Санкт-Петербургский государственный университет
информационных технологий, механики и оптики

Кафедра «Компьютерные технологии»

Г. Г. Удов, А. А. Шалыто

Построение визуализатора алгоритма пирамидальной сортировки набора чисел на базе технологии *Vizi*

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

На кафедре «Компьютерные технологии» СПбГУ ИТМО для разработки и реализации визуализаторов алгоритмов на основе конечных автоматов была предложена технология *Vizi* [1].

Визуализатор — это программа, в процессе работы которой на экране компьютера динамически демонстрируется применение алгоритма к выбранному набору данных. Алгоритм пирамидальной сортировки [2, 3, 4, 5, 6] является одним из фундаментальных методов сортировки набора чисел.

В данной работе строится логика и реализация визуализатора алгоритма пирамидальной сортировки набора чисел на базе технологии *Vizi*.

1. Анализ литературы

Информация о рассматриваемом алгоритме содержится в книгах Н. Вирта [2], Д. Кнута [3], И.В. Романовского [4] и Т. Кормена [6].

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 [2]. 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 [3], 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 [3].

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 [3] был предложен «лаконичный» способ построения пирамиды «на том же месте». Его процедура сдвига (функция `sift`) представлена на листинге 1. Здесь $h[1] \dots h[n]$ — некий массив, причем $h[m] \dots h[n] (m = n/2 + 1)$ 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 над-

лежащую 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 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

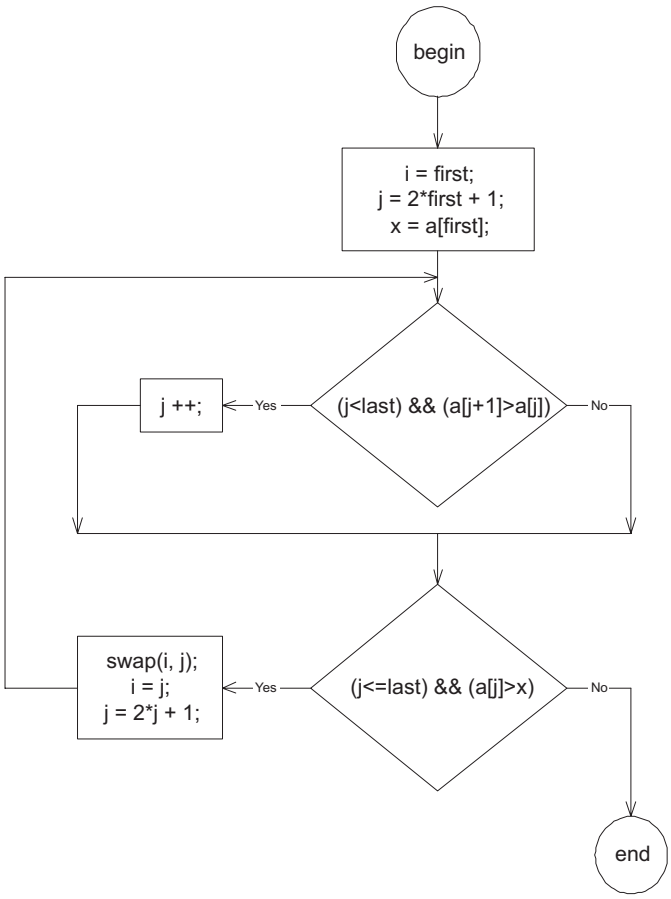

Рис. 1. Схема алгоритма сдвига

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 модели данных

Моделью данных называется класс, содержащий все необходимые алгоритму переменные и структуры данных. Для реализации алгоритма пирамидальной сортировки набора чисел требуется модель данных, содержащая:

- сортируемый массив целых чисел `a`;
- номер текущего просеиваемого элемента `l`;

- индекс элемента массива `r`, являющегося последним элементом пирамиды;
- временная переменная `tmp` для выполнения операций `swap`;
- переменная `numStadia` автомата Main, используемая исключительно в целях визуализации — номер стадии работы алгоритма;
- переменная автомата Sift — номер родительской вершины `i`;
- переменная автомата Sift — номер дочерней вершины `j`;
- переменная автомата Sift — значение в родительской(просеиваемой) вершине `x`;
- экземпляр апплета `visualizer`.

В последних версиях *Vizi* построение модели данных осуществляется автоматически на основе определенных в XML-представлении алгоритма глобальных и локальных переменных.

4. Описание алгоритма на языке XML

Описание алгоритма на языке XML, используемом в технологии *Vizi*, изоморфно строится по его реализации, приведенной в листинге 2, а также в работе [5]. Полученное XML-описание алгоритма, дополненное комментариями (текстовой информацией о визуализируемых шагах) и информацией о действиях по отрисовке состояний, помещается в файл `HeapSort-Algorithm.xml`, приведенный в приложении 1.

Определение каждой переменной производится с помощью тега `variable`. У этого тега имеется атрибут `description`, содержащий описание соответствующей переменной. Переменные, описанные внутри тегов `auto`, становятся локальными переменными соответствующей процедуры, а описанные вне данных тегов — глобальными переменными. В модель данных включаются все локальные и глобальные переменные. При этом к имени каждой локальной переменной добавляется префикс, состоящий из имени процедуры, в котором они определены, и символа подчеркивания.

В описании реализации алгоритма оператор присваивания представляется знаком `@=`, оператор ветвления — тегом `if`, цикл с предусловием — тегом `while`. У перечисленных тегов есть атрибуты, которые, в частности, содержат текстовые описания вызываемых операторов, используемые при составлении комментариев в автоматически генерируемых файлах реализации, а также комментарии для понимания выполняемого шага алгоритма.

Действия, которые необходимо выполнять при работе алгоритма, локализуются внутри тегов `action`. Они, в свою очередь, размещаются в тегах `step`, отмечающих шаги алгоритма, которые требуется визуализировать, заостряя на них внимание пользователя. Важнейшим атрибутом этого тега является атрибут `level`, хранящий целую величину, называемую «интересностью» данного шага. В текущей реализации *Vizi* «интересность» может принимать значения -1, 0, 1.

В случае если «интересность» имеет значение -1, данный шаг не будет выделяться визуализатором, и программа сразу перейдет к содержимому следующего тега `step`. Если «интересность» имеет значение 0, данный шаг будет выделяться визуализатором только при последовательном просмотре шагов, в неавтоматическом режиме. Если же «интересность» имеет значение 1, то данный шаг будет всегда выделяться визуализатором.

В XML-описании алгоритма также должен использоваться тег `draw`, содержащий вызов метода визуализатора, выполняющего перерисовку графического изображения массива. Так как дальнейшая реализация алгоритма осуществляется на основе автоматного подхода, все теги, которые отвечают за функционирование одного и того же автомата, собираются в одном теге `auto`. Эти теги, вместе с тегом `toString`, вкладываются в тег `algorithm`.

Кроме того, строится еще два файла. Первый из них (`HeapSort.xml`) содержит тег `visualizer`, хранящий общую информацию, такую как название визуализатора, имя автора и т. д., а также ссылки на два других файла — `HeapSort-Algorithm.xml` и `HeapSort-Configuration.xml`.

Третий файл (`HeapSort-Configuration.xml`) содержит параметры административного интерфейса визуализатора, рассматриваемого ниже.

