Teoría de la Información y Codificación

Práctica 3: Huffman como algoritmo de compresión

José A. Montenegro Montes

26 de septiembre de 2014

1. Enunciado

En esta práctica utilizaremos el algoritmo de Huffman para comprimir, basándonos en la codificación por símbolos.

Esta práctica será de utilidad para comparar posteriormente los algoritmos de compresión en bloques.

La práctica está basada en el código del libro Algorithms 4 Edición(Autores Robert Sedgewick y Kevin Wayne), que puede ser consultado en biblioteca.

2. Conclusiones

Además de realizar la comparativa con los códigos en bloques, la práctica ofrece un ejemplo de lectura y escritura binaria y cómo es posible utilizar la librería diffutils.

El algoritmo es aplicado a los tres libros disponibles en el campus virtual: **Alice.txt**, **MobyDick.txt** y **Quixote.txt**. La práctica genera el archivo comprimido con el sufijo C.txt y el archivo descomprimido con el sufijo D.txt. Los sufijos puede ser modificados en el código. Por ejemplo, con la ejecución sobre el libro Alice.txt obtendremos Alice.txtC.txt como archivo comprimido y Alice.txtD.txt como archivo descomprimido.

Además de utilizar los libros anteriormente citados, sería interesante aplicar la práctica a un **documento aleatorio** y comparar su salida posteriormente con la codificación en bloques.

La salida del programa será la siguiente por defecto la siguiente:

```
********************          Alice.txt          **********************
Size regular: 152146 Size compress: 87816 Ratio: 0.5771825
Fichero Descomprimido correctamente!!! Size regular: 152146 Size decompress: 152146
Tiempo Comprimir: 65872000 nanosegundos. Tiempo Descomprimir: 56971000 nanosegundos.
************************************************************
```

donde nos ofrecen información del ratio de compresión, el tiempo utilizado en comprimir y descomprimir. Además verificamos que el archivo descomprimido es igual al original.

Si ejecutamos la aplicación con la opción de estadísticas habilitada, nos mostrará la codificación utilizada, frecuencias y probabilidades de los caracteres del texto y finalmente la entropía y la longitud media obtenida.

```
********************          Alice.txt          **********************
Size regular: 152146 Size compress: 87816 Ratio: 0.5771825
Fichero Descomprimido correctamente!!! Size regular: 152146 Size decompress: 152146
Tiempo Comprimir: 65872000 nanosegundos. Tiempo Descomprimir: 56971000 nanosegundos.
*************************************************************


Caracter  (Frecuencia - Probabilidad)          Codificacion
-------------------------------------------------------
10          (3601 - 0.023668056)   01000
13          (3601 - 0.023668056)   01001
26          (1 - 6.572634E-6)   01011100110010010
32          (28917 - 0.19006087)   00
33          (450 - 0.0029576854)   111011011
34          (113 - 7.4270763E-4)   11101101010
39          (1763 - 0.011587554)   1110111
40          (56 - 3.680675E-4)   111011001111
41          (55 - 3.6149487E-4)   111011001110
42          (60 - 3.9435804E-4)   01011100100
44          (2418 - 0.015892629)   100111
45          (669 - 0.0043970924)   11000010
46          (978 - 0.0064280364)   0101011
48          (1 - 6.572634E-6)   01011100110011000
51          (1 - 6.572634E-6)   01011100110011001
58          (233 - 0.0015314238)   010101000
59          (194 - 0.0012750911)   1110110000
63          (202 - 0.0013276722)   1110110010
65          (638 - 0.0041933404)   10011011
66          (91 - 5.9810973E-4)   11000011001
67          (145 - 9.5303194E-4)   1001101001
68          (192 - 0.0012619458)   1100001111
69          (188 - 0.0012356553)   1100001110
70          (74 - 4.8637492E-4)   10011010100
71          (83 - 5.4552866E-4)   10011010111
72          (284 - 0.0018666281)   010111011
73          (733 - 0.004817741)   11000110
74          (8 - 5.2581072E-5)   0101110011000
75          (82 - 5.38956E-4)   10011010110
76          (98 - 6.441182E-4)   11101100110
77          (200 - 0.0013145269)   1110110001
78          (120 - 7.887161E-4)   0101110000
79          (176 - 0.0011567837)   1100001101
80          (65 - 4.272212E-4)   01011100101
81          (84 - 5.521013E-4)   11000011000
82          (140 - 9.2016876E-4)   0101110101
83          (218 - 0.0014328342)   1110110100
84          (472 - 0.0031022832)   01010101
85          (66 - 4.3379384E-4)   01011100111
86          (42 - 2.7605065E-4)   010111001101
```

```
87          (237 - 0.0015577143)   010101001
88          (4 - 2.6290536E-5)   010111001100101
89          (114 - 7.4928027E-4)   11101101011
90          (1 - 6.572634E-6)   01011100110010011
91          (2 - 1.3145268E-5)   0101110011001000
93          (2 - 1.3145268E-5)   0101110011001101
95          (4 - 2.6290536E-5)   010111001100111
96          (1109 - 0.007289051)   1001100
97          (8153 - 0.053586688)   0111
98          (1384 - 0.009096526)   1100000
99          (2254 - 0.014814718)   100001
100          (4739 - 0.031147713)   10010
101          (13386 - 0.08798128)   1101
102          (1927 - 0.012665466)   010110
103          (2448 - 0.016089808)   101001
104          (7090 - 0.046599977)   11111
105          (6781 - 0.044569034)   11100
106          (138 - 9.070235E-4)   0101110100
107          (1076 - 0.0070721544)   0101111
108          (4618 - 0.030352425)   10001
109          (1907 - 0.012534013)   010100
110          (6896 - 0.045324884)   11110
111          (7970 - 0.052383896)   0110
112          (1459 - 0.009589474)   1100010
113          (125 - 8.2157925E-4)   0101110001
114          (5297 - 0.034815244)   10101
115          (6282 - 0.04128929)   11001
116          (10217 - 0.067152604)   1011
117          (3402 - 0.022360101)   111010
118          (804 - 0.005284398)   11000111
119          (2438 - 0.016024083)   101000
120          (144 - 9.464593E-4)   1001101000
121          (2149 - 0.014124591)   100000
122          (77 - 5.0609285E-4)   10011010101

Entropia:       4.5675883
Longitud Media: 4.6123915
```

