Burrows-Wheeler Compression

[Problem Definition 2](#_Toc405748917)

[Steps in Detail 2](#_Toc405748918)

[[1] Burrows-Wheeler 2](#_Toc405748919)

[a. Transform 2](#_Toc405748920)

[b. Inverse 3](#_Toc405748921)

[[2] Move-to-front 5](#_Toc405748922)

[a. Encoding 5](#_Toc405748923)

[a. Decoding 6](#_Toc405748924)

[[3] Huffman Coding 6](#_Toc405748925)

[Project Requirements 6](#_Toc405748926)

[Input 6](#_Toc405748927)

[Output 6](#_Toc405748928)

[How to calculate execution time? 6](#_Toc405748929)

[Test Cases 7](#_Toc405748930)

[Deliverables 7](#_Toc405748931)

[Implementation (60%) 7](#_Toc405748932)

[Document (40%) 7](#_Toc405748933)

[Allowed Codes 7](#_Toc405748934)

[Milestones 7](#_Toc405748935)

[BONUS TASK 8](#_Toc405748936)

# Problem Definition

Implement the Burrows-Wheeler data compression algorithm based on Huffman Coding. This revolutionary algorithm out-compresses gzip and PKZIP. It consists of three steps, which are applied in succession:

1. *Burrows-Wheeler transform.* Given a normal English text file, transform it into a text file in which series of the same letter occur near each other many times.
2. *Move-to-front encoding.* Given a text file in which series of the same letter occur near each other many times, convert it into a stream of integers in which small integers appear more frequently than large ones.
3. *Huffman encoding.* Given the stream of integers in which small integers appear more frequently than large ones, compress it by encoding common integers with short codewords and rare integers with long codewords.

The only step that compresses the message is the final step. It is particularly effective because the first two steps result in a message in which certain symbols (integers) appear more frequently than others. To decode a message, apply the inverse operations in reverse order: first apply the Huffman decoding, then the move-to-front decoding, and finally the inverse Burrows-Wheeler transform.

# Steps in Detail

## [1] Burrows-Wheeler

### a. Transform

The goal of this step is not to compress a message, but rather to transform it into a form that is more suitable to compression. The transform rearranges the characters in the input so that that there are lots of clusters with repeated characters, but in such a way that it is still possible to recover the original input. It consists of three steps:

1. Generate the N suffixes of the input string, assuming the input string is cyclic
2. Sort these N suffixes in ascending order
3. The output string **t[]** is the last column in the suffix sorted list, preceded by the row number where the original string ends up

#### Example:

Here is how it works for the text message "**abfacadabfa**" of length 11.

1. The 11 original suffixes are
   1. abfacadabfa
   2. bfacadabfaa
   3. facadabfaab

…

1. aabfacadabfa

They appear in rows 0 through 10 of the table below.

1. Sorting these 11 strings yields the sorted suffixes, shown in table below. Ignore the "next" array for now - you will only need it for decoding.

i Original Suffixes Sorted Suffixes **t[]** next

-- --------------------- ---------------------- ----

0 **a b f a c a d a b f a** a a b f a c a d a b **f** 2

1 b f a c a d a b f a a a b f a a b f a c a **d** 5

**\*2** f a c a d a b f a a b **a** **b f a c a d a b f a** 6

3 a c a d a b f a a b f a c a d a b f a a b **f** 7

4 c a d a b f a a b f a a d a b f a a b f a **c** 8

5 a d a b f a a b f a c b f a a b f a c a d **a** 9

6 d a b f a a b f a c a b f a c a d a b f a **a** 10

7 a b f a a b f a c a d c a d a b f a a b f **a** 4

8 b f a a b f a c a d a d a b f a a b f a c **a** 1

9 f a a b f a c a d a b f a a b f a c a d a **b** 0

10 a a b f a c a d a b f f a c a d a b f a a **b** 3

1. The Burrows Wheeler transform **t[]**is the last column in the "Sorted Suffixes" list, preceded by the row number where the original string **abfacadabfa** ends up:

Row number: **2**

Output string (**t[]**): **fdafcaaaabb**

Notice how there are 4 a's in a row and 2 consecutive b's - this makes the file easier to compress by the second step.

