Assignment (60 points)

1 Overview

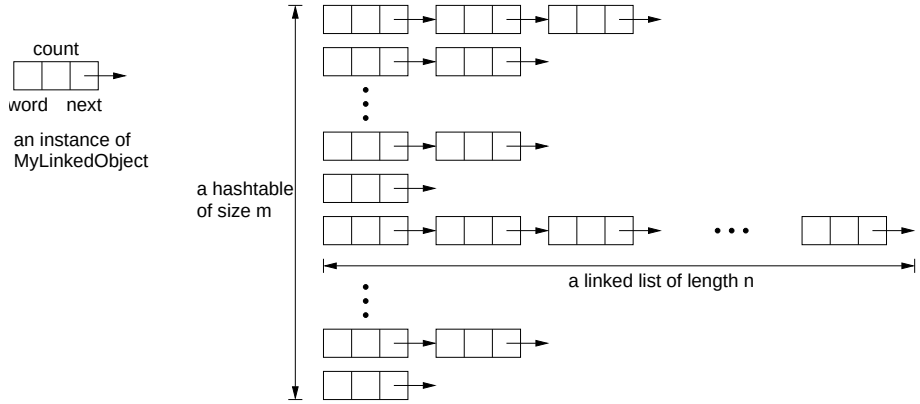
Development of language models (LMs) has a history of more than half of a century. LMs use statistics derived from a large amount of natural language texts in order to calculate the probability of words and word sequences. There have been a number of LM applications in the natural language processing (NLP) field, *e.g.*, automatic speech recognition (ASR), natural language generation (NLG), part-of-speech tagging and parsing, machine translation (MT), and information retrieval, to name only a few traditional ones. Large language models (LLMs) may be considered as the most recent advances in the field. LLMs are built on modern neural network technologies, such as the long short term memory (LSTM) and the transformer architecture, trained on a humongous amount of data, resulting in models with a enormous number of parameters.

Unfortunately LLMs are far beyond the scope of a single Java assignment. Instead this assignment is concerned with creating and experimenting a traditional LM, a statistical model that is much simpler to handle. The work starts with building your 'own' structure of a hash table that accommodates your 'own' linked objects, instead of utilising the ones from the Java Collections Framework. Using the above structure, you are required to develop a system that builds a model from an English language text, and explore various properties of your implementation. A graphical user interface (GUI) should also be prepared in order to handle a text input and to display outputs as well as model properties.

Hash Table, Hash Function and Linked Object — your 'own' structure

A hash table is an effective and practical data structure that realises dictionary operations in $O(1)$ time. In this assignment, you will first implement a simple hash table of your own. The hash table plays a major role in the rest of the assignment, where rudimentary attempt will be made to create and utilise a statistical LM.

Your hash table will have the following structure. First a choice of hash function is used to derive a hash table index that is specific to a given word sequence. (note: For this assignment a 'word sequence' implies a sequence of one or more words, separated by space(s).) Ideally different word sequences are mapped into different hash table indices, however some amount of clustering (*i.e.*, difference word sequences being mapped into the same index) is often unavoidable. In order to alleviate the clustering issue, we store an individual word sequence in a linked list structure that is selected by the hash function. A linked list can be implemented recursively using multiple linked objects, where each linked object should consist of three instance fields — '`word`' is a string entry of a word sequence (of one or more words), '`count`' carries the number of occurrences of that particular string, and finally '`next`' is linking to the next object in the linked list ('`null`' if it is the last object in the list). Within each linked list, objects are ordered by the ascending order of alphabets in the instance field, '`word`'. When a new word sequence arrives at a linked list selected by the hash function, a new object is inserted at the right position in the list. If it already exists in the list, '`count`' is incremented.



Shown above are an instance of a single linked list object (left) and a hash table of size m (right) with hash table indices $0, 1, \ldots, m-1$. Each linked list is a connected sequence of linked objects and is associated with one hash table index. The length of a linked list varies, and is as small as '0' (zero). The rightmost linked object is not connected to anywhere, hence '`null`' in the '`next`' field. You may find further description of a hash table and a hash function by searching the web — Wikipedia may be a good place to start with.

