**Burrows Wheeler**

**Problem Statement:**

The Burrows–Wheeler transform (BWT, also called block-sorting compression) rearranges a character string into runs of similar characters. This is useful for compression, since it tends to be easy to compress a string that has runs of repeated characters by techniques such as move-to-front transform and run-length encoding. More importantly, the transformation is reversible,

without needing to store any additional data except the position of the first original character.

**The following three algorithms were implemented:**

* **Burrows–Wheeler transform :** Given a typical English text file, transform it into a text file in which sequences of the same character occur near each other many times.
* **Move-to-front encoding :** Given a text file in which sequences of the same character occur near each other many times, convert it into a text file in which certain characters appear much more frequently than others.
* **Huffman compression :** Given a text file in which certain characters appear much more frequently than others, compress it by encoding frequently occurring characters with short codewords and infrequently occurring characters with long codewords.

**The following problems were solved:**

* Encode a given string by using MoveToFront method and decode the original string from the encoded pattern.
* Find the Circular Suffix Array for the given string.
* Implement Burrows-Wheeler Transform on the given string and obtain the original string by using Inverse Burrows-Wheeler Transform method.

**Related Concepts:**

* Programming Language - Java
* HashTables
* Suffix Arrays
* Circular Suffix Arrays

**API**

For all the methods below used stdIn for reading and stdOut for the output.

**Move to Front Encoding:**

The main idea is that each symbol in the data is replaced by its index in the stack of “recently used symbols”. Your task is to maintain an ordered sequence of the 256 extended ASCII characters. Initialize the sequence by making the ith character in the sequence equal to the ith extended ASCII character. Now, read each 8-bit character c from standard input, one at a time; output the 8-bit index in the sequence where c appears; and move c to the front.

**public class MoveToFront:**

**{**

**public static void encode():**

**Time Complexity : proportional nR**

**Space Complexity : proportional to n + R**

**public static void decode():**

**Time Complexity : proportional to n + R**

**Space Complexity : proportional to n + R**

**public static void main(String[] args)**

if args[0] is "-", apply move-to-front encoding

if args[0] is "+", apply move-to-front decoding

**}**

**Circular Suffix Array:**

The circular suffix array, describes the abstraction of a sorted array of the n circular suffixes of a string of length n.

**public class CircularSuffixArray {**

**public CircularSuffixArray(String s):**

circular suffix array of s

**Time Complexity : proportional to n^2**

**Space Complexity : proportional to n + R**

**public int length():**

length of s

**Time Complexity : O(1)**

**Space Complexity : O(1)**

**public int index(int i):**

returns index of ith sorted suffix

**Time Complexity : O(1)**

**Space Complexity : O(1)**

**public static void main(String[] args) }**

**Burrows Wheeler Transform:**

The goal of the Burrows–Wheeler transform is not to compress a message, but rather to transform it into a form that is more amenable for compression. The Burrows–Wheeler transform rearranges the characters in the input so 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 relies on the following intuition:

if you see the letters hen in English text, then, most of the time, the letter preceding it is either t or w. If you could somehow group all such preceding letters together

(mostly t’s and some w’s), then you would have a propitious opportunity for data compression.

**public class BurrowsWheeler {**

**public static void transform()**

Burrows-Wheeler transform was applied

**Time Complexity : proportional to n + R**

**Space Complexity : proportional to n + R**

**public static void inverseTransform()**

Burrows-Wheeler inverse transform was applied

**Time Complexity : proportional to n + R**

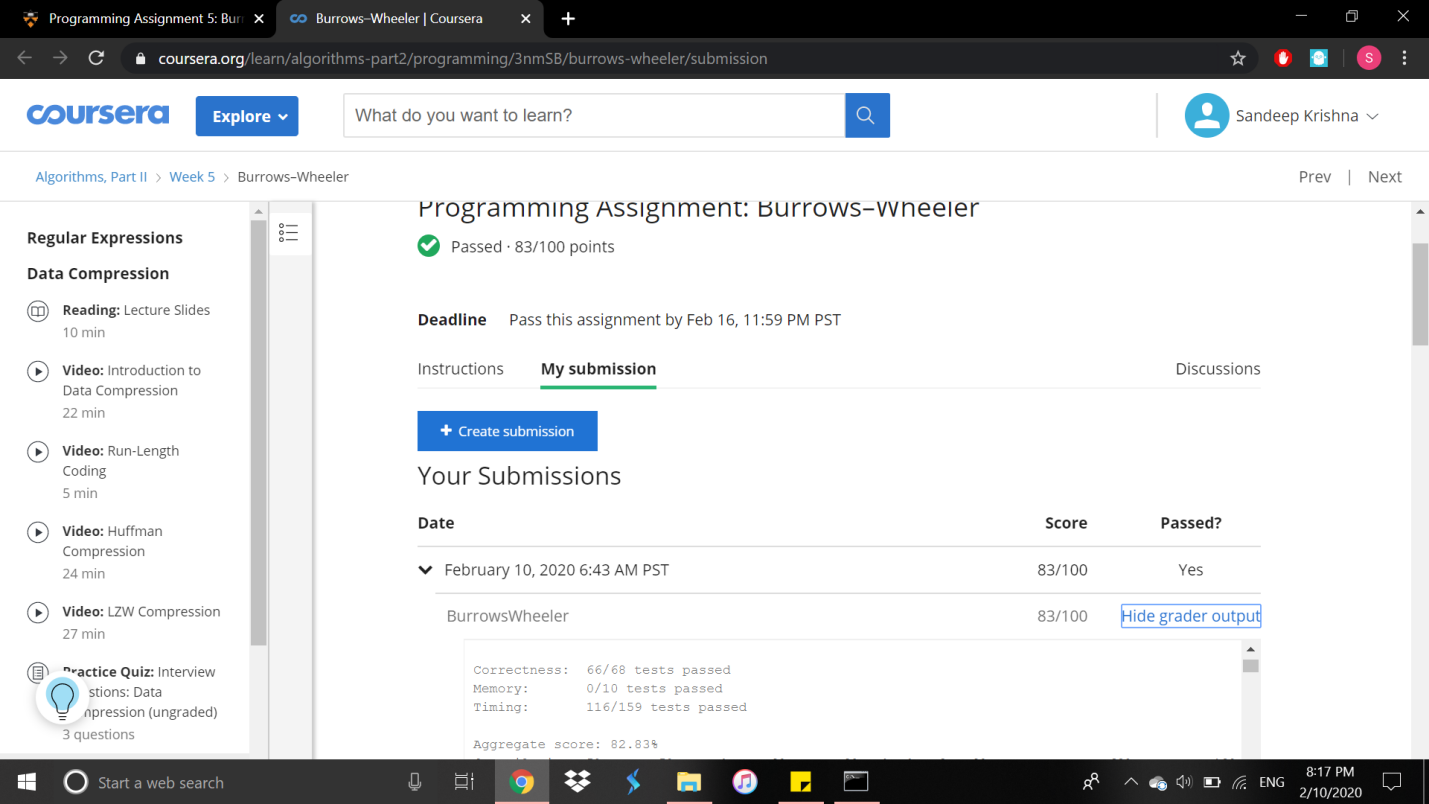
**Space Complexity : proportional to n + R**

**public static void main(String[] args)**

if args[0] is "-", apply Burrows-Wheeler transform

if args[0] is "+", apply Burrows-Wheeler inverse transform

**}**



**2019501035**

**Sandeep Krishna.**