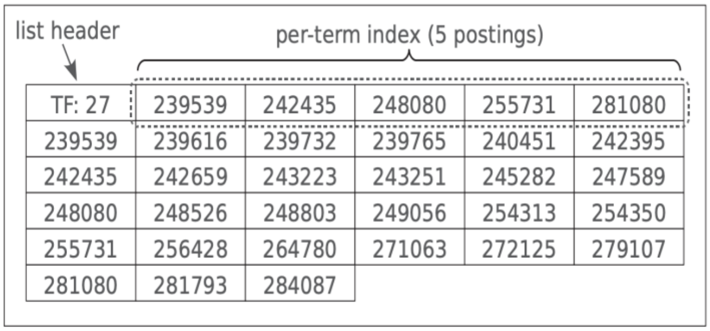
## Exercise 4.1

In Section 4.3 we introduced the concept of the per-term index as a means to improve the index's random access performance. Suppose the postings list for some term consists of 48 million postings, each of which consumes 8 bytes. In order to carry out a single random access into the term's postings list, the search engine needs to perform two disk read operations: 1. Loading the per-term index (list of synchronization points) into RAM. 2. Loading a block B of postings into RAM, where B is identified by means of binary search on the list of synchronization points. Let us call the number of postings per synchronization point the granularity of the per-term index. For the above access pattern, what is the optimal granularity (i.e., the one that minimizes disk I/O)? What is the total number of bytes read from disk?

Considering that the:

- Postings list for some term consists term 't': 48 million
- Each posting occupies/ consumes space of: 8 bytes


list header
per-term index (5 postings)

| | | | | | |
|---|---|---|---|---|---|
| TF: 27 | 239539 | 242435 | 248080 | 255731 | 281080 |
| 239539 | 239616 | 239732 | 239765 | 240451 | 242395 |
| 242435 | 242659 | 243223 | 243251 | 245282 | 247589 |
| 248080 | 248526 | 248803 | 249056 | 254313 | 254350 |
| 255731 | 256428 | 264780 | 271063 | 272125 | 279107 |
| 281080 | 281793 | 284087 | | | |

**Figure 4.4** Schema-independent postings list for "denmark" (extracted from the Shakespeare collection) with per-term index: one synchronization point for every six postings. The number of synchronization points is implicit from the length of the list: $\lceil 27/6 \rceil = 5$.

From the figure above we know that:

$$number\ of\ synchronization\ points = \frac{length\ of\ postings\ list}{per\_term\ index\ granularity}$$

*According to the textbook:*

Choosing the granularity of the per-term index, that is, the number of postings between two synchronization points, represents a trade-off. A greater granularity increases the amount of data between two synchronization points that need to be loaded into memory for every random access operation; a smaller granularity, conversely, increases the size of the per-term index and thus the amount of data read from disk when initializing the postings list.

In theory it is conceivable that, for a very long postings list containing billions of entries, the optimal per-term index (with a granularity that minimizes the total amount of disk activity) becomes so large that it is no longer feasible to load it completely into memory. In such a situation it is possible to build an index for the per-term index, or even to apply the whole procedure recursively. In the end this leads to a multi-level static B-tree that provides efficient random access into the postings list. In practice, however, such a complicated data structure is rarely necessary. A simple two-level structure, with a single per-term index for each on-disk postings list, is sufficient.

To compute the optimal granularity which minimizes the disk I/O for 48 million postings; the size of each per term index (in bytes) must be similar.

Optimal granularity: $\sqrt{48 * 10^6}$ = 6927
Number of postings in block B = 6927
Number of postings for last block = $48 * 10^6$ % 6927 = 2817

Which means we need one synchronization point for every 6927 postings.