2 Core Task Description

Core tasks start with implementation of your own linked object structure (**Task 1**) that can be connected to build a linked list, own hash functions (**Task 2**), and own hash table (**Task 3**) that builds on your implementation of linked lists and hash functions. You should use your implementation of a hash table in order to create a vocabulary list (**Task 4**) and 'so-called' *n*-gram LMs (**Task 5**).

2.1 Task 1 — `MyLinkedObject`

Objects of a linked list are arranged in a linear order, determined by a reference (an entry in the '`word`' field) in each object, providing a flexible representation for dynamic sets. Your first task is to write a class, `MyLinkedObject`:

```
public class MyLinkedObject { ... }
```

If you find it useful for your coding, the `MyLinkedObject` class can be arranged by extending some superclass and/or by implementing a suitable interface(s), *e.g.*,

```
public class MyLinkedObject extends ... implements ... { ... }
```

The class will have three private fields:

```
private String word;
private int count;
private MyLinkedObject next;
```

The following two methods should be implemented:

`public MyLinkedObject(String w)`
: This is the constructor that assigns the `word` field by the parameter `w`. It also assigns initial values for the `count` and the `next` fields.

`public void setWord(String w)`
: The important assumption here is that the parameter `w` is *alphabetically not smaller than* the `word` field of this object. It increments `count` if `w` is equal to `word`. If `w` and `word` are not equal and if the next object does not exist (*i.e.*, `next` is `null`), it creates a new object for `w`, that is linked to the `next` field of this object. If the next object exists, and if `w` is alphabetically smaller than the `word` field of the next object, it creates a new object for `w`, and that object is inserted between this and the next objects. Otherwise `w` is passed on to the next object — you may note the use of the recursive structure, eg, `next.setWord(w)`, within the `setWord` method of this object.

It is expected that you create additional method(s) where you find them useful for your implementation of this and the rest of tasks.

Discussion item in Part III of your report:

- Give a brief description for each method you have implemented for the `MyLinkedObject` class.

2.2 Task 2 — `MyHashFunction`

A hash function transfers a given word sequence to an integer value, which is then used as an index in a hash table. A good hash function should result in a uniform distribution of hash table indices so that an adverse effect of clustering may be reduced. This task requires you to do some study by implementing a super class `MyHashFunction` and multiple subclasses, each of which provides a single implementation of a hash function algorithm. One of them must use the first letter of a word sequence — *e.g.*, its unicode value modulo 'the hash table size', *i.e.*, a remainder of the division of a unicode value by the hash table size *m*. You should explore and implement one (or two) more hash function algorithms of your choice — for the later task, choose algorithm(s) you are able to test a range of values to derive the hash table size *m*.

Discussion item in Part III of your report:

- Describe your choice of hash function algorithm(s). Where appropriate, mathematics may be used. The source (*e.g.*, textbook, reference, web URL) of your choice must be provided.

2.3 Task 3 — `MyHashTable`

The `MyHashTable` class is created with *m* (the hash table size) linked lists of, *initially*, the length '0' (zero). A word sequence is stored to one of the linked lists, selected by your choice of a hash function, hence the array of linked objects grows, utilising instance methods provided by the `MyLinkedObject` class.

Discussion item in Part III of your report:

- Although this is not a mandatory requirement but a possible design consideration, you may also think about hiding the `MyLinkedObject` class and the `MyHashFunction` class within the `MyHashTable` class. The rest of tasks should only use the `MyHashTable` class, through which methods from the `MyLinkedObject` class and the `MyHashFunction` class may be utilised.
  Were you able to achieve this? If not, why not?

