
      String line;
      while ((line = reader.readLine()) != null) {
        data.add(line.split(",")[column]);
      }
    } catch (FileNotFoundException e) {
      System.out.println("Файл с именем " + DATA + " не найден");
    } catch (IOException e) {
      System.out.println("Ошибка при чтении файла");
    }
    return data;
  }
}
```