3. Código

Clase FuenteFile.

```
package Practica3;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.LinkedList;
import java.util.List;

import difflib.Delta;
import difflib.DiffUtils;
```

```
13 import difflib.Patch;
14
15 public class FuenteFile {
16
17 String path="/Users/";
18 String compressSufix="C.txt";
19 String decompressSufix="D.txt";
20 Huffman huffman;
21 String file;
22 long nanoCompIni,nanoCompFin;
23 long nanoDeCompIni,nanoDeCompFin;
24
25
26   public FuenteFile(String filep) throws FileNotFoundException{
27           file=filep;
28           huffman = new Huffman ();
29
30           nanoCompIni=System.nanoTime();
31          huffman.compress(path+file,path+file+compressSufix);
32          nanoCompFin=System.nanoTime();
33
34          nanoDeCompIni=System.nanoTime();
35          huffman.expand(path+file+compressSufix,path+file+decompressSufix);
36          nanoDeCompFin=System.nanoTime();
37
38
39 }
40 /*********
41  *  Metodo para Diff
42  * @param filename
43  * @return
44  */
45   private static List<String> fileToLines(String filename) {
46       List<String> lines = new LinkedList<String>();
47       String line = "";
48       BufferedReader in;
49       try {
50            in = new BufferedReader(new FileReader(filename));
51             while ((line = in.readLine()) != null) {
52                   lines.add(line);
53            }
54       } catch (IOException e) {
55                   e.printStackTrace();
56       }
57      return lines;
58  }
59 /************
60  *
61  */
62 public void stadistic(boolean printCodeTable){
63   long regularL,compressL,decompressL;
64   float ratio;
65
66   System.out.println("********************** \t "+file+" \t **********************");
67
68   File regular = new File(path+file);
69   regularL=regular.length();
```

```
70
71    File compress = new File(path+file+compressSufix);
72    compressL=compress.length();
73
74    ratio= (float) compressL / (float) regularL;
75    System.out.println("Size regular: "+regularL+" Size compress: "+compressL+ " Ratio: "+ ratio);
76
77
78    File decompress = new File(path+file+decompressSufix);
79    decompressL=decompress.length();
80
81    List<String> regularDiff = fileToLines(path+file);
82    List<String> decompressDiff  = fileToLines(path+file+decompressSufix);
83
84    // Compute diff. Get the Patch object. Patch is the container for computed deltas.
85     Patch patch = DiffUtils.diff(regularDiff, decompressDiff);
86     List<Delta> diff= patch.getDeltas();
87
88     if (diff.isEmpty()) {
89              System.out.print("Fichero Descomprimido correctamente!!!");
90       System.out.println("\t Size regular: "+regularL+" Size decompress: "+decompressL);
91       System.out.println("Tiempo Comprimir: "+(nanoCompFin-nanoCompIni)+" nanosegundos.
92                       Tiempo Descomprimir: "+(nanoDeCompFin-nanoDeCompIni)+" nanosegundos.");
93    }
94    else                                  System.out.println("Error al Descomprir!!!");
95
96
97   System.out.println("*****************************************************************");
98
99   if (printCodeTable) huffman.printCodeTableAndFrequency();
100 }
101
102
103
104 public static void main(String[] args) {
105
106           try {
107                 FuenteFile f1= new FuenteFile("Alice.txt");
108         f1.stadistic(true);
109
110                 FuenteFile f2= new FuenteFile("MobyDick.txt");
111                 f2.stadistic(true);
112
113                 FuenteFile f3= new FuenteFile("Quixote.txt");
114                 f3.stadistic(true);
115
116     } catch (FileNotFoundException e) {
117                 e.printStackTrace();
118         }
119
120     }
121
122 }
```

