Informatics 2 – Introduction to Algorithms and Data Structures (2021-22)

Coursework 1: A search engine for a large text corpus

John Longley

October 22, 2021

In this coursework, we will apply ideas from the course to the construction of a search engine for a potentially large set of text files, such as a corpus of classic literature. Part of the purpose is to illustrate how algorithmic ideas may be useful when processing large amounts of data. This is a credit-bearing coursework assignment worth 10% of the total course mark. It will be marked out of 100.

Submission will be via the Assessment page on the course Learn site. The deadline is **16:00 (GMT) on Friday 12 November**. Your marks and feedback will be returned to you by Friday 3 December.

The coursework consists of three Python programming tasks. Part A is intended to be relatively straightforward and should be attempted first. Parts B and C are more challenging, but are independent of each other and may be attempted in either order.

Many of the more routine parts of the code have already been provided for you, leaving it to you to supply some of the more algorithmically interesting parts. The total volume of code you are expected to write is not large; however, the instructions are quite complex, and it may take you some time to **understand the set-up** and find your way around the existing body of code. You are therefore advised to make an early start on the practical to give yourself time to assimilate all of this.

It is your responsibility to **test** your programs before you submit them, and in particular to ensure that they work correctly in conjunction with the provided code. For Part A, a sample output file is provided for you. For Parts B and C, you will need to devise your own ways to test your code. You may work mostly on your own machine, but before submitting you should do a final check that they run correctly on the `python3.8` installation on the DICE machines. Marking will be done by a combination of automated testing and human eye inspection of your code.

You should add enough **comments** to your code to make it readable. Not only is this a good habit to get into, but it may help your mark by allowing the human marker to see readily that you understand what you are doing. Comments should be short and not too numerous. Put yourself in the shoes of the human marker who is on a tight time budget of 15 minutes per student submission: you're wanting to steer their thoughts in the right direction as quickly as possible, so excessive commenting would be self-defeating.

If you feel you need help on the Python side of the coursework, **lab sessions** will be running at the advertised times, and the demonstrators there may be able to help you. You are also welcome to post queries on the **Piazza forum** – but if you do so, please observe the following important points:

- **STERN WARNING!!!** It is *absolutely forbidden* to post any part of a solution attempt – even a one-line snippet of code – or to share it with other students by any other means. This applies even if you know that your attempt is incorrect. Any breach of this rule will be treated as a very serious offence. Please err on the side of caution in the way you phrase your questions.
- Please select the 'cwk1' folder to tag your question as related to this coursework.
- Before posting your query, please do check through previous postings to see whether your question has already been asked and answered! We will try to answer queries within 48 hours, but please bear in mind that this may not always be possible. You are also encouraged to help one another by replying to queries if you know the answer – again as long as you don't give away any part of the solution.

Overview

The three tasks of the practical are:

A: Compiling an index for a large collection of documents using a file-based version of MergeSort. If you have already worked through the Python lab sheets, this part should be relatively straightforward and familiar. In view of this, Part A will be marked quite strictly using an automarker, so it is essential that your code yields the correct output.

B: Reducing the size of the in-memory component of the index as far as possible, using an ingenious *perfect hashing* scheme due to Belazzougui, Botelho and Dietzfelbinger (2008). This will give you experience of a relatively recent development in the field which is nonetheless fairly easy to implement.

C: Using such an index to process search queries. We shall focus on searching for places in the corpus where *at least two* of some specified list of words (e.g. *friends*, *romans*, *countrymen*) appear within a specified number of consecutive lines (e.g. within a 5-line window). You will be given a simple but inefficient implementation, and your task is to re-implement it more efficiently using heap-based priority queues.

For each of the three parts, you will be inserting code into one of the *template files* provided. Please add your code at the point marked `# TODO`,[1] and **do not** make changes to any other part of the code: it is essential that your code works correctly in conjunction with the provided code precisely as given. For each part, it is quite possible to give a natural solution within 40 lines of code. You need not worry if your solution is a little longer (or shorter) than this – but if you find yourself writing twice this much, you should stop to consider whether your approach is needlessly complicated.

