Assignment 1

Information Retrieval

Changelog

All important changes to the assignment specification and files will be listed here.

- `[01/03 13:00]` Assignment released

Aims

- To implement an information retrieval system using well-known tf-idf measures
- To give you practice with binary search trees

Admin

Marks	contributes 14% towards your final mark (see Assessment section for more details)
Submit	see the Submission section
Deadline	8pm on Monday of Week 7
Late penalty	1% off the maximum mark for each hour late. For example if an assignment worth 80% was submitted 15 hours late, the late penalty would have no effect. If the same assignment was submitted 24 hours late it would be awarded 76%, the maximum mark it can achieve at that time.

Prerequisite Knowledge

- Recursion
- Binary search trees (Week 3 only)

Background

The goal of this assignment is to implement an information retrieval system using a well-known term-weighting scheme called **tf-idf**.

Information retrieval is the process of obtaining information from a collection of resources. There are many applications of information retrieval, with search engines being the most visible and widely used.

Inverted index

The main goal of web search engines is to return the set of websites that are most relevant to the given query. But with the Internet having over a billion websites, efficient techniques need to be used, as scanning the entire Internet for every query would take far too long. Instead, search engines use a data structure that associates words or phrases to the set of websites in which they occur. This data structure is called an *inverted* index, because unlike a normal book index which lists the key words and phrases of a document (such as a textbook or journal article), an inverted index inverts this and maps key words/phrases to documents.

Inverted indexes allow for much faster searching at the cost of needing to preprocess each document that is added to the collection.

tf-idf

Although an inverted index allows us to quickly identify which documents are relevant, it doesn't help us decide which documents are *most* relevant.

One tool which search engines can use to rank documents is tf-idf. tf-idf is a statistic used to quantify how important a word is to a document in a collection, and is motivated by the following two observations:

- The more frequently a search term or set of search terms appears in a document, the more relevant the document is likely to be.
- Words which appear more frequently in the collection of documents (for example, "the") are less distinguishing, so their contribution to relevance should be smaller than less frequent words.

tf-idf takes the above observations into account by combining **term frequency**, which is the frequency of a term in a document, and **document frequency**, which is the proportion of documents in the collection that contain the term. tf-idf stands for **term frequency-inverse document frequency**.

The *term frequency* of a term t in a document d is defined as:

$$\mathrm{tf}(t,d) = \frac{\text{frequency of term } t \text{ in } d}{\text{number of words in } d}$$

Meanwhile, the *inverse document frequency* of a term t is calculated by dividing the total number of documents by the number of documents containing the term, and then taking the logarithm (base 10) of that quotient:

$$\mathrm{idf}(t,D) = \log_{10} \frac{N}{|\{d \in D : t \in d\}|}$$

where D is the collection of documents, $N = |D|$ is the total number of documents in the collection and $|\{d \in D : t \in d\}|$ is the number of documents that contain the term t.

The tf-idf of a term in a document is equal to the product of its tf and idf:

$$\mathrm{tfidf}(t,d,D) = \mathrm{tf}(t,d) \times \mathrm{idf}(t,D)$$

Notice how this formula ensures that tf-idf increases if a term appears more frequently in a document, but is significantly reduced if many documents contain the term.

One of the simplest ways to rank documents given a search term, which we will be using in this assignment, is to simply use the raw tf-idf value - the higher the tf-idf value of a document, the more relevant it is (although most search engines use a variation of tf-idf along with other techniques). If there are multiple search terms, then we can just sum the tf-idf values for each query term.

