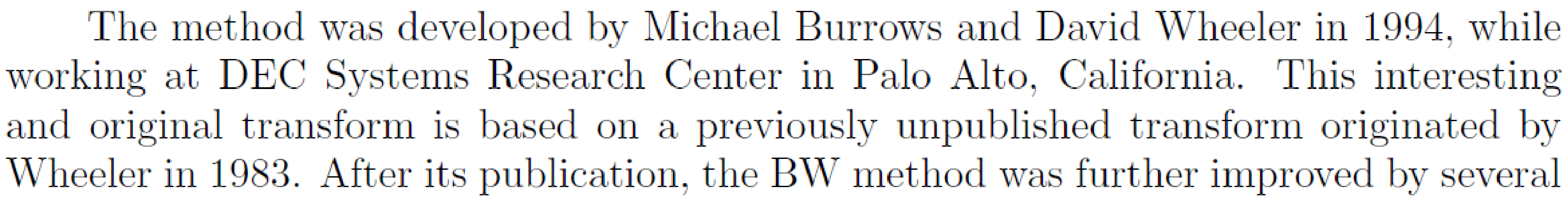
Burrows-Wheeler

CS6025 Data Encoding

Yizong Cheng

1-15-15

Burrows-Wheeler 1994



The method was developed by Michael Burrows and David Wheeler in 1994, while working at DEC Systems Research Center in Palo Alto, California. This interesting and original transform is based on a previously unpublished transform originated by Wheeler in 1983. After its publication, the BW method was further improved by several

Burrows-Wheeler

- Goal: Reduce entropy without changing data size so that average Huffman codeword length is shorter.
- Method: Permute data so that same symbol occur next to each other more often and then re-label the symbols.
- Requirements: permutation and relabeling must be invertible.

Permutation Method

- Generate all rotations of data.
- Arrange all rotations in lexicographical order.
- This generates a square matrix of symbols.
- Each row is a rotation of the original data.
  - One row is the original data (row I).
- Each column is a permutation of the data.
- The first column, F, is the sorted sequence of symbols in data, which can be generated from any other column by sorting.
- We can recover data from the last column L.

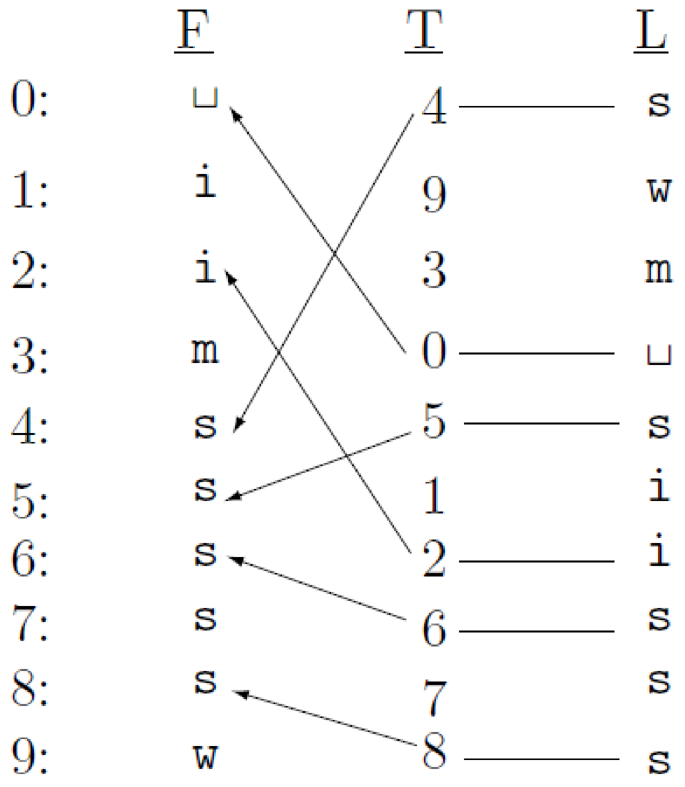
BW Steps

```
swiss␣miss
wiss␣misss
iss␣misssw
ss␣missswi
s␣missswis
␣missswiss
missswiss␣
issswiss␣m
ssswiss␣mi
sswiss␣mis
```