Clase Huffman.

```java
package Practica3;

import java.io.FileNotFoundException;

/*************************************************************************
 *  Codigo basado en http://algs4.cs.princeton.edu/55compression/
 *
 *************************************************************************/

public class Huffman {

    // alphabet size of extended ASCII
    private static final int R = 256;
    BinaryIn In;
    BinaryOut Out;
    String[] codeTable;
    int[] freq;

    public Huffman(){

    }

    // Huffman trie node
    private  class Node implements Comparable<Node> {
        private final char ch;
        private final int freq;
        private final Node left, right;

        Node(char ch, int freq, Node left, Node right) {
            this.ch    = ch;
            this.freq  = freq;
            this.left  = left;
            this.right = right;
        }

        // is the node a leaf node?
        private boolean isLeaf() {
            assert (left == null && right == null) || (left != null && right != null);
            return (left == null && right == null);
        }

        // compare, based on frequency
        public int compareTo(Node that) {
            return this.freq - that.freq;
        }
    }

/**********************
 * Compress
 * @param fileIn
 * @param fileOut
 * @throws FileNotFoundException
 */
    // compress bytes from standard input and write to standard output
    public  void compress(String fileIn, String fileOut) throws FileNotFoundException{
            In = new BinaryIn(fileIn);
        Out = new BinaryOut(fileOut);
```

```
58
59
60          // read the input
61          String s = In.readString();
62          char[] input = s.toCharArray();
63
64          // tabulate frequency counts
65          freq = new int[R];
66          for (int i = 0; i < input.length; i++)
67              freq[input[i]]++;
68
69          // build Huffman trie
70          Node root = buildTrie(freq);
71
72          // build code table
73          codeTable = new String[R];
74          buildCode(codeTable, root, "");
75
76          // print trie for decoder
77          writeTrie(root);
78
79          // print number of bytes in original uncompressed message
80          Out.write(input.length);
81
82          // use Huffman code to encode input
83          for (int i = 0; i < input.length; i++) {
84              String code = codeTable[input[i]];
85              for (int j = 0; j < code.length(); j++) {
86                  if (code.charAt(j) == '0') {
87                      Out.write(false);
88                  }
89                  else if (code.charAt(j) == '1') {
90                      Out.write(true);
91                  }
92                  else throw new IllegalStateException("Illegal state");
93              }
94          }
95
96          // close output stream
97          Out.close();
98
99
100     }
101
102 /************
103  * Imprime la codificacion
104  */
105     public void printCodeTableAndFrequency(){
106             long total=0;
107             float prob[]= new float [R];
108
109             System.out.println("\n\nCaracter  (Frecuencia - Probabilidad) \t Codificacion\n----------------------
110
111              for (int i=0;i<R;i++){
112                  if (codeTable[i]!=null){
113                          total+=freq[i];
114                  }
```

```
115          }
116
117              for (int i=0;i<R;i++){
118                   if (codeTable[i]!=null){
119                           prob[i]=(float) freq[i]/ (float) total;
120                          System.out.println(i+"\t ("+freq[i]+" - "+prob[i]+")   "+codeTable[i]);
121
122                   }
123                   else   prob[i]=0;
124           }
125
126              System.out.println("\nEntropia:       "+entropia(prob));
127              System.out.println("Longitud Media: "+longitudMedia(prob)+"\n");
128     }
129
130  /**********
131   *
132   * @param probabilidades
133   * @return
134   */
135     public float longitudMedia(float probabilidades[]) {
136                   float l = 0;
137
138                   for (int i = 0; i < R; i++) {
139                           if (probabilidades[i]!=0) l += probabilidades[i] * codeTable[i].length();
140                   }
141                   return l;
142           }
143
144  /***************
145   * Entropia
146   * @param probabilidades
147   * @return
148   */
149     public float entropia(float probabilidades[]) {
150                 float h = 0;
151
152                 for (int i = 0; i < R; i++) {
153                         if (probabilidades[i]!=0) h += probabilidades[i] * logBase2(1.0/probabilidades[i]);
154                 }
155                 return h;
156         }
157
158     /*******
159           * Calculo logartimo base 2 mediante cambio de base
160           * @param num
161           * @return
162           */
163         private  double logBase2(double num) {
164                 return logCambioBase(num,2);
165         }
166
167         private  double logCambioBase(double num, int b) {
168                 return Math.log(num)/Math.log(b);
169         }
170  /*****************
171   *   build the Huffman trie given frequencies
```

```
 * @param freq
 * @return
 */

   private  Node buildTrie(int[] freq) {

       // initialze priority queue with singleton trees
       MinPQ<Node> pq = new MinPQ<Node>();
       for (char i = 0; i < R; i++)
           if (freq[i] > 0)
               pq.insert(new Node(i, freq[i], null, null));

       // merge two smallest trees
       while (pq.size() > 1) {
           Node left  = pq.delMin();
           Node right = pq.delMin();
           Node parent = new Node('\0', left.freq + right.freq, left, right);
           pq.insert(parent);
       }
       return pq.delMin();
   }

/************
 *   write bitstring-encoded trie to standard output
 * @param x
 */
   private  void writeTrie(Node x) {
       if (x.isLeaf()) {
           Out.write(true);
           Out.write(x.ch, 8);
           return;
       }
       Out.write(false);
       writeTrie(x.left);
       writeTrie(x.right);
   }

   /*******
    * make a lookup table from symbols and their encodings
    *
    * @param st
    * @param x
    * @param s
    */

   private  void buildCode(String[] st, Node x, String s) {
       if (!x.isLeaf()) {
           buildCode(st, x.left,  s + '0');
           buildCode(st, x.right, s + '1');
       }
       else {
           st[x.ch] = s;
       }
   }

/******************
 * Expand
```

```
 * @param fileIn
 * @param fileOut
 * @throws FileNotFoundException
 */
   // expand Huffman-encoded input from standard input and write to standard output
   public  void expand(String fileIn, String fileOut) throws FileNotFoundException{
            In = new BinaryIn(fileIn);
        Out = new BinaryOut(fileOut);

        // read in Huffman trie from input stream
        Node root = readTrie();

        // number of bytes to write
        int length = In.readInt();

        // decode using the Huffman trie
        for (int i = 0; i < length; i++) {
            Node x = root;
            while (!x.isLeaf()) {
                boolean bit = In.readBoolean();
                if (bit) x = x.right;
                else     x = x.left;
            }
            Out.write(x.ch, 8);
        }
        Out.close();
    }

/*************
 *
 * @return
 */
   private  Node readTrie() {
        boolean isLeaf = In.readBoolean();
        if (isLeaf) {
            return new Node(In.readChar(), -1, null, null);
        }
        else {
            return new Node('\0', -1, readTrie(), readTrie());
        }
    }


}
```

