Assignment 2

Simple Search Engines

Changelog

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

- `[25/10 16:00]` Assignment released
- `[28/10 18:00]` Released resources
- `[31/10 00:00]` Corrected expected output files in `part3/02` to use 7 decimal places instead of 6
- `[07/11 23:30]` Updated the `find` commands in the Makefile to prevent plain text files in `part1`, `part2` and `part3` directories from being overwritten
  - Download the new `Makefile` here
- `[11/11 09:00]` Updated the dryrun to include sample correctness tests
- `[15/11 09:00]` Corrected a typo in the Part 3 background example - the minimum possible total scaled footrule distance is 1.4, not 1.1.

Admin

Marks	contributes 20% towards your final mark (see Assessment section for more details)
Submit	see the Submission section
Deadline	8pm on Friday of Week 10
Late penalty	0.2% per hour or part thereof deducted from the attained mark, submissions later than 5 days not accepted

Prerequisite Knowledge

- Abstract Data Types
- 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 a simplified version of the well-known Weighted PageRank algorithm. You should start by reading the Wikipedia article on PageRank (up to the section Damping Factor).

The main focus of this assignment is to build a graph structure from a set of web pages, calculate Weighted PageRanks and rank pages.

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.

Setting Up

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

```
$ unzip /web/cs2521/22T3/ass/ass2/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
`searchPageRank.c`	a stub file for Part 2
`scaledFootrule.c`	a stub file for Part 3
`part1`, `part2` and `part3`	directories containing test data for Part 1, Part 2 and Part 3 respectively

First, compile the original version of the files using the `make` command. This will produce three executables: `pageRank`, `searchPageRank` and `scaledFootrule`, and copy the executables to the directories in `part1`, `part2` and `part3`. When you test your program, you will need to change into one of these directories first.

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`. 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 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 - Weighted PageRanks

Resources

- How to Implement Part 1 (hints)
- Weighted PageRank Algorithm (paper)
- How to Calculate W^{out}

Your task is to implement a program in `pageRank.c` that calculates Weighted PageRanks for a collection of pages.

The URLs of the pages in the collection are given in `collection.txt`. 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 url32 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 `url11` is contained in `url11.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

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

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

In the above algorithm:

- $M(p_i)$ is a set of pages with outgoing links to p_i (ignore self-loops and parallel edges)
- $W^{in}_{(p_j,p_i)}$ and $W^{out}_{(p_j,p_i)}$ are defined in the paper above (see the link "Weighted PageRank Algorithm")
- t and $t+1$ correspond to iteration numbers. For example, $PR(p_i, 2)$ means "the PageRank of p_i after iteration 2".

Note: When calculating $W^{out}_{(p_j,p_i)}$, if a page k has zero outdegree (zero outlinks), O_u and O_p should be 0.5, not zero. This will avoid issues related to division by zero. **Note that this does not apply to the indegree.**

`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 Weighted PageRank for every page in the collection.

Here is an example of how the program is run:

```
$ ./pageRank 0.85 0.00001 1000
```

Each of the expected output files in the provided files were generated using the above command.

The program should output (to `stdout`) 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 Weighted PageRank (using the format string `"%.7lf"`). The values should be space-separated. Lines should be sorted in descending order by Weighted PageRank. Pages with the same Weighted PageRank should be sorted in increasing alphabetical order by URL.

Here is an example output. Note that this was not generated from any of the provided test files, **this is just an example** to demonstrate the output format.

```
url31 3 0.0724132
url21 1 0.0501995
url11 3 0.0449056
url22 4 0.0360005
url34 1 0.0276197
url23 3 0.0234734
url32 1 0.0232762
```

We will use a tolerance of 10^{-6} to check PageRank values when automarking. This means your answer for a PageRank value will be accepted if it is within 0.000001 of the expected value.

Testing

We have provided some test data for you to test Part 1 in the `part1` directory. Each directory in `part1` contains the following files:

`collection.txt` a file that contains a list of relevant URLs

`exp.txt` a file that contains the expected output for this collection of URLs

`log.txt` a file that contains information that may be useful for debugging

Here is an example of how to test Part 1.