Assume the size of per-term index as 'n' and the granularity as 'm'.
Our target is to minimize 'm + n'. To achieve this; we must find a value 'x' that minimizes the function and we take the derivative of this function as:

$$y = f(x) = \frac{x + (48000000 - x)}{(x + 1)} = \frac{(x^2 + 48000000)}{(x + 1)}$$

The minimum value for 'x' happens when y` = 0

$$x^2 + 2x - 48000000 = 0$$

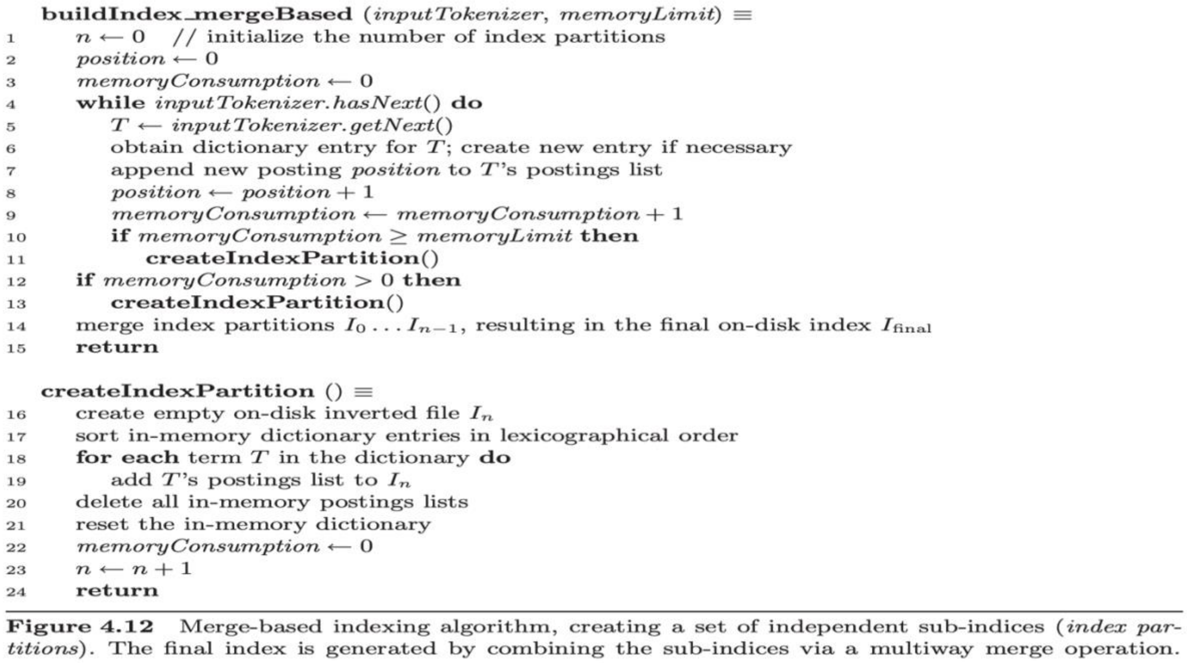
$$x = -2 \pm \frac{\sqrt{4 - 4 * 1 * (-48000000)}}{2} = 6927$$

$$y = \frac{6927 + 47993073}{(6927 + 1)} = 13854$$

Total number of bytes read from disk = 13854 * 8 bytes = 110832 bytes
Total number of bytes read from disk for last block = $(6927 + 2817) * 8$ = 77952 bytes

Exercise 4.4

In the algorithm shown in Fig 4.12 the memory limit is expressed as the number of postings that can be stored in RAM. What is the assumption that justifies this definition of the memory limit? Give an example of a text collection or an application in which the assumption does not hold.



```
buildIndex_mergeBased (inputTokenizer, memoryLimit) ≡
    $n \leftarrow 0$   // initialize the number of index partitions
    $position \leftarrow 0$
    $memoryConsumption \leftarrow 0$
    while inputTokenizer.hasNext() do
        $T \leftarrow$ inputTokenizer.getNext()
        obtain dictionary entry for $T$; create new entry if necessary
        append new posting position to $T$'s postings list
        $position \leftarrow position + 1$
        $memoryConsumption \leftarrow memoryConsumption + 1$
        if $memoryConsumption \geq memoryLimit$ then
            createIndexPartition()
    if $memoryConsumption > 0$ then
        createIndexPartition()
    merge index partitions $I_0 \ldots I_{n-1}$, resulting in the final on-disk index $I_{\text{final}}$
    return

createIndexPartition () ≡
    create empty on-disk inverted file $I_n$
    sort in-memory dictionary entries in lexicographical order
    for each term $T$ in the dictionary do
        add $T$'s postings list to $I_n$
    delete all in-memory postings lists
    reset the in-memory dictionary
    $memoryConsumption \leftarrow 0$
    $n \leftarrow n + 1$
    return
```