Clase MinPQ.

```
package Practica3;

/******************************************************************************
 *  Compilation:  javac MinPQ.java
 *  Execution:    java MinPQ < input.txt
 *
 *  Generic min priority queue implementation with a binary heap.
```

```
 *  Can be used with a comparator instead of the natural order.
 *
 *  % java MinPQ < tinyPQ.txt
 *  E A E (6 left on pq)
 *
 *  We use a one-based array to simplify parent and child calculations.
 *
 ******************************************************************************/

import java.util.Comparator;
import java.util.Iterator;
import java.util.NoSuchElementException;

/**
 *  Codigo basado en http://algs4.cs.princeton.edu/55compression/
 *
    The <tt>MinPQ</tt> class represents a priority queue of generic keys.
 *  It supports the usual <em>insert</em> and <em>delete-the-minimum</em>
 *  operations, along with methods for peeking at the minimum key,
 *  testing if the priority queue is empty, and iterating through
 *  the keys.
 *  <p>
 *  This implementation uses a binary heap.
 *  The <em>insert</em> and <em>delete-the-minimum</em> operations take
 *  logarithmic amortized time.
 *  The <em>min</em>, <em>size</em>, and <em>is-empty</em> operations take constant time.
 *  Construction takes time proportional to the specified capacity or the number of
 *  items used to initialize the data structure.
 *  <p>
 *  For additional documentation, see <a href="http://algs4.cs.princeton.edu/24pq">Section 2.4</a> of
 *  <i>Algorithms, 4th Edition</i> by Robert Sedgewick and Kevin Wayne.
 *
 *  @author Robert Sedgewick
 *  @author Kevin Wayne
 */
public class MinPQ<Key> implements Iterable<Key> {
    private Key[] pq;                    // store items at indices 1 to N
    private int N;                       // number of items on priority queue
    private Comparator<Key> comparator;  // optional comparator

    /**
     * Initializes an empty priority queue with the given initial capacity.
     * @param initCapacity the initial capacity of the priority queue
     */
    public MinPQ(int initCapacity) {
        pq = (Key[]) new Object[initCapacity + 1];
        N = 0;
    }

    /**
     * Initializes an empty priority queue.
     */
    public MinPQ() {
        this(1);
    }

    /**
```

```
65      * Initializes an empty priority queue with the given initial capacity,
66      * using the given comparator.
67      * @param initCapacity the initial capacity of the priority queue
68      * @param comparator the order to use when comparing keys
69      */
70     public MinPQ(int initCapacity, Comparator<Key> comparator) {
71         this.comparator = comparator;
72         pq = (Key[]) new Object[initCapacity + 1];
73         N = 0;
74     }
75
76     /**
77      * Initializes an empty priority queue using the given comparator.
78      * @param comparator the order to use when comparing keys
79      */
80     public MinPQ(Comparator<Key> comparator) { this(1, comparator); }
81
82     /**
83      * Initializes a priority queue from the array of keys.
84      * Takes time proportional to the number of keys, using sink-based heap construction.
85      * @param keys the array of keys
86      */
87     public MinPQ(Key[] keys) {
88         N = keys.length;
89         pq = (Key[]) new Object[keys.length + 1];
90         for (int i = 0; i < N; i++)
91             pq[i+1] = keys[i];
92         for (int k = N/2; k >= 1; k--)
93             sink(k);
94         assert isMinHeap();
95     }
96
97     /**
98      * Is the priority queue empty?
99      * @return true if the priority queue is empty; false otherwise
100     */
101    public boolean isEmpty() {
102        return N == 0;
103    }
104
105    /**
106     * Returns the number of keys on the priority queue.
107     * @return the number of keys on the priority queue
108     */
109    public int size() {
110        return N;
111    }
112
113    /**
114     * Returns a smallest key on the priority queue.
115     * @return a smallest key on the priority queue
116     * @throws java.util.NoSuchElementException if priority queue is empty
117     */
118    public Key min() {
119        if (isEmpty()) throw new NoSuchElementException("Priority queue underflow");
120        return pq[1];
121    }
```

```
122
123     // helper function to double the size of the heap array
124     private void resize(int capacity) {
125         assert capacity > N;
126         Key[] temp = (Key[]) new Object[capacity];
127         for (int i = 1; i <= N; i++) temp[i] = pq[i];
128         pq = temp;
129     }
130
131     /**
132      * Adds a new key to the priority queue.
133      * @param x the key to add to the priority queue
134      */
135     public void insert(Key x) {
136         // double size of array if necessary
137         if (N == pq.length - 1) resize(2 * pq.length);
138
139         // add x, and percolate it up to maintain heap invariant
140         pq[++N] = x;
141         swim(N);
142         assert isMinHeap();
143     }
144
145     /**
146      * Removes and returns a smallest key on the priority queue.
147      * @return a smallest key on the priority queue
148      * @throws java.util.NoSuchElementException if the priority queue is empty
149      */
150     public Key delMin() {
151         if (isEmpty()) throw new NoSuchElementException("Priority queue underflow");
152         exch(1, N);
153         Key min = pq[N--];
154         sink(1);
155         pq[N+1] = null;         // avoid loitering and help with garbage collection
156         if ((N > 0) && (N == (pq.length - 1) / 4)) resize(pq.length  / 2);
157         assert isMinHeap();
158         return min;
159     }
160
161
162    /***************************************************************************
163     * Helper functions to restore the heap invariant.
164     ***************************************************************************/
165
166     private void swim(int k) {
167         while (k > 1 && greater(k/2, k)) {
168             exch(k, k/2);
169             k = k/2;
170         }
171     }
172
173     private void sink(int k) {
174         while (2*k <= N) {
175             int j = 2*k;
176             if (j < N && greater(j, j+1)) j++;
177             if (!greater(k, j)) break;
178             exch(k, j);
```

```
179             k = j;
180         }
181     }
182
183     /***************************************************************************
184      * Helper functions for compares and swaps.
185      ***************************************************************************/
186     private boolean greater(int i, int j) {
187         if (comparator == null) {
188             return ((Comparable<Key>) pq[i]).compareTo(pq[j]) > 0;
189         }
190         else {
191             return comparator.compare(pq[i], pq[j]) > 0;
192         }
193     }
194
195     private void exch(int i, int j) {
196         Key swap = pq[i];
197         pq[i] = pq[j];
198         pq[j] = swap;
199     }
200
201     // is pq[1..N] a min heap?
202     private boolean isMinHeap() {
203         return isMinHeap(1);
204     }
205
206     // is subtree of pq[1..N] rooted at k a min heap?
207     private boolean isMinHeap(int k) {
208         if (k > N) return true;
209         int left = 2*k, right = 2*k + 1;
210         if (left  <= N && greater(k, left))  return false;
211         if (right <= N && greater(k, right)) return false;
212         return isMinHeap(left) && isMinHeap(right);
213     }
214
215
216     /***************************************************************************
217      * Iterators
218      ***************************************************************************/
219
220     /**
221      * Returns an iterator that iterates over the keys on the priority queue
222      * in ascending order.
223      * The iterator doesn't implement <tt>remove()</tt> since it's optional.
224      * @return an iterator that iterates over the keys in ascending order
225      */
226     public Iterator<Key> iterator() { return new HeapIterator(); }
227
228     private class HeapIterator implements Iterator<Key> {
229         // create a new pq
230         private MinPQ<Key> copy;
231
232         // add all items to copy of heap
233         // takes linear time since already in heap order so no keys move
234         public HeapIterator() {
235             if (comparator == null) copy = new MinPQ<Key>(size());
```

```
            else                    copy = new MinPQ<Key>(size(), comparator);
            for (int i = 1; i <= N; i++)
                copy.insert(pq[i]);
        }

        public boolean hasNext()  { return !copy.isEmpty();                     }
        public void remove()      { throw new UnsupportedOperationException();  }

        public Key next() {
            if (!hasNext()) throw new NoSuchElementException();
            return copy.delMin();
        }
    }


}
```