Example of tf-idf

This example comes from the [Wikipedia article on tf-idf](). Suppose our collection D contains two documents d_1 and d_2, and these are their term counts:

Document 1 (d_1)

Term	Term Count
this	1
is	1
a	2
sample	1

Document 2 (d_2)

Term	Term Count
this	1
is	1
another	2
example	3

The calculation of tf-idf for the term "this" is performed as follows:

The tf is just the frequency of the word "this" for each document. In each document, the word "this" appears once; but as Document 2 has more words, its relative frequency is smaller.

$$\mathrm{tf}(''this'', d_1) = \frac{1}{5} = 0.2$$

$$\mathrm{tf}(''this'', d_2) = \frac{1}{7} = 0.14$$

The idf accounts for the ratio of documents that include the word "this". In this case, both documents in our collection include the word "this". Therefore,

$$\mathrm{idf}(''this'', D) = \log_{10}(\frac{2}{2}) = 0$$

So tf-idf is zero for the word "this", which implies that the word is not very informative as it appears in all documents in the collection.

$$\mathrm{tfidf}(''this'', d_1, D) = 0.2 \times 0 = 0$$

$$\mathrm{tfidf}(''this'', d_2, D) = 0.14 \times 0 = 0$$

A slightly more interesting example arises from the word "example", which occurs three times but only in the second document:

$$\text{tf}(''example'', d_1) = \frac{0}{5} = 0$$

$$\text{tf}(''example'', d_2) = \frac{3}{7} = 0.429$$

$$\text{idf}(''example'', D) = \log_{10}(\frac{2}{1}) = 0.301$$

Finally,

$$\text{tfidf}(''example'', d_1, D) = \text{tf}(''example'', d_1) \times \text{idf}(''example'', D) = 0 \times 0.301 = 0$$

$$\text{tfidf}(''example'', d_2, D) = \text{tf}(''example'', d_2) \times \text{idf}(''example'', D) = 0.429 \times 0.301 = 0.13$$

These values show that the second document is more relevant to the search term "example". If multiple terms are searched for at the same time, we simply sum together the tf-idf values for each term to obtain a number which quantifies how relevant each document is.

Setting Up

Change into the directory you created for the assignment and run the following command:

```
$ unzip /web/cs2521/22T1/ass/ass1/downloads/files.zip
```

If you're working at home, download `files.zip` by clicking on the above link and then run the `unzip` command on the downloaded file.

You should now have the following files:

`Makefile`	a set of dependencies used to control compilation
`invertedIndex.h`	contains the definition of data structures and function prototypes - **cannot be modified**
`invertedIndex.c`	contains the functions you need to complete
`exmp1/` and `exmp2/`	directories containing test data and expected output
`file_io_demo/`	a directory containing a demonstration of file IO functions

`exmp1` and `exmp2` are test directories that contain test data (`.txt` files), a test program (`testInvertedIndex.c`) and expected output (`.exp` files). You should examine the test programs to see what they do. You can also edit the test programs to add more tests.

The `make` command will produce executables called `testInvertedIndex` in the `exmp1` and `exmp2` directories. You'll need to change into the `exmp1` and `exmp2` directories to run these executables.

Note that in this assignment, you are permitted to create as many supporting `.c` and `.h` files as you like. This allows you to compartmentalise your solution into different files. For example, you could implement your BST-related functions in one file, and your file list-related functions in another. To ensure that these files are actually included in the compilation, you will need to edit the Makefile; the provided Makefile contains instructions on how to do this.

File Reading

You will be required to read files in this assignment. In case you are unfamiliar with file IO, we have provided a sample program in the `file_io_demo` directory that demonstrates usage of the main file IO functions `fopen`, `fscanf`, `fprintf` and `fclose`. These are the only file IO functions you will need to use in this assignment. The sample program also demonstrates how to read words using the `%s` specifier.