### b. Inverse

Now we describe how to undo the Burrows-Wheeler transform and recover the original message. If the original suffix number ***j*** appears in row number ***i*** in the sorted order, then **next[i]** records the row in the sorted order where the next original suffix (number ***j+1***) appears.

#### Example:

The original suffix number 0 in the above example: **abfacadabfa** appears in row number 2 of the "Sorted Suffixes". The next original suffix (number 1) **bfacadabfaa** appears in row number 6 of the "Sorted Suffixes". So, **next[2] = 6**.

Knowing the array **next[]** makes decoding easy, as with the following pseudo-code:

**In the above Example:**

N = 11

next[] = [2, 5, 6, 7, 8, 9, 10, 4, 1, 0, 3]

t[] = [f, d, a, f, c, a, a, a, a, b, b]

**Pseudo-code to decode:**

index = 2

for i = 0 to N-1

index = next[index]

print t[index]

endfor

Amazingly, there's no need to store the **next[]** array, the information contained in the Burrows-Wheeler transform (**t[]**) is enough to reconstruct it, and hence the original message! First, we already know all of the characters in the original message from (**t[]**), even if they're permuted in the wrong order. This enables us to reconstruct the first column in the "Sorted Suffixes" by sorting the characters, as shown in the table below. Since **c** only occurs once in the message (at location 7 in the 1st column and location 2 in the last column) and the suffixes are formed using cyclic wrap-around, we can deduce that **next[7] = 4**. Similarly, **d** only occurs once (at location 8 in the 1st column and location 1 in the last column), so we can deduce that **next[8] = 1**.

i Sorted Suffixes t next

-- --------------------- ----

0 a ? ? ? ? ? ? ? ? ? **f**

1 a ? ? ? ? ? ? ? ? ? **d**

**\*2** a ? ? ? ? ? ? ? ? ? a

3 a ? ? ? ? ? ? ? ? ? **f**

4 a ? ? ? ? ? ? ? ? ? **c**

5 b ? ? ? ? ? ? ? ? ? a

6 b ? ? ? ? ? ? ? ? ? a

7 **c** ? ? ? ? ? ? ? ? ? a 4

8 **d** ? ? ? ? ? ? ? ? ? a 1

9 **f** ? ? ? ? ? ? ? ? ? b

10 **f** ? ? ? ? ? ? ? ? ? b

However, since **f** appears twice, it may seem ambiguous whether next[9] = 0 and next[10] = 3, or next[9] = 3 and next[10] = 0. Here's the key rule that resolves the ambiguity:

*If sorted row i and j both start with the same letter and i < j, then next[i] < next[j].*

This rule implies **next[9] = 0** and **next[10] = 3**. But why is this rule valid? The rows are sorted, so row 9 is alphabetically less than row 10. Thus the nine unknown characters in row 9 must be less than the nine unknown characters in row 10 (since both start with **f**). We also know that between the two rows that end with **f**, row 0 is less than row 3. But, the nine unknown characters in row 9 and 10 are precisely the first nine characters in rows 0 and 3. Thus, next[9] = 0 and next[10] = 3 or this would contradict the fact that the suffixes were sorted.

## [2] Move-to-front

### a. Encoding

It consists of three steps:

1. Maintain an ordered sequence of legal symbols,
2. Read in the next symbol from the input message and print out the position in which that symbol appears,
3. Move that symbol to the front of the list.

Repeat from 2 until finishing the input message.

#### Example:

If the initial ordering over a 6 symbol alphabet is **a b c d e f**, and we want to encode the input **fdafcaaaabb**, then we would update the move-to-front lists as follows:

move-to-front in out

------------- --- ---

**a b c d e f** f 5

f a b c d e d 4

d f a b c e a 2

a d f b c e f 2

f a d b c e c 4

c f a d b e a 2

a c f d b e a 0

a c f d b e a 0

a c f d b e a 0

a c b d e f b 2

b a c d e f b 0

b a c d e f

If the same letters occurs next to each other many times in the input, then many of the output values will be small integers like 0, 1, and 2. The extremely high frequency of certain symbols makes an ideal scenario for Huffman coding.