2.4 Task 4 — vocabulary list

The purpose of this task is to create a vocabulary list for words (*i.e.*, word sequences of length '1', or 'unigrams') appeared in a document. Using your implementation of the `MyHashTable` class and other classes, you are required to create a GUI, with which you are able to extract 'words' given a document, store them in the hash table structure, and to display some statistics of the internal state of the hash table and the collected words. In order to develop your GUI, a 'preprocessed' sample document, `news.txt`, is provided. Your GUI should be able to handle this sample document, as well as other documents having the same format.

note about '`news.txt`'

- Preprocessing was made for simplicity of the task; all words are spelt using lower case letters only. Digits are replaced with spelt words (*e.g.*, 'year nineteen ninety' instead of 'year 1990', or 'two point five million pounds' instead of '£2.5M'). All of punctuation are removed apart from a *full stop* (.) and an *apostrophe* (') (*e.g.*, 'mrs.', 'dr.', 'don't', 'that's').
- Any words with different spelling (*e.g.*, 'have', 'has', 'had', 'having') are considered as different words, thus they should be stored separately in the hash table.
- If you (or your program code) find a mis-spelt word (*e.g.*, 'haev' or 'havve' instead if 'have'), they should also be stored separately in the hash table.
- For this task you have to ignore the meaning of a word. Only a spelled sequence matters; for example, verb 'count' and noun 'count' (or even a noble person 'count') will have a single entry in the hash table.

It is safe to assume that documents contain a text of English language with multiple lines and arbitrary length, consisting of words separated by spaces and/or new lines. When reading the sample document `news.txt`, and if your GUI identifies any item that contains a unicode symbol that is not either of a *full stop*, an *apostrophe*, or lower case English letters, it is likely a mis-processed item — the GUI should not store that item but display a warning message together with that mis-processed item.

note You may consider importing the `java.util.regex` package, then use regular expressions — *e.g.*, `"[^a-z]"` is a regular expression that indicates 'any character except lower case letters'.

Once words from a document are collected by the hash table, your GUI should be able to display:

- a document (from an input file);
- a vocabulary list with two columns — words in the first column and their frequencies in the second (*i.e.*, `word` and `count` fields of instances from the `MyLinkedObject` class — using an alphabetically natural order of English;
- a vocabulary list with two columns (words and counts) which is first sorted by the decreasing number of word occurrences (*i.e.*, a word that occurred more frequently appear at a higher rank), then for words with the same frequency, by an alphabetically natural order of English;

  note
  * This can be achieved by using some classes readily available from the Collections Framework, however it may be interesting if you are able to implement this by rearranging the order of instances within each of linked lists that are initially ordered by alphabet.
  * Two types of vocabulary lists can be switched by a button.
- the total number of words in the document (*i.e.*, the sum of counts of all words in the vocabulary list);
- the total number of different words in the document (*i.e.*, the number of entries in the vocabulary list);

- some form of statistics presenting the distribution of the linked list lengths within the hash table (*e.g.*, lists, graphs, mathematical values such as an average and a standard deviation).

You may consider use of a scrollbar within the GUI in order to display a document and/or a vocabulary list.

Discussion items in Part III of your report:

- Describe your strategy for rearranging from the alphabetically ordered vocabulary list to the list with descending frequency order.
- What observation(s) you have made from the vocabulary lists created?
- Make comparison between hash function algorithms, *e.g.*, statistics for lengths of individual linked lists within the hash table. Discuss which algorithm is the better choice for your hash table. Also discuss any observation you have made when using various values for the hash table size m.
- For natural language processing tasks such as this, discuss potential benefit(s) of sorting 'linked lists' by the decreasing number of word occurrences, instead of ones ordered alphabetically.

2.5 Task 5 — n-gram LM

This task is concerned with development of simplest models for language, known as 'bigrams' (*i.e.*, word sequences of length '2') and 'trigrams' (*i.e.*, word sequences of length '3'), or simply n-grams for $n = 2, 3, \ldots$ For example, from the following word sequence:

'in sheffield today it's sunny'

you can identify five unigrams (*i.e.*, words), 'in', 'sheffield', 'today', 'it's', 'sunny', four bigrams, 'in sheffield', 'sheffield today', 'today it's', 'it's sunny', and three trigrams, 'in sheffield today', 'sheffield today it's', 'today it's sunny'.