(a)

```
␣missswiss
iss␣misssw
issswiss␣m
missswiss␣
s␣missswis
ss␣missswi
ssswiss␣mi
sswiss␣mis
swiss␣miss
wiss␣misss
```

(b)



	F	T	L
0:	␣	4	s
1:	i	9	w
2:	i	3	m
3:	m	0	␣
4:	s	5	s
5:	s	1	i
6:	s	2	i
7:	s	6	s
8:	s	7	s
9:	w	8	s

(c)

Figure 11.1: Principles of BW Compression.

Recovering Data From L and I

- L is the last column of the rotation matrix.
- I is the index of the row for the original data.
- F, the first column, is obtained by sorting L.
- F[I] is the first symbol of original data, S[0].
- S[1] is the first symbol of a row whose last symbol is S[0].
- There may be many rows with S[0] as the last symbol! Which one?

Multiple Occurrences of Symbols

- Suppose that there are two rows $j < k$ with the same first symbol. $F[j] = F[k]$.
- We have $j < k$ because the left-rotation of row j is lexicographically smaller than the left-rotation of row k.
- Multiple occurrences of the same symbol in F and in L map to each other with the same order.

The Transition Array T

- We can compute an array T so that T[j] is the index of the left rotation of row j.
- If F[j] = F[k] and j < k, then T[j] < T[k].
- F[j] = L[T[j]] and F[T[j]] is the symbol following F[j] in the original data.
- Output original data as F[I], F[T[I]], F[T[T[I]]], …
    - The kth symbol in the original data is $F[T^k(I)]$.

Properties of L

- L, along with I, is a transform of S, the original data.
- L has the property that the same symbols are clustered next to each other when they are followed by the same strings of symbols.
- There are long runs of the same symbols in L (if not as much as in F, which cannot be used to recover S).
- L has the same entropy as S.

Move-To-Front Coding

- Move-to-front is a way to replace recurrent symbols (bursts of the same symbol) with small numbers.
- Smaller numbers will have higher frequencies.
- Entropy may be reduced even when the number of symbols stays the same.
- Huffman coding can now be used on move-to-front coding to get better compression rate.

