Writeup and Comparison

1. Writeup a summary of all the files that you included. This includes a small description for each file.

➔ File : “index_module.py”, the file to import as a module, then can be used to call the following functions from their own file:
- o **def create_index(input_file, output_path, sorted)**
- o **def compress_index(bitmap_index, output_path, compression_method, word_size)**

➔ File : "create_index.py" includes the following functions:

def create_index(input_file, output_path, sorted)
· *expecting* *'input_file'* as the name of the file that will be use *'output_path'* as the path to output bitmap_index *'sorted'* as a boolean determines whether to sort the data or not · open the 'input_file' to readlines() store in 'data' · if sorted = True, sort the data and create a bitmap_index over the sorted data · else, create a bitmap_index of data from input_file

➔ File : "compress_index.py" will compress the given bitmap index using WAH and BBC compression. It includes functions as:

def compress_index(bitmap_index, output_path, compression_method, word_size)
· with the given 'output_path' it will create a name with the specified naming convention · try to open and read the 'bitmap_index', open and write to output file, if error occurs then exit · if successfully read the 'bitmap_index' and create the output file, read content from 'bitmap_index' store as list named 'data' · using < class: WAH > or < class: BBC> in the same file, compress each item in 'data' and write out to output_file

class WAH	**compress given 'data' and 'word_size' using WAH compression**
· there are several private method within the class, but the main method to considered are only: _ def __init__() which will store 'data' and 'word_size' _ def compress() which will call appropriate method and compress self.data into WAH compression. It also returns an iterable list which each item in the list is a compressed WAH bit string per column	
Class BBC	**compress given 'data' using BBC compression**
· main methods to considered are only: _ def __init__() to store number of run, literal and list of literal _ def compress(data) which will call appropriate method and compress data into BBC compression. It also returns an iterable list which each item in the list is a compressed BBC bit string per column	

2. Then you must compare the size of the bitmap indexes and compressed versions on the large test file. Write an analysis on why you think they are different size. Did sorting help with the compression and by how much? Did different word sizes have different compression ratios and why do you think that is? In addition to your analysis, include the number of fill words and literal words that were compressed for each file.

Bitmap Index	
Sorted / Unsorted	**File Size**
unsorted animals.txt	1,661 KB
sorted animals.txt	1,661 KB

WAH Compression				
Sorted / Unsorted	**Word Size**	**File Size**	**Number of Run**	**Number of Literal**
unsorted animals.txt	8	1,522 KB	76429	152131
	16	1,623 KB	14025	92631
	32	1,612 KB	1271	50329
	64	1,590 KB	26	25366
sorted animals.txt	8	51 KB	226996	1564
	16	56 KB	104962	1694
	32	113 KB	49838	1762
	64	222 KB	23604	1788

- ➔ As shown in tables above, compressing on unsorted animals.txt does not make any different. However, when compressing on sorted animals.txt, the size has cut down dramatically. Since the sorted animals.txt has its data in a sorted order, when compressing over column, it will guarantee to have more consecutive run than the unsorted file. In addition, when it has several run bit strings, WAH compression allows us to store as a single bit string to indicate how many run we have. This helps to reduce the size of the final bit string of that column.

- ➔ When compression over a sorted file using 64 word size, it reduces the file size down to about 1/7 of the original file size. But, if we use 8 word size, it reduces the size down to about 1/32 of the original file size.

- ➔ Different word size does have different compression ratios because when using smaller word size, chances of getting the same run of the bit string will be higher, and we can store those consecutive runs in one bit string only.