Using your implementation of the `MyHashTable` class and other classes, your GUI should be able to extract bigrams (or trigrams) given a document, store them in the hash table structure, and display some statistics of the internal state of the hash table and the collected n-grams. Once again, a preprocessed sample document `news.txt` may be used for GUI development. This task implements the approach to generating 'likely' word sequences using an n-gram LM that is stored in the hash table.

note You may also consider processing bigrams and trigrams that run across two lines. Suppose one line ends with a word 'L' and the next line starts with a word'F', then this task considers 'L F' as a bigram. Similarly for a trigram that runs across two lines.

Consider a sequence of K words, $w_1, w_2, \ldots, w_K$. The probability of this word sequence is calculated by applying the chain rule:

$$p(w_1, w_2, \ldots, w_K) = p(w_1)p(w_2|w_1)p(w_3|w_1, w_2)p(w_4|w_1, w_2, w_3)\ldots p(w_K|w_1, \ldots, w_{K-1})$$

Suppose that the probability of a word occurrence is affected by the previous one word only:

$$p(w_1, w_2, \ldots, w_K) \approx p(w_1)p(w_2|w_1)p(w_3|w_2)p(w_4|w_3)\ldots p(w_K|w_{K-1})$$

where the probability $p(w_k|w_{k-1})$ for $k = 2, \ldots, K$ can be calculated using the unigram count $c(w_{k-1})$ and the bigram count $c(w_{k-1}, w_k)$ as the following:

$$p(w_k|w_{k-1}) = \frac{c(w_{k-1}, w_k)}{c(w_{k-1})}$$

Similarly, suppose that the probability of a word occurrence is affected by the previous two words:

$$p(w_1, w_2, \ldots, w_K) \approx p(w_1)p(w_2|w_1)p(w_3|w_1, w_2)p(w_4|w_2, w_3)\ldots p(w_K|w_{K-2}, w_{K-1})$$

where the probability $p(w_k|w_{k-2}, w_{k-1})$ for $k = 3, \ldots, K$ can be calculated using the bigram count $c(w_{k-2}, w_{k-1})$ and the trigram count $c(w_{k-2}, w_{k-1}, w_k)$ as the following:

$$p(w_k|w_{k-2}, w_{k-1}) = \frac{c(w_{k-2}, w_{k-1}, w_k)}{c(w_{k-2}, w_{k-1})}$$

This task starts with collecting n-grams for $n = 1, 2, 3$ within your own hash table structure. Using collected n-gram LMs, and now given a first few words, your code is required to find a 'most likely' word sequence that follows the given few words.

note A crude, but possibly the simplest, approach to calculating the 'most likely' word sequence may be to use the probabilities, $p(w_k|w_{k-1})$ for $k = 2, \ldots$ that are derived from unigrams and bigrams — *e.g.*, suppose w_{k-1} is given, find w_k with the largest probability of $p(w_k|w_{k-1})$. Now w_k is decided, then find w_{k+1} having the largest probability of $p(w_{k+1}|w_k)$, and so on... Similarly, suppose trigrams are also available, find w_k having the largest probability of $p(w_k|w_{k-2}, w_{k-1})$ where w_{k-2} and w_{k-1} are both known, and next, find w_{k+1} having the largest probability of $p(w_{k+1}|w_{k-1}, w_k)$ given w_{k-1} and w_k. The procedure can produce a word sequence of various lengths.

Your GUI should be able to display:

- (suppose a first few words are entered from a text window) the most likely word sequence of up to 20 words — when using unigrams and bigrams only, as well as when using up to trigrams;
- some form of statistics for bigrams and trigrams presenting the distribution of the linked list lengths within the hash table (*e.g.*, lists, graphs, mathematical values such as an average and a standard deviation).