Move-To-Front Coding

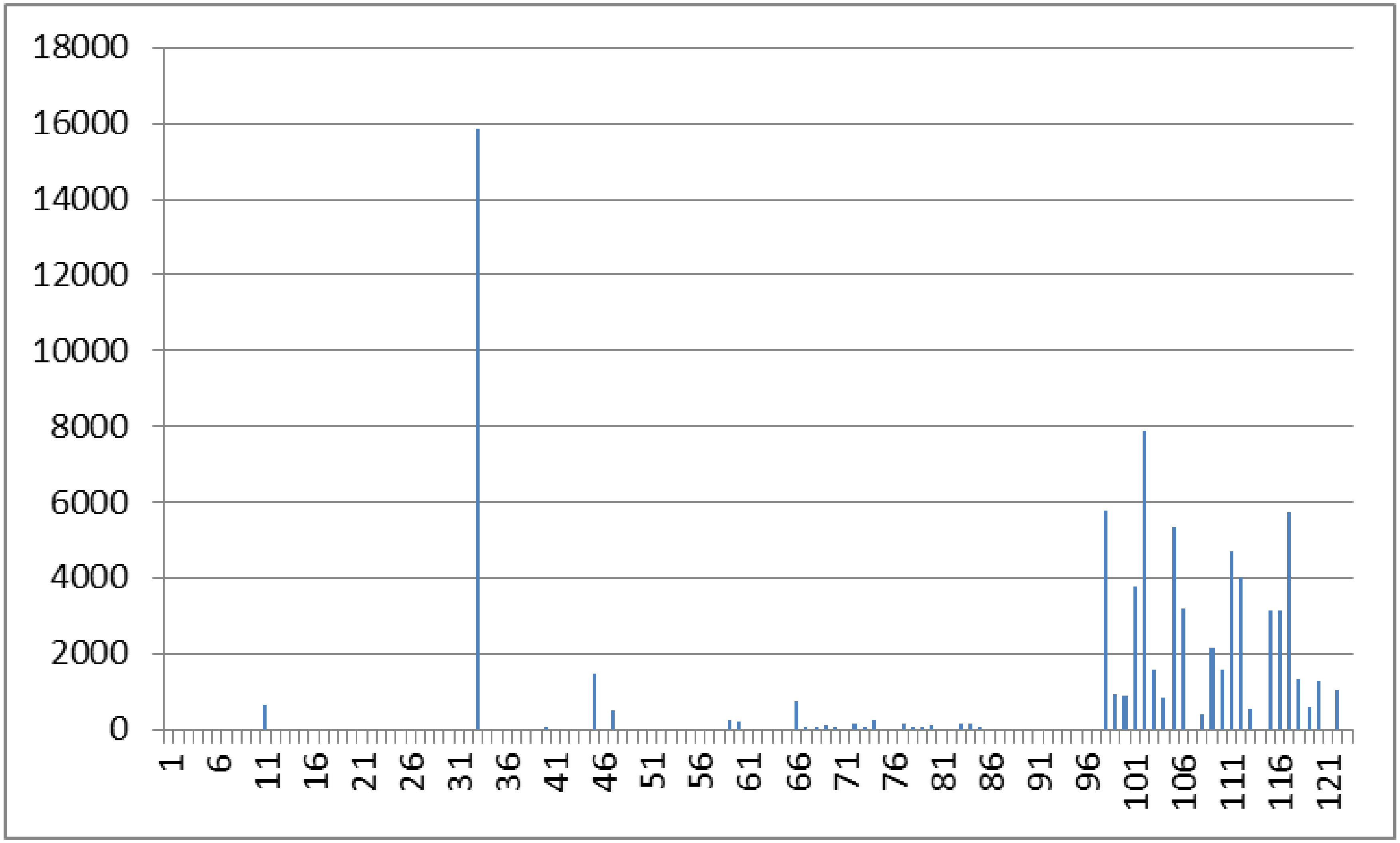
- Let array A be the set of symbols used in data.
- Replace each symbol by the current position of the symbol in A.
- After emitting the code for a symbol, move the symbol to the front of array A.
- When the same symbol occurs again, the code may become a small number because it’s been moved to front.

