Assignment

Simple Graph Structure-Based Search Engine

Changelog

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

- `[11/01 15:00]` Assignment released
- `[11/01 15:22]` Minor typos fixed
- `[11/01 16:31]` Part weights adjusted
- `[14/01 19:08]` Clarified that part 2 doesn't need to be implemented using a BST
- `[20/01 13:25]` Clarified what a "matching search term" is in part 3
- `[26/01 18:41]` Add in-text clarification that part 3 will not have access to `collection.txt` or any URL files

Admin

Marks contributes 25% towards your final mark (see Assessment section for more details)

Submit see the Submission section

Deadline 5pm on Friday of Week 5

Late penalty 0.2% per hour or part thereof deducted from the attained mark, submissions later than 10am Sunday of Week 5 not accepted.

Prerequisite Knowledge

- Abstract Data Types
- Linked Lists (carry-over from COMP1511)
- Trees & Binary Search Trees
- Graphs
  - Graph Basics
  - Graph Representations
  - Graph ADT
  - Directed Graphs
- Sorting (not absolutely necessary, but helpful)

Background

In this assignment, your task is to implement a simple search engine using the well-known PageRank algorithm (simplified for this assignment, of course!). You should start by reading the Wikipedia article on PageRank (up to the section Damping Factor). These topics will be discussed later, in lectures.

The main focus of this assignment is to build a graph structure, calculate PageRanks, and rank pages based one these values.

Mock web pages

To make things easier for you, you don't need to spend time crawling, collecting and parsing web pages for this assignment. Instead, you will be provided with a collection of mock web pages in the form of plain text files.

Each page has two sections:

- Section 1 contains URLs representing outgoing links. The URLs are separated by whitespace, and may be spread across multiple lines.
- Section 2 contains the actual content of the web page, and consists of zero or more words. Words are separated by whitespace, and may be spread across multiple lines.

Here is an example page:

```
#start Section-1

url2  url34  url1 url26
  url52 url21
url74  url6 url82

#end Section-1

#start Section-2

Mars has long been the subject of human interest. Early telescopic observations
revealed color changes on the surface that were attributed to seasonal vegetation
and apparent linear features were ascribed to intelligent design.

#end Section-2
```

Sections will always be delimited by lines containing `#start Section-1`, `#end Section-1`, `#start Section-2` and `#end Section-2`. There will be no other characters on these lines (no leading or trailing whitespace). There may be lines containing whitespace before or after any section.

Note that it is completely valid for Section 1 to be empty - this would mean that the page does not have any outgoing links.

Resources to help you get started

- How to Implement the Assignment (hints), to be discussed in lectures.
- Examples of how to implement certain things, including file I/O are provided here.

Setting Up

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

```
$ unzip /web/cs2521/23T0/ass/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.

If you've done the above correctly, you should now have the following files:

`Makefile`	a set of dependencies used to control compilation
`pagerank.c`	a stub file for Part 1
`invertedIndex.c`	a stub file for Part 2
`searchPagerank.c`	a stub file for Part 3
`test1`, `test2` and `test3`	directories containing test data

Note that in this assignment, you are permitted to create as many supporting files as you like. This allows you to compartmentalise your solution into different files. For example, you could implement a Graph ADT in `Graph.c` and `Graph.h`. You are encouraged to use a `Makefile` for this assignment, to make compilation faster and easier for you. The provided Makefile should be enough, for most cases.

File Reading

You will be required to read files in this assignment. In case you are unfamiliar with file IO, we have provided some sample programs in the `file_io_demo` directory that demonstrates the usage of the main file IO functions `fopen`, `fclose`, `fscanf` and `fgets`. These are the only file IO functions you will need to use in this assignment.

Part 1 - PageRank Calculation

Your task is to implement a program in `pagerank.c` that reads data from a given collection of pages in the files `collection.txt`, builds a graph structure using either an adjacency list or adjacency matrix from this data, and calculates the PageRank for every URL in the collection using the algorithm described below.

The URLs in `collection.txt` are separated by whitespace, and may be spread across multiple lines. Here is an example `collection.txt`:

```
url11 url21 url22
    url23
 url31 url24 url34
```

For each URL in the collection, there will be a text file that contains the associated page. To get the name of the file, simply append `.txt` to the URL. For example, the page associated with `url24` is contained in `url24.txt`.

You need to read all the pages and build a graph structure using a graph representation **of your choice**. Then, you must use the algorithm below to calculate the Weighted PageRank for each page.

PageRank(d, $diffPR$, $maxIterations$**)**

- Read pages from the collection in file `collection.txt` and build a graph structure using a graph representation of your choice
- N = number of URLs in the collection
- For each URL p_i in the collection
  - $PR(p_i, 0) = \frac{1}{N}$
- $iteration = 0$
- $diff = diffPR$
- While $iteration < maxIterations$ and $diff \geq diffPR$
  - $t = iteration$
  - For each URL p_i in the collection

$$PR(p_i, t+1) = \frac{1-d}{N} + d \times \sum_{p_j \in M(p_i)} \frac{PR(p_j, t)}{L(p_j)}$$

  - $diff = \sum_{i=1}^{N} |PR(p_i, t+1) - PR(p_i, t)|$
  - $iteration$++

In the above algorithm:

- $M(p_i)$ is a set of pages with outgoing links to p_i (ignore self-loops and parallel edges)
- $L(p_j)$ is the out-degree of the page p_j
- t and $t+1$ correspond to iteration numbers. For example, $PR(p_i, 2)$ means "the PageRank of p_i after iteration 2".

`pagerank.c` must take three command-line arguments: **d** (damping factor), **diffPR** (sum of PageRank differences), **maxIterations** (maximum number of iterations), and using the algorithm described in this section, calculate the PageRank for every page in the collection.

Here is an example of how the program is run:

```
$ ./pagerank 0.85 0.00001 1000
```

The program should output (to a file named `pagerankList.txt`) a list of pages, one per line, along with some information about each page. Each line should begin with the URL of a page, followed by its outdegree and PageRank (using the format string `"%.7lf"`). The values should be comma-separated. Lines should be sorted in descending order by PageRank.

Here is an example `pagerankList.txt`

```
url31, 3, 0.2623546
url21, 1, 0.1843112
url34, 6, 0.1576851
url22, 4, 0.1520093
url32, 6, 0.0925755
url23, 4, 0.0776758
url11, 3, 0.0733884
```

Use format string `"%.7f"` to output PageRank values. Please note that your PageRank values might be slightly different to those provided in the sample tests. This might be due to the way you carry out calculations. However, make sure that your PageRank values match the expected values to the first 6 decimal places. For example, if the expected value was `0.0776758`, your value could be `0.077675x` where `x` could be any digit.

All the sample files were generated using the following command:

```
$ ./pagerank 0.85 0.00001 1000
```

Assumptions/Constraints

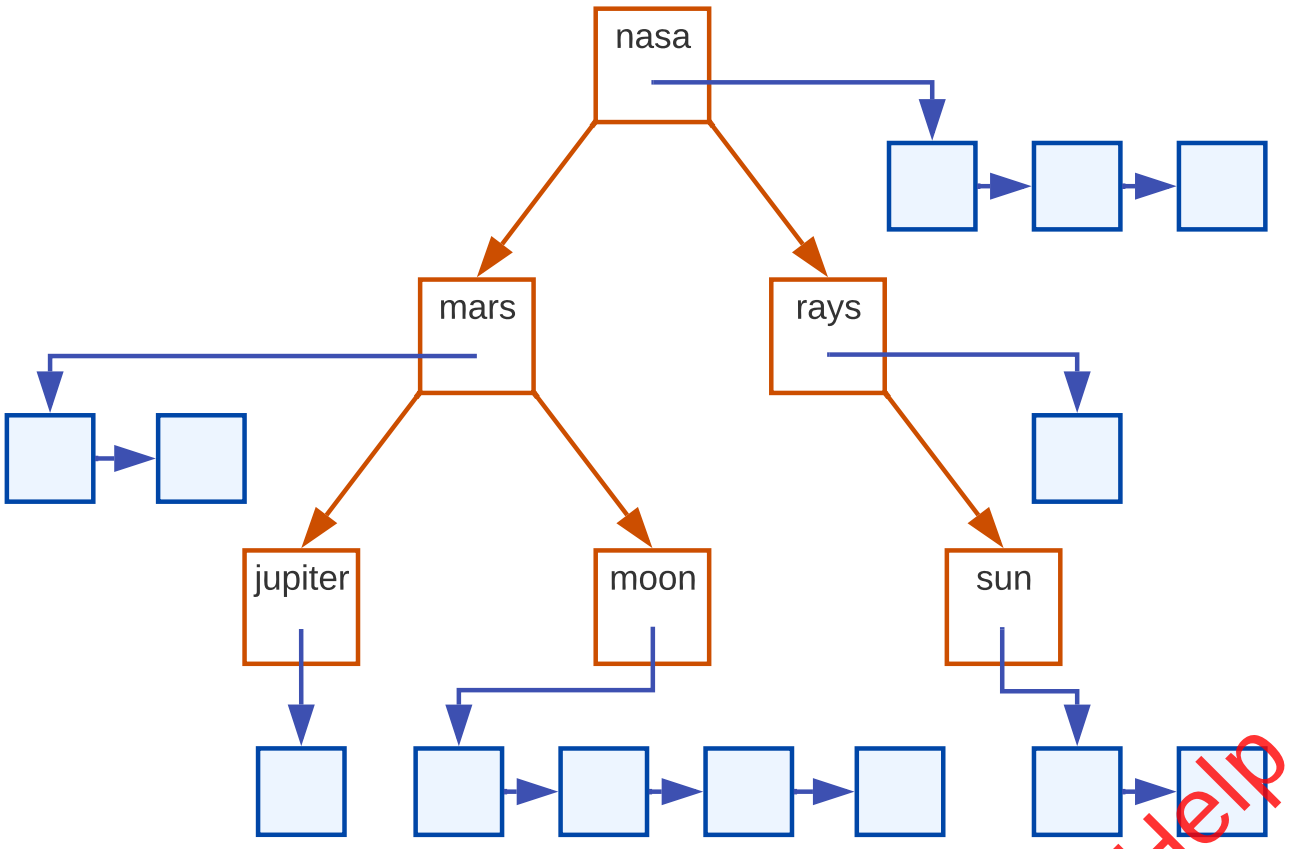
- All URLs in `collection.txt` are distinct.
- URLs consist of alphanumeric characters only, and are at most 100 characters in length.
- There is no limit on the number of filenames in the collection file.
- There is no limit on the number of words in a file.
- URLs **do not** necessarily start with `url`. For example, `pretzels.txt` is a valid filename, corresponding to the URL `pretzels`.
- Pages may have links to themselves (self links) and multiple links to the same page (duplicate links). You must ignore self links, and duplicate links should be treated as a single link.
- There will be no missing pages. This means each URL referenced in `collection.txt` or as an outlink in a page will have a text file associated with it that is appropriately named. Also, every URL referenced as an outlink in a page will be listed in `collection.txt`.
- The provided command-line arguments will be valid. This means d will be a double between 0 and 1, $diffPR$ will be a double greater than 0, and $maxIterations$ will be a positive integer.
- All text files required by the program will be in the current working directory when the program is executed.

Part 2 - Inverted Index

Your task is to implement a program in `invertedIndex.c` that reads data from a given collection of pages in `collection.txt` and generates an "inverted index" that provides a sorted list (set) of URLs for every word in a given collection of pages.

One possible (and recommended) implementation of 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. The inverted index tree must be ordered alphabetically (ascending) by word, and each file list must be sorted alphabetically (ascending) by filename.

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:



In this diagram, each of the blue linked lists contains the URLs of the pages which contain the corresponding word. Note that this is only a suggestion for how to implement this part. As long as your implementation works, and doesn't exceed the time limit of the autotests, you can receive full marks.

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 be inserted into the inverted index.

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)

Here is an example of how the program is run:

```
$ ./invertedIndex
```

Your program should print an inverted index to a file named `invertedIndex.txt`. Each line of the output should begin with a word from the inverted index followed by a space-separated list of filenames. 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:

```
design url2 url25 url31 url61
mars url101 url25 url31
vegetation url31 url61
```

On each line, a word and all the URLs should be separated by one or more spaces. The testing program will ignore additional spaces.

Assumptions/Constraints

- Relevant assumptions from previous parts apply.
- 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.
- Words are at most 1000 characters in length.

Part 3 - Simple Search Engine

Your task is to implement a program in `searchPagerank.c` that takes one or more search terms and outputs *relevant* URLs.

The program must make use of the data available in two files: `invertedIndex.txt` and `pagerankList.txt`. **All other text files are not relevant for this task and will not be present during testing (both the collection file, and all the individual URLs).**

`invertedIndex.txt` contains data about what pages contain what words, as generated in Part 2. The format of this file is the same as the expected format of the output produced by your `invertedIndex` program in Part 2.

`pagerankList.txt` contains information about pages including their URL, outdegree and PageRank. The format of this file is the same as the expected format of the output produced by your `pagerank` program in Part 1. To simplify parsing, you may assume that the URL, out-degree, and PageRank are separated by **`", "`** - that is, a comma followed by one space.

The program should take search terms as command-line arguments, find pages with **one or more** matching search terms and output the top 30 pages in descending order of the number of matching search terms to `stdout`. By matching search terms, we don't consider duplication; if a URL has `mars` appear in section 2 five times, and nothing else, searching for `mars` and `design` would count one matched search term, **not** five. You will **not have access** to any individual URL file for part 3, so you won't be able to determine how many occurrences there are regardless. Pages with the same number of matching search terms should be sorted in descending order by their PageRank.

Here is an example output:

```
$ ./searchPagerank mars design
url31
url25
```

Note that unlike more sophisticated search engines, your simple search engine program **should not** treat variants of a word as the same word. For example, "observe" and "observing" should be treated as different words. Words and their plural forms, such as "planet" and "planets," should also be treated as different words.

> We will test this part independently from your solution to Part 1 and Part 2. This means that we will use a reference solution with a correct `pagerank.c` and `invertedIndex.c` to generate `pagerankList.txt` and `invertedIndex.txt` to use with your program in this part.

Assumptions/Constraints

- Relevant assumptions from previous parts apply.
- Search terms will be unique.
- At least one search term will be given to the `searchPagerank` program.
- Search terms will consist entirely of lowercase letters.
- Lines in `invertedIndex.txt` are at most 1000 characters in length. (Note that this implies that words in `invertedIndex.txt` cannot be longer than 1000 characters.)
- Words in `invertedIndex.txt` will consist entirely of lowercase letters.
- The words at the start of each line in `invertedIndex.txt` will be unique (obviously).
- Each line of `invertedIndex.txt` contains at least one URL (obviously).
- The URLs on each line in `invertedIndex.txt` will be unique (obviously).
- It is not guaranteed that all the search terms will be in `invertedIndex.txt`. Search terms that are not in `invertedIndex.txt` should be ignored, since there are no pages that contain them.
- `pageRankList.txt` will contain valid URLs, out-degrees, and PageRanks.

- All URLs in `invertedIndex.txt` will be listed in `pageRankList.txt`.

Frequently Asked Questions

- **Are we allowed to `#include` any additional libraries?** You may only `#include` standard libraries, or your own header files.
- **Are we allowed to use *[insert library function here]*?** You can use any function provided by the standard libraries.
- **What are "standard" libraries?** Any library that compiles without either copying the source files from an external resource, or requiring linker options during compilation. If you aren't sure, ask on the forum.
- **The specification states that we can assume something. Do we need to check whether this assumption is satisfied?** No. If the specification says you can assume something, you don't need to check it.
- **What errors do we need to handle?** You should handle common errors such as `NULL` returned from `fopen`, or `NULL` returned from `malloc`, by printing an error message to `stderr` and terminating the program. You are not required to handle other errors.
- **What invalid inputs do we need to handle?** You are not required to handle invalid inputs, such as an invalid number of command-line arguments, invalid command-line arguments, or files which do not satisfy the given assumptions.
- **Are we allowed to share tests?** No. Testing and developing test cases is an important part of software design. Students who test their code more will likely have more correct code, so to ensure fairness, each student should independently develop their own tests.

Testing

We have provided three test cases for you to use. These test cases are very basic, but will give you a general idea whether your solution is working or not. **You are expected to do your own testing.**

To run these tests, a bash script is provided, which will change into each directory `test*` and run the test. To use it, assuming the script `runtests` is in your current directory, do the following:

```
$ ./runtests
========== Running test 1 ==========
========== Test 1: ./pagerank 0.85 0.00001 1000 ==========
Outputs match!
========== Test 1: ./invertedIndex ==========
Outputs match!
========== Test 1: ./searchPagerank mars design ==========
Outputs match!
========== Running test 2 ==========
========== Test 2: ./pagerank 0.85 0.00001 1000 ==========
Outputs match!
========== Test 2: ./invertedIndex ==========
Outputs match!
========== Test 2: ./searchPagerank volcano pretzel hello design ==========
Outputs match!
========== Running test 3 ==========
========== Test 3: ./pagerank 0.85 0.00001 1000 ==========
Outputs match!
========== Test 3: ./invertedIndex ==========
Outputs match!
========== Test 3: ./searchPagerank volcano nasa ==========
Outputs match!