Discussion items in Part III of your report:

- What do you do with $p(w_1)$ when calculating $p(w_1, w_2, \ldots, w_K)$ using unigrams and bigrams? Similarly, what do you do with $p(w_1)$ and $p(w_2|w_1)$ when calculating $p(w_1, w_2, \ldots, w_K)$ using up to trigrams?
- Using unigrams and bigrams only, find the first 20 words that most likely follow the following two words, 'you have'. Repeat the same for 'this is one'.
- Now using also trigrams in addition to unigrams and bigrams, find the first 20 words that most likely follow the following two words, 'you have'. Repeat the same for 'this is one'.
- Using unigrams, bigrams and/or trigrams, are you able to find the first 20 words that most likely follow the following three words, 'today in sheffield'? If not, why not? Repeat the same for 'in sheffield today'.
- Discuss any issue(s) of implementing n-grams with a larger value of n than '3'.

3 Further Notes

System extensions / sensible changes to the task description. This part is not a core task for this assignment, however read the following paragraph...

The marker is most willing to hear your idea for useful and interesting extension(s) that can be incorporated to the core tasks, as well as sensible change(s) to the task descriptions above, such as another approach(es) to predicting word sequences, or statistics that can expose the nature of n-gram LMs. If you are proposing such ideas, you should describe them clearly in your report and demonstrate them by implementing your design using Java.

Constraints on programming. Program code should compile and run on a console command line under **Java 17**. Where needed, you can import the `sheffield` package as well as any package(s) listed at **Java 17 API**

```
https://docs.oracle.com/en/java/javase/17/docs/api/index.html
```

but none of other packages.

4 Submission

Your submission should consists of a set of Java source code (`*.java` files) for your GUI and a report.

Java code. All source code should be placed within the '`code`' folder, and the `main` method is implemented in `MyLanguageModel.java`. The `code` folder should include (1) Java source code (`*.java`), (2) a sample document `news.txt` (the original one provided), and (3) the entire `sheffield` package (only if your system requires it), but nothing else. The `code` folder should not include, *e.g.*, bytecode (`*.class`), output files, or any documentation (*e.g.*, `html` files).

The marker will use the following two command lines with Java 17:

```
→ javac *.java
→ java MyLanguageModel
```

to compile and run your GUI within the `code` folder.

Report. Your report should consist of three parts. Part I should be a compact 'user manual' that outlines

- where/how to start/use the GUI.

- any extension(s) and/or some changes to the task requirements that were incorporated

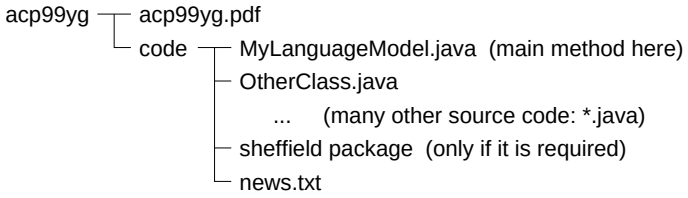
Part II is about your Java construct, where you may talk about

- use of object oriented programming concepts, such as data abstraction, encapsulation, inheritance and polymorphism — here, although not mandatory, UML can be used;
- use of the Collections Framework, GUI, and exceptions handling, if any.

In Part III, your responses are expected for a number of items raised at Tasks 1 to 5.

The report must use fonts of 11pt or larger size, and it should not be more than 5 (five) pages long using the paper size of A4. A cover page is not required, but if you create a cover page, it is also counted within the 5 page limit. The report can include not only text, but also figures, diagrams, and/or tables (or even maths) where appropriate. The report must be in the `pdf` format, and named as '`yourAccount.pdf`' (*e.g.*, suppose your university account is '`acp99yg`' then the report should be named as '`acp99yg.pdf`').

Handin procedure. Suppose your account is '`acp99yg`', then the `code` folder and the report should be arranged under the `acp99yg` folder as the following:



It should be zipped into a single file '`acp99yg.zip`', which is then submitted to Blackboard. Do not use other compression format such as `rar`, `tar`, `7z`, *etc.* A marker will upzip your handin by '`unzip yourAccount.zip`' in a command line.

Before the deadline, you are able to submit your work more than once through Blackboard. If you have made multiple submissions, only the latest version submitted will be marked.

Late handin rule. For each day late from the handin deadline, 5% will be deducted from your total mark. If the assignment is handed in more than one week late, it will not be marked (meaning that it results in a mark of '0').