**Figure 4.12** Merge-based indexing algorithm, creating a set of independent sub-indices (*index partitions*). The final index is generated by combining the sub-indices via a multiway merge operation.

Let us assume that each posting occupies the same fixed size of memory and let us denote that value using 'm'. Let memory limit be denoted as 'M'.
Then the total number of postings 't' can be calculated as:

$$\frac{\text{memory limit}}{\text{size of memory occupied by each posting}}$$

$$t = \frac{M}{m}$$

For a schema-dependent index the posting list is of the form:

$$(d, f_{t,d}, < p_0, \ldots, p_{f_{t,d}} >)$$

Where 'd' is the document id,

$f_{t,d}$ is the frequency of term 't' in document 'd'

$p_0, \ldots, p_{f_{t,d}}$ is the posting offsets for the term 't' within document 'd'

which has a variable size for the posting list.

In such an index our assumption may not hold; since if there are repeating words in the corpus and the index will have to store all the postings for the term and the size of the posting list will be very large. Considering the algorithm represented above which defines the memory limit as the number of postings that can be stored in RAM and assuming it for an index that has a very large posting list; the RAM memory space may not be sufficient to handle a posting very large if we set a memory limit as expressed in the algorithm.

Exercise 4.5

In Section 4.5.1 we discussed the performance characteristics of various dictionary data structures. We pointed out that hash-based implementations offer better performance for single-term lookups (performed during index construction), while sort-based solutions are more appropriate for multi-term lookups (needed for prefix queries). Design and implement a data structure that offers better single-term lookup performance than a sort-based dictionary and better prefix query performance than a hash-based implementation.

Hash-based implementations have better performance in case of single-term lookups; while sort-based implementations are more efficient for multi-term lookups like for example considering prefix queries.

The data structure we are going to design is a modified hash based method for dictionary construction; which includes prefix queries and has a better performance than hash based dictionary approach. For the original hash based approach which needs to do a linear scan for prefix queries and is thus less efficient than a sort based approach which uses the binary search and is better in terms of time complexity.

Indexing time:

For an example of two terms that occur in sequence; say term 1 is information' and term 2 is 'inform'.
During index construction; we build the index for term 1 'information' first and then when term 2 'inform' is read.
The term 'information' which has the posting offset of 1000; then we will store the prefix's of the term 'information' as keys in the dictionary.

i* -> PL [1000]
in* -> PL [1000]
inf* -> PL [1000]
info* -> PL [1000]
infor* -> PL [1000]
inform* -> PL [1000]
informa* -> PL [1000]
informat* -> PL [1000]
informati* -> PL [1000]
informatio* -> PL [1000]
information* -> PL [1000]

Given another term 'inform' which has the posting offset of 2000; during the index construction time we do:

i* -> PL [1000, 2000]
in* -> PL [1000, 2000]
inf* -> PL [1000, 2000]
info* -> PL [1000, 2000]
infor* -> PL [1000, 2000]

inform* -> PL [1000, 2000]
informa* -> PL [1000]
informat* -> PL [1000]
informati* -> PL [1000]
informatio* -> PL [1000]
information* -> PL [1000]

During index construction; when we read a second term ‘inform’ with the same prefix as an existing term say ‘information’; we append the posting offset of the second term to the posting offset of the first term that already has the same prefix as shown in the example above.

Query time:

Using the above data structure; when we process a prefix query like ‘inform*’ we return the postings as [1000, 2000].

This approach is better than the sort based method for prefix queries in terms of time complexity since this has a time complexity of *O(1*) which is better than *O( log n)*.