Total  tests: 9
Passed tests: 9
Failed tests: 0
Good job!
```

If you would only like to run specific tests, you can list them as arguments to the script. For example, to run only test 1 and test 3:

```
$ ./runtests 1 3
========== Running test 1 ==========
========== Test 1: ./pagerank 0.85 0.00001 1000 ==========
Outputs match!
========== Test 1: ./invertedIndex ==========
Outputs match!
========== Test 1: ./searchPagerank mars design ==========
Outputs match!
========== Running test 3 ==========
========== Test 3: ./pagerank 0.85 0.00001 1000 ==========
Outputs match!
========== Test 3: ./invertedIndex ==========
Outputs match!
========== Test 3: ./searchPagerank volcano nasa ==========
Outputs match!

Total  tests: 6
Passed tests: 6
Failed tests: 0
Good job!
```

You may also make your own tests, by creating a directory starting with `test`. Any directory starting with `test` will be included as a test when you run the `runtests` script. Testing Part 1 manually may be difficult, as the PageRanks aren't easy to calculate by hand, but you may still want to test Parts 2 and 3 using your own test directories. If a `*.exp` file is missing, the script will simply skip that part for that test. A typical test directory will look like this:

```
test4/
    collection.txt
    (url files)
    pagerankList.exp
    invertedIndex.exp
    searchTerms.txt
    searchPagerank.exp
```

Testing Part 3 requires an extra file beyond the expected output. The `searchTerms.txt` file contains the terms to search for, separated by spaces, on a single line. Take a look at the three provided test cases for how this works.

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, and you should be able to explain your debugging attempts.

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

> NOTE:
>
> If you want to use valgrind, you will need to remove the `-fsanitize=address,leak,undefined` option from the compilation command, as valgrind is not compatible with sanitizers. You can do this by opening the Makefile and changing `CFLAGS1` to `CFLAGS0` in the definition of `CFLAGS`. However, make sure you change it back afterwards, since we will be using sanitizers in our autotesting.

Submission

You must submit the three files: `pagerank.c`, `invertedIndex.c`, and `searchPagerank.c`. In addition, you can submit as many supporting files (`.c` and `.h`) as you want. Please note that none of your supporting files should contain a main function - only `pagerank.c`, `invertedIndex.c`, and `searchPagerank.c` should contain a main function.

You can submit via the command line using the `give` command:

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

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

Compilation

You must ensure that your final submission compiles on CSE machines. Submitting non-compiling code causes extra administrative overhead and will result in a 10% penalty. If the fix is not a small one, and is more fundamental to your program, then you will get at most 50% of the automark, at the discretion of your tutor.

Here, we describe how we will compile your files during autotesting.

To test Part 1, we will rename `invertedIndex.c` and `searchPagerank.c` so they do not end with `.c` and then compile all the remaining `.c` files together as follows to produce an executable called `pagerank`.

```
$ mv invertedIndex.c invertedIndex.c~
$ mv searchPagerank.c searchPagerank.c~
$ clang -Wall -Werror -g -fsanitize=address,leak,undefined *.c -o pagerank -lm
$ # run tests
```

To test Part 2, we will rename `pagerank.c` and `searchPagerank.c` so they do not end with `.c` and then compile all the remaining `.c` files together as follows to produce an executable called `invertedIndex`.

```
$ mv pagerank.c pagerank.c~
$ mv searchPagerank.c searchPagerank.c~
$ clang -Wall -Werror -g -fsanitize=address,leak,undefined *.c -o invertedIndex -lm
$ # run tests
```

To test Part 3, we will rename `pagerank.c` and `invertedIndex.c` so they do not end with `.c` and then compile all the remaining `.c` files together as follows to produce an executable called `searchPagerank`.

```
$ mv pagerank.c pagerank.c~
$ mv invertedIndex.c invertedIndex.c~
$ clang -Wall -Werror -g -fsanitize=address,leak,undefined *.c -o searchPagerank -lm
$ # run tests
```

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

Assessment

This assignment will contribute 25% to your final mark. The assignment mark will be made up from the following components:

Correctness

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

Part 1	30%
Part 2	30%
Part 3	20%

Memory errors/leaks

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.

The compilation flags that we use will cause the program to print an error message and exit whenever a memory error or leak occurs, failing the autotest, so you must ensure that your program is completely free of memory errors and leaks.

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

Note that **all** the code you submit will be assessed for style, including code that does not work. Therefore, you should get into the habit of writing good code from the very beginning, instead of writing bad code and then improving it just before submitting.

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.