Unfair means. Any work that you handin must be your own as this is an individual assignment. All code submitted may be examined using specialised software to identify evidence of code written by another student or downloaded from online resources.

You may review the University guidance on unfair means:

`https://www.sheffield.ac.uk/ssid/unfair-means/index`

5 Marking Scheme

The total points allocated for the assignment is **60 points**. Your work will be assessed based on the following criteria:

Overall system (20 points)

16-20 Stylishness and originality that stand out from others, in addition to fully satisfying the marking criteria below;

14-15 Construction of a functional system that fully achieves the core tasks; Sensible structure for the overall object oriented design and the system, *e.g.*, encapsulation, class hierarchies, abstraction;

10-13 The marking criteria above is roughly satisfied;

0-9 Overall development of the system is not satisfactory;

GUI design, use of the recursive structure *etc.* (20 points)

16-20 Stylishness and originality that stand out from others, in addition to fully satisfying the marking criteria below;

14-15 Easy to use, nice looking, clean and fully functional GUI; Clean and fully functional recursive design; (If any) sensible choices and the classes structure from the Collections Framework; Consideration for event/exception handling, where appropriate;

10-13 The marking criteria above is roughly satisfied;
0-9 Unsatisfactory development for GUI, or use of the recursive structure; *etc.*

Report and programming style (20 points)

16-20 The marking criteria below is achieved at an exceptional level;
14-15 The report is sensibly structured, clean, and unambiguous, consisting of a compact description of where/how to start/use the GUI and, if any, extensions and changes (Part I), a clear description of the Java construct (Part II), and concise responses to issues raised at Tasks 1 to 5 (Part III); Good programming style — things to be mindful include adequate commenting, clear indentation, layout, sensible naming for fields, methods, classes, *etc.*;
10-13 The marking criteria above is roughly satisfied;
0-9 Unsatisfactory report and programming style;

Point deductions

up to –10 (minus ten) **points** Deduction of –1 point if the submission contains any file, apart from the source code (`*.java`), the original sample document, `news.txt`, the `sheffield` package, and a report. Further deduction of up to –10 points, depending on the effort made to modify, compile and run the code within a limited amount of marking time; No deduction if the submitted code is unzipped, compiled and run on a command line without any modification;
up to –10 (minus ten) **points** Deduction of –1 point if the `pdf` format is not used for a report; Further deduction of –1 point per page beyond the report page limit, *e.g.*, –2 points if a report is 7 pages long; No deduction if the submitted report follows the handin rule, *e.g.*, the `pdf` format is used and it does not go beyond the page limit of 5 pages;

6 Q&A

Q. It appears that the document file, `news.txt`, is too large for my machine to handle.
A. Set a suitable upper limit in your code for reading the document. Do not modify (or cut short) `news.txt`, which should be included in the submission 'as is'.

Q. Instead of my own hash table, can I just use the Collections Framework for programing Tasks 4 and 5?
A. Yes, if you like, however some point deduction will be made.

Q. Can I use 3rd party packages with my program code?
A. No — apart from the `sheffield` package (if it is needed), it is expected that your design of the system uses packages 'only' from the standard Java platform (below):

```
https://docs.oracle.com/en/java/javase/17/docs/api/index.html
```

and NOT from a 3rd party.

Q. I would like to create my own package and include it in my submission.
A. Yes, you can — as long as it does not contain any 3rd party development. Your submission still should compile and run in two command lines (*i.e.*, '`javac *.java`' and '`java MyLanguageModel`').

Q. Should I include UML diagram(s) in my report?
A. It is NOT a core requirement, although it may be useful when you write your Java construct in Part II of your report.

Q. Does the report page limit (5 pages) apply for each of Parts I, II, and III?
A. No — the report should not be longer than 5 pages for Parts I, II, and III, all together.

Q. Can I use an IDE for this assignment?
A. Yes, you are free to use an IDE to develop your code. However, for submission, it is required that program code compiles and runs on a console command line without requiring a marker to change any part of your program code.