Part 1 - Inverted Index

The goal of Part 1 is to create an inverted index from a collection of files and compute the term frequency (tf) for each word.

generateInvertedIndex

The `generateInvertedIndex` function has the signature:

```
InvertedIndexBST generateInvertedIndex(char *collectionFilename);
```

This function should generate an inverted index from the files listed in the collection file, whose filename is given as an argument

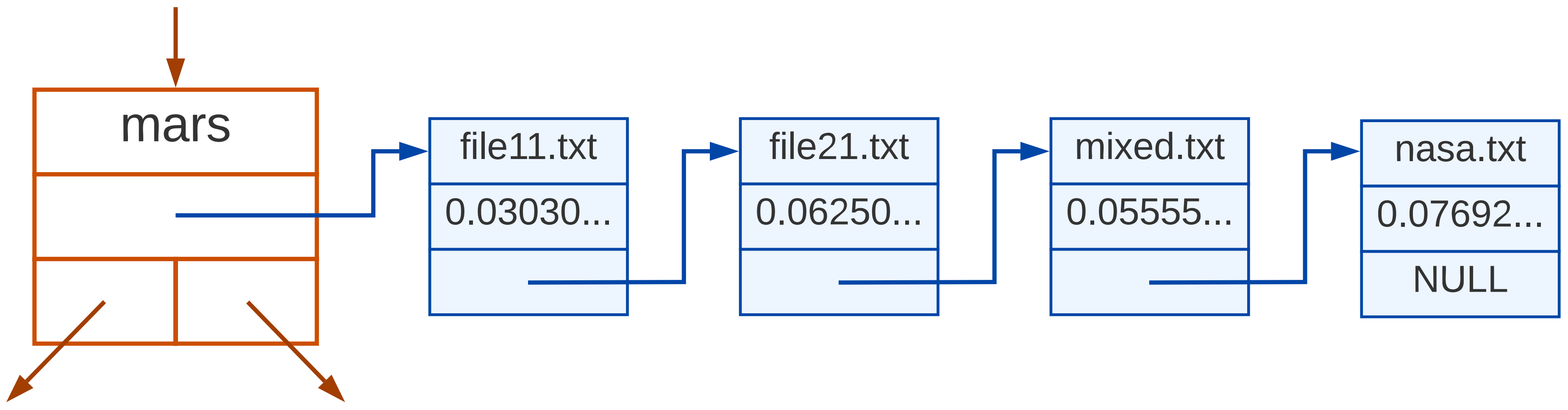
This function should generate an inverted index from the files listed in the collection file, whose filename is given as an argument.

Note: The collection file is a plain text file which contains a list of filenames. You will need to read these filenames, open each of these files and read the words from these files to generate the inverted index. For example, the collection file may contain the following:

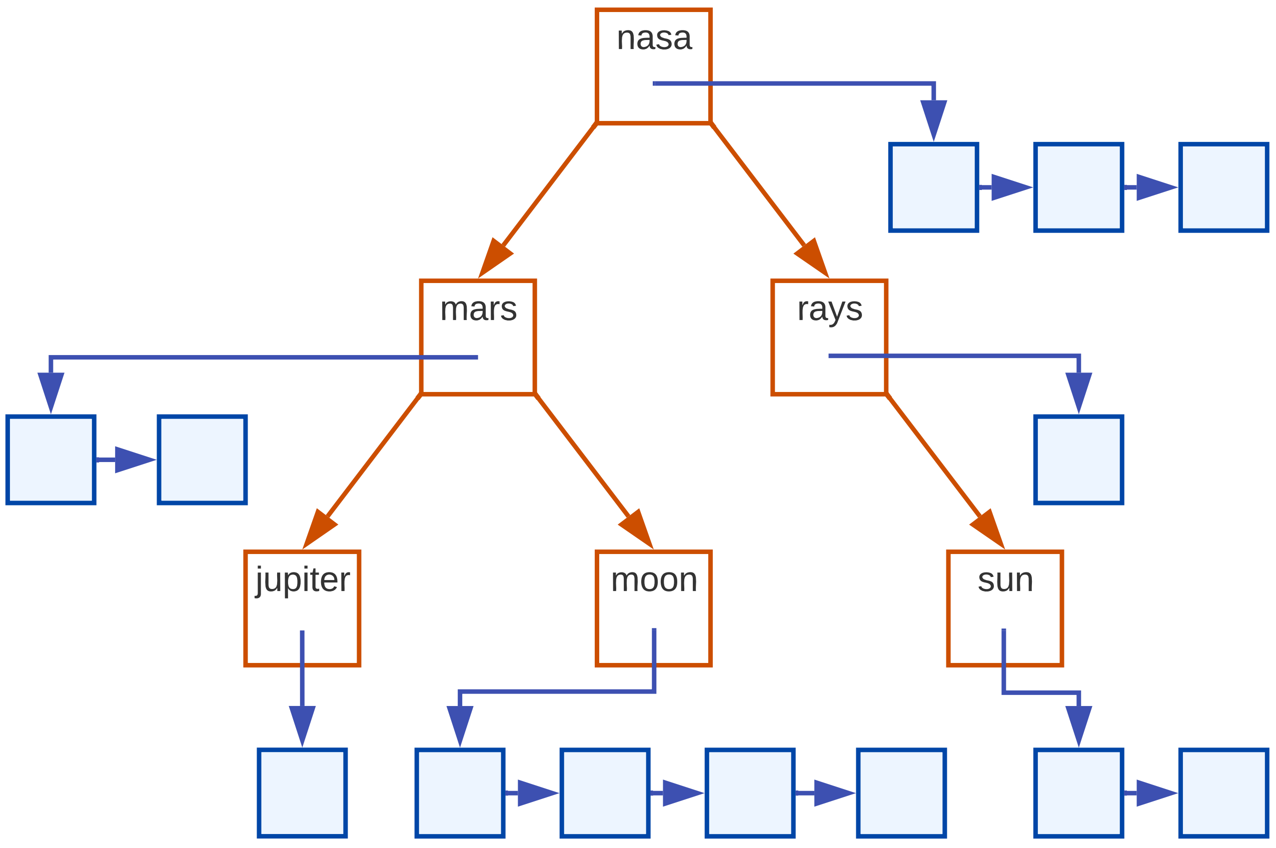
```
nasa.txt
news1.txt
file11.txt
mixed.txt
planets.txt
file21.txt
info31.txt
```

An inverted index is a binary search tree where each node contains a word and a corresponding file list, which is a linked list. Each node of the file list contains the filename of a file that contains the word, as well as the term frequency of the word in that file. The inverted index tree must be ordered alphabetically (ascending) by word, and each file list must be sorted alphabetically (ascending) by filename.

Here is an example of an inverted index node and its corresponding file list:



The full inverted index will contain many inverted index nodes arranged in a binary search tree (ordered alphabetically by word), and each inverted index node will contain a file list. For example:



Normalisation

Before inserting words into your inverted index, you must **normalise** them by converting all letters to lowercase and removing the following punctuation marks **from the end** of the words:

- **.** (dot)
- **,** (comma)
- **:** (colon)
- **;** (semicolon)
- **?** (question mark)
- ***** (asterisk)

If there are multiple of the above punctuation marks at the end of a word, you should remove all of them. Punctuation marks at the start of a word or in the middle of a word should not be removed. Other punctuation marks should not be removed. If a word becomes empty after removing punctuation marks, then it should not count as a word (for the purposes of calculating tf) and it should not be inserted into the inverted index.

Here are some examples of normalisation:

Here are some examples of normalisation:

Word	Normalised word
Data	data
BSTs	bsts
algorithms.	algorithms
Why?	why
graphs*.	graphs
.NET	.net
unsw.edu.au.	unsw.edu.au
Sydney!	sydney!
.,!.,:;	.,!
new..........s	new..........s
*	(empty word)

printInvertedIndex

The `printInvertedIndex` function has the signature:

```
void printInvertedIndex(InvertedIndexBST tree, char *filename);
```

This function should print an inverted index to a file with the given filename. Each line of the output should begin with a word from the inverted index followed by a space-separated list of filenames with the tf value of each file in parentheses (using the format string `"%.7lf"`). Lines should be ordered alphabetically (ascending) by the initial word, and each list of files should be ordered alphabetically (ascending) by filename.