5. Описание интерфейса визуализатора

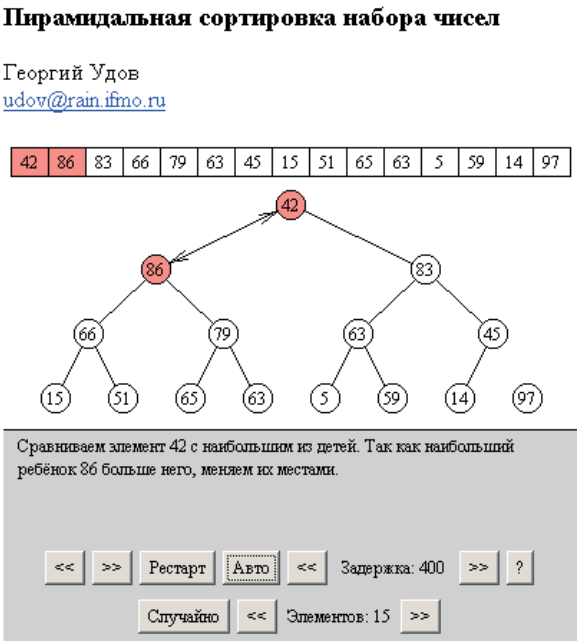

Рис. 2. Внешний вид аплета

Рассмотрим пользовательский интерфейс визуализатора (рис. 2).

Управление процессом визуализации пользователь осуществляет посредством нажатия на кнопки. Рассмотрим верхний ряд кнопок (слева направо). Первая и вторая кнопки, соответственно, осуществляют переход к следующему и предыдущему шагу визуализации. Кнопка «Рестарт» позволяет запустить процесс визуализации сначала. Кнопка «Авто» включает автоматический режим. В автоматическом режиме переход к следующему шагу визуализации осуществляется автоматически, с задержкой, длительность которой устанавливается при помощи пятой и шестой кнопок и отображается в текстовой области, расположенной между ними. Седьмая кнопка (на которой изображен вопросительный знак) выводит краткую информацию о визуализаторе — название алгоритма, имена и адреса электронной почты автора, руководителя и создателя технологии.

Аналогично рассмотрим нижний ряд кнопок. Кнопка «Случайно» позволяет заполнить сортируемый массив произвольными числами. Следующая за ней группа кнопок и текстовой области позволяет изменить размер сортируемого массива. Нажатие любой из кнопок нижнего ряда приводит к изменению входных данных алгоритма, поэтому процесс визуализации его работы начинается сначала.

Пользователю на каждом визуализируемом шаге работы алгоритма предоставляется развернутый комментарий, а также графическое изображение массива в его ленточном и пирамидальном представлении. Комментарий отражает происходящее на данном шаге. При этом элементы массива, с которыми идет работа, выделяются определенным (в текущей конфигурации — красным) цветом и, при необходимости, наличием стрелок на соответствующем ребре пирамиды.

6. Автоматически сгенерированный исходный код, реализующий логику визуализатора

Рассмотрим исходный код, сгенерированный *Vizi* (Приложение 2). Этот код содержит класс `HeapSort`, содержащий, в свою очередь, классы `Data` — модель данных, `Main` — реализация пары автоматов для движения по алгоритму `Main` в «прямом» и «обратном» направлениях (эти автоматы имеют по девять состояний), `Sift` — реализация пары автоматов

для движения по алгоритму `Sift` в «прямом» и «обратном» направлениях (эти автоматы имеют по четырнадцать состояний). Таким образом, логика визуализатора реализуется четырьмя автоматами, имеющими в сумме 46 состояний.

Каждый автомат реализуется двумя операторами `switch`. В автомате для прямого прохода первый оператор обеспечивает переход в следующее состояние, а второй — действия в текущем состоянии. В автомате для обратного прохода первый оператор обеспечивает обращение действий в текущем состоянии, а второй — переход в предыдущее состояние.

Основными методами класса Main являются методы `stepForward` и `stepBackward`. Первый из них обеспечивает переход в следующее состояние автомата с выполнением соответствующих действий, а второй — переход в обратном направлении. При этом каждый из этих методов может совершить несколько шагов за один вызов, в зависимости от того, считается ли данное состояние «интересным». «Интересность» состояния определяется методом `isInteresting`, использующим значение атрибута `level` тега `step` (разд. 4). Вывод комментариев к состояниям осуществляется методом `getComment`, а отрисовка состояний — методом `drawState`. Привязка к состояниям комментариев и действий по отрисовке также осуществляется автоматически с помощью двух дополнительных операторов `switch`.

7. Исходный код пользовательского интерфейса визуализатора

В приложении 3 приведен исходный код пользовательского интерфейса визуализатора, написанный вручную на языке Java с использованием библиотеки *Vizi*.

Пользовательский интерфейс визуализатора реализован финальным классом `HeapSortVisualizer`, являющимся наследником класса `Base` библиотеки *Vizi*.

Данный класс имеет следующие поля:

- `auto` — экземпляр автомата,
- `data` — экземпляр модели данных автомата,
- `elements` — спин-панель для количества элементов сортируемого массива,
- `maxValue` — максимальное корректное значение элемента массива,
- `maxValueString` — строковое представление `maxValue`,
- `styleSet` — набор стилей,
- `radius` — радиус окружностей, изображающих элементы пирамиды,
- `m_activeCell1` — номер первого активного на данном шаге элемента,
- `m_activeCell2` — номер второго активного на данном шаге элемента,

- `m_activeStyle` — номер стиля, которым следует отрисовывать активные элементы.

Конструктор данного класса загружает данные конфигурации визуализатора, выделяет память для структур данных пользовательского интерфейса, создает пользовательский интерфейс.

Переопределенный метод `createControlsPane` добавляет кнопку «Случайно» к стандартной панели управления элементами массива библиотеки *Vizi*.

Переопределенный метод `paintClient` производит обновление клиентской области аплета посредством отрисовки ленточного и пирамидального представлений массива методами библиотеки Java awt. Данный метод использует ряд вспомогательных математических функций.

8. Описание административного интерфейса (конфигурации) визуализатора

Конфигурирование визуализатора производится посредством изменения файла `HeapSort-Configuration.xml` (приложение 1). Все параметры конфигурации указываются в секции `configuration`. Используя XML-тег `property`, можно изменить такие параметры конфигурации, как высоту области для комментария (параметр `comment-height`), максимальное значение элемента массива, присваиваемое генератором случайных чисел (параметр `max-value`).

Так как входными данными для визуализируемого алгоритма является массив, необходимо предоставить пользователю возможность изменять его длину, а также устанавливать значения его элементов в произвольные величины. Для изменения размера массива создается спин-панель (совокупность текстовой области, отображающей значение определенной величины, и кнопок для ее изменения) `elements`, в XML-описании которой указываются все названия кнопок, подсказки (hints), минимальное и максимальное целое число, которое панель позволяет ввести. Для создания спин-панели используется XML-тег `spin-panel`, формат которого интуитивно ясен. Для установки значений элементов массива в произвольные величины создается кнопка «Случайно». Кнопка создается при помощи XML-тега `button` с параметром `button-random`.

Для предоставления возможности изменить цветовую гамму визуализатора без перекомпиляции аплета в *Vizi* существует механизм таблиц стилей (stylesets). В данном аплете используется одна таблица стилей — `array`. Первый стиль таблицы задает цвет контуров и шрифт, второй — не используется (существует для совместимости), третий задает цвет заливки выделенных элементов массива и пирамиды. Таблицы стилей создаются при помощи XML-тега `styleset`, их элементы при помощи XML-тега — `style`.