Part of the idea of this practical is to illustrate how algorithms such as MergeSort can help us when the data involved becomes too large to fit in main memory, and has to be stored e.g. on disk. Nowadays, of course, even a modest laptop is likely to have 16 gigabytes or so of memory, so working with a dataset that *really* didn't fit into memory would require you to download some large and unwieldy files, and would also entail excessive amounts of computation time. For the purpose of illustration, therefore, we shall investigate how a relatively small amount of working memory (around 1 megabyte) can be used to process a dataset many times larger. You can imagine how this models on a small scale some issues that arise with much larger datasets.

[1] This `# TODO` comment may be recognized by certain IDEs, which may be useful.

To get started, download the file `IADS_cwk1_materials.zip` from the course Learn page, and unzip it. On the DICE machines, this can be done using the command:

```
unzip IADS_cwk1_materials.zip
```

This will yield a directory containing four Python source files (three of which are incomplete templates), five plaintext files containing literary works from various cultures and historical periods,[2] one sample output file, and two pdf files (one being this document).

Part A: Building an index [30 marks]

Open the template file `index_build.py` in a text editor. The following instructions will walk you through some parts of the existing code, giving you what you need to know in order to create the sorted index.

Import the contents of this file into Python, e.g. using

```
import index_build
from index_build import *
```

Look at the declaration of `CorpusFiles` near the start of the file. This creates a Python dictionary for the text files, with a three-letter code for each file serving as the key. The entries for the two largest files (Shakespeare and Tolstoy) are commented out. Whilst developing your code, we suggest that you use just the three smaller files to keep things lightweight – once it is working, you are encouraged to add in the weightier offerings.[3]

Take a quick look at the contents of the 'Alice in Wonderland' file to see what it's like.

Now move down the Python file to the declaration of `generateAllIndexEntries`. You can try this out using:

```
generateAllIndexEntries('raw_entries')
```

Looking in your directory, you'll see that this has created a text file `raw_entries`. Take a look at it. You will see that it has a one-line entry for each word occurrence in the files in question (including the boilerplate at the beginning of the file). The entry contains the three-letter code for the text file followed by a line number, padded so that all entries for a given file use the same number of digits (this facilitates sorting). Note that we convert everything to lowercase, and we also suppress words of length 3 or less.[4]

Now look again at the code for `generateAllIndexEntries` and the preceding functions, and check that you can see broadly how it is working.

To create an index for our search engine, we want to *sort* these raw entries by their words, which can obviously be achieved by sorting the entries themselves. Since we are pretending we only have 1MB of working memory available, we do this by breaking the list of raw entries into chunks that are individually small enough to be sorted in-memory, and then merging all of these chunks together.

The first part of this process is already done for you. Look at the definition of the file `splitIntoSortedChunks`. You can try out this function as follows:

[2] The text files were obtained from the totally wonderful Project Gutenberg site (`www.gutenberg.org`), which makes a enormous number of literary works freely and legally available for purposes such as this. If you wish, you're welcome to add further literary works to your searchable corpus for your own interest: go to the Gutenberg site and look for the plaintext (`utf-8` encoding) versions of the books you want.

[3] OK – you *are* allowed to edit this bit of the provided code, just to vary the selection of files.

[4] This is a simple measure to avoid cluttering our index with vast numbers of occurrences of words like *the*. Unfortunately, it also cuts out perfectly reasonable words like *sun*. A better solution (more typical of real search engines) would be to index all words not appearing in a specified set of *stopwords* (typically short, commonly occurring grammatical function-words).

```
splitIntoSortedChunks('raw_entries')
```

This will create (in this instance) three new text files called `temp_0_1`, `temp_1_2`, `temp_2_3`, `temp_3_4` and `temp_4_5`. Take a look to see what they are like. Each contains around a fifth of the entries from `raw_entries`, and each has been sorted in-memory using the built-in Python `sort` method.[5] Note too that the call to `splitIntoSortedChunks` returns the number of chunks created (5 in this case).

Take some time to understand how the code for `splitIntoSortedChunks` is working. You will notice that this makes use of classes for buffered input and output defined in the source file `buffered_io.py`. (These are slightly adapted versions of classes that appeared in Lab Sheet 3. The explanation given there may be helpful.) Study the code in both `buffered_io.py` and `index_build.py` until you understand how these classes are used in practice. Note that in this version, a real number ≤ 1.0 is used to specify the *proportion* of the available memory to be allocated to the buffer in question.[6]

Now for the part you have to implement, at the point marked `# TODO`. If you've done the MergeSort exercise on Lab Sheet 2, you'll have a headstart here. There are two subtasks:

1. Write a function `mergeFiles(a,b,c)` which, given three integers a, b, c, merges the contents of two existing, already sorted input files called `temp_`a`_`b and `temp_`b`_`c to create a new sorted file in the same format called `temp_`a`_`c. (For instance, calling `mergeFiles(0,1,2)` with the files you already have should create `temp_0_2`.)