Move-To-Front Encoding and Decoding

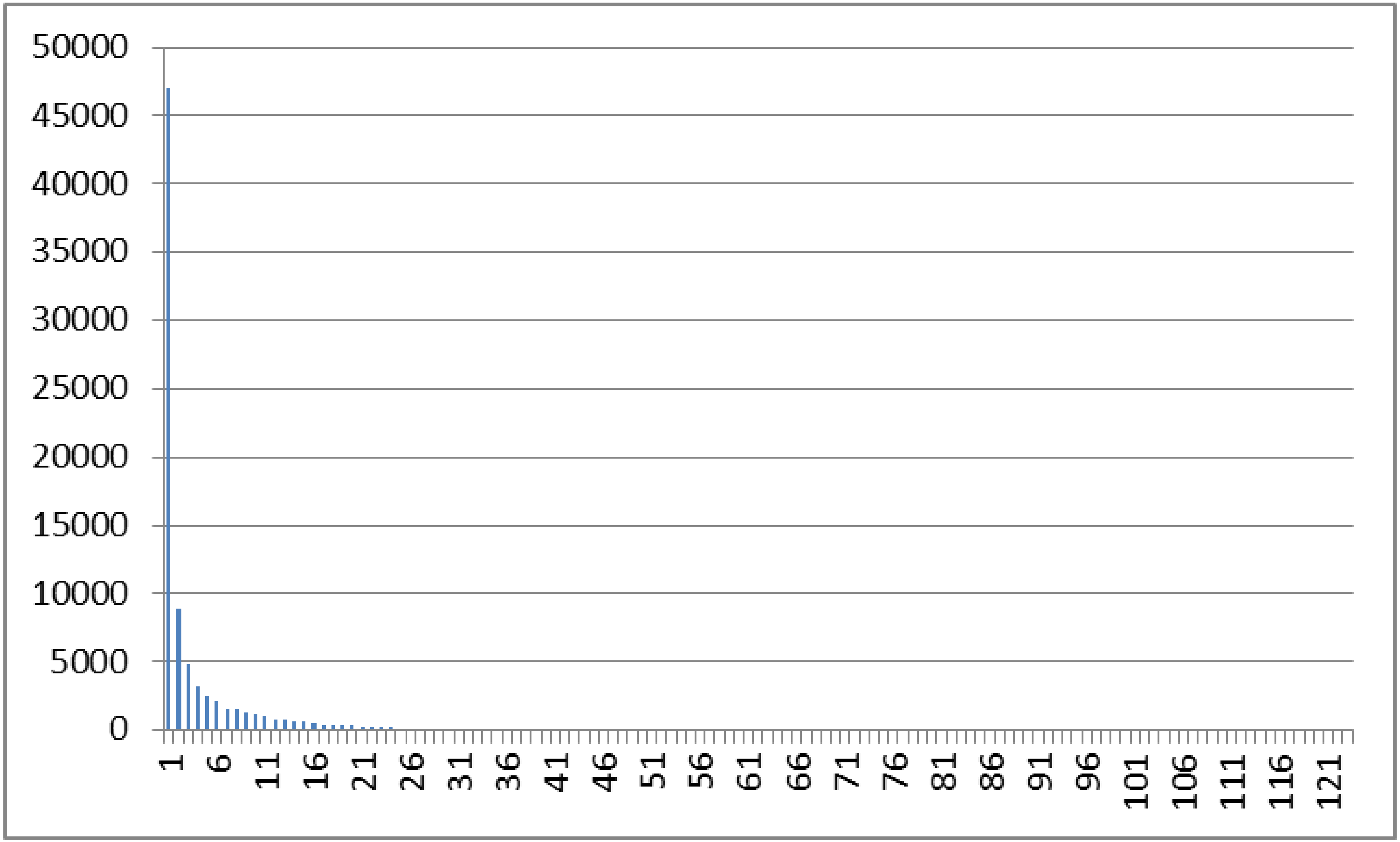
L	A	Code	C	A	L
s	␣imsw	3	3	␣imsw	s
w	s␣imw	4	4	s␣imw	w
m	ws␣im	4	4	ws␣im	m
␣	mws␣i	3	3	mws␣i	␣
s	␣mwsi	3	3	␣mwsi	s
i	s␣mwi	4	4	s␣mwi	i
i	is␣mw	0	0	is␣mw	i
s	is␣mw	1	1	is␣mw	s
s	si␣mw	0	0	si␣mw	s
s	si␣mw	0	0	si␣mw	s
	(a)			(b)	

Figure 11.2: Encoding/Decoding L by Move-to-Front.