Заключение

Хотелось бы отметить ряд преимуществ использования технологии *Vizi* перед реализациями, написанными вручную «с нуля», как это изложено в работе [5]:

- построение по XML-описанию «прямого» алгоритма визуализатора не только его прямого прохода, но и обратного;
- логика алгоритма реализована с помощью четырех автоматов, по два («прямой» и «обратный») для каждой процедуры. Общее число состояний у автоматов равно 46. Каждый автомат реализуется двумя операторами `switch`;
- описание алгоритма при помощи языка XML позволяет автоматически вводить комментарии в код (отличающиеся от комментариев, вводимых для визуализации), а особенности валидации XML-кода технологией *Vizi* делают написание этих комментариев неизбежным, что значительно повышает читабельность кода, а следовательно упрощает его дальнейшее сопровождение;
- привязка к состояниям комментариев и действий по отрисовке также осуществляется автоматически с помощью дополнительных операторов `switch`;
- использование для построения визуализаторов единой технологии стандартизирует процесс разработки и позволяет во многих случаях избежать дублирования кода, уменьшая вероятность появления ошибок;
- стандартный эргономичный интерфейс визуализаторов имеет большое значение и обуславливает удобство изучения коллекций визуализаторов алгоритмов;
- компактность XML-описания — 9 страниц этого описания против 25 страниц соответствующего (автоматически сгенерированного) исходного кода.

Обратим внимание, что при построении визуализатора на основе автоматного подхода [5] используется вручную построенный автомат, содержащий только три состояния. При этом визуализация обратного прохода алгоритма реализуется без использования автоматов. Таким образом, технология *Vizi* приводит к построению большего числа более сложных автоматов, однако в виду того, что они строятся автоматически и решают одинаковым образом задачи построения прямого и обратного проходов, это не является недостатком указанной технологии.

Список литературы

[1] Казаков М.А., Корнеев Г.А., Шалыто А.А. Разработка логики визуализаторов алгоритмов на основе конечных автоматов //Телекоммуникации и информатизация образования. 2003, № 6, с. 27–58.

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

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

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

[5] Удов Г.Г., Шалыто А.А. Классическая и автоматная реализации алгоритма пирамидальной сортировки набора чисел. СПб.: СПбГУ ИТМО, 2003. http://is.ifmo.ru, раздел «Визуализаторы».

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

Приложение 1. XML-описание визуализатора

Файл HeapSort.xml (основные параметры визуализатора)

```
<?xml version="1.0" encoding="WINDOWS-1251"?>

<!--
    "HeapSort" visualizer description
    Version: $Id: HeapSort.xml,v 1.0 2003/11/18 $
-->

<!DOCTYPE visualizer PUBLIC
    "-//IFMO Vizi//Visualizer description"
    "http://ips.ifmo.ru/vizi/dtd/visualizer.dtd"
[
    <!ENTITY algorithm SYSTEM "HeapSort-Algorithm.xml">
    <!ENTITY configuration SYSTEM "HeapSort-Configuration.xml">
]>

<visualizer
    id="HeapSort"
    package="ru.ifmo.vizi.heapsort"
    main-class="HeapSortVisualizer"

    preferred-width="400"
    preferred-height="350"

    name-ru= "Пирамидальная сортировка набора чисел"
    name-en="HeapSort"

    author-ru="Георгий Удов"
    author-en="George Udov"
    author-email="udov@rain.ifmo.ru"

    supervisor-ru="Анатолий Шалыто"
    supervisor-en="Anatoly Shalyto"
    supervisor-email="shalyto@mail.ifmo.ru"

    copyright-ru="Copyright \u00A9 Кафедра КТ, СПб ГИТМО (ТУ), 2003"
    copyright-en="Copyright \u00A9 Computer Technologies Department, SPb IFMO
        , 2003"
>
    &algorithm;
    &configuration;
</visualizer>
```

