Санкт-Петербургский государственный университет
информационных технологий, механики и оптики

Кафедра «Компьютерные технологии»

Г. Г. Удов, А. А. Шалыто

Классическая и автоматная реализации алгоритма пирамидальной сортировки набора чисел

Программирование с явным выделением состояний

Проектная документация

Проект создан в рамках
«Движения за открытую проектную документацию»
http://is.ifmo.ru

Санкт-Петербург
2004

Содержание

Введение

Алгоритм пирамидальной сортировки [1, 2, 3, 4] является одним из фундаментальных методов сортировки набора чисел. Алгоритм требует $O(1)$ дополнительной памяти, а его трудоемкость в худшем случае составляет $O(nlog_2(n))$, что является пределом для общего случая. При построении этого алгоритма используется не типичная для сортировок технология — применяется структура данных, называемая пирамидой.

В работе выполнены две реализации алгоритма пирамидальной сортировки набора чисел — классическая [5, 6] и с использованием SWITCH-технологии [7, 8, 9], и осуществлен их сравнительный анализ.

Реализации оформлены в виде Java-аплетов. Для каждого из них обеспечены визуализация алгоритма, дружественный пользовательский интерфейс, возможность загрузки приложения через интернет и запуска практически на любой аппаратной платформе. Широкие возможности по конфигурированию аплетов позволяют разместить их практически на любой интернет-странице без нарушения внешнего вида, а также «перевести» на любой язык.

Проектная документация помимо постановки задачи содержит описание и анализ рассматриваемого алгоритма, описание и анализ реализаций, а также исходные коды программ.

1. Анализ литературы

Информация о рассматриваемом алгоритме содержится в книгах Н. Вирта [1], Д. Кнута [2], И.В. Романовского [3] и Т. Кормена [4].

Описание алгоритма у И.В. Романовского — достаточно краткое и математичное. Н. Вирт описывают проблему достаточно сжато и лаконично. Т. Кормен сочетает доступность и подробность изложения, однако не уделяет должного внимания рассмотрению тех случаев, в которых именно этот алгоритм сортировки является наиболее подходящим. Отметим, что проблеме оптимальной сортировки должное внимание уделено только в фундаментальном труде Д. Кнута. Однако манера изложения Д. Кнута весьма специфична, так как использование языка ассемблера при реализации алгоритмов в настоящее время считается нецелесообразным.

В данной работе синтезированы и переработаны изложения Н. Вирта, Д. Кнута и Т. Кормена. Примеры программ написаны авторами на языке Java.

2. Описание и анализ алгоритма пирамидальной сортировки

2.1. Описание алгоритма

Пусть задан массив из n элементов. Сначала рассмотрим «школьный» алгоритм сортировки с помощью прямого выбора [1]. В его основе лежит поиск наименьшего элемента

в данном массиве. Найденный наименьший элемент меняют местами с первым элементом набора, затем эту процедуру повторяют для «хвоста» набора (для всех элементов, кроме первого). Это продолжается до тех пор, пока в «хвосте» не останется только один элемент. Таким образом, трудоемкость метода равна

$$n + (n-1) + (n-2) + \cdots + 1 = \frac{n^2 - n}{2}.$$

Как известно, для задачи сортировки массива решение, обладающее квадратичной трудоемкостью, не является оптимальным.

Улучшим алгоритм прямого выбора. Будем оставлять после каждого прохода больше информации, чем просто идентификация единственного минимального элемента. Это осуществляется следующим образом: сделав $n/2$ сравнений, можно определить в каждой паре элементов наименьший, с помощью $n/4$ сравнений — меньший из пары уже выбранных наименьших и т. д. Проделав $n-1$ сравнений, удается построить дерево выбора и идентифицировать его корень как требуемый наименьший элемент.

Второй этап сортировки — идентификация пути, отмеченного наименьшим элементом, и исключение его из дерева путем замены либо на пустой элемент (дырку) в самом низу, либо на элемент из соседней ветви в промежуточных вершинах. Элемент, передвинувшийся в корень дерева, вновь будет наименьшим (теперь уже вторым), и его можно исключить. После таких n шагов дерево станет пустым (в нем останутся только дырки), и процесс сортировки закончится.

Обратим внимание — на каждом из n шагов выбора требуется только $log_2(n)$ сравнений. Поэтому на весь процесс понадобится порядка $nlog_2(n)$ элементарных операций и еще n шагов на построение дерева. Это весьма существенное улучшение не только метода прямого выбора, требующего n^2 шагов, но и даже метода Шелла [2], где необходимо $n^{1.2}$ шагов. Естественно, что сохранение дополнительной информации делает задачу более изощренной — в сортировке по дереву каждый отдельный шаг усложняется. Это происходит потому, что для сохранения избыточной информации, получаемой при начальном проходе, создается некоторая древовидная структура. Поэтому следующая задача состоит в поиске приемов эффективной организации этой информации.

Во-первых, необходимо избавиться от дырок, которыми в конечном итоге будет заполнено все дерево, так как они порождают много ненужных сравнений. Во-вторых, требуется найти такое представление дерева из n элементов, которое потребует n единиц памяти, а не $2n-1$, как это было раньше.

Эти цели достигаются в рассматриваемом методе *Heapsort* (пирамидальная сортировка), изобретенном Д. Уилльямсом [2].

Пирамида определяется как последовательность элементов $h[L], h[L+1], \ldots, h[R]$, такая, что

$$\forall i = L \ldots R/2 \text{ справедливо } (h[i] \le h[2i]) \& (h[i] \le h[2i+1]). \qquad (1)$$

Если любое двоичное дерево рассматривать как массив, то $h[1]$ — наименьший элемент данного массива. Пусть задана пирамида с элементами $h[L+1], \ldots, h[R]$ для некоторых

значений L и R и требуется добавить новый элемент x, образуя расширенную пирамиду $h[L], \ldots, h[R]$. Она получается следующим образом: сначала x ставится наверх древовидной структуры, а затем он постепенно опускается вниз каждый раз по направлению наименьшего из примыкающих к нему элементов, а сам этот элемент передвигается вверх. Данный способ не нарушает условий (1), определяющих пирамиду.

Р. Флойдом в работе [2] был предложен «лаконичный» способ построения пирамиды «на том же месте». Его процедура сдвига (функция `sift`) представлена на листинге 1. Здесь $h[1] \ldots h[n]$ — некий массив, причем $h[m] \ldots h[n] (m = n/2 + 1)$ уже образуют пирамиду, поскольку индексов i, j, удовлетворяющих соотношениям $j = 2i$ и $j = 2i + 1$, просто не существует. Эти элементы образуют нижний слой соответствующего двоичного дерева. Для них никакой упорядоченности не требуется.

Листинг 1. Функция `sift`

```
// Меняет значения переменных массива с индексами a и b
void swap(index a, index b)
{
  item c;
  c = activeArray[a];
  activeArray[a] = activeArray[b];
  activeArray[b] = c;
}

void sift(index first, index last)
{
  index i, j;
  item x;

  i = first;
  j = 2*first + 1;
  x = activeArray[first];

  if(j < last && activeArray[j+1] < activeArray[j])
    j ++;

  while(j <= last && activeArray[j] > x)
  {
    swap(i, j);
    i = j;
    j = 2 * j + 1;
```

```
    if(j < last && activeArray[j+1] < activeArray[j])
      j ++;
  }
}
```

Теперь пирамида расширяется влево, каждый раз добавляется и сдвигами ставится в надлежащую позицию новый элемент. Следовательно, процесс формирования пирамиды из n элементов $h[1] \dots h[n]$ на том же самом месте описывается следующим образом:

```
l = (number / 2);
r = number - 1;
while(l > 0) sift(--l, r);
```

Для того, чтобы получить не только частичную, но и полную упорядоченность среди элементов, требуется проделать n сдвигающих шагов, причем после каждого шага на вершину дерева выталкивается очередной(наименьший) элемент. И вновь возникает вопрос: где хранить «всплывающие» верхние элементы и можно ли или нельзя проводить обращение на том же месте? Существует такой выход: каждый раз брать последнюю компоненту пирамиды (скажем, это будет x), прятать верхний элемент в освободившемся теперь месте, а x сдвигать в нужное место. Процесс описывается с помощью функции `sift` таким образом:

```
r = number - 1;
while(r > 0)
{
  swap(0, r--);
  sift(l, r);
}
```

Однако получившийся порядок фактически является обратным. Это можно легко исправить, изменив направление упорядочивающего отношения в функции `sift`. В конце концов получаем функцию пирамидальной сортировки `heapSort`.

Листинг 2. Функция `heapSort`

```
void swap(index a, index b)
{
  item c;
  c = activeArray[a];
  activeArray[a] = activeArray[b];
  activeArray[b] = c;
}

void sift(index first, index last)
```

```
{
  index i, j;
  item x;

  i = first;
  j = 2*first + 1;
  x = activeArray[first];

  if(j < last && activeArray[j+1] > activeArray[j])
    j ++;

  while(j <= last && activeArray[j] > x)
  {
    swap(i, j);
    i = j;
    j = 2 * j + 1;
    if(j < last && activeArray[j+1] > activeArray[j])
      j ++;
  }
}

void heapSort()
{
  l = (number / 2);
  r = number - 1;
  while(l > 0) sift(--l, r);
  while(r > 0)
  {
    swap(0, r--);
    sift(l, r);
  }
}
```

В окончание приведем схему алгоритма сдвига `sift` (рис. 1). Здесь для компактности иллюстрации `activeArray` обозначен через `a`.

2.2. Анализ алгоритма

На первый взгляд совсем не очевидно, что такой метод сортировки дает хорошие результаты. Ведь в конце концов бóльшие элементы, прежде чем попадут на свое место в правой части, сначала сдвигаются влево. И действительно, процедуру не рекомендуется применять

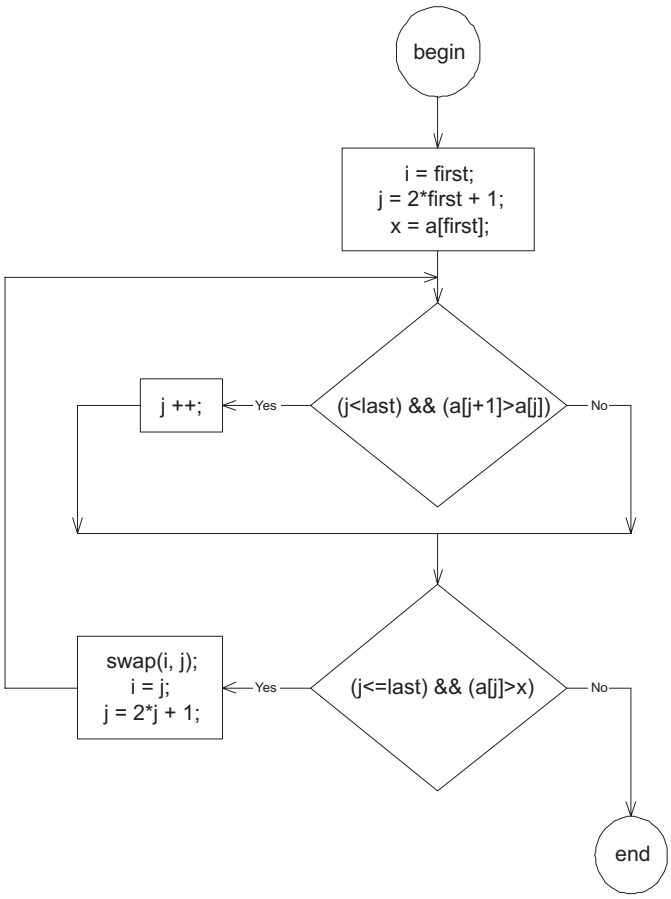

Рис. 1. Схема алгоритма сдвига

для небольшого количества элементов.

В худшем случае нужно $n/2$ сдвигающих шагов, они сдвигают элементы на $log_2(n/2)$, $log_2(n/2-1), \ldots, log_2(n-1)$ позиций (логарифмы округляются везде до следующего меньшего целого). Следовательно, фаза сортировки требует $n-1$ сдвигов с как максимум $log_2(n-1), log_2(n-2), \ldots, 1$ перемещениями. Кроме того, нужно еще $n-1$ перемещений для «просачивания» сдвинутого элемента на некоторое расстояние вправо. Из этих соображений следует, что даже в самом плохом из возможных случаев пирамидальная сортировка потребует $nlog_2(n)$ шагов. Хорошая производительность в таких плохих случаях — одно из привлекательных свойств пирамидальной сортировки.