Clase BinaryIn.

```
package Practica3;
/*************************************************************************
 *  Compilation:  javac BinaryStdIn.java
 *  Execution:    java BinaryStdIn < input > output
 *
 *  Supports reading binary data from standard input.
 *
 *  % java BinaryStdIn < input.jpg > output.jpg
 *  % diff input.jpg output.jpg
 *
 *************************************************************************/

import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

/**
 *  <i>Binary standard input</i>. This class provides methods for reading
 *  in bits from standard input, either one bit at a time (as a <tt>boolean</tt>),
 *  8 bits at a time (as a <tt>byte</tt> or <tt>char</tt>),
 *  16 bits at a time (as a <tt>short</tt>), 32 bits at a time
 *  (as an <tt>int</tt> or <tt>float</tt>), or 64 bits at a time (as a
 *  <tt>double</tt> or <tt>long</tt>).
 *  <p>
 *  All primitive types are assumed to be represented using their
 *  standard Java representations, in big-endian (most significant
 *  byte first) order.
 *  <p>
 *  The client should not intermix calls to <tt>BinaryStdIn</tt> with calls
 *  to <tt>StdIn</tt> or <tt>System.in</tt>;
 *  otherwise unexpected behavior will result.
 *
 *  @author Robert Sedgewick
```

```
 *  @author Kevin Wayne
 */
public final class BinaryIn {
    private  BufferedInputStream in;
    private  final int EOF = -1;    // end of file

    private  int buffer;            // one character buffer
    private  int N;                 // number of bits left in buffer

    // static initializer
    //static { fillBuffer(); }

    // don't instantiate
    public BinaryIn(String file) throws FileNotFoundException {

             in = new BufferedInputStream(new FileInputStream(file));
             fillBuffer();
    }

    private  void fillBuffer() {
        try { buffer = in.read(); N = 8; }
        catch (IOException e) { System.out.println("EOF"); buffer = EOF; N = -1; }
    }

   /**
     * Close this input stream and release any associated system resources.
     */
    public  void close() {
        try {
            in.close();
        }
        catch (IOException e) {
            e.printStackTrace();
            throw new RuntimeException("Could not close BinaryStdIn");
        }
    }

   /**
     * Returns true if standard input is empty.
     * @return true if and only if standard input is empty
     */
    public  boolean isEmpty() {
        return buffer == EOF;
    }

   /**
     * Read the next bit of data from standard input and return as a boolean.
     * @return the next bit of data from standard input as a <tt>boolean</tt>
     * @throws RuntimeException if standard input is empty
     */
    public  boolean readBoolean() {
        if (isEmpty()) throw new RuntimeException("Reading from empty input stream");
        N--;
        boolean bit = ((buffer >> N) & 1) == 1;
        if (N == 0) fillBuffer();
        return bit;
    }
```

```
93
94      /**
95        * Read the next 8 bits from standard input and return as an 8-bit char.
96        * Note that <tt>char</tt> is a 16-bit type;
97        * to read the next 16 bits as a char, use <tt>readChar(16)</tt>
98        * @return the next 8 bits of data from standard input as a <tt>char</tt>
99        * @throws RuntimeException if there are fewer than 8 bits available on standard input
100       */
101     public  char readChar() {
102         if (isEmpty()) throw new RuntimeException("Reading from empty input stream");
103
104         // special case when aligned byte
105         if (N == 8) {
106             int x = buffer;
107             fillBuffer();
108             return (char) (x & 0xff);
109         }
110
111         // combine last N bits of current buffer with first 8-N bits of new buffer
112         int x = buffer;
113         x <<= (8-N);
114         int oldN = N;
115         fillBuffer();
116         if (isEmpty()) throw new RuntimeException("Reading from empty input stream");
117         N = oldN;
118         x |= (buffer >>> N);
119         return (char) (x & 0xff);
120         // the above code doesn't quite work for the last character if N = 8
121         // because buffer will be -1
122     }
123
124     /**
125       * Read the next r bits from standard input and return as an r-bit character.
126       * @param r number of bits to read.
127       * @return the next r bits of data from standard input as a <tt>char</tt>
128       * @throws RuntimeException if there are fewer than r bits available on standard input
129       * @throws RuntimeException unless 1 &le; r &le; 16