Файл HeapSort-Algorithm.xml (xml-представление алгоритма)

```
<?xml version="1.0" encoding="WINDOWS-1251"?>

<!--
    "HeapSort" algoritm description (using auto-reverse feature)
    Version: $Id: HeapSort-Algorithm.xml,v 1.0 2003/11/18 $
-->

<algorithm>
    <variable
        description="Массив для поиска"
```

```
    name="a"
    type="int[]"
    value="new int[] {}"
/>
<variable
    description="Номер просеиваемого элемента"
    name="l"
    type="int"
    value="0"
/>
<variable
    description="Индекс элемента массива, являющегося последним элементом
       пирамиды"
    name="r"
    type="int"
    value="0"
/>
<variable
    description="Временная переменная для осуществления операции swap"
    name="tmp"
    type="int"
    value="0"
/>
<variable
    description="Экземпляр апплета"
    name="visualizer"
    type="HeapSortVisualizer"
    value="null"
/>

<toString>
    StringBuffer s = new StringBuffer();

    return s.toString();
</toString>

<auto id="Sift" description="Просеивает элемент номер l в массиве. r -
   последний элемент.">
    <variable
        description="Номер родительской вершины"
        name="i"
        type="int"
    />
    <variable
        description="Номер дочерней вершины"
        name="j"
        type="int"
    />
    <variable
        description="Значение в родительской(просеиваемой) вершине"
        name="x"
        type="int"
    />
    <step level="-1"
          description="Инициализируем переменные автомата Sift">
        <action>
          @i @= @l;
          @j @= 2*@l + 1;
          @x @= @a[@l];
```

```
    </action>
</step>

<if description="Определяем максимального ребенка"
    test="(@j &lt; @r) &amp;&amp; (@a[@j+1] &gt; @a[@j])"
    level="-1">
    <then>
      <step description="Максимальный ребенок - следующий"
            level="-1">
        <action>
          @j @= @j+1;
        </action>
      </step>
    </then>
</if>

<while description="Пока не дойдем до корня пирамиды"
       test="(@j &lt;= @r) &amp;&amp; (@a[@j] &gt; @x)"
       level="-1">
    <step description="Продвигаемся к корню"
          comment-ru="Сравниваем элемент {0} с наибольшим из детей. Так
              как наибольший ребенок {1} больше него, меняем их
             местами."
          comment-en="Comparing item {0} with the greatest child. Since
              the greatest child {1} is greater then it, swapping them
             ."
          comment-args="new Integer(@a[@i]), new Integer(@a[@j])"
          level="1"
          id="id0">
      <draw>
          @visualizer.updateArray(@i, @j, 2);
      </draw>
      <action>
      </action>
    </step>

    <step description="Продвигаемся к корню"
          comment-ru="Сравниваем элемент {0} с наибольшим из детей. Так
              как наибольший ребенок {1} больше него, меняем их
             местами."
          comment-en="Comparing item {0} with the greatest child. Since
              the greatest child {1} is greater then it, swapping them
             ."
          comment-args="new Integer(@a[@j]), new Integer(@a[@i])"
          level="1"
          id="id1">
      <draw>
          @visualizer.updateArray(@i, @j, 2);
      </draw>
      <action>
        @tmp   @= @a[@i];
        @a[@i] @= @a[@j];
        @a[@j] @= @tmp;
      </action>
    </step>

    <step description="Продвигаемся к корню" comment-ru="Продвигаемся к
        корню" level="-1">
      <action>
```

```
            @i @= @j;
            @j @= 2*@j + 1;
          </action>
        </step>

        <if description="Определяем максимального ребенка"
            test="(@j &lt; @r) &amp;&amp; (@a[@j+1] &gt; @a[@j])"
            level="-1">
            <then>
              <step description="Максимальный ребенок - следующий" level =
                  "-1" id="id2">
                <action>
                  @j @= @j+1;
                </action>
              </step>
            </then>
        </if>
    </while>

    <if description="Визуализируем отсутствие необходимости просеивать
       дальше"
        level="-1"
        test="(@j &lt;= @r)">
        <then>
            <step description="Визуализируем отсутствие необходимости
               просеивать дальше"
                comment-ru="Элемент {0} больше своего максимального
                   ребенка {1}, следовательно просеивание завершено."
                comment-en="The item {0} is greater then its greatest
                   child {1}, so sifting is finished."
                comment-args="new Integer(@a[@i]), new Integer(@a[@j])"
                level="1"
                id="id3">
              <draw>
                  @visualizer.updateArray(@i, @j, 2);
              </draw>
              <action>
              </action>
            </step>
        </then>
    </if>

</auto>

<auto id="Main" description="Сортирует массив методом heapSort">
    <variable
        description="Переменная для визуализации - номер стадии"
        name="numStadia"
        type="int"
    />
    <start
        comment-ru="На экране изображен массив и его представление в виде
           пирамиды"
        comment-en="There is an array on the display and its heap
           representation"
    >
        <draw>
            @visualizer.updateArray(-1, -1, 0);
        </draw>
```

```
</start>

<step description="Инициализация"
      level="1"
      comment-ru='Первая стадия работы алгоритма - просеиваем каждый
         элемент пирамиды, имеющий "детей".'
      comment-en="First phase of the algorithm - sifting all heap's
         items that have children"
      id="id4">
    <draw>
        @visualizer.updateArray(-1, -1, 0);
    </draw>
    <action>
      @l @= @a.length / 2;
      @r @= @a.length - 1;
      @numStadia = 1;
    </action>
</step>

<while description="Просев всех элементов дерева"
       test="@l &gt; 0"
       level="-1">
    <step description="Переходим к просеиванию предыдущего элемента"
       level="-1">
      <action>
        @l @= @l - 1;
      </action>
    </step>
    <call-auto id="Sift"/>
</while>

<step description="Переходим ко второй стадии работы алгоритма"
      comment-ru="Переходим ко второй стадии работы алгоритма - меняем
         местами первый и последний элементы пирамиды, отрываем
         последний элемент от пирамиды, а затем восстанавливаем ее
         иерархическую упорядоченность, просеивая корневой элемент."
      comment-en="Switching to the second phase of the algorithm -
         swapping the first and the last heap's items, disconnecting
         the last item from the heap, and then restoring hierarchical
         ordering of the heap by sifting the root item."
      level="1"
      id="id5">
    <draw>
        @visualizer.updateArray(0, -1, 2);
    </draw>
    <action>
        @numStadia = 2;
    </action>
</step>

<while description="Пока дерево имеет положительное число элементов"
       test="@r &gt; 0" level="-1">

    <step description="Меняем местами первый и последний элементы
       дерева"
          comment-ru="Наибольший в пирамиде элемент сейчас находится на
              первом месте, а должен быть на последнем. Меняем местами
              элементы {0} и {1} и отрываем элемент {0} от пирамиды."
```

```
                comment-en="The maximum heap item is first now, but it must
                    be last. Swapping items {0} and {1} and disconnecting
                    item {0} from the heap."
                comment-args="new Integer(@a[0]), new Integer(@a[@r])"
                id="id6">
              <draw>
                  @visualizer.updateArray(0, @r, 2);
              </draw>
              <action>
              </action>
            </step>

            <step description="Меняем местами первый и последний элементы
                дерева"
                comment-ru="Наибольший в пирамиде элемент сейчас находится на
                    первом месте, а должен быть на последнем. Меняем местами
                    элементы {0} и {1} и отрываем элемент {0} от пирамиды."
                comment-en="The maximum heap item is first now, but it must
                    be last. Swapping items {0} and {1} and disconnecting
                    item {0} from the heap."
                comment-args="new Integer(@a[@r+1]), new Integer(@a[0])"
                id="id7">
              <draw>
                  @visualizer.updateArray(0, @r+1, 2);
              </draw>
              <action>
                  @tmp   @= @a[0];
                  @a[0]  @= @a[@r];
                  @a[@r] @= @tmp;
                  @r     @= @r - 1;
              </action>
            </step>
            <call-auto id="Sift"/>
        </while>

        <finish
            comment-ru="Сортировка завершена"
            comment-en="Sorting finished"
        >
            <draw>
                @visualizer.updateArray(-1, -1, 0);
            </draw>
        </finish>
    </auto>

</algorithm>
```

Файл HeapSort-Configuration.xml (конфигурация визуализатора)

```
<?xml version="1.0" encoding="WINDOWS-1251"?>

<!--
    "HeapSort" visualizer configuration.
    Version: $Id: HeapSort-Configuration.xml,v 1.0 2003/11/18 11:42:21 geo Exp
        $
-->

<configuration>
    <property
```

```
    description = "Comment pane height"
    param       = "comment-height"
    value       = "80"
/>
<spin-panel
    param       = "elements"
    caption-ru  = "Элементов: {0,number,####}"
    caption-en  = "Elements: {0,number,####}"
    hint-ru     = "Количество элементов в массиве"
    hint-en     = "Number of elements in the array"
    value       = "15"
    min-value   = "5"
    max-value   = "20"
    step        = "1"
>
    <button
        param       = "button-less"
        caption-ru  = "&lt;&lt;"
        caption-en  = "&lt;&lt;"
        hint-ru     = "Уменьшить количество элементов"
        hint-en     = "Decrease number of elements"
    />
    <button
        param       = "button-more"
        caption-ru  = ">>"
        caption-en  = ">>"
        hint-ru     = "Увеличить количество элементов"
        hint-en     = "Increase number of elements"
    />
</spin-panel>
<button
    param       = "button-random"
    caption-ru  = "Случайно"
    caption-en  = "Random"
    hint-ru     = "Заполнить массив случайными значениями"
    hint-en     = "Fill the array with random values"
/>
<message
    param       = "max-message"
    message-ru  = "max = {0}"
    message-en  = "max = {0}"
/>
<styleset
    description = "Array style set"
    param       = "array"
>
    <style
        text-color      = "000000"
        text-align      = "0.5"
        border-color    = "000000"
        border-status   = "true"
        fill-color      = "8080ff"
        fill-status     = "true"
        aspect-status   = "false"
        padding         = "0.2"
    >
        <font
            face        = "Serif"
            size        = "12"
```

```
                style       = "plain"
            />
        </style>
        <style
            fill-color      = "80ff80"
        />
        <style
            fill-color      = "ff8080"
        />
    </styleset>
    <style
        param           = "max-style"
        border-status   = "false"
        fill-status     = "false"
    >
        <font size="20"/>
    </style>
    <property
        param   = "max-value"
        value   = "99"
    />
    <property
        param   = "max-value-string"
        value   = "88"
    />
</configuration>
```