Совсем не ясно, когда ожидать наихудшей (или наилучшей) производительности. Но вообще-то кажется, что алгоритм `HeapSort` хорошо работает на последовательностях, в которых элементы более или менее отсортированы в обратном порядке. Поэтому ее поведение несколько неестественно. Если мы имеем дело с обратным порядком, то фаза порождения пирамиды не требует каких-либо перемещений. Среднее число перемещений приблизительно равно $nlog_2(n)/2$, причем отклонения от этого значения относительно невелики.

3. Описание интерфейса визуализаторов

3.1. Описание пользовательского интерфейса визуализаторов

Оба визуализатора выполнены в виде Java-аплетов и имеют одинаковый интерфейс пользователя (рис. 2). Занимаемую аплетом часть html-страницы будем называть клиентской областью аплета.

Визуализатор отображает три графических элемента, соответствующих текущему шагу работы алгоритма. Рассмотрим их сверху вниз. В верхней части клиентской области — ленточное представление массива. Следом за ним — пирамидальное. И непосредственно под пирамидой — текстовый комментарий, отделенный горизонтальной линией.

На определенных шагах необходимо каким-либо образом выделить отдельные элементы массива. При прорисовке таких элементов используется особый (на рис. 2 — красный) цвет. Если данные элементы являются соседними в пирамиде, то у соединяющего их ребра дополнительно рисуются стрелки на концах.

В ходе работы алгоритма происходит «отрывание» от пирамиды последних элементов массива. Это изображается отсутствием связи (ребра) у «родителей» с такими элементами.

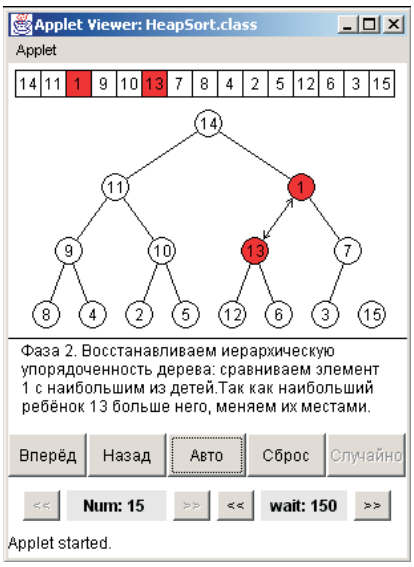

Рис. 2. Внешний вид аплета

Управление процессом визуализации пользователь выполняет посредством нажатия на кнопки. Кнопка «Вперед» осуществляет переход к следующему визуализируемому шагу, кнопка «Назад» — к предыдущему. Кнопка «Авто» включает так называемый автоматический режим — переход к следующему визуализируемому шагу происходит автоматически, с задержкой, длительность которой в мс указывается в правом нижнем текстовом поле. При включенном автоматическом режиме кнопка «Авто» превращается в кнопку «Пауза», назначение которой понятно из названия, и которая по нажатии превращается обратно в кнопку «Авто». Кнопка «Сброс» прекращает отображение визуализируемых шагов — отображается исходный массив и его пирамидальное представление. В данном режиме становится возможным заполнить массив произвольными значениями, нажав кнопку «Случайно», а также изменить его размер. Количество элементов в сортируемом массиве указывается в левом нижнем текстовом поле.

Величину вышеуказанной задержки и количество элементов в сортируемом массиве можно изменять при помощи кнопок, расположенных справа и слева от соответствующих текстовых полей. Величина задержки не может быть меньше 100 мс, количество элементов может изменяться от 7 до 15.

3.2. Описание административного интерфейса (конфигурации) визуализаторов

При размещении визуализатора на определенном интернет-сайте обычно необходимо соблюсти ряд стилевых требований. Например, цветовая гамма и шрифты, используемые в аплете, должны быть выполнены в том же стиле, что и другие элементы сайта. Для того, чтобы администратор или web-мастер мог легко, без модификации и перекомпиляции исходного кода соблюсти подобные требования, ряд параметров аплета необходимо сделать модифицируемыми. Однако в данном случае конечный пользователь не должен быть извещен о существовании этих параметров.

Совокупность параметров визуализатора, предназначенных для изменения администратором сайта или web-мастером, назовем конфигурацией визуализатора. Механизм реализации конфигурации назовем административным интерфейсом.

Оба визуализатора имеют одинаковый административный интерфейс. Конфигурирование осуществляется путем изменения значений параметров аплета в html-файле (приложение 1). Данный способ организации административного интерфейса является наиболее естественным и распространенным. Рассмотрим параметры аплета и их значения в прилагаемых визуализаторах.

Параметры `fgcolor`, `bkcolor`, `activecolor` соответствует цветам, соответственно, линий (контуров), фона аплета, выделенных элементов массива (пирамиды) и в текущей реализации имеют значения черный, белый, красный. Все цвета задаются четырехбайтным целым числом, первый байт которого соответствует синей составляющей цвета, второй — зеленой, третий — красной (байты нумеруем с единицы и начиная с младшего).

Параметр `number` — это количество элементов в массиве сразу после запуска аплета. Параметр `values` представляет собой строку, в которой записано значение по умолчанию (сразу после запуска аплета) всех элементов массива. Строка должна состоять из `number` целых чисел, разделенных пробелами. Параметр `delay` — это величина задержки автоматического режима сразу после запуска аплета.

Параметры `phase1mess`, `comparemess`, `changemess`, `compareendmess`, `firstlastmess`, `phase2mess`, `sortfinishmess` — это строки комментариев, выводимые визуализатором в соответствующих случаях. Выяснить, какая строка в каком случае выводится, достаточно просто, запустив визуализатор и посмотрев на содержимое строк. В ряде случаев строки содержат управляющие последовательности, указываемые в фигурных скобках — в такой ситуации визуализатор во время работы подставляет вместо них определенное число.

Параметры `next`, `prev`, `reset`, `faster`, `slower`, `auto`, `stop`, `randomize`, `More`, `Less` соответствуют названиям кнопок, соответственно, «Вперед», «Назад», «Сброс», уменьшающей задержку, увеличивающей задержку при автоматическом режиме, «Авто», «Пауза», «Случайно», увеличивающей количество элементов в массиве, уменьшающей количество элементов.

Для локализации визуализатора достаточно перевести строки комментариев и названия

кнопок на нужный язык.

4. Описание реализаций визуализаторов

4.1. Технические особенности реализации визуализаторов

При реализации визуализаторов необходимо, во-первых, решить ряд чисто технических вопросов, и, во-вторых, реализовать собственно алгоритм. Технические вопросы включают обеспечение функционирования пользовательского и административного интерфейсов, механизм обмена данными между реализациями алгоритма и названными интерфейсами. Целесообразно в обоих визуализаторах технические вопросы решать единообразно.

Вся реализация аплетов содержится в классе `HeapSort`. Данный класс является наследником стандартного класса `Applet` и реализацией стандартных интерфейсов `Runnable` и `ActionListener`.

Приведем структурную схему технической функциональности класса `HeapSort` на рис. 3 и опишем ее подробнее. Формат структурной схемы приведен в статье [10]. Методы и свойства, относящиеся к реализациям алгоритма, на данной структурной схеме не показаны.

Для того, чтобы возложить на класс `HeapSort` функциональность java-аплета, его необходимо сделать наследником стандартного класса `Applet`.

Непосредственно после загрузки аплета из сети виртуальная java-машина программы-браузера вызывает метод `init` наследника класса `Applet` (инициализирует аплет). В рассматриваемых аплетах данный метод перегружен и используется для загрузки параметров конфигурации в соответствующие свойства и последующего построения интерфейса на основе данных свойств. При загрузке параметров используются вспомогательные внутренние методы `getIntParm`, `readInt`, `readColor`.

Метод `getIntParm` ищет параметр с именем `parameterName` и в случае наличия такого параметра возвращает его значение, а в случае отсутствия — возвращает `defaultValue`. Методы `readInt` и `readColor` производят преобразование соответственно числа и цвета из строкового представления (тип `String`) в рабочее (тип `int`), а при неудаче возвращают `defaultValue` (`defaultColor`).

При инициализации сначала загружается информация о цветах в свойства `foregroundColor`, `backgroundColor`, `activeColor`. Затем загружаются фразы комментариев в свойства `phase1Mess`, `compareMess`, `changeMess`, `compareEndMess`, `firstLastMess`, `phase2Mess`, `sortFinishMess`.

Далее создаются кнопки. Для этого используется внутренняя функция `createButton`, в качестве параметров которой передается имя параметра html-файла, в котором хранится название соответствующей кнопки. Данная функция создает объект класса `Button`, передавая в конструктор параметром загруженное из html-файла название кнопки, и устанавливает текущую реализацию класса `HeapSort` (ключевое слово `this`) в качестве получателя событий от данной кнопки. При этом используется метод `addActionListener` класса `Button`.

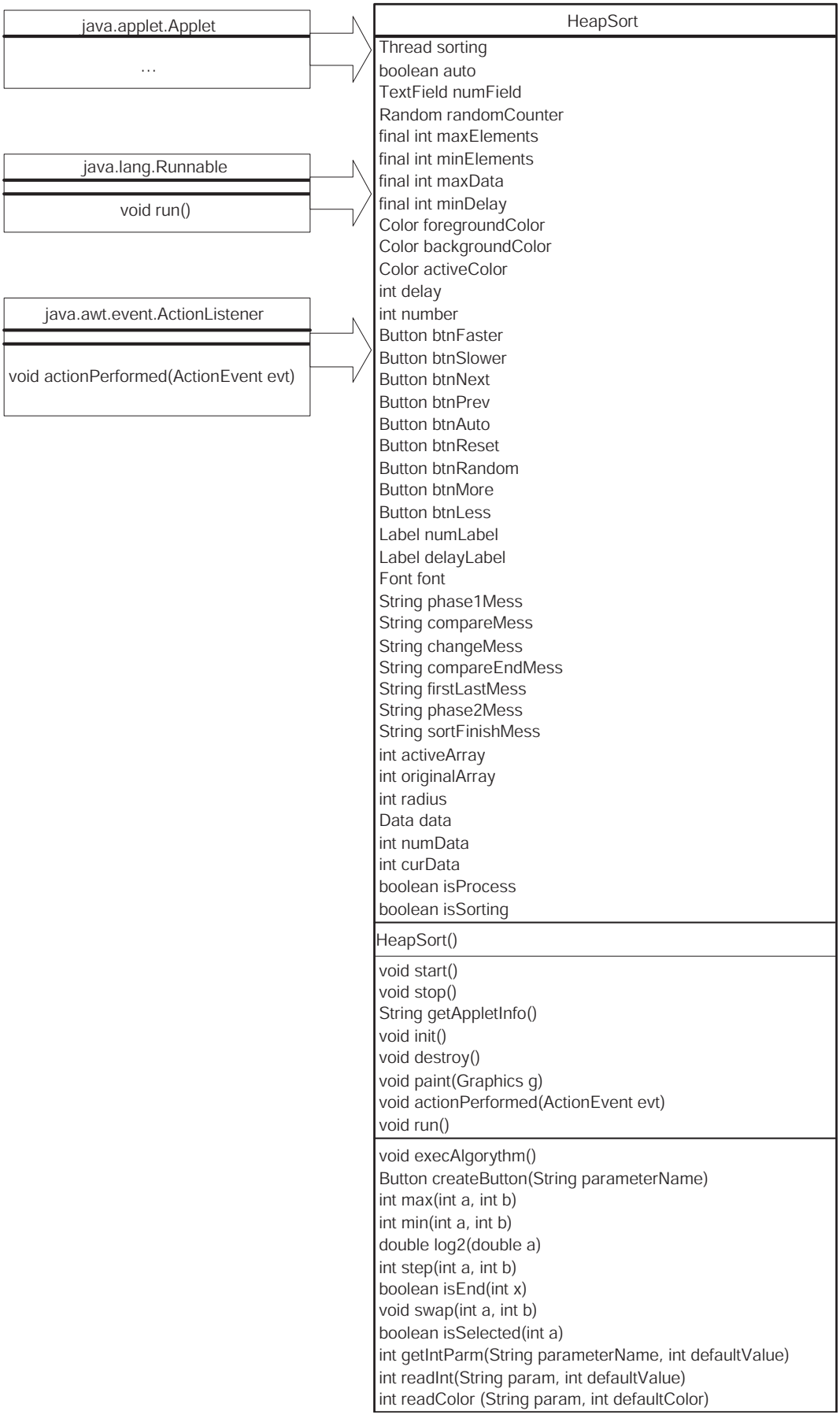

Рис. 3. Структурная схема технической функциональности класса HeapSort

После создания кнопок происходит загрузка из html-файла (секция параметров аплета) величины задержки при автоматическом режиме и количества элементов в сортируемом массиве. Данные величины непосредственно после загрузки проверяются — величина задержки не может быть меньше константы `minDelay`, количество элементов должно находиться в диапазоне между константами `minElements` и `maxElements`.