Entropy = 4.33 English Text



Entropy = 2.68 after BW



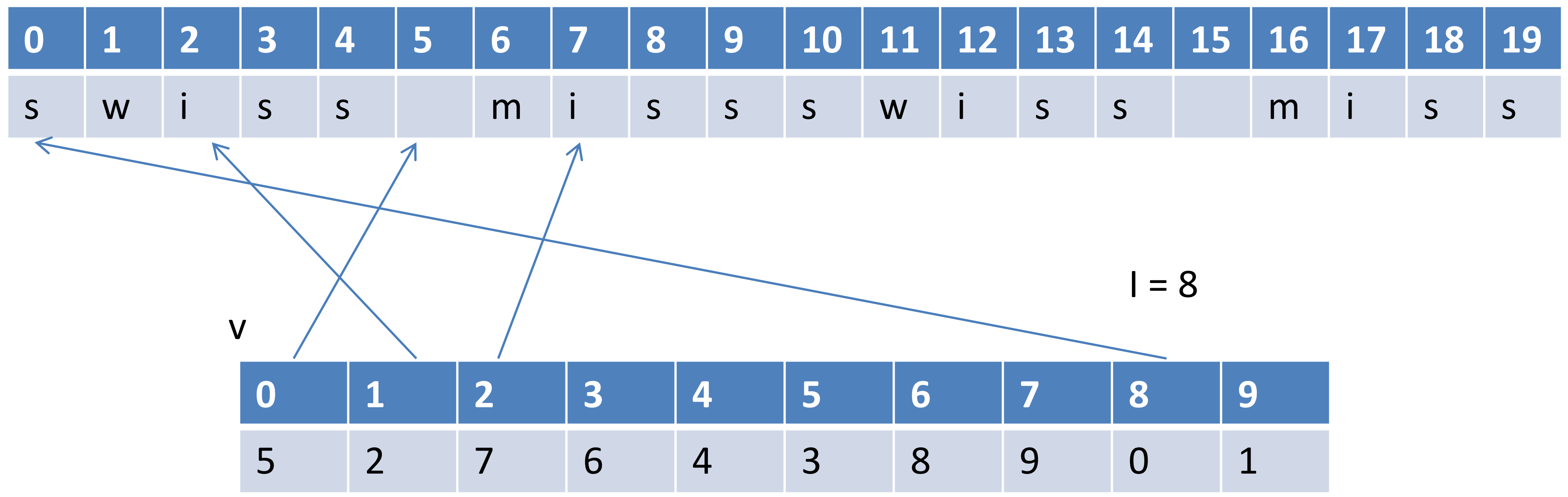
Burrows-Wheeler Encoding

```
class H2A{
  static int BLOCKSIZE = 16384;
  static int numberOfSymbols = 256;
  int[] s = new int[BLOCKSIZE * 2];
   // text block repeated twice for sorting
  int length = 0;  // length of block
  int[] v = new int[BLOCKSIZE]; // vector for suffix sorting
  int[] L = new int[BLOCKSIZE];
   // the Burrows-Wheeler transform
  int I;  // the position of text after suffix sort
  int[] A = new int[numberOfSymbols];
```

Reading a Block

```
void readBlock(){
  byte[] buffer = new byte[BLOCKSIZE];
  try{
    length = System.in.read(buffer);
  }catch(IOException e){
     System.err.println(e.getMessage());
     System.exit(1);
  }
  if (length <= 0) return;
  for (int i = 0; i < length; i++){
      int c = buffer[i];
      if (c < 0) c += 256;
      s[i] = s[length + i] = c;
   }
}
```

Suffix Sorting



s

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
s	w	i	s	s		m	i	s	s	s	w	i	s	s		m	i	s	s

l = 8

v

0	1	2	3	4	5	6	7	8	9
5	2	7	6	4	3	8	9	0	1

L

s	w	m		s	i	i	s	s	s

Suffix and SuffixSort

```
class Suffix implements Comparable{
  int position;
  public Suffix(int p){ position = p; }
  public int compareTo(Object obj){
    Suffix o = (Suffix)obj;
    int k = 0;
    while (k < length && s[position + k] == s[o.position + k])
       k++;
    if (k == length) return position - o.position;
    else return s[position + k] - s[o.position + k];
  }
 }

  void suffixSort(){
    TreeSet<Suffix> tset = new TreeSet<Suffix>();
    for (int i = 0; i < length; i++) tset.add(new Suffix(i));
    int j = 0;
    for (Suffix o: tset) v[j++] = o.position;
  }
```