Here is an example to demonstrate the expected format:

```
and file11.txt (0.0303030)
apparent file11.txt (0.0303030)
ascribed file11.txt (0.0303030) file21.txt (0.0625000)
attributed file11.txt (0.0303030) file21.txt (0.0625000)
been file11.txt (0.0303030)
changes file11.txt (0.0303030) file21.txt (0.0625000)
circle mixed.txt (0.1111111) planets.txt (0.2222222)
color file11.txt (0.0303030) file21.txt (0.0625000)
```

freeInvertedIndex

The `freeInvertedIndex` function has the signature:

```
void freeInvertedIndex(InvertedIndexBST tree);
```

This function should free all memory associated with the given inverted index.

Assumptions/Constraints/Notes

- All arguments are valid.
- Filenames and words are at most 100 characters long.
- Filenames do not contain whitespace characters.
- All filenames in the collection file are distinct.
- There is no limit on the number of filenames in the collection file.
- There is no limit on the number of words in a file.
- When we say "alphabetical order", we mean as determined by `strcmp`.
- All files required by the program (i.e., the collection file and all the files listed in the collection file) are valid readable/openable text files, and will be in the current working directory when the program is executed.
- The `printInvertedIndex` function should always behave as if the given file was empty or didn't exist. If the file had any content, that content should be removed before printing the inverted index to the file.
- We will use a tolerance of 10^{-6} to check tf values when testing `generateInvertedIndex` and `printInvertedIndex`. This means your answer for a tf value will be accepted if it is within 0.000001 of the expected value.

Part 2 - Information Retrieval

The goal of Part 2 is to implement an information retrieval engine that accepts one or more query terms and returns relevant documents.

searchOne

The `searchOne` function has the signature:

```
TfIdfList searchOne(InvertedIndexBST tree, char *searchWord, int D);
```

This function takes an inverted index (the same as that produced in Part 1), a **single** search word, and the total number of documents in the collection, and returns a list of all the files that contain that search word. Each node of the list should contain a filename and the tf-idf value of the word for that file. The list must be in *descending order* of tf-idf. Files with the same tf-idf should be ordered alphabetically by filename in *ascending order*.

searchMany

The `searchMany` function has the signature:

```
TfIdfList searchMany(InvertedIndexBST tree, char *searchWords[], int D);
```

This function takes an inverted index (as in `searchOne`), **one or more** search words, and the total number of documents in the collection, and returns a list of all the files that contain at least one of the given search words. Each node of the list should contain a filename and the summation of tf-idf values of all the matching search words for that file. The list must be in *descending order* of summation of tf-idf values. Files with the same tf-idf sum should be ordered alphabetically by filename in *ascending order*.

The `searchWords` array is a NULL-terminated array of strings. For example:

```
char *words[] = { "nasa", "mars", "earth", NULL };
TfIdfList list = searchMany(index, words, 7);
```

Example

Suppose that the lists returned by `searchOne` for two different words are:

```
word 1: file11.txt (0.4), file12.txt (0.3), file13.txt (0.1)
word 2: file13.txt (0.4), file12.txt (0.3), file14.txt (0.2)
```

Then a search for the same two words together using `searchMany` should result in the following list:

```
file12.txt (0.6), file13.txt (0.5), file11.txt (0.4), file14.txt (0.2)
```

freeTfIdfList

The `freeTfIdfList` function has the signature:

```
void freeTfIdfList(TfIdfList list);
```

This function frees all memory associated with the given list.