Your task is to maintain an ordered list of the 256 extended ASCII symbols. Initialize the list by making the ***ith*** symbol equal to the ***ith*** extended ASCII symbol. Now, read in each character from standard input one at a time, output the index in the array where this character appears, and move it to the front of the list.

As an example, the move-to-front encoding of

a b b b a a b b b b a c c a b b a a a b c

is given by the following sequence of integers:

97 98 0 0 1 0 1 0 0 0 1 99 0 1 2 0 1 0 0 1 2

Note that 'a' is 97 in ASCII and that we have printed out the indices as type int with separating whitespace (for displaying only) rather than as type byte without separating whitespace (for your actual code).

### a. Decoding

It consists of three steps:

1. Initialize an ordered list of 256 characters, where extended ASCII character ***i*** appears ***ith*** in the list,
2. Read in each integer ***i*** from the input numbers, one at a time, and print the ***ith*** character in the list,
3. Move that character to the front of the list.

Repeat from 2 until finishing the input numbers.

## [3] Huffman Coding

Refer to [this presentation](Huffman%20%5bfrom%20Lecture7%5d.pptx), from Lecture7, for encoding steps, decoding steps and examples.

# Project Requirements

1. Implement the Burrows-Wheeler transform and its inverse using suitable sorting algorithm.
2. Implement the Move-to-front encoding and decoding.
3. Implement the Huffman encoding and decoding.
4. Use the three steps to compress and decompress text files.

## Input

* Text file to be compressed

## Output

1. Compressed file
2. Decompressed file
3. Compression ratio
4. Execution time of compression & decompression

### How to calculate execution time?

To calculate time of certain piece of code:

1. Get the system time before the code
2. Get the system time after the code
3. Subtract both of them to get the time of your code

To get system time in milliseconds using C#, you can use System.Environment.TickCount

## Test Cases

* [Sample Cases:](Test%20Files/Sample%20Cases) to debug on it
  1. Lecture7 example
  2. Above example
  3. Other simple examples
* [Large Cases:](Test%20Files/Large%20Cases) for final testing
  1. Small sizes (~100 KB)
  2. Medium sizes (~5 MB)
  3. Large sizes (~30 MB)

# Deliverables

## Implementation (60%)

1. Burrows-Wheeler transform and its inverse using suitable sorting algorithm
2. Move-to-front encoding and decoding
3. Huffman encoding and decoding
4. Use the three steps to compress and decompress text files.

## Document (40%)

1. Source code of the first two steps ONLY in both the compression and the decompression
2. Detailed analysis of it
3. Compression ratio of "Large Cases"
4. Execution time of the compression and the decompression for "Large Cases"

## Allowed Codes

1. Open-source code for sorting. You should understand and analyze it!
2. Open-source code for priority queue. You should understand and analyze it!

# Milestones

|  |  |  |
| --- | --- | --- |
|  | **Deliverables** | **Due to** |
| **Milestone1** | * Burrows-Wheeler transform and its inverse using suitable sorting algorithm | Week10: start at SAT 13-12-2014  During LABS and/or Office Hours |
| **Milestone2** | 1. Move-to-front encoding and decoding. 2. Huffman encoding. | Week11: start at SAT 20-12-2014  During LABS and/or Office Hours |
| **Milestone3** | 1. Complete project (compress & decompress) 2. documentation | Final Delivery |
| **For Milestone1 & 2:**   * + deliver your current status **even** it does not work correctly   + each group should deliver in the section time of the majority members (e.g. if a group consists of 3 members from sec.1 and two members from sect.2, they should deliver in the time of sec.1) | | |

# BONUS TASK

1. Efficient implementation for the 1st step (Burrows-Wheeler transform) and/or its inverse. Compare its execution time with the original one.
2. Efficient implementation for the 2nd step (Move-to-front). Compare its execution time with the original one.
3. Efficient implementation for Huffman Coding using heap-based priority queue