Далее загружаются элементы массива. Как уже было упомянуто, в файле они хранятся в одной строке, разделенные пробелами. Для разбора строки используется средство `StringTokenizer` языка Java.

Затем устанавливаются шрифт (посредством функции `setFont` ассоциированного с данным аплетом объекта `Graphics`) и цвет фона (посредством функции `setBackground` унаследованной классом `HeapSort` от класса `Applet`).

Далее производится настройка интерфейсных панелей. Сначала устанавливается объект-компоновщик, чья задача — располагать компоненты на рабочей области. В качестве объекта-компоновщика аплета используется экземпляр класса `BorderLayout`. Установка осуществляется при помощи метода `setLayout` класса `Applet`. Далее создается объект класса `Panel` (переменная `controlPanel`). Он после заполнения будет добавлен к клиентской области аплета с нижней («South») стороны. К нему же, в свою очередь, после заполнения добавляются две другие панели (за них отвечают локальные переменные `controlPanel1` и `controlPanel2`). Панель `controlPanel1` заполняется кнопками `btnNext`, `btnPrev`, `btnAuto`, `btnReset`, `btnRandom`. Панель `controlPanel2` — кнопками `btnLess`, `numLabel`, `btnMore`, `btnSlower`, `delayLabel`, `btnFaster`.

В завершение инициализируются массивы `data`, `originalArray`, `activeArray`, переменные `numData`, `curData` (назначение этих полей будет описано позже).

Для запуска аплета виртуальная java-машина программы-браузера вызывает метод `start()` наследника класса `Applet`, а для приостановки аплета — метод `stop()`. В данных реализациях эти методы используются для порождения (соответственно, удаления) нити `sorting`, отвечающей за сортировку в автоматическом режиме. Кроме этой нити существует нить по умолчанию, в которой собственно java-машина браузера запускает аплет.

При создании нить `sorting` инициализируется ссылкой на класс, реализующий интерфейс `Runnable`. В данных аплетах для обеспечения компактности кода было решено возложить реализацию этого интерфейса непосредственно на класс `HeapSort`. Таким образом, нить `sorting` инициализируется ссылкой на данный экземпляр класса `HeapSort` (`this`). Интерфейс `Runnable` имеет единственный метод `run`, который необходимо реализовать. Поэтому метод `run` класса `HeapSort` будет запущен в нити `sorting`. Работа данного метода может быть приостановлена вызовом методов `sleep` или `suspend` нити. Метод `suspend` приостанавливает выполнение нити до вызова метода `resume`. Метод `sleep` приостанавливает выполнение нити на заданный период времени. Также работа нити может быть начата (метод `start`) и прервана (метод `stop`).

Сразу после создания нить запускается и сразу же приостанавливается. Это необходимо для того, чтобы включение и выключение автоматического режима можно было реализовать

как восстановление и приостановку нити. Перед удалением нить прерывается.

При завершении работы аплета виртуальная java-машина программы-браузера вызывает метод `destroy()` наследника класса `Applet`. Обычно этот метод используется для освобождения потребляемых аплетом ресурсов и т. п. В рассматриваемых реализациях не применяются какие-либо ресурсы, требующие освобождения. Поэтому данный метод оставлен пустым.

Метод `getAppletInfo` вызывается виртуальной java-машиной программы-браузера когда пользователь средствами интерфейса браузера запрашивает информацию об аплете. Данная функция возвращает строку, содержащую имя визуализатора и информацию об авторе.

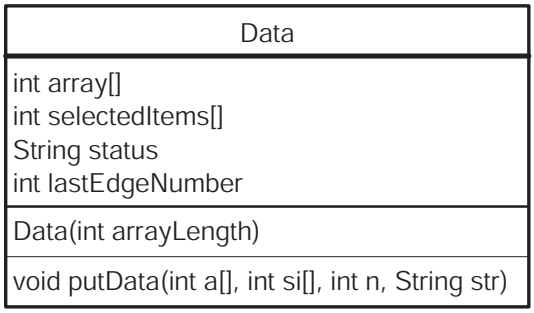

Рис. 4. Структурная схема класса Data

Теперь остановимся на механизме обмена данными между реализацией алгоритма и интерфейсом визуализатора. Для этой цели используется класс `Data`, структурная схема которого приведена на рис. 4. Каждый визуализируемый шаг описывается экземпляром класса `Data`. Данный класс имеет следующие поля. Поле `array` — собственно сортируемый массив на данном шаге. Поле `selectedItems` — массив, содержащий индексы «активных» элементов сортируемого массива, иначе говоря, номера элементов, которые при отображении должны быть выделены определенным (в конфигурации по умолчанию — красным) цветом. Поле `status` — строка, содержащая текстовый комментарий к данному шагу, который при отображении будет выведен внизу клиентской области аплета. Поле `lastEdgeNumber` — номер последнего элемента массива, являющегося элементом рассматриваемой пирамиды (соответствует переменной `r` в листинге 2). Элементы пирамиды, соответствующие элементам массива с бóльшими номерами, при отображении должны быть нарисованы «оторванными» от пирамиды. Также данный класс имеет конструктор, выделяющий память для массивов `array` и `selectedItems` и метод `putData`, осуществляющий присвоение значений полям класса. Следует заметить, что данный класс рассматривается не как программный объект, наделенный определенной функциональностью, а как некоторый контейнер (структура) для хранения данных, вспомогательный для класса `HeapSort`. Поэтому доступ к его полям намеренно сделан явным, а не через методы.

Существует следующее соглашение. При вызове метода `execAlgorythm` массив `data` наполняется информацией обо всех визуализируемых шагах. При этом переменная `numData` получает значение количества визуализируемых шагов. Во время визуализации также задействована переменная `curData`, она хранит номер текущего визуализируемого шага. Данное соглашение позволяет абстрагировать реализацию алгоритма от реализации технических моментов, массив `data` при этом играет роль протокола обмена данными. Таким образом, единственная разница классической и автоматной реализаций визуализаторов алгоритма — содержание метода `execAlgorythm`.

Следует упомянуть еще об одном соглашении, на котором базируется процесс визуализации. Визуализатор может находиться в двух режимах — процесс визуализации запущен

(при этом переменная `isSorting` имеет значение истина, а массив `data` хранит актуальный набор визуализируемых шагов) и не запущен (`isSorting` имеет значение ложь). Сразу после запуска аплета процесс визуализации не запущен. Однако по нажатии кнопки «Авто» или «Вперед» происходит запуск процесса визуализации. При этом становится невозможным менять размер или содержимое массива (соответствующие кнопки интерфейса запрещаются), массив `data` и переменная `numData` получают актуальные значения (вызывается метод `execAlgorythm`), переменная `curData` инициализируется нулем.

Для того, чтобы прервать процесс визуализации, пользователь должен нажать на кнопку «Сброс». При этом переменная `isSorting` получает значение ложь, становится возможным изменять размер и содержимое сортируемого массива (соответствующие кнопки разрешаются), но становится невозможным перейти к следующему визуализируемому шагу (кнопка «Назад» запрещается).

Когда пользователь нажимает какую-либо кнопку, виртуальная java-машина программы-браузера передает управление функции `actionPerformed` класса `HeapSort`. Это происходит потому, что у всех кнопок в качестве получателя событий был указан данный класс. В этом методе анализируется, какая кнопка была нажата, и выполняются необходимые действия.

В случае нажатия кнопки `btnNext`, если переменная `isSorting` имеет значение ложь, то происходит запуск процесса визуализации, в противном случае осуществляется переход к следующему визуализируемому шагу. Если был осуществлен переход на последний визуализируемый шаг, кнопки `btnNext` и `btnAuto` запрещаются. Если был осуществлен переход на второй визуализируемый шаг, разрешается кнопка `btnPrev`. Кроме того, в любом случае инкрементируется переменная `curData` и перерисовывается содержимое клиентской области аплета (метод `repaint`).

В случае нажатия кнопки `btnPrev` выполняется переход к предыдущему визуализируемому шагу. При этом разрешаются кнопки `btnNext` и `btnAuto`, которые могли быть запрещены в результате перехода на последний шаг. В случае перехода на первый шаг (равенство нулю переменной `curData`) кнопка `btnPrev` запрещается. Кроме того, в любом случае декрементируется переменная `curData` и перерисовывается содержимое клиентской области аплета (метод `repaint`).

Визуализатор может находиться в автоматическом и ручном режиме. Переменная `isProcess` имеет значение истина в автоматическом режиме и ложь в ручном. Автоматический режим включается нажатием кнопки `btnAuto`, если переменная `isProcess` имеет значение ложь (визуализатор уже не находится в автоматическом режиме). В этом случае кнопка `btnAuto` имеет название, загружаемое из параметра html-файла с именем auto (в текущей конфигурации — «Авто»). При включении автоматического режима данная кнопка меняет свое название на «Пауза» (параметр html-файла `stop`), переменная `isProcess` получает значение истина, восстанавливается работа нити `sorting`. Автоматический режим выключается нажатием кнопки `btnAuto` если переменная `isProcess` имеет значение истина (визуализатор находится в автоматическом режиме) или нажатием кнопки `btnReset`. При этом кнопка `btnAuto` меняет свое название обратно на значение параметра html-файла `auto`,

переменная `isProcess` получает значение ложь, приостанавливается работа нити `sorting`.

Метод `run` состоит из бесконечного цикла. В цикле осуществляется переход к следующему шагу визуализации и перерисовка клиентской области аплета. После выполнения данных действий нить засыпает — вызывается ее метод `sleep` с параметром `delay` — указанное пользователем время задержки. При переходе на последний шаг автоматический режим выключается.

В случае нажатия кнопок `btnFaster` и `btnSlower` происходит, соответственно, уменьшение и увеличение величины задержки при автоматическом режиме, которая хранится в переменной `delay`, на определенную величину (в данной реализации — 50 мс). Уменьшение величины задержки меньше 100 мс не допускается. Также осуществляется обновление текстового поля, содержащего задержку.

В случае нажатия кнопок `btnLess` и `btnMore` происходит, соответственно, уменьшение и увеличение размера сортируемого массива. Не допускается выход размера массива за границы отрезка, ограниченного свойствами `minElements` и `maxElements`. Также осуществляется обновление текстового поля, содержащего количество элементов в сортируемом массиве.

Обновление клиентской области производится методом `paint` класса `HeapSort`, также вызываемым виртуальной java-машиной программы-браузера при необходимости перерисовки окна либо по требованию программиста. Осуществляется прорисовка ленточного представления массива, пирамиды, разбиение и прорисовка строки комментария. Информация для прорисовки берется из элемента `curData` массива `data`. Элементы, индексы которых являются членами массива `selectedItems`, прорисовываются выделенными и в ленточном представлении, и в пирамиде. Также если выделенные элементы являются соединенными одним ребром в пирамиде, то на концах данного ребра дополнительно рисуются стрелки.

4.2. Особенности классической реализации алгоритма

Как уже упоминалось, исполнение алгоритма является функциональностью метода `execAlgorythm` класса `HeapSort`. В классической реализации метод `execAlgorythm` по сути идентичен функции `heapSort` (листинг 2), с тем лишь исключением, что он еще должен дополнительно заполнить массив `data`. Метод `execAlgorythm` производит добавление шага к массиву `data` до и после вызова метода `swap` (приложение 2). В качестве комментария в данном случае должна выступать фраза: «Наибольший в дереве элемент сейчас находится на первом месте, а должен быть на последнем. Меняем местами первый и последний элементы и отрываем последний элемент от пирамиды»

Аналогичным образом метод `sift` до и после вызова метода `swap` добавляет шаг к массиву `data` (наибольший ребенок больше данного элемента). Также шаг необходимо добавить и в том случае, когда данный элемент больше своего наибольшего ребенка. Более подробно узнать механизмы классической реализации алгоритма можно просмотрев методы `sift` и `execAlgorythm` в приложении 2.

4.3. Особенности автоматной реализации алгоритма

Функция `heapSort` (листинг 2) реализована как цикл, так как в ней выделение состояний не является естественным.

Построение автомата по функции `sift` осуществляется не формальным путем (введение соответствующего кодирования в схеме алгоритма), а за счет «естественного» выделения состояний и переходов между ними.

Выделим три состояния — начало и конец процесса (нулевое состояние); просев элемента, имеющего два дочерних элемента (первое состояние); просев элемента, имеющего один дочерний элемент (второе состояние). Опишем переходы, условия и действия.