   You should broadly follow the merge procedure described in lectures, but should pay particular attention to what happens when one of the input files is exhausted before the other. You may define other auxiliary functions if you wish.

   All file input/output should be managed via the classes in `buffered_io.py`. You should use three buffers each of proportional size 0.3 (notionally leaving free a small amount of memory for other program purposes). All buffers should be flushed/closed after use, and your function should conclude by deleting the two input files (use `os.remove(`*filename*`)`).

2. Now use `mergeFiles` to define a recursive function `mergeFilesInRange(a,c)` which will merge all the chunk files `temp_`a`_`$(a+1)$ to `temp_`$(c-1)$`_`c inclusive (as generated by `splitIntoSortedChunks`) to produce a single output file `temp_`a`_`c. You should follow the pattern of the MergeSort algorithm described in lectures. In addition, your function `mergeFilesInRange` should return the name of the output file it creates.

Your `mergeFilesInRange` function will be called by `sortRawEntries`, which is defined immediately after your code. The bottom line is that executing

```
sortRawEntries('raw_entries')
```

should have the effect of sorting the whole file `raw_entries` into a single output file called (in this case) `temp_0_5`, and should return the string `'temp_0_5'`. You should also check that this still works correctly with Shakespeare and Tolstoy thrown in; in this case, of course, the number of chunks will be much larger.[7]

[5] This uses a somewhat idiosyncratic variant of MergeSort with various bells and whistles, modestly dubbed by its developer Tim Peters as 'Timsort'.

[6] In the buffer classes provided, the realization of this idea of 'memory allowance' is rather crude and imperfect, and is intended only to illustrate the concept in broad terms.

[7] If you open a window in your file manager to view the contents of your working directory, it's quite fun to watch the temporary files come and go as the computation proceeds.

The functions provided after `sortRawEntries` perform some further stages of processing. First, we 'compress' our sorted list of entries into a master index file called `index.txt`, in which all the entries for a given word are combined in a single line. This will be the main index used by our search engine. Next, we read through this file to create an in-memory Python dictionary, called the *meta-index*, which we can use to locate the entry for a given word. For instance, if `MetaIndex['above']` returns 26, this tells us that the index entry listing all occurrences of *above* may be found at line 26 of `index.txt`.

The function `buildIndex()` draws together all stages of the index creation process as described above. If your MergeSort implementation works correctly, then calling this function will create the whole index from scratch and report success. You should inspect the generated file `index.txt`, checking that it has the same form as the provided file `sample_index.txt`.

You should now be able to look up index entries for particular words like this:

```
indexEntryFor('above')
```

Caution: some things *not* to do!

1. Don't attempt to manipulate the input/output files except through the classes provided in `buffered_io.py`. Let these classes do all the file handling 'behind the scenes'. Use only the methods `readln`, `close`, `writeln`, `flush` from these classes.

2. When merging files, don't try to store their entire contents in memory all at once. This would defeat the point, which is to see what we'd do if the file contents were too large to fit into memory.

3. Don't make use of any built-in sorting function within the code you write.

Marking scheme

Marks for Part A will be awarded as follows:

Correctness: 15 marks. We will test whether your code produces correct output on a certain corpus of text files. This part will be marked quite strictly: any errors may occur a significant penalty.

Time and Space Efficiency: 10 marks. Any reasonable implementation of the right algorithm is likely to qualify for these marks, so don't spend too long fine-tuning your code to gain those last few percent in runtime!

Code clarity: 5 marks. The readability of your code will be assessed by a human marker.

Part B: Meta-index compression via perfect hashing [40 marks]

The search engine we have constructed keeps most of its index information on disk (in `index.txt`), but also requires an in-memory dictionary `MetaIndex`. This itself still occupies significant space, since (for instance) it includes as keys all the words (of length ≥ 4) that appear in the corpus. We now explore whether we can get away with a much smaller 'meta-index'. Not only would this economize on space, load time etc., but it might also allow the meta-index to live in 'fast memory' where it can be accessed very quickly.