Приложение 2. Сгенерированный исходный код автомата

```
package ru.ifmo.vizi.heapsort;

import ru.ifmo.vizi.base.auto.*;
import java.util.Locale;

public final class HeapSort extends BaseAutoReverseAutomata {
    /**
      * Модель данных.
      */
    public final Data d = new Data();

    /**
      * Конструктор для языка
      */
    public HeapSort(Locale locale) {
        super("ru.ifmo.vizi.heapsort.Comments", locale);
        init(new Main(), d);
    }

    /**
      * Данные.
      */
    public final class Data {
        /**
          * Массив для поиска.
          */
        public int[] a = new int[] {};

        /**
          * Номер просеиваемого элемента.
          */
        public int l = 0;

        /**
          * Индекс элемента массива, являющегося последним элементом пирамиды.
          */
        public int r = 0;

        /**
          * Временная переменная для осуществления операции swap.
          */
        public int tmp = 0;

        /**
          * Экземпляр апплета.
          */
        public HeapSortVisualizer visualizer = null;

        /**
          * Номер родительской вершины (Процедура Sift).
          */
        public int Sift_i;

        /**
          * Номер дочерней вершины (Процедура Sift).
          */
```

```
    public int Sift_j;

    /**
      * Значение в родительской(просеиваемой) вершине (Процедура Sift).
      */
    public int Sift_x;

    /**
      * Переменная для визуализации - номер стадии (Процедура Main).
      */
    public int Main_numStadia;

    public String toString() {
        StringBuffer s = new StringBuffer();

        return s.toString();
    }
}

/**
  * Просеивает элемент номер l в массиве. r - последний элемент..
  */
private final class Sift extends BaseAutomata implements Automata {
    /**
      * Начальное состояние автомата.
      */
    private final int START_STATE = 0;

    /**
      * Конечное состояние автомата.
      */
    private final int END_STATE = 15;

    /**
      * Конструктор.
      */
    public Sift() {
        super(
            "Sift",
            0, // Номер начального состояния
            15, // Номер конечного состояния
            new String[]{
                "Начальное состояние",
                "Инициализируем переменные автомата Sift",
                "Определяем максимального ребенка",
                "Определяем максимального ребенка (окончание)",
                "Максимальный ребенок - следующий",
                "Пока не дойдем до корня пирамиды",
                "Продвигаемся к корню",
                "Продвигаемся к корню",
                "Продвигаемся к корню",
                "Определяем максимального ребенка",
                "Определяем максимального ребенка (окончание)",
                "Максимальный ребенок - следующий",
                "Визуализируем отсутствие необходимости просеивать дальше",
                "Визуализируем отсутствие необходимости просеивать дальше (
                   окончание)",
                "Визуализируем отсутствие необходимости просеивать дальше",
                "Конечное состояние"
```

```
            }, new int[]{
                Integer.MAX_VALUE, // Начальное состояние,
                -1, // Инициализируем переменные автомата Sift
                -1, // Определяем максимального ребенка
                -1, // Определяем максимального ребенка (окончание)
                -1, // Максимальный ребенок - следующий
                -1, // Пока не дойдем до корня пирамиды
                1, // Продвигаемся к корню
                1, // Продвигаемся к корню
                -1, // Продвигаемся к корню
                -1, // Определяем максимального ребенка
                -1, // Определяем максимального ребенка (окончание)
                -1, // Максимальный ребенок - следующий
                -1, // Визуализируем отсутствие необходимости просеивать
                   дальше
                -1, // Визуализируем отсутствие необходимости просеивать
                   дальше (окончание)
                1, // Визуализируем отсутствие необходимости просеивать
                   дальше
                Integer.MAX_VALUE, // Конечное состояние
            }
        );
    }

    /**
      * Сделать один шаг автомата вперед.
      */
    protected void doStepForward(int level) {
        // Переход в следующее состояние
        switch (state) {
            case START_STATE: { // Начальное состояние
                state = 1; // Инициализируем переменные автомата Sift
                break;
            }
            case 1: { // Инициализируем переменные автомата Sift
                state = 2; // Определяем максимального ребенка
                break;
            }
            case 2: { // Определяем максимального ребенка
                if ((d.Sift_j < d.r) && (d.a[d.Sift_j+1] > d.a[d.Sift_j]))
                    {
                    state = 4; // Максимальный ребенок - следующий
                } else {
                    stack.pushBoolean(false);
                    state = 3; // Определяем максимального ребенка (
                       окончание)
                }
                break;
            }
            case 3: { // Определяем максимального ребенка (окончание)
                stack.pushBoolean(false);
                state = 5; // Пока не дойдем до корня пирамиды
                break;
            }
            case 4: { // Максимальный ребенок - следующий
                stack.pushBoolean(true);
                state = 3; // Определяем максимального ребенка (окончание)
                break;
            }
```

```
case 5: { // Пока не дойдем до корня пирамиды
    if ((d.Sift_j <= d.r) && (d.a[d.Sift_j] > d.Sift_x)) {
        state = 6; // Продвигаемся к корню
    } else {
        state = 12; // Визуализируем отсутствие необходимости
            просеивать дальше
    }
    break;
}
case 6: { // Продвигаемся к корню
    state = 7; // Продвигаемся к корню
    break;
}
case 7: { // Продвигаемся к корню
    state = 8; // Продвигаемся к корню
    break;
}
case 8: { // Продвигаемся к корню
    state = 9; // Определяем максимального ребенка
    break;
}
case 9: { // Определяем максимального ребенка
    if ((d.Sift_j < d.r) && (d.a[d.Sift_j+1] > d.a[d.Sift_j]))
        {
        state = 11; // Максимальный ребенок - следующий
    } else {
        stack.pushBoolean(false);
        state = 10; // Определяем максимального ребенка (
            окончание)
    }
    break;
}
case 10: { // Определяем максимального ребенка (окончание)
    stack.pushBoolean(true);
    state = 5; // Пока не дойдем до корня пирамиды
    break;
}
case 11: { // Максимальный ребенок - следующий
    stack.pushBoolean(true);
    state = 10; // Определяем максимального ребенка (окончание)
    break;
}
case 12: { // Визуализируем отсутствие необходимости просеивать
    дальше
    if ((d.Sift_j <= d.r)) {
        state = 14; // Визуализируем отсутствие необходимости
            просеивать дальше
    } else {
        stack.pushBoolean(false);
        state = 13; // Визуализируем отсутствие необходимости
            просеивать дальше (окончание)
    }
    break;
}
case 13: { // Визуализируем отсутствие необходимости просеивать
    дальше (окончание)
    state = END_STATE;
    break;
}
```

```
        case 14: { // Визуализируем отсутствие необходимости просеивать
            дальше
            stack.pushBoolean(true);
            state = 13; // Визуализируем отсутствие необходимости
               просеивать дальше (окончание)
            break;
        }
    }

    // Действие в текущем состоянии
    switch (state) {
        case 1: { // Инициализируем переменные автомата Sift
            startSection();
            storeField(d, "Sift_i");
            d.Sift_i = d.l;
            storeField(d, "Sift_j");
            d.Sift_j = 2*d.l + 1;
            storeField(d, "Sift_x");
            d.Sift_x = d.a[d.l];
            break;
        }
        case 2: { // Определяем максимального ребенка
            break;
        }
        case 3: { // Определяем максимального ребенка (окончание)
            break;
        }
        case 4: { // Максимальный ребенок - следующий
            startSection();
            storeField(d, "Sift_j");
            d.Sift_j = d.Sift_j+1;
            break;
        }
        case 5: { // Пока не дойдем до корня пирамиды
            break;
        }
        case 6: { // Продвигаемся к корню
            startSection();
            break;
        }
        case 7: { // Продвигаемся к корню
            startSection();
            storeField(d, "tmp");
            d.tmp   = d.a[d.Sift_i];
            storeArray(d.a, d.Sift_i);
            d.a[d.Sift_i] = d.a[d.Sift_j];
            storeArray(d.a, d.Sift_j);
            d.a[d.Sift_j] = d.tmp;
            break;
        }
        case 8: { // Продвигаемся к корню
            startSection();
            storeField(d, "Sift_i");
            d.Sift_i = d.Sift_j;
            storeField(d, "Sift_j");
            d.Sift_j = 2*d.Sift_j + 1;
            break;
        }
        case 9: { // Определяем максимального ребенка
```

```
                break;
            }
            case 10: { // Определяем максимального ребенка (окончание)
                break;
            }
            case 11: { // Максимальный ребенок - следующий
                startSection();
                storeField(d, "Sift_j");
                d.Sift_j = d.Sift_j+1;
                break;
            }
            case 12: { // Визуализируем отсутствие необходимости просеивать
                дальше
                break;
            }
            case 13: { // Визуализируем отсутствие необходимости просеивать
                дальше (окончание)
                break;
            }
            case 14: { // Визуализируем отсутствие необходимости просеивать
                дальше
                startSection();
                break;
            }
        }
    }

    /**
      * Сделать один шаг автомата назад.
      */
    protected void doStepBackward(int level) {
        // Обращение действия в текущем состоянии
        switch (state) {
            case 1: { // Инициализируем переменные автомата Sift
                restoreSection();
                break;
            }
            case 2: { // Определяем максимального ребенка