Assumptions/Constraints/Notes

- All arguments will be valid.
- All given search words will already be normalised.
- All given search words to `searchMany` will be unique.
- You should use the `log10` function from the `<math.h>` library for calculating idf.
- When we say "alphabetical order", we mean as determined by `strcmp`.
- Since the total number documents in the collection is given as an argument, you need not and **should not** use the collection file. No files should be opened in this part of the assignment.
- We will use a tolerance of 10^{-6} to check tf-idf values when testing `searchOne` and `searchMany`. This means your answer for a tf-idf value will be accepted if it is within 0.000001 of the expected value.

Testing

We have provided two test directories `exmp1` and `exmp2`. Each directory contains test data, a test program called `testInvertedIndex.c` and expected output. You should examine `testInvertedIndex.c` to see what the tests do and how the tests call your functions.

To test your code, you should first run `make` in your main assignment 1 directory, which compiles your code and creates an executable called `testInvertedIndex` in each of the test directories, and then change into one of the directories to run the executable. Each test program takes a test number as a command-line argument.

```
$ make
...
$ cd exmp1
$ ./testInvertedIndex 1
$ ./testInvertedIndex 2
```

Each test produces a file ending with `.out`. You can compare this with the corresponding `.exp` file, which contains the expected output for that test. For example, test 2 of `exmp1` produces a file called `mars.out`, and the expected output is in `mars.exp`.

You can create your own tests by either extending the given test programs (by adding a new case to the switch statement in `main()` and a new test function) or by creating your own test directory with your own data and testing program. If you do the latter, you'll need to add instructions for compiling the new test program to the `Makefile`. The `Makefile` includes details on how to do this.

Implementation Details

This section is intended to address questions such as "can we do *X*?" and "can we use algorithm *Y*?".

You can implement your solution **any way you want** as long as it complies with the rest of the specification and the following restrictions:

- Your solution must not start or communicate with any external programs

Debugging

Debugging is an important and inevitable aspect of programming. We expect everyone to have an understanding of basic debugging methods, including using print statements, knowing basic GDB commands and running Valgrind. In the forum and in help sessions, tutors will expect you to have made a reasonable attempt to debug your program yourself using these methods before asking for help.

You can learn about GDB and Valgrind in the Debugging with GDB and Valgrind lab exercise.

Frequently Asked Questions

- **Are we allowed to create our own functions?** Of course.
- **Are we allowed to create our own `#defines` and structs?** Yes.
- **Are we allowed to modify `invertedIndex.h` in any way?** No.
- **Are we allowed to modify the structs provided in `invertedIndex.h`?** You are not allowed to modify `invertedIndex.h`, so no.
- **Are we allowed to change the signatures of the given functions?** No. If you change these, your code won't compile and we won't be able to test it.
- **What errors do we need to handle?** You should handle common errors such as `NULL` returned from `fopen` and `NULL` returned from `malloc` by printing an error message to `stderr` and terminating the program. You are not required to handle other errors.
- **Will we be assessed on our tests?** No. You will not be submitting any test files, and therefore you will not be assessed on your tests.
- **Are we allowed to share tests?** No. Testing is an important part of software development. Students that test their code more will likely have more correct code, so to ensure fairness, each student should independently develop their own tests.

Submission

You must submit the file `invertedIndex.c`. In addition, you can submit as many supporting files (`.c` and `.h`) as you want. Please note that none of your files should contain a main function, and you should not submit `invertedIndex.h` as you are not permitted to modify that file. You can submit via the command line using the `give` command:

```
$ give cs2521 ass1 invertedIndex.c your-supporting-files...
```

or you can submit via WebCMS. You can submit multiple times. Only your last submission will be marked. You can check the files you have submitted here.

WARNING:

After you submit, you **must** check that your submission was successful by going to your submissions page. Check that the timestamp is correct. If your submission does not appear under Last Submission or the timestamp is not correct, then resubmit.

Compilation

You must ensure that your final submission compiles on CSE machines. Submitting non-compiling code leads to extra administrative overhead and will result in a 10% penalty.

Every time you make a submission, a dryrun test will be run on your code to check that it compiles. Please ensure that your final submission successfully compiles, even for parts that you have not completed.