The intention is that if we can construct a *perfect hash function* H mapping our keys to *distinct* integers, then we may simply use this as our meta-index, provided we reshuffle

our main index file so that the entry for each key w does indeed appear at the line number given by $H(w)$.[8] The approach we follow, due to Belazzougui, Botelho and Dietzfelbinger (2008), enables us to construct perfect hash functions that can be stored very compactly in memory. The instructions below give all the information you will need, but the provided file `BBD_slides.pdf` may serve as a useful supplement (including some pictures).

Your solution to Part B will be added within the file `perfect_hashing.py`. Take a look at this file. There is some simple machinery for hashing of strings modulo a number p (maybe prime, maybe not). The precise details aren't important, but you should experiment a bit with the functions `modHash` and `buildModHashTable` to get the feel of what they do.

There is then some code for selecting a prime number close to a given n, followed by a function `miniHash` which, for a given number m, yields a whole family of (reasonably 'independent') hash functions mapping strings to integers $0, \ldots, m-1$. For example:

```
miniHash(1000,0)('chair')
miniHash(1000,1)('chair')
```

Our goal is to construct a *perfect hash function* mapping a specified list of n words (given by `keys`) to integers $0, \ldots, m-1$ for some specified m. (The intention is that $m > n$, but not by too much.) The method proceeds as follows.

First, we 'mod hash' our set of keys down to numbers $0, \ldots, r-1$, where r is some suitably selected prime number. (This is done by the code for the `Hasher` class later in the file.) This results in a list `L` of r *buckets*, where each bucket is itself a (possibly empty) list of strings. Next, we try to assign to each of these buckets a number j such that `miniHash(m,j)` is a suitable mini-hash function to apply to that bucket. 'Suitable' means that all these mini-hash functions *taken together* should map the contents of all buckets to numbers $0, \ldots, m-1$ with no clashes.

Your task is to write a function `hashCompress(L,m)` which takes a list L of buckets and a desired table size m, and suitably selects a number j for each bucket as above, storing the number j for the ith bucket at position i in a list R, whose length is r (the number of buckets). This list R is what your function should return; it will become the value of the field `hashChoices` in the `Hasher` class. The `hash` method for this class then gives the hash function we have constructed, which is intended to be a perfect hash function.

How does one go about choosing j-values to ensure this? We may adopt the following approach:

- First, sort the list of buckets L in order of decreasing size. In the sorted list, it will be useful to pair each bucket with a number i indicating the position in original list `L` where this bucket came from.
- Create a list T of m booleans, initially all `False`, intended to record which slots in the main table are currently 'taken'. Also create a list R of integers (initially zero), one for each bucket.
- Go through the buckets in order of decreasing size. For each bucket B, search for the smallest integer $j \geq 0$ such that the mini-hash function `miniHash(m,j)` is *suitable* for B. This means that:
  - it maps all elements of B to currently free slots in the main table,
  - it doesn't map any two elements of B to the same slot.

[8] In the interests of keeping Parts B and C independent, we have not actually implemented this reshuffling. Doing so is not necessary here, since there are other ways to test your solution.

Once a suitable value of j is found,[9] record this value as $R[i]$ (where i is the original position of the bucket B), and mark all the slots to which we are sending elements of B as 'taken'.

Although this may seem complicated, the list comprehension syntax of Python allows it to be coded quite economically. Your code for `hashCompress` should be added at the point marked `# TODO`. You may also define any auxiliary functions you find helpful.

The provided class `Hasher` will then call your code to create a hash table. As you can see from the code, this class gives the user control over various parameters. A typical use of this (with your meta-index from Part A loaded in) would be:

```
H = Hasher (MetaIndex.keys(), 5.0, 0.8)
```

where the 5.0 controls the value of r via $r = n/5.0$, and the 0.8 is the desired load on the resulting table (i.e. n/m). Strings can then be hashed by calling e.g. `H.hash('chair')`. You may also use the function `checkPerfectHasher` to check whether a hasher is a perfect one: this works simply by tabulating its hash table for the given set of keys.