                break;
            }
            case 3: { // Определяем максимального ребенка (окончание)
                break;
            }
            case 4: { // Максимальный ребенок - следующий
                restoreSection();
                break;
            }
            case 5: { // Пока не дойдем до корня пирамиды
                break;
            }
            case 6: { // Продвигаемся к корню
                restoreSection();
                break;
            }
            case 7: { // Продвигаемся к корню
                restoreSection();
                break;
            }
            case 8: { // Продвигаемся к корню
```

```
            restoreSection();
            break;
        }
        case 9: { // Определяем максимального ребенка
            break;
        }
        case 10: { // Определяем максимального ребенка (окончание)
            break;
        }
        case 11: { // Максимальный ребенок - следующий
            restoreSection();
            break;
        }
        case 12: { // Визуализируем отсутствие необходимости просеивать
            дальше
            break;
        }
        case 13: { // Визуализируем отсутствие необходимости просеивать
            дальше (окончание)
            break;
        }
        case 14: { // Визуализируем отсутствие необходимости просеивать
            дальше
            restoreSection();
            break;
        }
    }

    // Переход в предыдущее состояние
    switch (state) {
        case 1: { // Инициализируем переменные автомата Sift
            state = START_STATE;
            break;
        }
        case 2: { // Определяем максимального ребенка
            state = 1; // Инициализируем переменные автомата Sift
            break;
        }
        case 3: { // Определяем максимального ребенка (окончание)
            if (stack.popBoolean()) {
                state = 4; // Максимальный ребенок - следующий
            } else {
                state = 2; // Определяем максимального ребенка
            }
            break;
        }
        case 4: { // Максимальный ребенок - следующий
            state = 2; // Определяем максимального ребенка
            break;
        }
        case 5: { // Пока не дойдем до корня пирамиды
            if (stack.popBoolean()) {
                state = 10; // Определяем максимального ребенка (
                    окончание)
            } else {
                state = 3; // Определяем максимального ребенка (
                    окончание)
            }
            break;
```

```
            }
            case 6: { // Продвигаемся к корню
                state = 5; // Пока не дойдем до корня пирамиды
                break;
            }
            case 7: { // Продвигаемся к корню
                state = 6; // Продвигаемся к корню
                break;
            }
            case 8: { // Продвигаемся к корню
                state = 7; // Продвигаемся к корню
                break;
            }
            case 9: { // Определяем максимального ребенка
                state = 8; // Продвигаемся к корню
                break;
            }
            case 10: { // Определяем максимального ребенка (окончание)
                if (stack.popBoolean()) {
                    state = 11; // Максимальный ребенок - следующий
                } else {
                    state = 9; // Определяем максимального ребенка
                }
                break;
            }
            case 11: { // Максимальный ребенок - следующий
                state = 9; // Определяем максимального ребенка
                break;
            }
            case 12: { // Визуализируем отсутствие необходимости просеивать
                дальше
                state = 5; // Пока не дойдем до корня пирамиды
                break;
            }
            case 13: { // Визуализируем отсутствие необходимости просеивать
                дальше (окончание)
                if (stack.popBoolean()) {
                    state = 14; // Визуализируем отсутствие необходимости
                       просеивать дальше
                } else {
                    state = 12; // Визуализируем отсутствие необходимости
                       просеивать дальше
                }
                break;
            }
            case 14: { // Визуализируем отсутствие необходимости просеивать
                дальше
                state = 12; // Визуализируем отсутствие необходимости
                   просеивать дальше
                break;
            }
            case END_STATE: { // Начальное состояние
                state = 13; // Визуализируем отсутствие необходимости
                   просеивать дальше (окончание)
                break;
            }
        }
    }
```

```
    /**
      * Комментарий к текущему состоянию
      */
    public String getComment() {
        String comment = "";
        Object[] args = null;
        // Выбор комментария
        switch (state) {
            case 6: { // Продвигаемся к корню
                comment = HeapSort.this.getComment("Sift.id0");
                args = new Object[]{new Integer(d.a[d.Sift_i]), new Integer
                    (d.a[d.Sift_j])};
                break;
            }
            case 7: { // Продвигаемся к корню
                comment = HeapSort.this.getComment("Sift.id1");
                args = new Object[]{new Integer(d.a[d.Sift_j]), new Integer
                    (d.a[d.Sift_i])};
                break;
            }
            case 8: { // Продвигаемся к корню
                comment = HeapSort.this.getComment("Sift.");
                break;
            }
            case 14: { // Визуализируем отсутствие необходимости просеивать
                дальше
                comment = HeapSort.this.getComment("Sift.id3");
                args = new Object[]{new Integer(d.a[d.Sift_i]), new Integer
                    (d.a[d.Sift_j])};
                break;
            }
        }

        return java.text.MessageFormat.format(comment, args);
    }

    /**
      * Выполняет действия по отрисовке состояния
      */
    public void drawState() {
        switch (state) {
            case 6: { // Продвигаемся к корню
                d.visualizer.updateArray(d.Sift_i, d.Sift_j, 2);
                break;
            }
            case 7: { // Продвигаемся к корню
                d.visualizer.updateArray(d.Sift_i, d.Sift_j, 2);
                break;
            }
            case 14: { // Визуализируем отсутствие необходимости просеивать
                дальше
                d.visualizer.updateArray(d.Sift_i, d.Sift_j, 2);
                break;
            }
        }
    }
}

/**
```

```
  * Сортирует массив методом heapSort.
  */
private final class Main extends BaseAutomata implements Automata {
    /**
      * Начальное состояние автомата.
      */
    private final int START_STATE = 0;

    /**
      * Конечное состояние автомата.
      */
    private final int END_STATE = 10;

    /**
      * Конструктор.
      */
    public Main() {
        super(
            "Main",
            0, // Номер начального состояния
            10, // Номер конечного состояния
            new String[]{
                "Начальное состояние",
                "Инициализация",
                "Просев всех элементов дерева",
                "Переходим к просеиванию предыдущего элемента",
                "Просеивает элемент номер l в массиве. r - последний
                   элемент. (автомат)",
                "Переходим ко второй стадии работы алгоритма",
                "Пока дерево имеет положительное число элементов",
                "Меняем местами первый и последний элементы дерева",
                "Меняем местами первый и последний элементы дерева",
                "Просеивает элемент номер l в массиве. r - последний
                   элемент. (автомат)",
                "Конечное состояние"
            }, new int[]{
                Integer.MAX_VALUE, // Начальное состояние,
                1, // Инициализация
                -1, // Просев всех элементов дерева
                -1, // Переходим к просеиванию предыдущего элемента
                CALL_AUTO_LEVEL, // Просеивает элемент номер l в массиве. r
                    - последний элемент. (автомат)
                1, // Переходим ко второй стадии работы алгоритма
                -1, // Пока дерево имеет положительное число элементов
                0, // Меняем местами первый и последний элементы дерева
                0, // Меняем местами первый и последний элементы дерева
                CALL_AUTO_LEVEL, // Просеивает элемент номер l в массиве. r
                    - последний элемент. (автомат)
                Integer.MAX_VALUE, // Конечное состояние
            }
        );
    }

    /**
      * Сделать один шаг автомата вперед.
      */
    protected void doStepForward(int level) {
        // Переход в следующее состояние
        switch (state) {
```

```
case START_STATE: { // Начальное состояние
    state = 1; // Инициализация
    break;
}
case 1: { // Инициализация
    stack.pushBoolean(false);
    state = 2; // Просев всех элементов дерева
    break;
}
case 2: { // Просев всех элементов дерева
    if (d.l > 0) {
        state = 3; // Переходим к просеиванию предыдущего
            элемента
    } else {
        state = 5; // Переходим ко второй стадии работы
            алгоритма
    }
    break;
}
case 3: { // Переходим к просеиванию предыдущего элемента
    state = 4; // Просеивает элемент номер l в массиве. r -
        последний элемент. (автомат)
    break;
}
case 4: { // Просеивает элемент номер l в массиве. r -
    последний элемент. (автомат)
    if (child.isAtEnd()) {
        child = null;
        stack.pushBoolean(true);
        state = 2; // Просев всех элементов дерева
    }
    break;
}
case 5: { // Переходим ко второй стадии работы алгоритма
    stack.pushBoolean(false);
    state = 6; // Пока дерево имеет положительное число
        элементов
    break;
}
case 6: { // Пока дерево имеет положительное число элементов
    if (d.r > 0) {
        state = 7; // Меняем местами первый и последний
            элементы дерева
    } else {
        state = END_STATE;
    }
    break;
}
case 7: { // Меняем местами первый и последний элементы дерева