Assessment

This assignment will contribute 14% to your final mark.

Correctness

80% of the marks for this assignment will be based on the correctness of your code, and will be based on autotesting. Marks for correctness will be distributed as follows:

Part 1	generateInvertedIndex	40% of the correctness mark
	printInvertedIndex	10% of the correctness mark
	freeInvertedIndex	See **memory errors/leaks** section below
Part 2	searchOne	20% of the correctness mark
	searchMany	30% of the correctness mark
	freeTfIdfList	See **memory errors/leaks** section below

Efficiency

You must ensure that your program is reasonably efficient. Our autotests will apply a 3 second time limit to each normal execution (i.e., without `valgrind`) of `./testInvertedIndex`. If your program takes longer than 3 real-time seconds on a particular test, the test will be automatically terminated and you will receive 0 marks for that test.

The largest test (in terms of the size of the collection and files) will be about the same size as the `exmp2` directory, and you can assume that `searchMany` is given at most 20 search terms.

Memory errors/leaks

Additionally, you must ensure that your code does not contain memory errors or leaks, as code that contains memory errors is unsafe and it is bad practice to leave memory leaks in your program. Note that our tests will always call `freeInvertedIndex` and/or `freeTfIdfList` at the end of the program to free all memory associated with the inverted index/tf-idf list.

Submissions that contain any amount of memory errors or leaks will receive a penalty of 10% of the correctness mark, deducted from the attained mark. Specifically, there will be a separate memory error/leak check for each part, and the penalty will be 10% of the marks for that part. You can check for memory errors and leaks with the `valgrind` command:

```
$ valgrind -q --leak-check=full ./testInvertedIndex test-number > /dev/null
```

A program that contains no memory errors or leaks will produce no output from this command. You can learn more about memory errors and leaks in the Debugging with GDB and Valgrind lab exercise.

Style

20% of the marks for this assignment will come from hand marking of the readability of the code you have written. These marks will be awarded on the basis of clarity, commenting, elegance and style. The following is an indicative list of characteristics that will be assessed, though your program will be assessed wholistically so other characteristics may be assessed too (see the style guide for more details):

- Good separation of logic into files
- Consistent and sensible indentation and spacing
- Using blank lines and whitespace
- Using functions to avoid repeating code
- Decomposing code into functions and not having overly long functions

- Using comments effectively and not leaving planning or debugging comments

The course staff may vary the assessment scheme after inspecting the assignment submissions but it will remain broadly similar to the description above.

Originality of Work

This is an individual assignment. The work you submit must be your own work and only your work apart from the exceptions below. Joint work is not permitted. At no point should a student read or have a copy of another student's assignment code.

You may use any amount of code from the lectures and labs of the **current iteration** of this course. You must clearly attribute the source of this code in a comment.

You may use small amounts (< 10 lines) of general purpose code (not specific to the assignment) obtained from sites such as Stack Overflow or other publicly available resources. You must clearly attribute the source of this code in a comment.

You are not permitted to request help with the assignment apart from in the course forum, help sessions or from course lecturers or tutors.

Do not provide or show your assignment work to any other person (including by posting it publicly on the forum) apart from the teaching staff of COMP2521. When posting on the course forum, teaching staff will be able to view the assignment code you have recently submitted with give.

The work you submit must otherwise be entirely your own work. Submission of work partially or completely derived from any other person or jointly written with any other person is not permitted. The penalties for such an offence may include negative marks, automatic failure of the course and possibly other academic discipline. Assignment submissions will be examined both automatically and manually for such issues.

Relevant scholarship authorities will be informed if students holding scholarships are involved in an incident of plagiarism or other misconduct. If you knowingly provide or show your assignment work to another person for any reason, and work derived from it is submitted, then you may be penalised, even if the work was submitted without your knowledge or consent. This may apply even if your work is submitted by a third party unknown to you.