Wheeler Transform

```
void wheeler(){
    L = new int[length];
    for (int i = 0; i < length; i++)
       if (v[i] == 0){
          I=i;   // position of text
          L[i] = s[length - 1];   // The last character
       }
       else L[i] = s[v[i] - 1];
    System.out.write(I / 256); System.out.write(I % 256);
 }
```

Move-To-Front

```
  void initializeA(){
     for (int i = 0; i < numberOfSymbols; i++) A[i] = i;
  }




public void moveToFront(){
   int i,j,k;
   for (i = 0; i < length; i++){
      int t = L[i];
      for (j = 0; t != A[j]; j++);
      System.out.write(j);        // j is the position of L[i]
      for (k = j; k > 0; k--)
         A[k] = A[k-1];        // move L[i] to front
      A[0] = t;

   }
}
```

Encoding the Whole File

```
void encode(){
   initializeA();
   while (true){
     readBlock();
     if (length <= 0) break;
     suffixSort();
     wheeler();
     moveToFront();
     if (length < BLOCKSIZE) break;
   }
   System.out.flush();
}


public static void main(String[] args){
   H2A h2 = new H2A();
   h2.encode();
 }
```

H2B.java: Decoder

```
class H2B{
  static int BLOCKSIZE = 16384;
  static int numberOfSymbols = 256;
  int length = 0;  // length of block
  int[] A = new int[numberOfSymbols];
  int[] L = new int[BLOCKSIZE];
   // the Burrows-Wheeler transform
  int[] F = new int[BLOCKSIZE];
  int[] T = new int[BLOCKSIZE];
  int I;  // the position of text after suffix sort

 void initializeA(){
    for (int i = 0; i < numberOfSymbols; i++) A[i] = i;
 }
```

```java
void readBlock(){
  byte[] buffer = new byte[BLOCKSIZE + 2];
  try{
    length = System.in.read(buffer) - 2;
  }catch(IOException e){
     System.err.println(e.getMessage());
     System.exit(1);
  }
  if (length <= 0) return;
  int i1 = buffer[0]; if (i1 < 0) i1 += 256;
  int i0 = buffer[1]; if (i0 < 0) i0 += 256;
  I = i1 * 256 + i0;
  for (int i = 0; i < length; i++){
     int j = buffer[i + 2];
     if (j < 0) j += 256;
     int t = A[j];
     L[i] = t;
     for (int k = j; k > 0; k--) A[k] = A[k-1];
     A[0] = t; // move to front
  }
}
```

Inverse Burrows-Wheeler

```
void deBW(){
   for (int i = 0; i < length; i++){
     int j = i - 1; for (; j >= 0; j--)
       if (L[i] < F[j]) F[j + 1] = F[j];
       else break;
     F[j + 1] = L[i];
   }
   int j = 0;
   for (int i = 0; i < length; i++){
    if (i > 0 && F[i] > F[i - 1]) j = 0;
    for (; j < length; j++) if (L[j] == F[i]) break;
    T[i] = j++;
   }
 // Now we have I, L, F, and T
 // Your code here for printing the decoded block
 // using System.out.write().
 }
```

Decoding the Whole File

```
void decode(){
  initializeA();
  while (true){
    readBlock();
    if (length <= 0) return;
    deBW();
    if (length < BLOCKSIZE) return;
  }
}

public static void main(String[] args){
 H2B h2 = new H2B();
 h2.decode();
}
```

Homework 2: due 1-26-15

- Complete the H2B.deBW() function and apply it to test2.bw to restore the original file transformed by H2A and submit the first and last pages of the decoded file.
- Apply Huffman coding H1A to test2.bw and report the compression ratio. Compare this with the compression ratio when H1A is applied to the decoded (original) file.
- Compare the entropy of test2.bw and that of the original file.