    state = 8; // Меняем местами первый и последний элементы
        дерева
    break;
}
case 8: { // Меняем местами первый и последний элементы дерева
    state = 9; // Просеивает элемент номер l в массиве. r -
        последний элемент. (автомат)
    break;
}
```

```
        case 9: { // Просеивает элемент номер l в массиве. r -
            последний элемент. (автомат)
            if (child.isAtEnd()) {
                child = null;
                stack.pushBoolean(true);
                state = 6; // Пока дерево имеет положительное число
                    элементов
            }
            break;
        }
    }

    // Действие в текущем состоянии
    switch (state) {
        case 1: { // Инициализация
            startSection();
            storeField(d, "l");
            d.l = d.a.length / 2;
            storeField(d, "r");
            d.r = d.a.length - 1;
            d.Main_numStadia = 1;
            break;
        }
        case 2: { // Просев всех элементов дерева
            break;
        }
        case 3: { // Переходим к просеиванию предыдущего элемента
            startSection();
            storeField(d, "l");
            d.l = d.l - 1;
            break;
        }
        case 4: { // Просеивает элемент номер l в массиве. r -
            последний элемент. (автомат)
            if (child == null) {
                child = new Sift();
                child.toStart();
            }
            child.stepForward(level);
            step--;
            break;
        }
        case 5: { // Переходим ко второй стадии работы алгоритма
            startSection();
            d.Main_numStadia = 2;
            break;
        }
        case 6: { // Пока дерево имеет положительное число элементов
            break;
        }
        case 7: { // Меняем местами первый и последний элементы дерева
            startSection();
            break;
        }
        case 8: { // Меняем местами первый и последний элементы дерева
            startSection();
            storeField(d, "tmp");
            d.tmp   = d.a[0];
            storeArray(d.a, 0);
```

```
                d.a[0]   = d.a[d.r];
                storeArray(d.a, d.r);
                d.a[d.r] = d.tmp;
                storeField(d, "r");
                d.r      = d.r - 1;
                break;
            }
            case 9: { // Просеивает элемент номер l в массиве. r -
                последний элемент. (автомат)
                if (child == null) {
                    child = new Sift();
                    child.toStart();
                }
                child.stepForward(level);
                step--;
                break;
            }
        }
    }

    /**
      * Сделать один шаг автомата назад.
      */
    protected void doStepBackward(int level) {
        // Обращение действия в текущем состоянии
        switch (state) {
            case 1: { // Инициализация
                restoreSection();
                break;
            }
            case 2: { // Просев всех элементов дерева
                break;
            }
            case 3: { // Переходим к просеиванию предыдущего элемента
                restoreSection();
                break;
            }
            case 4: { // Просеивает элемент номер l в массиве. r -
                последний элемент. (автомат)
                if (child == null) {
                    child = new Sift();
                    child.toEnd();
                }
                child.stepBackward(level);
                step++;
                break;
            }
            case 5: { // Переходим ко второй стадии работы алгоритма
                restoreSection();
                break;
            }
            case 6: { // Пока дерево имеет положительное число элементов
                break;
            }
            case 7: { // Меняем местами первый и последний элементы дерева
                restoreSection();
                break;
            }
            case 8: { // Меняем местами первый и последний элементы дерева
```

```
            restoreSection();
            break;
        }
        case 9: { // Просеивает элемент номер l в массиве. r -
            последний элемент. (автомат)
            if (child == null) {
                child = new Sift();
                child.toEnd();
            }
            child.stepBackward(level);
            step++;
            break;
        }
    }

    // Переход в предыдущее состояние
    switch (state) {
        case 1: { // Инициализация
            state = START_STATE;
            break;
        }
        case 2: { // Просев всех элементов дерева
            if (stack.popBoolean()) {
                state = 4; // Просеивает элемент номер l в массиве. r
                    - последний элемент. (автомат)
            } else {
                state = 1; // Инициализация
            }
            break;
        }
        case 3: { // Переходим к просеиванию предыдущего элемента
            state = 2; // Просев всех элементов дерева
            break;
        }
        case 4: { // Просеивает элемент номер l в массиве. r -
            последний элемент. (автомат)
            if (child.isAtStart()) {
                child = null;
                state = 3; // Переходим к просеиванию предыдущего
                    элемента
            }
            break;
        }
        case 5: { // Переходим ко второй стадии работы алгоритма
            state = 2; // Просев всех элементов дерева
            break;
        }
        case 6: { // Пока дерево имеет положительное число элементов
            if (stack.popBoolean()) {
                state = 9; // Просеивает элемент номер l в массиве. r
                    - последний элемент. (автомат)
            } else {
                state = 5; // Переходим ко второй стадии работы
                    алгоритма
            }
            break;
        }
        case 7: { // Меняем местами первый и последний элементы дерева
```

```
                state = 6; // Пока дерево имеет положительное число
                    элементов
                break;
            }
            case 8: { // Меняем местами первый и последний элементы дерева
                state = 7; // Меняем местами первый и последний элементы
                    дерева
                break;
            }
            case 9: { // Просеивает элемент номер l в массиве. r -
                последний элемент. (автомат)
                if (child.isAtStart()) {
                    child = null;
                    state = 8; // Меняем местами первый и последний
                        элементы дерева
                }
                break;
            }
            case END_STATE: { // Начальное состояние
                state = 6; // Пока дерево имеет положительное число
                    элементов
                break;
            }
        }
    }

    /**
     * Комментарий к текущему состоянию
     */
    public String getComment() {
        String comment = "";
        Object[] args = null;
        // Выбор комментария
        switch (state) {
            case START_STATE: { // Начальное состояние
                comment = HeapSort.this.getComment("Main.START_STATE");
                break;
            }
            case 1: { // Инициализация
                comment = HeapSort.this.getComment("Main.id4");
                break;
            }
            case 4: { // Просеивает элемент номер l в массиве. r -
                последний элемент. (автомат)
                comment = child.getComment();
                args = new Object[0];
                break;
            }
            case 5: { // Переходим ко второй стадии работы алгоритма
                comment = HeapSort.this.getComment("Main.id5");
                break;
            }
            case 7: { // Меняем местами первый и последний элементы дерева
                comment = HeapSort.this.getComment("Main.id6");
                args = new Object[]{new Integer(d.a[0]), new Integer(d.a[d.
                    r])};
                break;
            }
            case 8: { // Меняем местами первый и последний элементы дерева
```

```
                comment = HeapSort.this.getComment("Main.id7");
                args = new Object[]{new Integer(d.a[d.r+1]), new Integer(d.
                    a[0])};
                break;
            }
            case 9: { // Просеивает элемент номер l в массиве. r -
                последний элемент. (автомат)
                comment = child.getComment();
                args = new Object[0];
                break;
            }
            case END_STATE: { // Конечное состояние
                comment = HeapSort.this.getComment("Main.END_STATE");
                break;
            }
        }

        return java.text.MessageFormat.format(comment, args);
    }

    /**
      * Выполняет действия по отрисовке состояния
      */
    public void drawState() {
        switch (state) {
            case START_STATE: { // Начальное состояние
                d.visualizer.updateArray(-1, -1, 0);
                break;
            }
            case 1: { // Инициализация
                d.visualizer.updateArray(-1, -1, 0);
                break;
            }
            case 4: { // Просеивает элемент номер l в массиве. r -
                последний элемент. (автомат)
                child.drawState();
                break;
            }
            case 5: { // Переходим ко второй стадии работы алгоритма
                d.visualizer.updateArray(0, -1, 2);
                break;
            }
            case 7: { // Меняем местами первый и последний элементы дерева
                d.visualizer.updateArray(0, d.r, 2);
                break;
            }
            case 8: { // Меняем местами первый и последний элементы дерева
                d.visualizer.updateArray(0, d.r+1, 2);
                break;
            }
            case 9: { // Просеивает элемент номер l в массиве. r -
                последний элемент. (автомат)
                child.drawState();
                break;
            }
            case END_STATE: { // Конечное состояние
                d.visualizer.updateArray(-1, -1, 0);
                break;
            }
```

```
                }
            }
        }
    }
```

