PART I

1. Exercise 7.2 out of the book, but where k=4 and postings are all 2 bytes

Solution: To prove that the total number of bytes transferred from/to disk is less than $5 \times s$, where s is the list's size in bytes, we can analyze the Inplace strategy with proportional pre-allocation.

In the Inplace strategy with proportional pre-allocation, when a block of postings is fetched from disk, space is pre-allocated for the postings in the in-memory buffer. The pre-allocation factor is denoted by k, and in this case, $k=4$.

Let's consider the worst-case scenario where every posting is a new term that needs to be loaded into the buffer. In this case, the number of postings loaded into the buffer will be k times the number of postings that can fit in a block.

Let b be the number of postings that fit in one block. Then, the number of postings loaded into the buffer is $k \times b$.

Now, let s be the total size of the list in bytes, and n be the total number of postings in the list. If every posting has a constant size of 2 bytes, then $s=2 \times n$.

In the worst case, $k \times b$ postings are loaded into the buffer for every disk block. The total number of disk blocks needed is given by the ceiling of n/b, denoted as $\lceil n/b \rceil$.

Therefore, the total number of postings loaded into the buffer is: $k \times b \times \lceil n/b \rceil$

Now, we know that $s=2 \times n$, and the worst-case number of postings loaded into the buffer is $k \times b \times \lceil n/b \rceil$. Therefore, the total number of bytes transferred from/to disk is:
$2 \times k \times b \times \lceil n/b \rceil$

Now, let's substitute
b with $s/(2 \times n)$ (since $s=2 \times n$):
$2 \times k \times (s/(2xn)) \times \lceil n/(s/(2xn)) \rceil$

Simplify the expression:
$k \times (s/n) \times \lceil 2 \rceil$

Since $\lceil 2 \rceil = 2$:
$2 \times k \times (s/n)$

Substitue k = 4:
8 x (s/n)

Now, substitute s = 2 x n:
8 x ((2xn)/n)

Simplify the expression:
16

So, the total number of bytes transferred from/to disk is
16×n. Since s=2×n, the total number of bytes transferred is 8×s.

Therefore, we have shown that the total number of bytes transferred from/to disk is less than 5 x s.

2. Come up with pseudo-code for maybe also firstDoc, lastDoc, nextDoc, prevDoc which works with an index that is being stored using Logarithmic merging

Solution:

```
 firstDoc( term):
  if term in self.inverted_index:
      return min(self.inverted_index[term])
  else:
      return None

lastDoc(term):
  if term in self.inverted_index:
      return max(self.inverted_index[term])
  else:
      return None

nextDoc( term, current_doc):
  if term in self.inverted_index:
      doc_ids = self.inverted_index[term]
      doc_ids.sort()
      for doc_id in doc_ids:
          if doc_id > current_doc:
              return doc_id
      return None
  else:
```

```
        return None

    prevDoc(term, current_doc):
        if term in self.inverted_index:
            doc_ids = self.inverted_index[term]
            doc_ids.sort(reverse=True)
            for doc_id in doc_ids:
                if doc_id < current_doc:
                    return doc_id
            return None
        else:
            return None
```

We accumulate postings in an in-memory auxiliary index, and when the limit is reached, we transfer the postings to a new index created on disk. The next time the auxiliary index is full, we merge it with the existing index to create a larger index.

3. In a plain text ASCII document, define a paragraph to be any block of text preceded by a start of document or two blanks line and continuing until an end of document or two blank lines. Consider the punctuation symbols period, comma, and semi-colon. Write a map reduce algorithm to compute the average number of occurrences of these symbols in a paragraph for a corpus of plain text documents.

Solution:

Given a corpus of plain text documents, we want to calculate the average occurrences of the specified punctuation symbols within each paragraph.

Approach:

1. Map Phase:
    - For each document in the corpus:
        - Split the document into paragraphs based on the start of document or two blank lines.
        - For each paragraph:
            - Tokenize the paragraph into words.
            - Count the occurrences of the specified punctuation symbols (period, comma, and semi-colon).
            - Emit key-value pairs: (paragraph_id, (punctuation_count, 1)).

2. Shuffle and Sort:
    - Group the key-value pairs by paragraph ID.
3. Reduce Phase:
    - For each paragraph ID:
        - Sum up the punctuation counts and the number of paragraphs.
        - Calculate the average punctuation count per paragraph.
        - Emit the result: (paragraph_id, average_punctuation_count).

Pseudo-Code (MapReduce Algorithm):

```
# Mapper
def map(doc_id, document):
    paragraphs = document.split("\n\n")  # Split by two blank lines
    for para_id, paragraph in enumerate(paragraphs):
        words = tokenize(paragraph)  # Tokenize into words
        punctuation_count = count_punctuation(words)
        yield para_id, (punctuation_count, 1)

# Reducer
def reduce(para_id, counts):
    total_punctuation, num_paragraphs = 0, 0
    for count, num in counts:
        total_punctuation += count
        num_paragraphs += num
    average_punctuation = total_punctuation / num_paragraphs
    yield para_id, average_punctuation

# Example usage
corpus = [...]  # List of plain text documents
result = {}
for doc_id, document in enumerate(corpus):
    for para_id, avg_punctuation in map(doc_id, document):
        result[para_id] = avg_punctuation

# Print the average punctuation counts for each paragraph
for para_id, avg_punctuation in sorted(result.items()):
    print(f"Paragraph {para_id}: Average Punctuation = {avg_punctuation:.2f}")
```

4. For a corpus of plain text documents, write a map reduce algorithm which computes for all terms t the conditional probability of t occurring in a paragraph given that the paragraph has a term 'calvacade' in it. i.e., p(t in paragraph |'calvacade' in paragraph).

Solution:

1. Map Phase:
   - Read each document (paragraph) from the input corpus.
   - Tokenize the document into words.
   - For each word **w** in the document:
     - Emit a key-value pair: **(w, 1)** if **w ≠ 'calvacade'**.
     - Emit a special key-value pair: **('calvacade', w)** if **w ≠ 'calvacade'**.
     - This step ensures that we capture the co-occurrence of **'calvacade'** with other terms.
2. Shuffle and Sort Phase:
   - Group the emitted key-value pairs by key (term).
3. Reduce Phase:
   - For each term **t**:
     - Initialize counters: **count_t** (total occurrences of **t**) and **count_t_given_calvacade** (occurrences of **t** when **'calvacade'** is present).
     - For each value **v** associated with **t**:
       - If **v = 1**, increment **count_t**.
       - If **v ≠ 1**, increment **count_t_given_calvacade**.
     - Calculate the conditional probability:
       - **p(t in paragraph | 'calvacade' in paragraph) = count_t_given_calvacade / count_t**
4. Output Phase:
   - Emit the term **t** along with its conditional probability.

Here's a simplified Python pseudo-code for the MapReduce algorithm:

```
# Mapper
def mapper(document):
    for word in tokenize(document):
        if word != 'calvacade':
            emit(word, 1)
        else:
            emit('calvacade', word)

# Reducer
```

```
def reducer(term, values):
    count_t = 0
    count_t_given_calvacade = 0
    for value in values:
        if value == 1:
            count_t += 1
        else:
            count_t_given_calvacade += 1
    if count_t > 0:
        conditional_probability = count_t_given_calvacade / count_t
        emit(term, conditional_probability)

# Driver code
for document in input_corpus:
    mapper_output = mapper(document)
    shuffled_data = shuffle_and_sort(mapper_output)
    reducer_output = reducer(shuffled_data)
    output(reducer_output)
```