Переход из нулевого состояния в первое осуществляется всегда безусловно.

Из первого состояния необходимо осуществить переход в нулевое, если у просеиваемого элемента нет дочерних элементов, и во второе — если дочерний элемент только один. Далее необходимо поменять просеиваемый элемент со своим наименьшим ребенком, если он меньше наименьшего ребенка. Это реализовывается двумя следующими петлями. Если просеиваемый элемент меньше своего левого ребенка и левый ребенок не меньше правого, то меняем местами элемент и его левого ребенка. Если просеиваемый элемент меньше своего правого ребенка, то меняем местами элемент и его правого ребенка. Если же оба этих условия не справедливы, то это значит, что просев завершен и нужно перейти в нулевое состояние.

Из второго состояния необходимо перейти в нулевое безусловно, но если элемент меньше своего левого ребенка, нужно поменять их местами.

Таким образом, можно выделить пять условий (`x1`, `x2`, `x3`, `x4`, `x5`) и три действия(`z1`, `z2`, `z3`).

Условие `x1` истинно тогда и только тогда, когда просеиваемый элемент не имеет дочерних. Условие `x2` истинно тогда и только тогда, когда просеиваемый элемент имеет только один дочерний элемент. Условие `x3` истинно тогда и только тогда, когда просеиваемый элемент меньше своего левого ребенка. Условие `x4` истинно тогда и только тогда, когда левый ребенок просеиваемого элемента не меньше правого ребенка. Условие `x5` истинно тогда и только тогда, когда просеиваемый элемент меньше своего правого ребенка.

Перед описанием действий напомним, что функция `sift` имеет два параметра — индекс элемента массива с которого начинается процесс просева и индекс элемента массива которым заканчивается дерево. В автоматном представлении также уместно выделить одну локальную переменную — индекс текущего просеиваемого элемента. Действие `z1` устанавливает значением индекса текущего просеиваемого элемента индекс элемента массива, с которого начинается процесс просева. Действие `z2` (соответственно, `z3`) меняет местами значения и индексы текущего просеиваемого элемента и его левого (соответственно, правого) ребенка. Таким образом, при просеве изменяется индекс элемента, но сам элемент остается неизменным. Так как действия изменяют локальную переменную (индекс текущего просеиваемого элемента), они оформлены в виде функций, возвращающих результат — новое значение данной переменной.

Схема связей получившегося автомата представлена на рис. 5, а граф переходов — на рис. 6.

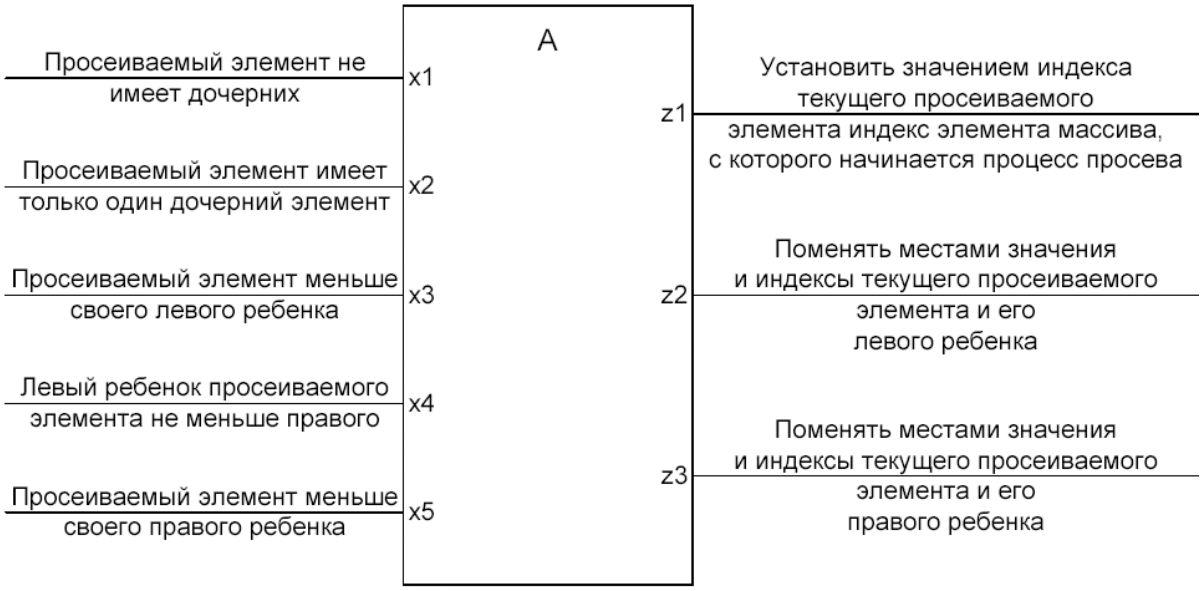

Рис. 5. Схема связей автомата, реализующего просев элемента

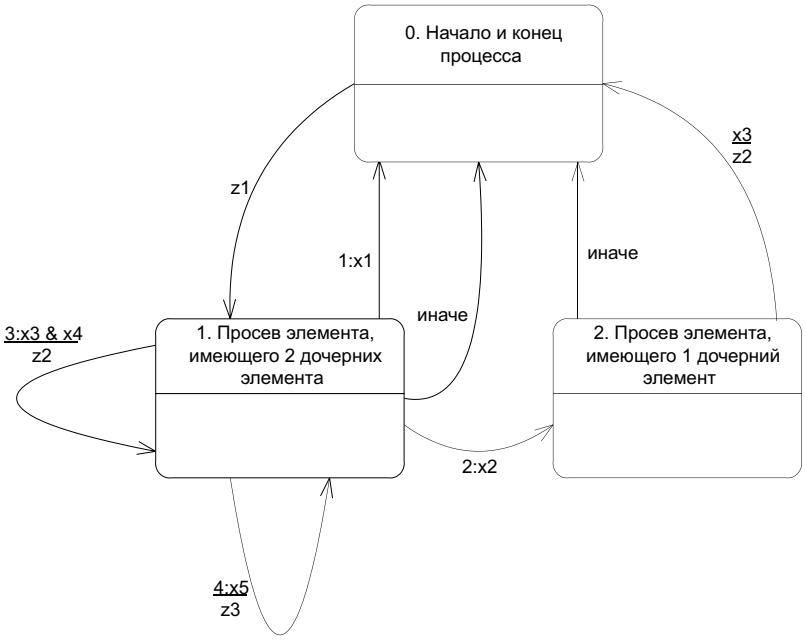

Рис. 6. Граф переходов автомата, реализующего просев элемента

SWITCH-технология предполагает ведение полного протокола работы автомата. В данном проекте протоколирование обеспечивает добавление информации о визуализируемых шагах в массив `data`. Добавление информации о шаге визуализации происходит до и после осуществления действия на переходе. В случае отсутствия действия в зависимости от необходимости визуализации добавление информации может происходить перед переходом или не происходить вообще.

Заключение

В работе приведены две реализации алгоритма пирамидальной сортировки — классическая и автоматная.

Автоматная реализация расширяет использование конечных автоматов в программировании. Их применение делает естественным процесс визуализации, так как не вводя понятие состояния в общем случае не понятно, в какой момент необходимо выполнять шаг визуализации.

В силу того, что в функции `sift` состояния выделились естественно, то для нее автоматный подход может применяться не только при построении визуализатора, но и непосредственно при реализации.

Отметим, что граф переходов автомата (рис. 6) получился значительно проще схемы алгоритма (рис. 1), а совместно со схемой связей, описывающей интерфейс автоматов (рис. 5) — еще и ее понятнее.

Обратим внимание, что при использовании технологии *Vizi* [11] логика визуализатора реализуется четырьмя автоматами, общее число состояний в которых равно 46. В то время как на основе предлагаемого подхода автомат содержит только три состояния. При этом, однако, в отличие от технологии *Vizi* автомат строится вручную, а обратный проход реализуется и вовсе без использования автоматов, что нельзя отнести к достоинствам предлагаемого метода.

Список литературы

[1] Вирт Н. *Алгоритмы и структуры данных.* М.: Мир, 1989.

[2] Кнут Д. *Искусство программирования для ЭВМ. Т. 3. Сортировка и поиск.* М.: Мир, 1978.

[3] Романовский И.В. *Дискретный анализ.* СПб.: «Невский диалект», 1999.

[4] Cormen T., Leiserson C., Rivest R., Stein C. *Introduction to Algorithms, Second edition.* Massachusetts: The MIT Press, 2001.

[5] Столяр С.Е., Осипова Т.Г., Крылов И.П., Столяр С.С. Об инструментальных средствах для курса информатики //Вторая Всероссийская конференция «Компьютеры в образовании». СПб.: 1994.

[6] Казаков М.А., Столяр С.Е. Визуализаторы алгоритмов как элемент технологии преподавания дискретной математики и программирования //Международная научно-методическая конференция «Телематика'2000». СПб.: 2000.

[7] Шалыто А.А. *SWITCH-технология. Алгоритмизация и программирование задач логического управления.* СПб.: Наука, 1998.

[8] Шалыто А.А., Туккель Н.И. Программирование с явным выделением состояний //Мир ПК. 2001. № 8. С.116–121. № 9. С.132–138. http://is.ifmo.ru, раздел «Статьи».

[9] Казаков М.А., Шалыто А.А. Использование автоматного программирования для реализации визуализаторов //Компьютерные инструменты в образовании. 2004. http://is.ifmo.ru, раздел «Статьи».

[10] Шалыто А.А., Туккель Н.И. Автоматы и танки. Объектно-ориентированное программирование с явным выделением состояний. СПб.: СПбГУ ИТМО, 2003. http://is.ifmo.ru, раздел «Статьи».

[11] Удов Г.Г., Шалыто А.А. Построение визуализатора алгоритма пирамидальной сортировки набора чисел на базе технологии*vizi.* СПб.: СПбГУ ИТМО, 2003. http://is.ifmo.ru, раздел «Визуализаторы».

Приложение 1. html-файл

```
<html>

<title>HeapSort</title>
</head>
<body>
<hr>
<table width=100%>
<tr> <td>
<applet code=HeapSort.class id=Shift1 width=300 height=350 >
    <param name="fgcolor"        value="0x00000000">
    <param name="bkcolor"        value="0x00ffffff">
    <param name="activecolor"    value="0x00ff0000">
    <param name="number"         value="15">
    <param name="values"         value="
       1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17">

    <param name="changemess"     value="Так как наибольший ребенок {0} больше
       него, меняем их местами.">
    <param name="compareendmess" value="Так как он больше наибольшего ребенка,
       утопление завершено.">

    <param name="phase2mess"     value="Фаза 2. Восстанавливаем иерархическую
       упорядоченность дерева: ">
    <param name="sortfinishmess" value="Сортировка завершена.">
    <param name="next"           value="Вперед">
</applet>

  </td><td>

        <p align=justify>

        </td></tr>

</table>

<hr>

<I>Copyright 2003 by <a href="mailto:udov@rain.ifmo.ru">Georgy G. Udov</a></I>

</body>

</html>
```