130       */
131     public  char readChar(int r) {
132         if (r < 1 || r > 16) throw new RuntimeException("Illegal value of r = " + r);
133
134         // optimize r = 8 case
135         if (r == 8) return readChar();
136
137         char x = 0;
138         for (int i = 0; i < r; i++) {
139             x <<= 1;
140             boolean bit = readBoolean();
141             if (bit) x |= 1;
142         }
143         return x;
144     }
145
146     /**
147       * Read the remaining bytes of data from standard input and return as a string.
148       * @return the remaining bytes of data from standard input as a <tt>String</tt>
149       * @throws RuntimeException if standard input is empty or if the number of bits
```

```
150      * available on standard input is not a multiple of 8 (byte-aligned)
151      */
152     public  String readString() {
153         if (isEmpty()) throw new RuntimeException("Reading from empty input stream");
154 
155         StringBuilder sb = new StringBuilder();
156         while (!isEmpty()) {
157             char c = readChar();
158             sb.append(c);
159         }
160         return sb.toString();
161     }
162 
163 
164    /**
165      * Read the next 16 bits from standard input and return as a 16-bit short.
166      * @return the next 16 bits of data from standard input as a <tt>short</tt>
167      * @throws RuntimeException if there are fewer than 16 bits available on standard input
168      */
169     public  short readShort() {
170         short x = 0;
171         for (int i = 0; i < 2; i++) {
172             char c = readChar();
173             x <<= 8;
174             x |= c;
175         }
176         return x;
177     }
178 
179    /**
180      * Read the next 32 bits from standard input and return as a 32-bit int.
181      * @return the next 32 bits of data from standard input as a <tt>int</tt>
182      * @throws RuntimeException if there are fewer than 32 bits available on standard input
183      */
184     public  int readInt() {
185         int x = 0;
186         for (int i = 0; i < 4; i++) {
187             char c = readChar();
188             x <<= 8;
189             x |= c;
190         }
191         return x;
192     }
193 
194    /**
195      * Read the next r bits from standard input and return as an r-bit int.
196      * @param r number of bits to read.
197      * @return the next r bits of data from standard input as a <tt>int</tt>
198      * @throws RuntimeException if there are fewer than r bits available on standard input
199      * @throws RuntimeException unless 1 &le; r &le; 32
200      */
201     public  int readInt(int r) {
202         if (r < 1 || r > 32) throw new RuntimeException("Illegal value of r = " + r);
203 
204         // optimize r = 32 case
205         if (r == 32) return readInt();
206 
```

```
        int x = 0;
        for (int i = 0; i < r; i++) {
            x <<= 1;
            boolean bit = readBoolean();
            if (bit) x |= 1;
        }
        return x;
    }

   /**
     * Read the next 64 bits from standard input and return as a 64-bit long.
     * @return the next 64 bits of data from standard input as a <tt>long</tt>
     * @throws RuntimeException if there are fewer than 64 bits available on standard input
     */
    public  long readLong() {
        long x = 0;
        for (int i = 0; i < 8; i++) {
            char c = readChar();
            x <<= 8;
            x |= c;
        }
        return x;
    }


   /**
     * Read the next 64 bits from standard input and return as a 64-bit double.
     * @return the next 64 bits of data from standard input as a <tt>double</tt>
     * @throws RuntimeException if there are fewer than 64 bits available on standard input
     */
    public  double readDouble() {
        return Double.longBitsToDouble(readLong());
    }

   /**
     * Read the next 32 bits from standard input and return as a 32-bit float.
     * @return the next 32 bits of data from standard input as a <tt>float</tt>
     * @throws RuntimeException if there are fewer than 32 bits available on standard input
     */
    public  float readFloat() {
        return Float.intBitsToFloat(readInt());
    }


   /**
     * Read the next 8 bits from standard input and return as an 8-bit byte.
     * @return the next 8 bits of data from standard input as a <tt>byte</tt>
     * @throws RuntimeException if there are fewer than 8 bits available on standard input
     */
    public  byte readByte() {
        char c = readChar();
        byte x = (byte) (c & 0xff);
        return x;
    }


}
```