Приложение 3. Исходный код интерфейса визуализатора

```
package ru.ifmo.vizi.heapsort;

import ru.ifmo.vizi.base.ui.*;
import ru.ifmo.vizi.base.*;
import ru.ifmo.vizi.base.widgets.Bresenham;
import ru.ifmo.vizi.base.widgets.ShapeStyle;

import java.awt.*;
import java.util.Stack;

/**
 * HeapSort applet.
 *
 * @author  George Udov
 * @version $Id: HeapSortVisualizer.java,v 1.0 2003/11/18 $
 */
public final class HeapSortVisualizer extends Base
{
  /**
   * Find maximum automata instance.
   */
  private final HeapSort auto;

  /**
   * Find maximum automata data.
   */
  private final HeapSort.Data data;

  /**
   * Number of elements in array.
   */
  private final SpinPanel elements;

  /**
   * Maximal array value.
   */
  private final int maxValue;

  /**
   * Maximal array value string.
   */
  private final String maxValueString;

  /**
   * Array shape style set.
   */
  private final ShapeStyle[] styleSet;

  /**
   * Radius of heap's items (interface parameter -
   * maybe should be moved to style)
   */
  private final int radius = 10;

  /**
   * Number of first active cell
```

```
 */
private int m_activeCell1 = -1;

/**
 * Number of second active cell
 */
private int m_activeCell2 = -1;

/**
 * Number of style for active cells
 */
private int m_activeStyle = 0;

/**
 * Creates a new Heap Sort visualizer.
 *
 * @param parameters visualizer parameters.
 */
public HeapSortVisualizer(VisualizerParameters parameters)
{
  super(parameters);
  auto = new HeapSort(locale);
  data = auto.d;
  data.visualizer = this;
  styleSet = ShapeStyle.loadStyleSet(config, "array");
  elements = new SpinPanel(config, "elements")
  {
      protected void click(double value)
      {
          setArraySize(getIntValue());
      }
  };

  maxValue = config.getInteger("max-value");
  maxValueString = config.getParameter("max-value-string",
  Integer.toString(maxValue));
  setArraySize(elements.getIntValue());

  createInterface(auto);
}

/**
 * This method creates panel with visualizer controls.
 *
 * @return controls pane.
 */
public Component createControlsPane()
{
  Panel panel = new Panel(new BorderLayout());

  panel.add(new AutoControlsPane(config, auto, forefather, false),
    BorderLayout.CENTER);

  Panel bottomPanel = new Panel();
  bottomPanel.add(new HintedButton(config, "button-random")
  {
    protected void click()
    {
      randomize();
```

```
    }
  });
  bottomPanel.add(elements);
  panel.add(bottomPanel, BorderLayout.SOUTH);
  return panel;
}

/**
 * Sets new array size.
 *
 * @param size new array size.
 */
private void setArraySize(int size)
{
  auto.d.a = new int[size];
  data.r = data.a.length-1;
  clientPane.doLayout();
  randomize();
}

/**
 * Randomizes array values.
 */
private void randomize()
{
  auto.toStart();
  for(int i = 0; i < data.a.length; i++)
  {
    data.a[i] = (int) (Math.random() * maxValue) + 1;
  }
  updateArray(-1, -1, 0);
}

/**
 * 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);
}

/**
 * Returns whether x is end of layer or not (maybe temporary)
 */
private boolean isEnd(int x)
{
  x += 1;
  while(x % 2 == 0)
    x /= 2;
```

```
    if(x == 1)
      return true;
    return false;
  }

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

  /**
   * Возвращает, является ли элемент с индексом a
   * в данном состоянии выделенным
   */
  private boolean isSelected(int a)
  {
    return (a == m_activeCell1) || (a == m_activeCell2);
  }

  /**
   * This method must paint current state of the algorithm.
   *
   * @param g graphics context to use for painting.
   * @param inputWidth graphics width.
   * @param inputHeight graphics height.
   */
  protected void paintClient(Graphics g, int inputWidth, int inputHeight)
  {
    int x0, y0, width, height;
    int screenWidth  = inputWidth;
    int screenHeight = inputHeight;
    int rSin, rCos;
    double tmp;
    String str;
    int maxElements = 20;
    int array[] = new int [maxElements];
    int lastEdgeNumber;
    int number = data.a.length;

    array = data.a;
    lastEdgeNumber = data.r;

    // Цвета
    Color foregroundColor = styleSet[0].getBorderColor();
    Color activeColor     = styleSet[2].getFillColor();

    // Шрифт
    Font font = styleSet[0].getTextFont();

    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;
// Высота дерева в элементах
height = (int)Math.ceil(log2(number + 1));
// 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) / (height-1);
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);
    Bresenham.fillEllipse(g, x0-radius, y0-radius, 2*radius, 2*radius);
  }
  g.setColor(foregroundColor);
  Bresenham.drawEllipse(g, x0-radius, y0-radius, 2*radius, 2*radius);

  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;
}

/**
 * Updates array view.
 *
 * @param activeCell1 current active cell.
 * @param activeCell2 current active cell.
```

```
     * @param activeStyle style of active cell.
     */
    public void updateArray(int activeCell1,
      int activeCell2, int activeStyle)
    {
      m_activeCell1 = activeCell1;
      m_activeCell2 = activeCell2;
      m_activeStyle = activeStyle;

      update(true);
    }
}
```