Приложение 2. Исходный текст классической реализации алгоритма

```
/**
   Applet visualising HeapSort algorithm
   @author Georgy Udov (udov@green.ifmo.ru)
*/

import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.math.*;
import java.text.MessageFormat;

public class HeapSort extends Applet implements Runnable, ActionListener
{
  Thread sorting = null;
  private boolean auto = false;
  private TextField numField = new TextField(1);
  Random randomCounter = new Random();

  // Limits
  private final int maxElements = 15;
  private final int minElements = 07;
  private final int maxData     = 200;
  private final int minDelay    = 100;

  private Color foregroundColor = null;
  private Color backgroundColor = null;
  private Color activeColor = null;

  private int delay, number;

  private Button btnFaster;
  private Button btnSlower;
  private Button btnNext;
  private Button btnPrev;
  private Button btnAuto;
  private Button btnReset;
  private Button btnRandom;
  private Button btnMore;
  private Button btnLess;

  private Label numLabel   = new Label ();
  private Label delayLabel = new Label ();

  private Font font = new Font ("Arial", Font.PLAIN, 12);

  private String phase1Mess, compareMess, changeMess, compareEndMess,
                 firstLastMess, phase2Mess, sortFinishMess;

  public int activeArray[]   = new int [maxElements];
  public int originalArray[] = new int [maxElements];

  private int radius = 10;
```

```
// Data which we'll see on screen(array, etc)
public Data data[] = new Data [200];
private int numData = 0, curData = 0;
boolean isProcess = false, isSorting = false;

public HeapSort()
{
}

public String getAppletInfo()
{
  return "Name: HeapSort visualiser\r\n" +
  "Author: Georgy G. Udov(udov@green.ifmo.ru)\r\n";
}

private Button createButton(String parameterName)
{
  final Button button = new Button(getParameter(parameterName));
  button.addActionListener(this);
  return button;
}

public void init()
{
  foregroundColor = new Color (readColor (getParameter ("fgcolor"), 0));
  backgroundColor = new Color (readColor (getParameter ("bkcolor"), 0
     x00ffffff));
  activeColor     = new Color (readColor (getParameter ("activecolor"), 0
     x00ff0000));

  phase1Mess     = getParameter("phase1mess");
  compareMess    = getParameter("compareMess");
  changeMess     = getParameter("changemess");
  compareEndMess = getParameter("compareendmess");
  firstLastMess  = getParameter("firstlastmess");
  phase2Mess     = getParameter("phase2mess");
  sortFinishMess = getParameter("sortfinishmess");

  btnRandom = createButton("randomize");
  btnMore   = createButton("more");
  btnLess   = createButton("less");
  btnNext   = createButton("next");
  btnPrev   = createButton("prev");
  btnAuto   = createButton("auto");
  btnFaster = createButton("faster");
  btnSlower = createButton("slower");
  btnReset  = createButton("reset");

  delay  = getIntParm("delay", minDelay);
  if(delay < minDelay) delay = minDelay;

  number = getIntParm("number", maxElements);
  if (number > maxElements) number = maxElements;
  if (number < minElements) number = minElements;

  int valueCount = 0;

  StringTokenizer st = new StringTokenizer(getParameter("values"), " ");
  while (st.hasMoreTokens() && valueCount < number)
```

```
  {
    activeArray[valueCount++] = readInt(st.nextToken(), 0);
  }

  getGraphics().setFont(font);

  setBackground(backgroundColor);
  setLayout (new BorderLayout ());

  numLabel.setAlignment(1);
  numLabel.setSize(50, 0);
  numLabel.setBackground(new Color(0x00dfdfdf));
  numLabel.setFont(new Font("Arial", Font.BOLD, 12));
  numLabel.setText("  Num: " + number + "    ");

  delayLabel.setAlignment(1);
  delayLabel.setSize(60, 0);
  delayLabel.setBackground(new Color(0x00dfdfdf));
  delayLabel.setFont(new Font("Arial", Font.BOLD, 12));
  delayLabel.setText("wait: " + Integer.toString((int)delay));

  Panel controlPanel = new Panel ();
  Panel controlPanel1 = new Panel ();
  Panel controlPanel2 = new Panel ();

  controlPanel.setLayout(new GridLayout (2, 1, 3, 5));
  controlPanel1.setLayout(new GridLayout (1, 5, 1, 5));

  controlPanel1.add(btnNext);
  controlPanel1.add(btnPrev);
  controlPanel1.add(btnAuto);
  controlPanel1.add(btnReset);
  controlPanel1.add(btnRandom);

  controlPanel2.add(btnLess);
  controlPanel2.add(numLabel);
  controlPanel2.add(btnMore);
  controlPanel2.add(btnSlower);
  controlPanel2.add(delayLabel);
  controlPanel2.add(btnFaster);

  controlPanel.add(controlPanel1);
  controlPanel.add(controlPanel2);

  add("South", controlPanel);

  // Initialising the data
  for(int i = 0; i < maxData; i ++)
    data[i] = new Data(maxElements);

  originalArray = activeArray;
  numData = curData = 0;
  btnPrev.setEnabled(false);
}

public void start()
{
  if (sorting == null)
  {
```

```
    sorting = new Thread(this);
    sorting.start();
    sorting.suspend();
  }
}

 public void stop()
 {
   if (sorting != null)
   {
     sorting.stop();
     sorting = null;
   }
 }

public void destroy()
{
}

public void run()
{
  for(;;)
  {
    if(curData < numData - 1)
    {
      curData ++;
      if(curData == numData - 1)
      {
        btnNext.setEnabled(false);
        btnAuto.setEnabled(false);
        btnPrev.setEnabled(true);
        btnAuto.setLabel(getParameter("auto"));
        isProcess = false;
      }
    }
    repaint();
    if(isProcess)
    try
    {
      Thread.sleep(delay);
    }
    catch (InterruptedException e)
    {
      stop();
    }
    if(curData >= numData - 1)
    {
      sorting.suspend();
    }
  }
}

private int max(int a, int b)
{
  return a > b ? a : b;
}

private int min(int a, int b)
{
```

```
  return a < b ? a : b;
}

private double log2(double a)
{
  return Math.log(a)/Math.log(2.0);
}

private int step(int a, int b)
{
  int res = 1;
  for(int i = 0; i < b; i ++)
    res *= a;
  return res;
}

// Return whether x is end of layer or not
boolean isEnd(int x)
{
  x += 1;
  while(x % 2 == 0)
    x /= 2;
  if(x == 1)
    return true;
  return false;
}

// Attention: a and b are INDEXES of values in array which
// will be swaped. It is NOT variables.
private void swap(int a, int b)
{
  int c;
  c = activeArray[a];
  activeArray[a] = activeArray[b];
  activeArray[b] = c;
}

private void sift(int first, int last, int numStadia)
{
  int i, j, x;
  int selected[] = new int [3];
  String str;

  i = first;
  j = 2*first + 1;
  x = activeArray[first];
  if(j < last && activeArray[j+1] > activeArray[j])
    j ++;
  while(j <= last && activeArray[j] > x)
  {
    selected[0] = i;
    selected[1] = j;
    selected[2] = -1;
    str = numStadia == 1 ? phase1Mess : phase2Mess;
    Object[] tmp1 = {new Integer(activeArray[i])};
    Object[] tmp2 = {new Integer(activeArray[j])};
    str += MessageFormat.format(compareMess, tmp1) +
           MessageFormat.format(changeMess, tmp2)  + " ";
    data[numData].putData(activeArray, selected, last, str);
```

```
      numData ++;
      swap(i, j);
      selected[0] = i;
      selected[1] = j;
      selected[2] = -1;
      data[numData].putData(activeArray, selected, last, str);
      numData ++;
      i = j;
      j = 2 * j + 1;
      if(j < last && activeArray[j+1] > activeArray[j])
        j ++;
    }
    if(j <= last)
    {
      selected[0] = i;
      selected[1] = j;
      selected[2] = -1;
      str  = numStadia == 1 ? phase1Mess : phase2Mess;
      Object [] tmp3 = {new Integer(activeArray[i])};
      str += MessageFormat.format(compareMess, tmp3) +
             compareEndMess + " ";
      data[numData].putData(activeArray, selected, last, str);
      numData ++;
    }
  }

  private void execAlgorythm()
  {
    int x, l, r;
    int selected[] = new int [3];
    String str;

    l = (number / 2);
    r = number - 1;
    while(l > 0)
    {
      l --;
      sift(l, r, 1);
    }

    while(r > 0)
    {
      selected[0] = 0;
      selected[1] = r;
      selected[2] = -1;
      Object [] tmp4 = {new Integer(activeArray[0]), new Integer(activeArray[r
          ])};
      str = MessageFormat.format(firstLastMess, tmp4);
      data[numData].putData(activeArray, selected, r, str);
      numData ++;
      swap(0, r--);
      // Outputing the status
      selected[0] = 0;
      selected[1] = r+1;
      selected[2] = -1;
      data[numData].putData(activeArray, selected, r, str);
      numData ++;
      sift(l, r, 2);
    }
```

```
    selected[0] = -1;
    selected[1] = -1;
    selected[2] = -1;
    data[numData].putData(activeArray, selected, r, sortFinishMess);
    numData ++;
  }

  private boolean isSelected(int a)
  {
    if(isSorting)
    {
      for(int i = 0; i < 3; i ++)
        if(a == data[curData].selectedItems[i])
          return true;
    }
    return false;
  }

  public void paint(Graphics g)
  {
    int x0, y0, width, height;
    int screenWidth  = getSize().width;
    int screenHeight = getSize().height;
    int rSin, rCos;
    double tmp;
    String str, activeString;
    int array[] = new int [maxElements];
    int lastEdgeNumber;

    if(isSorting)
    {
      array = data[curData].array;
      lastEdgeNumber = data[curData].lastEdgeNumber;
      activeString = data[curData].status;
    }
    else
    {
      array = originalArray;
      lastEdgeNumber = number - 1;
      activeString = "";
    }

    g.setFont(font);
    g.setColor(foregroundColor);

    // Painting the Array ...
    /* Painting the "grid" of the members */

    x0 = 5;
    y0 = 5;
    height = 20;
    width = (screenWidth - 10) / number;
    for(int i = 0; i < number; i ++)
    {
      if(isSelected(i))
      {
        g.setColor(activeColor);
        g.fillRect(x0, y0, width, height);
      }
```

```
  g.setColor(foregroundColor);
  g.drawRect(x0, y0, width, height);
  str = Integer.toString(array[i]);
  g.drawString(str, x0 + width/2 - str.length()*font.getSize()/4,
               y0 + height/2 + font.getSize()/2 - 1);
  x0 += width;
}

/* Painting the tree of members */

y0 += height + 20;
x0 = screenWidth / 2;
height = (int)Math.ceil(log2(number + 1)); //This is height of tree in
    units
// This is Width of the last "layer" of tree in units
width = max(step(2, height - 1), number - (step(2, height - 1) - 1));
// This is a width of unit in pixels
width = (screenWidth - 20)/(width /*- 1*/);
width *= step(2, (height - 1)); // Incrementing...
//This is height of unit in pixels