Clase BinaryOut.

```
1  package Practica3;
2  /*************************************************************************
3   *  Compilation:  javac BinaryStdOut.java
4   *  Execution:    java BinaryStdOut
5   *
6   *  Write binary data to standard output, either one 1-bit boolean,
7   *  one 8-bit char, one 32-bit int, one 64-bit double, one 32-bit float,
8   *  or one 64-bit long at a time.
9   *
10  *  The bytes written are not aligned.
11  *
12  *************************************************************************/
13
14 import java.io.BufferedOutputStream;
15 import java.io.FileNotFoundException;
16 import java.io.FileOutputStream;
17 import java.io.IOException;
18
19 /**
20  *  <i>Binary standard output</i>. This class provides methods for converting
21  *  primtive type variables (<tt>boolean</tt>, <tt>byte</tt>, <tt>char</tt>,
22  *  <tt>int</tt>, <tt>long</tt>, <tt>float</tt>, and <tt>double</tt>)
23  *  to sequences of bits and writing them to standard output.
24  *  Uses big-endian (most-significant byte first).
25  *  <p>
26  *  The client must <tt>flush()</tt> the output stream when finished writing bits.
27  *  <p>
28  *  The client should not intermixing calls to <tt>BinaryStdOut</tt> with calls
29  *  to <tt>StdOut</tt> or <tt>System.out</tt>; otherwise unexpected behavior
30  *  will result.
31  *
32  *  @author Robert Sedgewick
33  *  @author Kevin Wayne
34  */
35 public final class BinaryOut  {
36          String file =null;
37         private   BufferedOutputStream out;
38
39
40     private  int buffer;     // 8-bit buffer of bits to write out
41     private  int N;          // number of bits remaining in buffer
42
43     // don't instantiate
44     public BinaryOut(String filep) throws FileNotFoundException {
45             file =filep;
46
47             out=new BufferedOutputStream(new FileOutputStream(file));
48     }
49
50    /**
51      * Write the specified bit to standard output.
52      */
53     private  void writeBit(boolean bit) {
54         // add bit to buffer
55         buffer <<= 1;
```

```
56          if (bit) buffer |= 1;
57
58          // if buffer is full (8 bits), write out as a single byte
59          N++;
60          if (N == 8) clearBuffer();
61      }
62
63      /**
64        * Write the 8-bit byte to standard output.
65        */
66      private  void writeByte(int x) {
67          assert x >= 0 && x < 256;
68
69          // optimized if byte-aligned
70          if (N == 0) {
71              try { out.write(x); }
72              catch (IOException e) { e.printStackTrace(); }
73              return;
74          }
75
76          // otherwise write one bit at a time
77          for (int i = 0; i < 8; i++) {
78              boolean bit = ((x >>> (8 - i - 1)) & 1) == 1;
79              writeBit(bit);
80          }
81      }
82
83      // write out any remaining bits in buffer to standard output, padding with 0s
84      private  void clearBuffer() {
85          if (N == 0) return;
86          if (N > 0) buffer <<= (8 - N);
87          try { out.write(buffer); }
88          catch (IOException e) { e.printStackTrace(); }
89          N = 0;
90          buffer = 0;
91      }
92
93      /**
94        * Flush standard output, padding 0s if number of bits written so far
95        * is not a multiple of 8.
96        */
97      public  void flush() {
98          clearBuffer();
99          try { out.flush(); }
100         catch (IOException e) { e.printStackTrace(); }
101     }
102
103     /**
104       * Flush and close standard output. Once standard output is closed, you can no
105       * longer write bits to it.
106       */
107     public  void close() {
108         flush();
109         try { out.close(); }
110         catch (IOException e) { e.printStackTrace(); }
111     }
112
```

```
113
114     /**
115       * Write the specified bit to standard output.
116       * @param x the <tt>boolean</tt> to write.
117       */
118      public  void write(boolean x) {
119          writeBit(x);
120      }
121
122     /**
123       * Write the 8-bit byte to standard output.
124       * @param x the <tt>byte</tt> to write.
125       */
126      public  void write(byte x) {
127          writeByte(x & 0xff);
128      }
129
130     /**
131       * Write the 32-bit int to standard output.
132       * @param x the <tt>int</tt> to write.
133       */
134      public  void write(int x) {
135          writeByte((x >>> 24) & 0xff);