Finally (and purely for interest), you may use `bestCompression` to glean some statistics on how much space would be required to store the hash function, given a simple compression scheme. A typical value for a hasher `H` constructed above would be 1.6 bits per key, or less – obviously tiny compared with the original `MetaIndex`!

Marking scheme

Marks for Part B will be awarded as follows:

Correctness: 20 marks. We will test whether your code indeed works correctly on certain sets of keys.

Efficiency: 15 marks. We will time the execution of your code on certain tests. Again, do not worry too much about fine-tuning: any correct implementation whose runtime comes within a factor of 1.5 of our sample implementation will earn all of these marks.

Code clarity: 5 marks. Assessed by a human marker.

Part C: Processing search queries [30 marks]

Before attempting Part C, you should have a working solution to Part A, and you should have used this to generate an index file for some selection of the books. You may then load in the file `search_queries.py`.

Take a look at the source code for the function `search` near the end of the file. This searches for places where two or more of the specified keys occur within the specified line window (the default is just a single line). You may also specify the number of hits to be returned (you can later request others using the function `more`).

The following examples illustrate what is going on: try them with all five books loaded.[10]

[9] Actually, there's no absolute guarantee that a suitable j will *ever* be found. But as long as our mini-hash functions are 'sufficiently random and independent', this will happen 'almost certainly', and the approach works extremely well in practice. You need not worry about the possibility that the search will fail: in the highly unlikely event that such a situation arises during marking, you won't be penalized.

[10] For good results, make your window wide enough to fit the displayed lines of text.

```
search(['black','white','blue','green'])
more()
more(10)
search(['black','white','blue','green'],3,10)
search(['palpable','very'])
```

For each hit returned, a portion of text of length `lineWindow` is displayed, and line number on the left is a reference to the *first* line of this portion.

There are a few subtleties around what happens if some region of the text qualifies as a hit for more than one reason. In the interests of reducing clutter in the output, and also for ease of implementation, we have adopted the following policies:

- A search will never return the same line number twice. For instance, if our corpus contained the line

  > black and white and blue and green

  then `search(['black','white','blue','green'])` would return this line only once, even though (arguably) it is a hit for 6 different reasons.

- A hit requires that we have an occurrence of one of our keywords, say A, followed by an occurrence of a different keyword, say B, within the specified line window. However, such a hit will be detected and returned only when these occurrences are 'as close as possible': that is, when the intervening text does not itself contain any keywords. For instance, suppose our text consists of the six lines:

  > you say 'black'
  > I say 'black'
  > you say 'white'
  > I say 'white'
  > you say 'green'
  > I say 'green'

  and we perform `search(['black','white','blue','green'],3)`. Then lines 2 and 4 will be returned as hits, so lines 2–4 and then 4–6 will be displayed. However, line 1 won't be returned, even though we have 'black' on line 1 and 'white' on line 3, because there's an intervening occurrence of 'black' on line 2, which together with the 'white' on line 3 forms a "closer" hit. Similarly, line 3 will not be returned.

Now let's look at how all this is implemented.

Clearly, we can use `indexEntryFor` (from Part A) to retrieve the entries for each of our keys, which in effect list all their occurrences in the corpus. One's first thought might be to transform each entry into an actual Python list, then 'merge' these lists, and finally scan through the resulting list looking for hits.

This procedure would work fine if the number of occurrences of each of our keys was fairly small. Suppose, however, that we had a very large corpus in which each key appeared several million times. Then the above method would require us to process our entire index entries before we could detect even a single hit for our search – even if such a hit were to occur very early in the corpus. In this practical, we therefore explore a different approach: we process the index entries 'lazily' – that is, a little bit at a time, on demand – using them to generate *streams* of items which can be queried as required.

Have a look at the definition of the class `ItemStream` within the file `search_queries.py`. This performs the transformation of index entries into streams. You will see what it does by importing the file, and trying the following:

```
indexEntryFor('above')
a = _
A = ItemStream(a)
A.pop()
```

Now repeat `A.pop()` lots of times, occasionally interspersed by `A.peek()`. You will see that the latter allows us to view the next item without advancing to the next position. Once there are no more items, calling `pop` or `peek` will simply return `None` forever.