height = (screenHeight - y0 - 2*radius - (font.getSize()+2)*5 - 20)/height;
for(int i = 0; i < number; i ++)
{
  // Calculating R*sin alpha and R*cos alpha
  tmp = Math.sqrt(width / 2.0 * width / 2.0 / 4.0 + height * height);
  rSin = (int)Math.round( (double)radius * (double)(width / 2) / tmp / 2.0)
      ;
  rCos = (int)Math.round( (double)radius * (double)height / tmp);
  if(isSelected(i))
  {
    g.setColor(activeColor);
    g.fillArc(x0 - radius, y0 - radius, 2 * radius + 1, 2 * radius
        + 1, 0, 360);
  }
  g.setColor(foregroundColor);
  g.drawArc(x0 - radius, y0 - radius, 2 * radius, 2 * radius, 0, 360);

  str = Integer.toString(array[i]);
  g.drawString(str, x0 - str.length()*font.getSize() / 4,
                    y0 + font.getSize() / 2 - 1);
  if( 2*i+1 <= lastEdgeNumber )
  {
    // Painting the left edge
    g.drawLine(x0 - (width / 2)/2 + rSin, y0 + height - rCos,
      x0 - rSin, y0 + rCos);
    if(isSelected(2*i+1) && isSelected(i))
    {
      int addX = (width / 4 - 2 * rSin);
      int addY = (height - 2 * rCos);
      double k = (double)addX / (double)addY;
      addX /= 6;
      addY /= 6;
      double x1 = x0 - (width / 2)/2 + rSin + addX;
      double y1 = y0 + height - rCos - addY;
      double b  = y1 - k*x1;
      double d = (2*k*b - 2*x1 - 2*y1*k)*(2*k*b - 2*x1 - 2*y1*k) -
                 4*(1 + k*k)*(x1*x1 + b*b + y1*y1 - 2*y1*b - 9);
      double xOne = (-(2*k*b - 2*x1 - 2*y1*k) + Math.sqrt(d))/(2*(1+k*k));
      double xTwo = (-(2*k*b - 2*x1 - 2*y1*k) - Math.sqrt(d))/(2*(1+k*k));
```

```
      double yOne = k*xOne + b;
      double yTwo = k*xTwo + b;
      int xZero = x0 - (width / 2)/2 + rSin + 1;
      int yZero = y0 + height - rCos - 1;
      g.drawLine(xZero, yZero,
                 (int)Math.round(xOne), (int)Math.round(yOne));
      g.drawLine(xZero, yZero,
                 (int)Math.round(xTwo), (int)Math.round(yTwo));
      // Other side
      addX = (width / 4 - 2 * rSin);
      addY = (height - 2 * rCos);
      k = (double)addX / (double)addY;
      addX /= 6;
      addY /= 6;
      x1 = x0 - rSin - addX;
      y1 = y0 + rCos + addY;
      b  = y1 - k*x1;
      d = (2*k*b - 2*x1 - 2*y1*k)*(2*k*b - 2*x1 - 2*y1*k) -
                 4*(1 + k*k)*(x1*x1 + b*b + y1*y1 - 2*y1*b - 9);
      xOne = (-(2*k*b - 2*x1 - 2*y1*k) + Math.sqrt(d))/(2*(1+k*k));
      xTwo = (-(2*k*b - 2*x1 - 2*y1*k) - Math.sqrt(d))/(2*(1+k*k));
      yOne = k*xOne + b;
      yTwo = k*xTwo + b;
      xZero = x0 - rSin - 1;
      yZero = y0 + rCos + 1;
      g.drawLine(xZero, yZero,
                 (int)Math.round(xOne), (int)Math.round(yOne));
      g.drawLine(xZero, yZero,
                 (int)Math.round(xTwo), (int)Math.round(yTwo));

    }
  }
  if( 2*i+2 <= lastEdgeNumber )
  {
    // Painting the right edge
    g.drawLine(x0 + (width / 2)/2 - rSin,
               y0 + height - rCos, x0 + rSin, y0 + rCos);
    if(isSelected(2*i+2) && isSelected(i))
    {
      int addX = (width / 4 - 2 * rSin);
      int addY = (height - 2 * rCos);
      double k = -(double)addX / (double)addY;
      addX /= 6;
      addY /= 6;
      double x1 = x0 + (width / 2)/2 - rSin - addX;
      double y1 = y0 + height - rCos - addY;
      double b  = y1 - k*x1;
      double d  = (2*k*b - 2*x1 - 2*y1*k)*(2*k*b - 2*x1 - 2*y1*k) -
                   4*(1 + k*k)*(x1*x1 + b*b + y1*y1 - 2*y1*b - 9);
      double xOne = (-(2*k*b - 2*x1 - 2*y1*k) + Math.sqrt(d))/(2*(1+k*k));
      double xTwo = (-(2*k*b - 2*x1 - 2*y1*k) - Math.sqrt(d))/(2*(1+k*k));
      double yOne = k*xOne + b;
      double yTwo = k*xTwo + b;
      int xZero = x0 + (width / 2)/2 - rSin - 1;
      int yZero = y0 + height - rCos - 1;
      g.drawLine(xZero, yZero,
                 (int)Math.round(xOne), (int)Math.round(yOne));
      g.drawLine(xZero, yZero,
                 (int)Math.round(xTwo), (int)Math.round(yTwo));
```

```
        // Other side
        addX = (width / 4 - 2 * rSin);
        addY = (height - 2 * rCos);
        k = -(double)addX / (double)addY;
        addX /= 6;
        addY /= 6;
        x1 = x0 + rSin + addX;
        y1 = y0 + rCos + addY;
        b  = y1 - k*x1;
        d = (2*k*b - 2*x1 - 2*y1*k)*(2*k*b - 2*x1 - 2*y1*k) -
                    4*(1 + k*k)*(x1*x1 + b*b + y1*y1 - 2*y1*b - 9);
        xOne = (-(2*k*b - 2*x1 - 2*y1*k) + Math.sqrt(d))/(2*(1+k*k));
        xTwo = (-(2*k*b - 2*x1 - 2*y1*k) - Math.sqrt(d))/(2*(1+k*k));
        yOne = k*xOne + b;
        yTwo = k*xTwo + b;
        xZero = x0 + rSin + 1;
        yZero = y0 + rCos + 1;
        g.drawLine(xZero, yZero,
                    (int)Math.round(xOne), (int)Math.round(yOne));
        g.drawLine(xZero, yZero,
                    (int)Math.round(xTwo), (int)Math.round(yTwo));

      }
    }

    // Modifying the X0 and Y0
    if(isEnd(i+1) && i != number - 1)
    {
      width /= 2;
      x0 = screenWidth - x0 - width/2;
      y0 += height;
    }
    else
      x0 += width;
  }
  x0 = 10;
  y0 += 2 * radius - 2;

  if(isSorting)
  {
    g.drawLine(0, y0, screenWidth, y0);
    y0 += 2 + font.getSize();

    // Wrapping and outputting the string
    StringTokenizer tokens = new StringTokenizer(activeString, " ");
    FontMetrics metrics = g.getFontMetrics();
    String words[] = new String[tokens.countTokens()];
    for (int i = 0; i < words.length; i++)
    {
      words[i] = tokens.nextToken();
    }

    str = words[0];
    for (int i = 1; i < words.length; i++)
    {
      str += ' ' + words[i];
      if (metrics.stringWidth(str) > screenWidth - 2 * x0)
      {
        str = str.substring(0, str.lastIndexOf(' '));
```

```
            g.drawString(str.trim(), x0, y0);
            y0 += font.getSize() + 2;
            str = "";
            i --;
          }
        }
        g.drawString(str.trim(), x0, y0);
    }
}

public void actionPerformed(ActionEvent evt)
{
    if(evt.getSource() == btnNext)
    {
        if(!isSorting)
        {
            isSorting = true;
            btnMore.setEnabled(false);
            btnLess.setEnabled(false);
            btnRandom.setEnabled(false);
            numData = 0;
            curData = -1;
            originalArray = activeArray;
            execAlgorythm();
        }
        if(curData < numData - 1)
        {
            if(curData == numData - 1)
            {
                btnNext.setEnabled(false);
                btnAuto.setEnabled(false);
            }
            curData ++;
            repaint();
        }
        if(curData >= 1) btnPrev.setEnabled(true);
        return;
    }
    if(evt.getSource() == btnPrev)
    {
        if(curData > 0)
        {
            btnNext.setEnabled(true);
            btnAuto.setEnabled(true);
            curData --;
            repaint();
        }
        if(curData == 0) btnPrev.setEnabled(false);
        return;
    }
    if(evt.getSource() == btnAuto)
    {
        // If present sorting process
        if(isProcess)
        {
            btnAuto.setLabel(getParameter("auto"));
            if(curData > 0) btnPrev.setEnabled(true);
            sorting.suspend();
            isProcess = false;
```

```
    }
    else
    {
      btnAuto.setLabel(getParameter("stop"));
      btnPrev.setEnabled(false);
      if(!isSorting)
      {
        isSorting = true;
        btnMore.setEnabled(false);
        btnLess.setEnabled(false);
        btnRandom.setEnabled(false);
        numData = 0;
        curData = -1;
        originalArray = activeArray;
        execAlgorythm();
      }
      isProcess = true;
      sorting.resume();
    }
    return;
  }
  if(evt.getSource() == btnReset)
  {
    numData = curData = 0;
    isSorting = isProcess = false;
    sorting.suspend();
    btnMore.setEnabled(true);
    btnLess.setEnabled(true);
    btnRandom.setEnabled(true);
    btnPrev.setEnabled(false);
    btnNext.setEnabled(true);
    btnAuto.setEnabled(true);
    btnAuto.setLabel(getParameter("auto"));
    repaint();
    return;
  }
  if(evt.getSource() == btnRandom)
  {
    for(int i = 0; i < number; i ++)
    {
      activeArray[i] = originalArray[i] =
                       (int)Math.round(randomCounter.nextDouble() * 99);
    }
    repaint();
    return;
  }
  if(evt.getSource() == btnMore)
  {
    if(number < maxElements)
    {
      numLabel.setText("  Num: " + (++number) + "    ");
      repaint();
    }
    return;
  }
  if(evt.getSource() == btnLess)
  {
    if(number > minElements)
    {
```

```
      numLabel.setText("  Num: " + (--number) + "    ");
      repaint();
    }
    return;
  }
  if(evt.getSource() == btnFaster)
  {
    delay += 50;
    delayLabel.setText("wait: " + Integer.toString((int)delay));
    return;
  }
  if(evt.getSource() == btnSlower)
  {
    if(delay > 100)
      delay -= 50;
    delayLabel.setText("wait: " + Integer.toString((int)delay));
    return;
  }
}

/**
 * Gets integer parameter. If parameter not specified, returns defaultValue
 * @param parameterName The name of parameter
 * @param defaultValue  Default value
 * @see ReadInt()
 */
private int getIntParm(String parameterName, int defaultValue)
{
  return readInt(getParameter(parameterName), defaultValue);
}

private int readInt(String param, int defaultValue)
{
  try
  {
    return Integer.parseInt(param);
  }
  catch (NumberFormatException e)
  {
    return defaultValue;
  }
}

private int readColor (String param, int defaultColor)
{
  if (param == null)
  {
    return defaultColor;
  }

  int color = defaultColor;
  if (param.startsWith ("0x"))
  {
    try
    {
      color = Integer.parseInt (param.substring (2), 1);
    }
    catch (NumberFormatException e)
    {
```

```
        color = defaultColor;
      }
    }
    else
      if (param.startsWith ("#"))
      {
        try
        {
          color = Integer.parseInt (param.substring (1), 1);
        }
        catch (NumberFormatException e)
        {
          color = defaultColor;
        }
      }
      else
        color = defaultColor;
    return color;
  }
}

class Data
{
  int array[];
  int selectedItems[];
  String status;
  int lastEdgeNumber;

  public Data(int arrayLength)
  {
    array = new int [arrayLength];
    selectedItems = new int [3];
  }

  public void putData(int a[], int si[], int n, String str)
  {
    for(int i = 0; i < a.length; i++)
    {
      array[i] = a[i];
    }
    for(int i = 0; i < si.length; i ++)
    {
      selectedItems[i] = si[i];
    }
    lastEdgeNumber = n;
    status = str;
  }
}
```