```
# assuming you are currently in your assignment 2 directory
$ make
...
$ cd part1/01
$ ./pageRank 0.85 0.00001 1000
...
# compare the output with the expected output in exp.txt
```

The test cases we have provided are very basic - you are expected to develop your own test cases that cover different scenarios. Some scenarios you could test include, but are not limited to: (1) different number of URLs, (2) different values of d, $diffPR$ and $maxIterations$, and (3) URLs with the same PageRank. Carefully read this section of the spec (including the assumptions/constraints below) to find other scenarios that you could test.

To create your own test data, create a new directory in the `part1` directory. Inside this directory, create a `collection.txt` file and create appropriately formatted `.txt` files for each URL in the collection file (see the Background section for the format of pages).

If your program is producing different PageRanks from the expected output and you are not sure why, you can check `log.txt` which contains some expected intermediate values (such as W^{in} and W^{out} for each link).

Assumptions/Constraints

- All URLs in `collection.txt` are distinct.
- URLs consist of alphanumeric characters only, and are at most 100 characters in length.
- URLs **do not** necessarily start with `url`.
- 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.
- When we say "alphabetical order", we mean as determined by `strcmp`.
- All text files required by the program will be in the current working directory when the program is executed.

Part 2 - 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.**

`invertedIndex.txt` contains data about what pages contain what words. Each line of the file contains one word followed by an alphabetically ordered list of URLs of pages that contain that word. Lines are ordered alphabetically by the initial word. Here is an example `invertedIndex.txt`:

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

`pageRankList.txt` contains information about pages including their URL, outdegree and Weighted PageRank. The format of this file is the same as the expected format of the output produced by your `pageRank` program in Part 1.

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`. Pages with the same number of matching search terms should be sorted in descending order by their Weighted PageRank. Pages with the same number of matching search terms *and* the same Weighted PageRank should be sorted in increasing alphabetical order by URL.

Here is an example output. Note that this was not necessarily generated from any of the provided test files, **this is just an example** to demonstrate the output format.

```
$ ./searchPageRank mars design
url31
url25
url2
url101
```

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.

Testing

We have provided some test data for you to test Part 2 in the `part2` directory. Each directory in `part2` contains `invertedIndex.txt` and `pageRankList.txt`, which are formatted as described above.

Here is an example of how to test Part 2.

```
# assuming you are currently in your assignment 2 directory
$ make
...
$ cd part2/01
$ ./searchPageRank mars design
...
# now verify the output yourself
```

You are expected to develop your own test cases that cover different scenarios and verify the output of your program yourself. To create your own test data, create a new directory in the `part2` directory and inside this directory, create appropriately formatted `invertedIndex.txt` and `pageRankList.txt` files.

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, outdegrees and PageRanks.
- All URLs in `invertedIndex.txt` will be listed in `pageRankList.txt`.

Part 3 - Hybrid/Meta Search Engine using Rank Aggregation

Background

Rank Aggregation

Suppose that τ_1, τ_2, ... , τ_k are search results (rankings) obtained from k different sources (e.g., search engines). We want to aggregate these results into a single search result that agrees with the original results as much as possible.

In other words, if C is the set of all pages in the original rankings, then we want to find a permutation of C that agrees with the original rankings (τ_1, τ_2, ... , τ_k) as much as possible.

Clearly, some permutations of C will agree more with the original rankings than others. For example, suppose we have the following two rankings that were obtained from two different sources:

τ_1	τ_2
url1	url1
url4	url5
url5	url4
url2	url3
url3	

Now consider the following two permutations of C:

C_1	C_2
url1	url3
url4	url2
url5	url5
url2	url4
url3	url1

Clearly, C_1 agrees more with the original rankings than C_2, since in both τ_1 and τ_2, url1 is the first result and url3 is the last result.

Scaled Footrule Rank Aggregation

One way to quantify the level of agreement or disagreement between an aggregate ranking and the original rankings is to consider how the position of each page in the aggregate ranking differs from its position in the original rankings. This metric is called the total scaled footrule distance. The aggregate ranking with the minimum total scaled footrule distance is the one that agrees the most with the original rankings.

Let C be the set of pages to be ranked, and F be the set of positions. For each page c and position p, the cost of placing page c at position p in the aggregate ranking (called the **scaled footrule distance**) is given by the following formula:

$$W(c, p) = \sum_{i=1}^{k} \left| \frac{\tau_i(c)}{|\tau_i|} - \frac{p}{n} \right|$$

where

- k is the number of rankings
  - and τ_1, τ_2, ..., τ_k are the original rankings
- n is the number of pages in C
- $|\tau_i|$ is the number of pages in τ_i
- $\tau_i(c)$ is the position of c in τ_i (between 1 and $|\tau_i|$)

The **total scaled footrule distance** of a given aggregate ranking a is then:

$$D = \sum_{c \in C} (W(c, a(c)))$$

where $a(c)$ is the position of c in the aggregate ranking.

Consider the following example with **two** original rankings (the table on the left), **five** total pages, and one possible aggregate ranking (the table on the right):

size of τ_1 is 5 *size of τ_2 is 4*

	τ_1	τ_2
1	url1	url3
2	url3	url2
3	url5	url1
4	url4	url4
5	url2	

c	p	W(c, p) for τ_1	W(c, p) for τ_2	W(c, p)
url1	1	|1/5 – 1/5|	|3/4 – 1/5|	0.55
url2	3	|5/5 – 3/5|	|2/4 – 3/5|	0.50
url3	2	|2/5 – 2/5|	|1/4 – 2/5|	0.15
url4	5	|4/5 – 5/5|	|4/4 – 5/5|	0.20
url5	4	|3/5 – 4/5|		0.20

The total scaled footrule distance of this particular aggregate ranking is 1.6, which is the sum of all the orange cells (and also the sum of all the yellow cells). However, this is not the most optimal aggregate ranking, since the minimum possible total scaled footrule distance is 1.4. Note that since `url5` does not appear in τ_2, we don't calculate its distance for τ_2.

Task

Resources

- How to Get Started with Part 3

Your task is to implement a program in `scaledFootrule.c` that combines search results (rankings) from multiple sources using Scaled Footrule Rank Aggregation (described above).

The program should take the names of files containing search results as command-line arguments, and output (to `stdout`) the minimum total scaled footrule distance (using the format `"%.7lf"`) on the first line, followed by the corresponding aggregated list, with one URL per line.

We will use a tolerance of 10^{-6} when automarking. This means your minimum scaled footrule distance will be accepted if it is within 0.000001 of the expected value.

Note: You are not permitted to adapt or copy algorithms from the Internet for this task.

Efficiency

There are many different ways to assign positions to the given pages. When $n = 5$, there are $5! = 25$ possibilities. When $n = 10$, there are $10! = 3,628,800$ possibilities. A very simple but inefficient approach would be to use brute-force search and generate all possible permutations, calculate the total scaled-footrule distance for each, and find the permutation with the minimum scaled-footrule distance.

If you use such a brute-force search, you will receive a maximum of 65% for Part 3. However, if you implement a "smart" algorithm that avoids generating unnecessary alternatives, you can receive up to 100% for Part 3. You must document your algorithm such that your tutor can easily understand your logic, and clearly outline how you algorithm reduces the search space, otherwise you will not be awarded marks for your algorithm. Yes, it's only 35% of Part 3 marks, but if you try it, you will find it very challenging and rewarding.

Examples

Example 1

Suppose that `rankA.txt` and `rankB.txt` contain the following:

`rankA.txt`	`rankB.txt`
url4 url3 url2 url1 url5	url3 url2 url1 url4

This is how the program would be run:

```
$ ./scaledFootrule rankA.txt rankB.txt
```

The program should output the following:

```
1.1000000
url4
url3
url2
url1
url5
```

Example 2

There may be multiple aggregate lists that result in the minimum scaled footrule distance. In this case, you must only output one of them. Both will be considered correct.

Suppose that `rankA.txt` and `rankB.txt` contain the following:

rankA.txt	rankB.txt
url1	url3
url3	url2
url5	url1
url4	url4
url2	

There are two possible correct outputs:

```
1.4000000
url1
url3
url5
url2
url4
```

```
1.4000000
url1
url3
url5
url4
url2
```

Example 3

Your program should be able to handle more than two rank files, for example:

```
$ ./scaledFootrule google.txt bing.txt yahoo.txt myRank.txt
```

Testing

We have provided a few simple test cases for you to test Part 3 in the `part3` directory. Each directory in `part3` contains ranking files (which contain search results), and `exp.txt` which contains the expected output.

Here is an example of how to test Part 3.

```
# assuming you are currently in your assignment 2 directory
$ make
...
$ cd part3/01
$ ./scaledFootrule rankA.txt rankB.txt
...
# now verify the output yourself
```

Again, you are expected to develop your own test cases to cover different scenarios, for example: (1) different numbers of URLs, (2) different numbers of files.

Assumptions/Constraints

- Relevant assumptions from previous parts apply.
- At least two ranking files will be given.
- All given ranking files will have different filenames.
- Each ranking file will contain at least one URL.
- No ranking file will contain two of the same URL.
- Each ranking file will contain exactly one URL per line, and no other text. Lines do not contain leading or trailing whitespace.

Frequently Asked Questions

- **Are we allowed to `#include` any additional libraries?** No. However, you are allowed to `#include` your own `.h` files.
- **Are we allowed to use *[insert library function here]*?** You can use any function provided by the `#include`d libraries.
- **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`, `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.
- **What is the point of Section 2 of the pages?** Section 2 is not used in the assignment, since we have already provided `invertedIndex.txt`.

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 recipes for each of the executables. 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`, `searchPageRank.c` and `scaledFootrule.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`, `searchPageRank.c` and `scaledFootrule.c` should contain a main function.

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

```
$ give cs2521 ass2 pageRank.c searchPageRank.c scaledFootrule.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.

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.

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

To test Part 1, we will rename `searchPageRank.c` and `scaledFootrule.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 searchPageRank.c searchPageRank.c~
$ mv scaledFootrule.c scaledFootrule.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 `scaledFootrule.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 scaledFootrule.c scaledFootrule.c~
$ clang -Wall -Werror -g -fsanitize=address,leak,undefined *.c -o searchPageRank -lm
$ # run tests
```

To test Part 3, 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 `scaledFootrule`.

```
$ mv pageRank.c pageRank.c~
$ mv searchPageRank.c searchPageRank.c~
$ clang -Wall -Werror -g -fsanitize=address,leak,undefined *.c -o scaledFootrule -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 20% 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	40%
Part 2	20%
Part 3	20% Note: A correct brute-force algorithm is worth 65% of this. You will only receive 100% of this if you devise a "smart" algorithm that avoids checking all possible permutations. See Part 3 for details.

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, 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.

COMP2521 22T3: Data Structures and Algorithms is brought to you by

the School of Computer Science and Engineering

at the University of New South Wales, Sydney.

For all enquiries, please email the class account at cs2521@cse.unsw.edu.au

CRICOS Provider 00098G