Next, look at the code for the crucial class `HitStream`, which in effect does 'online merging' of a given list of item streams, checking for hits as it does so (with a specified line window). An object of this class may be constructed by supplying a list `itemStreams` of objects of class `ItemStream` along with an integer `lineWindow`. Once such an object has been created, calling its `next` method repeatedly will return the hits in succession (until there are no more hits, after which `None` will be returned).

Study the `HitStream` code carefully with the help of the comments provided, also noting how this class (under the alias `HS`) is used in the implementation of `search` and `displayHits`. You should expect to devote some time to getting a good understanding of how this code is working.

Once you have understood this code, you may realize that it will not be very efficient if the number of keys is large.[11] The inefficiency centres around the method `fetchNextItem`, which does a fresh scan through the head items of all the streams each time it is called. The runtime of this will be $\Theta(k)$, where k is the number of item streams (= number of keys).

Your task is to add a more efficient implementation of hit streams by making use of heap-based *priority queues*, as described in Lecture 11 and as implemented by the Python module `heapq`. You should study the documentation page for this class carefully, noting any small differences between this and the version presented in lectures:

`https://docs.python.org/3/library/heapq.html`

It is for you to think through how these heap queues might be used to improve the efficiency. Once you have understood the idea, find the `# TODO` comment in the source file (just under `import heapq`), and at this point add code for a class called `HitStreamQ`, which does the same job as `HitStream`, but more efficiently. Like `HitStream`, your class should contain an initializer with arguments `itemStreams` and `lineWindow`, and a method `next` with no arguments. It *may* also contain equivalents of the helper methods `nextItem` and `isHit`, but it is not required to do so.

Do not delete or modify the existing code for `HitStream` – you will need both `HitStream` and `HitStreamQ` around in order to compare their behaviours. It is completely fine for you to copy or adapt parts of the `HitStream` code in your implementation of `HitStreamQ` – indeed, you may find that the amount of fresh code you have to write is quite small. In any case, your code for `HitStreamQ` should not be very much longer in total than the existing code for `HitStream`.

The bottom line is that any uses of `search` and `more` should return the same results regardless of whether `HitStream` or `HitStreamQ` is used (you can switch between them by editing the definition of `HS`).

[11] It might seem odd to want to do searches with a large number of keys. However, a semi-plausible motivation for this is as follows. Suppose we wish to search our corpus for passages relating e.g. to astronomy (perhaps because we're writing a paper on astronomy, and want to give it a touch of class by including some well-chosen literary quotation). We could do this by constructing a long list of likely key words (e.g. `star`, `moon`, `planet`, `telescope`, ...), and then searching for all places in the corpus where at least two of these occur within the space of (say) 5 lines.

As regards efficiency, there are two criteria. The first is that each iteration of the 'main loop' within your `next` method should execute in $O(\lg k)$ time, where k is the number of item streams. (The analysis of heap behaviour in Lecture 12 should enable you to confirm that this is the case.) The second concerns the observed runtime. You are given a function `allHits` that extracts the complete list of hits from a hit stream, and a list of keys called `ordinals`. With an index for all five books installed, the execution of `allHits` on a hit stream for `ordinals` (and some small choice of `lineWindow`) should run much faster when `HitStreamQ` is used. A typical speed-up factor of 10 or more will get you a good mark.

You should perform sufficient tests to satisfy yourself of both the correctness and the efficiency of your implementation, but you are not required to submit your test code.

Marking scheme

Marks for Part C will be awarded as follows:

Correctness: 15 marks. We will test whether your implementation produces the same results as `HitStream` on a selection of queries of varying levels of complexity.

Runtime efficiency: 10 marks. We will measure the speed-up factor of your implementation on various test cases, and also check by inspection that your code has the desired asymptotic efficiency property.

Code clarity: 5 marks. Assessed by a human marker.

Submission instructions

You should submit your coursework via Learn. Go to the 'Assessment' page from which you obtained this instruction document, and click on the link bearing the title of this coursework. This will allow you to upload your three files (or as many of these as you have attempted):

```
index_build.py    perfect_hashing.py    search_queries.py
```

Do not submit any files other than these.

You may re-submit as many times as you wish up to the deadline (16:00 (GMT) on Friday 12 November). **However,** if you do this, then please re-submit *all* of the files that you have worked on, not just those that have changed! Only the files included in your most recent submission made before the deadline will be marked.

Good luck!

John Longley, October 2021