Приложение 3. Исходный текст реализации алгоритма с использованием SWITCH-технологии

```
/**
   Applet visualising HeapSort algorithm
   @author Georgy Udov (udov@green.ifmo.ru)
*/

import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.math.*;
import java.text.MessageFormat;

public class HeapSort extends Applet implements Runnable, ActionListener
{
  // Нить для автоматического режима (используется таймер)
  Thread sorting = null;

  // Определяет, включен ли автоматический (использующий таймер) режим
  private boolean auto = false;

  // Текстовая область для вывода размера сортируемого массива
  private TextField numField = new TextField(1);

  // Генератор случайных чисел
  Random randomCounter = new Random();

  // Ограничения (соответственно, максимальное и минимальное количество
  // элементов в массиве, максимальное количество запоминаемых
  // состояний, минимальная задержка в автоматическом режиме)
  private final int maxElements = 15;
  private final int minElements = 07;
  private final int maxData     = 200;
  private final int minDelay    = 100;

  // Цвета линий, фона, активного элемента
  // (значения будут загружены их html-файла)
  private Color foregroundColor = null;
  private Color backgroundColor = null;
  private Color activeColor = null;

  // Текущее значение задержки и текущее визуализирумое состояние
  private int delay, number;

  // Переменные для кнопок (быстрее, медленнее, вперед, назад,
  // автоматический режим, сброс, случайно, увеличить/уменьшить
  // количество элементов в массиве)
  private Button btnFaster;
  private Button btnSlower;
  private Button btnNext;
  private Button btnPrev;
  private Button btnAuto;
  private Button btnReset;
  private Button btnRandom;
  private Button btnMore;
```

```
private Button btnLess;

// Текстовые области для вывода текущих значений количества элементов в
// массиве и задержки в автоматическом режиме
private Label numLabel   = new Label ();
private Label delayLabel = new Label ();

// Шрифт для вывода словесного описания визуализируемого шага
private Font font = new Font ("Arial", Font.PLAIN, 12);

// Строки (различные фразы, используемые в описании)
private String phase1Mess, compareMess, changeMess, compareEndMess,
               firstLastMess, phase2Mess, sortFinishMess;

// Сортируемый и входной массивы
public int activeArray[]   = new int [maxElements];
public int originalArray[] = new int [maxElements];

// Радиус элементов пирамиды
private int radius = 10;

// Массив визуализируемых шагов
public Data data[] = new Data [200];

// Количество визуализируемых шагов и текущий шаг
private int numData = 0, curData = 0;

// Определяет, соответственно, включен ли автоматический режим и
// была ли осуществлена сортировка
boolean isProcess = false, isSorting = false;

/**
 * Конструктор по умолчанию
 */
public HeapSort()
{
}

/**
 * Возвращает информацию об апплете
 */
public String getAppletInfo()
{
  return "Name: HeapSort visualiser\r\n" +
  "Author: Georgy G. Udov(udov@green.ifmo.ru)\r\n";
}

/**
 * Создает кнопку
   @param parameterName имя параметра,
   соответствующго кнопке (в html-файле)
 */
private Button createButton(String parameterName)
{
  final Button button = new Button(getParameter(parameterName));
  button.addActionListener(this);
  return button;
}
```

```
/**
 * Инициализирует апплет
 */
public void init()
{
  // Загрузка и создание цветов
  foregroundColor = new Color (readColor
    (getParameter ("fgcolor"), 0));
  backgroundColor = new Color (readColor
    (getParameter ("bkcolor"), 0x00ffffff));
  activeColor     = new Color (readColor
    (getParameter ("activecolor"), 0x00ff0000));

  // Загрузка фраз сообщений
  phase1Mess     = getParameter("phase1mess");
  compareMess    = getParameter("comparemess");
  changeMess     = getParameter("changemess");
  compareEndMess = getParameter("compareendmess");
  firstLastMess  = getParameter("firstlastmess");
  phase2Mess     = getParameter("phase2mess");
  sortFinishMess = getParameter("sortfinishmess");

  // Создание кнопок
  btnRandom = createButton("randomize");
  btnMore   = createButton("more");
  btnLess   = createButton("less");
  btnNext   = createButton("next");
  btnPrev   = createButton("prev");
  btnAuto   = createButton("auto");
  btnFaster = createButton("faster");
  btnSlower = createButton("slower");
  btnReset  = createButton("reset");

  // Загрузка величины задержки при автоматическом режиме
  delay  = getIntParm("delay", minDelay);
  if(delay < minDelay) delay = minDelay;

  // Загрузка количества элементов в рабочем массиве
  number = getIntParm("number", maxElements);
  if (number > maxElements) number = maxElements;
  if (number < minElements) number = minElements;

  // Загрузка элементов рабочего массива
  int valueCount = 0;
  StringTokenizer st = new
    StringTokenizer(getParameter("values"), " ");
  while (st.hasMoreTokens() && valueCount < number)
  {
    activeArray[valueCount++] = readInt(st.nextToken(), 0);
  }

  // Установка шрифта
  getGraphics().setFont(font);

  // Установка цвета фона
  setBackground(backgroundColor);

  /**
   * Настройка интерфейсных панелей
```

```
     */
    setLayout (new BorderLayout ());

    numLabel.setAlignment(1);
    numLabel.setSize(50, 0);
    numLabel.setBackground(new Color(0x00dfdfdf));
    numLabel.setFont(new Font("Arial", Font.BOLD, 12));
    numLabel.setText("  Num: " + number + "    ");

    delayLabel.setAlignment(1);
    delayLabel.setSize(60, 0);
    delayLabel.setBackground(new Color(0x00dfdfdf));
    delayLabel.setFont(new Font("Arial", Font.BOLD, 12));
    delayLabel.setText("wait: " + Integer.toString((int)delay));

    Panel controlPanel = new Panel ();
    Panel controlPanel1 = new Panel ();
    Panel controlPanel2 = new Panel ();

    controlPanel.setLayout(new GridLayout (2, 1, 3, 5));
    controlPanel1.setLayout(new GridLayout (1, 5, 1, 5));

    controlPanel1.add(btnNext);
    controlPanel1.add(btnPrev);
    controlPanel1.add(btnAuto);
    controlPanel1.add(btnReset);
    controlPanel1.add(btnRandom);

    controlPanel2.add(btnLess);
    controlPanel2.add(numLabel);
    controlPanel2.add(btnMore);
    controlPanel2.add(btnSlower);
    controlPanel2.add(delayLabel);
    controlPanel2.add(btnFaster);

    controlPanel.add(controlPanel1);
    controlPanel.add(controlPanel2);

    add("South", controlPanel);

    // Initialising the data
    for(int i = 0; i < maxData; i ++)
      data[i] = new Data(maxElements);

    originalArray = activeArray;
    numData = curData = 0;
    btnPrev.setEnabled(false);
  }

  /**
   * Методы интерфейса Applet - запуск и приостановка аплета
   */
  public void start()
  {
    if (sorting == null)
    {
      sorting = new Thread(this);
      sorting.start();
      sorting.suspend();
```

```
  }
}

public void stop()
{
  if (sorting != null)
  {
    sorting.stop();
    sorting = null;
  }
}

public void destroy()
{
}

/**
 * Метод интерфейса Runnable - реализация автоматического режима
 */
public void run()
{
  for(;;)
  {
    if(curData < numData - 1)
    {
      curData ++;
      if(curData == numData - 1)
      {
        btnNext.setEnabled(false);
        btnAuto.setEnabled(false);
        btnPrev.setEnabled(true);
        btnAuto.setLabel(getParameter("auto"));
        isProcess = false;
      }
    }
    repaint();
    if(isProcess)
    try
    {
      Thread.sleep(delay);
    }
    catch (InterruptedException e)
    {
      stop();
    }
    if(curData >= numData - 1)
    {
      sorting.suspend();
    }
  }
}

/**
 * Вспомогательные математические функции
 */
private int max(int a, int b)
{
  return a > b ? a : b;
}
```

```
private int min(int a, int b)
{
  return a < b ? a : b;
}

private double log2(double a)
{
  return Math.log(a)/Math.log(2.0);
}

private int step(int a, int b)
{
  int res = 1;
  for(int i = 0; i < b; i ++)
    res *= a;
  return res;
}

// Return whether x is end of layer or not
boolean isEnd(int x)
{
  x += 1;
  while(x % 2 == 0)
    x /= 2;
  if(x == 1)
    return true;
  return false;
}

// Attention: a and b are INDEXES of values in array which
// will be swaped. It is NOT variables.
private void swap(int a, int b)
{
  int c;
  c = activeArray[a];
  activeArray[a] = activeArray[b];
  activeArray[b] = c;
}


/**
 * Реализация автомата A0
 */

private boolean x1(int r, int last)
{
  return 2*r+1 > last;
}

private boolean x2(int r, int last)
{
  return 2*r+1 == last;
}

private boolean x3(int r, int last)
{
  return activeArray[r] < activeArray[2*r+1];
}
```

```
private boolean x4(int r, int last)
{
  return activeArray[2*r+1] >= activeArray[2*r+2];
}

private boolean x5(int r, int last)
{
  return activeArray[r] < activeArray[2*r+2];
}

private int z1(int r, int first)
{
  r = first;

  return r;
}

private int z2(int r, int first)
{
  swap(r, 2*r+1);
  r = 2*r+1;
  return r;
}

private int z3(int r, int first)
{
  swap(r, 2*r+2);
  r = 2*r+2;
  return r;
}

private void sift(int first, int last, int numStadia)
{
  // Состояние автомата
  int state;

  // Рабочая переменная алгоритма
  int r = 0;

  // Вспомогательные переменные для вывода информации
  // в визуализатор (протоколирования)
  int selected[] = new int [3];
  String str;

  r = z1(r, first);
  state = 1;

  while(state != 0)
  {
    switch(state)
    {
    case 1:
      if(x1(r, last))
      {
        state = 0;
      }
      else if(x2(r, last))
      {
```

```
      state = 2;
    }
    else if(x3(r, last) && x4(r, last))
    {
      // Протоколирование
      selected[0] = r;
      selected[1] = 2*r+1;
      selected[2] = -1;
      str = numStadia == 1 ? phase1Mess : phase2Mess;
      Object[] tmp1 = {new Integer(activeArray[r])};
      Object[] tmp2 = {new Integer(activeArray[2*r+1])};
      str += MessageFormat.format(compareMess, tmp1) +
             MessageFormat.format(changeMess, tmp2)  + " ";
      data[numData++].putData(activeArray, selected, last, str);

      r = z2(r, first);

      // Протоколирование
      data[numData++].putData(activeArray, selected, last, str);
    }
    else if(x5(r, last))
    {
      // Протоколирование
      selected[0] = r;
      selected[1] = 2*r+2;
      selected[2] = -1;
      str = numStadia == 1 ? phase1Mess : phase2Mess;
      Object[] tmp1 = {new Integer(activeArray[r])};
      Object[] tmp2 = {new Integer(activeArray[2*r+2])};
      str += MessageFormat.format(compareMess, tmp1) +
             MessageFormat.format(changeMess, tmp2)  + " ";
      data[numData++].putData(activeArray, selected, last, str);

      r = z3(r, first);

      // Протоколирование
      data[numData++].putData(activeArray, selected, last, str);
    }
    else
    {
      // Протоколирование
      selected[0] = r;
      selected[1] = activeArray[2*r+2] > activeArray[2*r+1] ?
        2*r+2 : 2*r+1;
      selected[2] = -1;
      str  = numStadia == 1 ? phase1Mess : phase2Mess;
      Object [] tmp3 = {new Integer(activeArray[r])};
      str += MessageFormat.format(compareMess, tmp3) +
             compareEndMess + " ";
      data[numData++].putData(activeArray, selected, last, str);

      state = 0;
    }
    break;
  case 2:
    if(x3(r, last))
    {
      // Протоколирование
      selected[0] = r;
```

```
        selected[1] = 2*r+1;
        selected[2] = -1;
        str = numStadia == 1 ? phase1Mess : phase2Mess;
        Object[] tmp1 = {new Integer(activeArray[r])};
        Object[] tmp2 = {new Integer(activeArray[2*r+1])};
        str += MessageFormat.format(compareMess, tmp1) +
               MessageFormat.format(changeMess, tmp2)  + " ";
        data[numData++].putData(activeArray, selected, last, str);

        r = z2(r, first);
        state = 0;

        // Протоколирование
        data[numData++].putData(activeArray, selected, last, str);
      }
      else
      {
        // Протоколирование
        selected[0] = r;
        selected[1] = 2*r+1;
        selected[2] = -1;
        str  = numStadia == 1 ? phase1Mess : phase2Mess;
        Object [] tmp3 = {new Integer(activeArray[r])};
        str += MessageFormat.format(compareMess, tmp3) +
               compareEndMess + " ";
        data[numData++].putData(activeArray, selected, last, str);

        state = 0;
      }
      break;
    }
  }
}

/**
 * Реализация алгоритма
 * (функция реализовывает алгоритм на входных данных,
 * визуализируемые шаги запоминает в массиве data)
 */
private void execAlgorythm()
{
  int x, l, r;
  int selected[] = new int [3];
  String str;

  l = (number / 2);