136          writeByte((x >>> 16) & 0xff);
137          writeByte((x >>>  8) & 0xff);
138          writeByte((x >>>  0) & 0xff);
139      }
140
141     /**
142       * Write the r-bit int to standard output.
143       * @param x the <tt>int</tt> to write.
144       * @param r the number of relevant bits in the char.
145       * @throws RuntimeException if <tt>r</tt> is not between 1 and 32.
146       * @throws RuntimeException if <tt>x</tt> is not between 0 and 2<sup>r</sup> - 1.
147       */
148      public  void write(int x, int r) {
149          if (r == 32) { write(x); return; }
150          if (r < 1 || r > 32)        throw new RuntimeException("Illegal value for r = " + r);
151          if (x < 0 || x >= (1 << r)) throw new RuntimeException("Illegal " + r + "-bit char = " + x);
152          for (int i = 0; i < r; i++) {
153              boolean bit = ((x >>> (r - i - 1)) & 1) == 1;
154              writeBit(bit);
155          }
156      }
157
158
159
160
161
162     /**
163       * Write the 64-bit double to standard output.
164       * @param x the <tt>double</tt> to write.
165       */
166      public  void write(double x) {
167          write(Double.doubleToRawLongBits(x));
168      }
169
```

```
170     /**
171       * Write the 64-bit long to standard output.
172       * @param x the <tt>long</tt> to write.
173       */
174      public  void write(long x) {
175          writeByte((int) ((x >>> 56) & 0xff));
176          writeByte((int) ((x >>> 48) & 0xff));
177          writeByte((int) ((x >>> 40) & 0xff));
178          writeByte((int) ((x >>> 32) & 0xff));
179          writeByte((int) ((x >>> 24) & 0xff));
180          writeByte((int) ((x >>> 16) & 0xff));
181          writeByte((int) ((x >>>  8) & 0xff));
182          writeByte((int) ((x >>>  0) & 0xff));
183      }
184
185     /**
186       * Write the 32-bit float to standard output.
187       * @param x the <tt>float</tt> to write.
188       */
189      public  void write(float x) {
190          write(Float.floatToRawIntBits(x));
191      }
192
193     /**
194       * Write the 16-bit int to standard output.
195       * @param x the <tt>short</tt> to write.
196       */
197      public  void write(short x) {
198          writeByte((x >>>  8) & 0xff);
199          writeByte((x >>>  0) & 0xff);
200      }
201
202     /**
203       * Write the 8-bit char to standard output.
204       * @param x the <tt>char</tt> to write.
205       * @throws RuntimeException if <tt>x</tt> is not betwen 0 and 255.
206       */
207      public  void write(char x) {
208          if (x < 0 || x >= 256) throw new RuntimeException("Illegal 8-bit char = " + x);
209          writeByte(x);
210      }
211
212     /**
213       * Write the r-bit char to standard output.
214       * @param x the <tt>char</tt> to write.
215       * @param r the number of relevant bits in the char.
216       * @throws RuntimeException if <tt>r</tt> is not between 1 and 16.
217       * @throws RuntimeException if <tt>x</tt> is not between 0 and 2<sup>r</sup> - 1.
218       */
219      public  void write(char x, int r) {
220          if (r == 8) { write(x); return; }
221          if (r < 1 || r > 16)        throw new RuntimeException("Illegal value for r = " + r);
222          if (x < 0 || x >= (1 << r)) throw new RuntimeException("Illegal " + r + "-bit char = " + x);
223          for (int i = 0; i < r; i++) {
224              boolean bit = ((x >>> (r - i - 1)) & 1) == 1;
225              writeBit(bit);
226          }
```

```
227     }
228
229    /**
230      * Write the string of 8-bit characters to standard output.
231      * @param s the <tt>String</tt> to write.
232      * @throws RuntimeException if any character in the string is not
233      * between 0 and 255.
234      */
235     public  void write(String s) {
236         for (int i = 0; i < s.length(); i++)
237             write(s.charAt(i));
238     }
239
240    /**
241      * Write the String of r-bit characters to standard output.
242      * @param s the <tt>String</tt> to write.
243      * @param r the number of relevants bits in each character.
244      * @throws RuntimeException if r is not between 1 and 16.
245      * @throws RuntimeException if any character in the string is not
246      * between 0 and 2<sup>r</sup> - 1.
247      */
248     public  void write(String s, int r) {
249         for (int i = 0; i < s.length(); i++)
250             write(s.charAt(i), r);
251     }
252
253
254 }
```