  r = number - 1;
  while(l > 0)
  {
    l --;
    sift(l, r, 1);
  }

  while(r > 0)
  {
    selected[0] = 0;
    selected[1] = r;
    selected[2] = -1;
    Object [] tmp4 = {new Integer(activeArray[0]),
```

```
      new Integer(activeArray[r])};
    str = MessageFormat.format(firstLastMess, tmp4);
    data[numData].putData(activeArray, selected, r, str);
    numData ++;
    swap(0, r--);
    // Outputing the status
    selected[0] = 0;
    selected[1] = r+1;
    selected[2] = -1;
    data[numData].putData(activeArray, selected, r, str);
    numData ++;
    sift(l, r, 2);
  }
  selected[0] = -1;
  selected[1] = -1;
  selected[2] = -1;
  data[numData].putData(activeArray, selected, r, sortFinishMess);
  numData ++;
}

/**
 * Возвращает, является ли элемент с индексом a
 * на данном шаге выделенным
 */
private boolean isSelected(int a)
{
  if(isSorting)
  {
    for(int i = 0; i < 3; i ++)
      if(a == data[curData].selectedItems[i])
        return true;
  }
  return false;
}

/**
 * Перерисовывает рабочую область
 */
public void paint(Graphics g)
{
  int x0, y0, width, height;
  int screenWidth  = getSize().width;
  int screenHeight = getSize().height;
  int rSin, rCos;
  double tmp;
  String str, activeString;
  int array[] = new int [maxElements];
  int lastEdgeNumber;

  if(isSorting)
  {
    array = data[curData].array;
    lastEdgeNumber = data[curData].lastEdgeNumber;
    activeString = data[curData].status;
  }
  else
  {
    array = originalArray;
    lastEdgeNumber = number - 1;
```

```
  activeString = "";
}

g.setFont(font);
g.setColor(foregroundColor);

/**
 * Перерисовываем массив (ленточное представление)
 */

x0 = 5;
y0 = 5;
height = 20;
width = (screenWidth - 10) / number;
for(int i = 0; i < number; i ++)
{
  if(isSelected(i))
  {
    g.setColor(activeColor);
    g.fillRect(x0, y0, width, height);
  }
  g.setColor(foregroundColor);
  g.drawRect(x0, y0, width, height);
  str = Integer.toString(array[i]);
  g.drawString(str, x0 + width/2 - str.length()*font.getSize()/4,
               y0 + height/2 + font.getSize()/2 - 1);
  x0 += width;
}

/**
 * Перерисовываем пирамиду (пирамидальное представление массива)
 */

y0 += height + 20;
x0 = screenWidth / 2;
height = (int)Math.ceil(log2(number + 1)); // высота дерева в элементах
// Ширина последнего слоя дерева в элементах
width = max(step(2, height - 1), number - (step(2, height - 1) - 1));
// Ширина элемента в пикселях
width = (screenWidth - 20) / (width);
width *= step(2, (height - 1)); // Увеличиваем...
// Высота элемента в пикселях
height = (screenHeight - y0 - 2*radius -
  (font.getSize() + 2)*5 - 20) / height;
for(int i = 0; i < number; i ++)
{
  // Вычисляем R*sin(alpha) и R*cos(alpha)
  tmp = Math.sqrt(width / 2.0 * width / 2.0 / 4.0 + height * height);
  rSin = (int)Math.round( (double)radius *
    (double)(width / 2) / tmp / 2.0);
  rCos = (int)Math.round( (double)radius * (double)height / tmp);
  if(isSelected(i))
  {
    g.setColor(activeColor);
    g.fillArc(x0 - radius, y0 - radius, 2 * radius + 1,
      2 * radius + 1, 0, 360);
  }
  g.setColor(foregroundColor);
  g.drawArc(x0 - radius, y0 - radius, 2 * radius, 2 * radius, 0, 360);
```

```
str = Integer.toString(array[i]);
g.drawString(str, x0 - str.length()*font.getSize() / 4,
                  y0 + font.getSize() / 2 - 1);
if( 2*i+1 <= lastEdgeNumber )
{
  // Отрисовываем левое ребро (дугу)
  g.drawLine(x0 - (width / 2)/2 + rSin, y0 + height - rCos,
    x0 - rSin, y0 + rCos);
  if(isSelected(2*i+1) && isSelected(i))
  {
    int addX = (width / 4 - 2 * rSin);
    int addY = (height - 2 * rCos);
    double k = (double)addX / (double)addY;
    addX /= 6;
    addY /= 6;
    double x1 = x0 - (width / 2)/2 + rSin + addX;
    double y1 = y0 + height - rCos - addY;
    double b  = y1 - k*x1;
    double d = (2*k*b - 2*x1 - 2*y1*k)*(2*k*b - 2*x1 - 2*y1*k) -
               4*(1 + k*k)*(x1*x1 + b*b + y1*y1 - 2*y1*b - 9);
    double xOne = (-(2*k*b - 2*x1 - 2*y1*k) +
      Math.sqrt(d))/(2*(1+k*k));
    double xTwo = (-(2*k*b - 2*x1 - 2*y1*k) -
      Math.sqrt(d))/(2*(1+k*k));
    double yOne = k*xOne + b;
    double yTwo = k*xTwo + b;
    int xZero = x0 - (width / 2)/2 + rSin + 1;
    int yZero = y0 + height - rCos - 1;
    g.drawLine(xZero, yZero,
               (int)Math.round(xOne), (int)Math.round(yOne));
    g.drawLine(xZero, yZero,
               (int)Math.round(xTwo), (int)Math.round(yTwo));
    // Другая сторона
    addX = (width / 4 - 2 * rSin);
    addY = (height - 2 * rCos);
    k = (double)addX / (double)addY;
    addX /= 6;
    addY /= 6;
    x1 = x0 - rSin - addX;
    y1 = y0 + rCos + addY;
    b  = y1 - k*x1;
    d = (2*k*b - 2*x1 - 2*y1*k)*(2*k*b - 2*x1 - 2*y1*k) -
      4*(1 + k*k)*(x1*x1 + b*b + y1*y1 - 2*y1*b - 9);
    xOne = (-(2*k*b - 2*x1 - 2*y1*k) +
      Math.sqrt(d))/(2*(1+k*k));
    xTwo = (-(2*k*b - 2*x1 - 2*y1*k) -
      Math.sqrt(d))/(2*(1+k*k));
    yOne = k*xOne + b;
    yTwo = k*xTwo + b;
    xZero = x0 - rSin - 1;
    yZero = y0 + rCos + 1;
    g.drawLine(xZero, yZero,
      (int)Math.round(xOne), (int)Math.round(yOne));
    g.drawLine(xZero, yZero,
      (int)Math.round(xTwo), (int)Math.round(yTwo));

  }
}
```

```
if( 2*i+2 <= lastEdgeNumber )
{
  // Отрисовываем правое ребро (дугу)
  g.drawLine(x0 + (width / 2)/2 - rSin,
             y0 + height - rCos, x0 + rSin, y0 + rCos);
  if(isSelected(2*i+2) && isSelected(i))
  {
    int addX = (width / 4 - 2 * rSin);
    int addY = (height - 2 * rCos);
    double k = -(double)addX / (double)addY;
    addX /= 6;
    addY /= 6;
    double x1 = x0 + (width / 2)/2 - rSin - addX;
    double y1 = y0 + height - rCos - addY;
    double b  = y1 - k*x1;
    double d  = (2*k*b - 2*x1 - 2*y1*k)*(2*k*b - 2*x1 - 2*y1*k) -
      4*(1 + k*k)*(x1*x1 + b*b + y1*y1 - 2*y1*b - 9);
    double xOne = (-(2*k*b - 2*x1 - 2*y1*k) +
      Math.sqrt(d))/(2*(1+k*k));
    double xTwo = (-(2*k*b - 2*x1 - 2*y1*k) -
      Math.sqrt(d))/(2*(1+k*k));
    double yOne = k*xOne + b;
    double yTwo = k*xTwo + b;
    int xZero = x0 + (width / 2)/2 - rSin - 1;
    int yZero = y0 + height - rCos - 1;
    g.drawLine(xZero, yZero,
               (int)Math.round(xOne), (int)Math.round(yOne));
    g.drawLine(xZero, yZero,
               (int)Math.round(xTwo), (int)Math.round(yTwo));
    // Другая сторона
    addX = (width / 4 - 2 * rSin);
    addY = (height - 2 * rCos);
    k = -(double)addX / (double)addY;
    addX /= 6;
    addY /= 6;
    x1 = x0 + rSin + addX;
    y1 = y0 + rCos + addY;
    b  = y1 - k*x1;
    d = (2*k*b - 2*x1 - 2*y1*k)*(2*k*b - 2*x1 - 2*y1*k) -
               4*(1 + k*k)*(x1*x1 + b*b + y1*y1 - 2*y1*b - 9);
    xOne = (-(2*k*b - 2*x1 - 2*y1*k) + Math.sqrt(d))/(2*(1+k*k));
    xTwo = (-(2*k*b - 2*x1 - 2*y1*k) - Math.sqrt(d))/(2*(1+k*k));
    yOne = k*xOne + b;
    yTwo = k*xTwo + b;
    xZero = x0 + rSin + 1;
    yZero = y0 + rCos + 1;
    g.drawLine(xZero, yZero,
               (int)Math.round(xOne), (int)Math.round(yOne));
    g.drawLine(xZero, yZero,
               (int)Math.round(xTwo), (int)Math.round(yTwo));
  }
}

// Модифицируем X0 и Y0
if(isEnd(i+1) && i != number - 1)
{
  width /= 2;
  x0 = screenWidth - x0 - width/2;
  y0 += height;
```

```
    }
    else
      x0 += width;
  }
  x0 = 10;
  y0 += 2 * radius - 2;

  if(isSorting)
  {
    g.drawLine(0, y0, screenWidth, y0);
    y0 += 2 + font.getSize();

    // Разбиваем и выводим строку (описание текущего шага)
    StringTokenizer tokens = new StringTokenizer(activeString, " ");
    FontMetrics metrics = g.getFontMetrics();
    String words[] = new String[tokens.countTokens()];
    for (int i = 0; i < words.length; i++)
    {
      words[i] = tokens.nextToken();
    }

    str = words[0];
    for (int i = 1; i < words.length; i++)
    {
      str += ' ' + words[i];
      if (metrics.stringWidth(str) > screenWidth - 2 * x0)
      {
        str = str.substring(0, str.lastIndexOf(' '));
        g.drawString(str.trim(), x0, y0);
        y0 += font.getSize() + 2;
        str = "";
        i --;
      }
    }
    g.drawString(str.trim(), x0, y0);
  }
}

/**
 * Обработка событий от кнопок
 */
public void actionPerformed(ActionEvent evt)
{
  if(evt.getSource() == btnNext)
  {
    if(!isSorting)
    {
      isSorting = true;
      btnMore.setEnabled(false);
      btnLess.setEnabled(false);
      btnRandom.setEnabled(false);
      numData = 0;
      curData = -1;
      originalArray = activeArray;
      execAlgorythm();
    }
    if(curData < numData - 1)
    {
      if(curData == numData - 1)
```

```
    {
      btnNext.setEnabled(false);
      btnAuto.setEnabled(false);
    }
    curData ++;
    repaint();
  }
  if(curData >= 1) btnPrev.setEnabled(true);
  return;
}
if(evt.getSource() == btnPrev)
{
  if(curData > 0)
  {
    btnNext.setEnabled(true);
    btnAuto.setEnabled(true);
    curData --;
    repaint();
  }
  if(curData == 0) btnPrev.setEnabled(false);
  return;
}
if(evt.getSource() == btnAuto)
{
  if(isProcess)
  {
    btnAuto.setLabel(getParameter("auto"));
    if(curData > 0) btnPrev.setEnabled(true);
    sorting.suspend();
    isProcess = false;
  }
  else
  {
    btnAuto.setLabel(getParameter("stop"));
    btnPrev.setEnabled(false);
    if(!isSorting)
    {
      isSorting = true;
      btnMore.setEnabled(false);
      btnLess.setEnabled(false);
      btnRandom.setEnabled(false);
      numData = 0;
      curData = -1;
      originalArray = activeArray;
      execAlgorythm();
    }
    isProcess = true;
    sorting.resume();
  }
  return;
}
if(evt.getSource() == btnReset)
{
  numData = curData = 0;
  isSorting = isProcess = false;
  sorting.suspend();
  btnMore.setEnabled(true);
  btnLess.setEnabled(true);
  btnRandom.setEnabled(true);
```

```
      btnPrev.setEnabled(false);
      btnNext.setEnabled(true);
      btnAuto.setEnabled(true);
      btnAuto.setLabel(getParameter("auto"));
      repaint();
      return;
    }
    if(evt.getSource() == btnRandom)
    {
      for(int i = 0; i < number; i ++)
      {
        activeArray[i] = originalArray[i] =
          (int)Math.round(randomCounter.nextDouble() * 99);
      }
      repaint();
      return;
    }
    if(evt.getSource() == btnMore)
    {
      if(number < maxElements)
      {
        numLabel.setText("  Num: " + (++number) + "    ");
        repaint();
      }
      return;
    }
    if(evt.getSource() == btnLess)
    {
      if(number > minElements)
      {
        numLabel.setText("  Num: " + (--number) + "    ");
        repaint();
      }
      return;
    }
    if(evt.getSource() == btnFaster)
    {
      delay += 50;
      delayLabel.setText("wait: " + Integer.toString((int)delay));
      return;
    }
    if(evt.getSource() == btnSlower)
    {
      if(delay > 100)
        delay -= 50;
      delayLabel.setText("wait: " + Integer.toString((int)delay));
      return;
    }
  }

  /**
   * Возвращает целый параметр. Если параметр не указан -
   * возвращает defaultValue
   * @param parameterName Имя параметра
   * @param defaultValue  Значение по умолчанию
   * @see ReadInt()
   */
  private int getIntParm(String parameterName, int defaultValue)
  {
```

```
    return readInt(getParameter(parameterName), defaultValue);
  }

  private int readInt(String param, int defaultValue)
  {
    try
    {
      return Integer.parseInt(param);
    }
    catch (NumberFormatException e)
    {
      return defaultValue;
    }
  }

  private int readColor (String param, int defaultColor)
  {
    if (param == null)
    {
      return defaultColor;
    }

    int color = defaultColor;
    if (param.startsWith ("0x"))
    {
      try
      {
        color = Integer.parseInt (param.substring (2), 1);
      }
      catch (NumberFormatException e)
      {
        color = defaultColor;
      }
    }
    else
      if (param.startsWith ("#"))
      {
        try
        {
          color = Integer.parseInt (param.substring (1), 1);
        }
        catch (NumberFormatException e)
        {
          color = defaultColor;
        }
      }
      else
        color = defaultColor;
    return color;
  }
}

class Data
{
  int array[];
  int selectedItems[];
  String status;
  int lastEdgeNumber;
```

```
  public Data(int arrayLength)
  {
    array = new int [arrayLength];
    selectedItems = new int [3];
  }

  public void putData(int a[], int si[], int n, String str)
  {
    for(int i = 0; i < a.length; i++)
    {
      array[i] = a[i];
    }
    for(int i = 0; i < si.length; i ++)
    {
      selectedItems[i] = si[i];
    }
    lastEdgeNumber = n;
    status = str;
  }
}
```