CPSC 457: Principles of Operating Systems

Assignment 4: deadlocks, CPU scheduling

Weight: 22%

Collaboration

Discussing the assignment requirements with others is a reasonable thing to do and an excellent way to learn. However, the work you hand in must ultimately be your work. This is essential for you to benefit from the learning experience and for the instructors and TAs to grade you fairly. Handing in work that is not your original work but is represented as such is plagiarism and academic misconduct. Penalties for academic misconduct are outlined in the university calendar.

Here are some tips to avoid plagiarism in your programming assignments.

1. Cite all sources of code you hand in that are not your original work. You can put the citations into comments in your program. For example, if you find and use code found on a website, include a comment that says, for example:

   # The following code is from https://www.quackit.com/python/tutorial/python_hello_world.cfm.

   Use the complete URL so that the marker can check the source.

2. A tool like chat-GPT can be used to improve small code blocks. For example, three lines of code. If you get help from code assistance like Chat-GPT, you should comment above the block of code you requested assistance on debugging or improving and cite the tool used to get that suggestion. Using a tool like chat-GPT to write the majority of your assignment requirements will be treated as plagiarism if found without citation, and with citation, it will be treated as 0 for the component the student did not complete. Code improvement of short length will get credit if commented/cited properly.
3. Citing sources avoids accusations of plagiarism and penalties for academic misconduct. **However, you may still get a low grade if you submit code not primarily developed by yourself. Cited material should never be used to complete core assignment specifications unless clearly approved. Before submitting, you can and should verify any code you are concerned about with your instructor/TA.**
4. Discuss and share ideas with other programmers as much as you like, but make sure that when you write your code, it is your own. A good rule of thumb is to wait 20 minutes after talking with somebody before writing your code. If you exchange code with another student, write code while discussing it with a fellow student, or copy code from another person's screen, this code is not yours.
5. **Collaborative coding is strictly prohibited. Your assignment submission must be strictly your code**. Discussing anything beyond assignment requirements and ideas is a strictly forbidden form of collaboration. This includes sharing code, discussing the code itself, or modelling code after another student's algorithm. **You can not use (even with citation) another student's code.**
6. Making your code available, even passively, for others to copy or potentially copy is also plagiarism.
7. We will look for plagiarism in all code submissions, possibly using automated software designed for the task. For example, see Measures of Software Similarity (MOSS - https://theory.stanford.edu/~aiken/moss/).
8. Remember, if you are having trouble with an assignment, it is always better to go to your TA and/or instructor for help rather than plagiarizing. A common penalty is an F on a plagiarized assignment.

Late Policy

All students will have 6 total days during the semester that can be used for none, one, some, or all the 6 assignments. Students can use these days all on one assignment, or some on each of the six.

Each 24-hour period late after an assignment deadline counts as one full day regardless of how many hours the assignment was late within that period. For example, deadlines are generally Fridays 11:59pm local time. That means an assignment submitted any time Saturday before 11:59pm local time will be considered as 1 day late and count against the students 5 total days.

As long as a student still has days left to use, their assignment will be graded without penalty. If a student has no more days left or their submission exceeds the days they have remaining, then they will receive a 0 grade for a late assignment. Students who use up late days on earlier assignments will not have them available for later assignments.

TAs will indicate the student's late day usage in grading feedback.

Due date is posted on D2L. Your D2L submission should include the files requested and a link to a Gitlab repository you used while completing the assignment with your TA added as a **Developer** role. Help with Gitlab Clone/Developer role access is available in D2L video.

Q1. Programming question – deadlock detection [50 marks]

For this question you will write a deadlock detection algorithm for a system state with a single instance per resource type. The input will be an ordered sequence of request and assignment edges. Your algorithm will start by initializing an empty system state (e.g. empty graph), and then process the edges one at a time. For each edge, your algorithm will update the system state (e.g. insert the edge into a graph), and then run a deadlock detection algorithm (e.g. topological sort). If a deadlock is detected after processing an edge, your algorithm will stop processing further edges and return results immediately.

Below is the signature of the **find_deadlock** function you need to implement:

```
struct Result {
      int index;
      std::vector<std::string> procs;
};
Result find_deadlock(const std::vector<std::string> & edges);
```

The parameter **edges** is an ordered list of strings, each representing an edge. The function returns an instance of **Result** containing two fields as described below.
Your function will start with an empty system state – by initializing an empty graph data structure. For each string **edges[i]** it will parse it to determine if it represents an assignment edge or request edge and update the graph accordingly. After inserting each edge into the

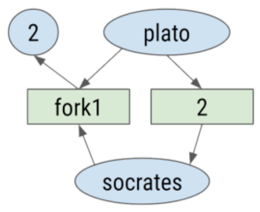
graph, the function will run an algorithm that will look for a deadlock (by detecting if a cycle is present in the graph). If deadlock is detected, your function will stop processing any more edges and immediately return **Result**:

- with **index=i,** where **i** indicates which **edges[i]** caused the deadlock; and
- with **procs[]** containing process names that are involved in a deadlock, in arbitrary order.

If no deadlock is detected after processing all edges, your function will indicate this by returning Result with **index=-1** and an empty **procs[]**.

Edge description Your function will be given the edges as a vector of strings, where each string will represent and edge. A request edge will have the format **"process -> resource"**, and assignment edge will be of the form **"process <- resource"**, where **process** and **resource** are the names of the process and resource, respectively. Here is a sample input, and its graphical representation:

```
edges =
  [ "   2     <- fork1",
    " plato   -> fork1",
    " plato   ->   2  ",
    "socrates -> fork1",
    "socrates <-   2  "
  ]
```



The input above represents a system with three processes: **"plato"**, **"socrates"** and **"2"**, and two resources: **"fork1"** and **"2".** The first line **"2 <- fork1"** represents an assignment edge, and it denotes process **"2"** currently holding resource **"fork1"**. The second line **"plato -> fork1"** is a request edge, meaning that process **"plato"** is requesting resource **"fork1"**. The resource allocation graph on the right is a graphical representation of the input. **Process and resource names are independent from each other, and it is therefore possible for processes and resources to have the same names**.

Notice that each individual string representing an edge may contain an arbitrary number of white spaces. Feel free to use the provided **split_line()** function to help you parse these strings, as it correctly deals with white spaces.

Starter code

Start by downloading the following starter code:

```
$ git clone https://csgit.ucalgary.ca/jwhudson/cpsc457w24.git
$ cd cpsc457w24/deadlock
$ make
```

You need to implement **find_deadlock()** function by modifying the **find_deadlock.cpp** file. **Only modify** file **find_deadlock.cpp,** and **do not** modify any other files.

The included driver (**main.cpp**) will read edge descriptions from standard input. It parses them into an array of strings, calls your **find_deadlock()** and finally prints out the results. The driver will ensure that the input passed to your function is syntactically valid, i.e. every string in **edges[]** will contain a valid edge. Here is how you run it on file **test1.txt**:

```
$ ./deadlock < test1.txt
Reading in lines from stdin...
Running find_deadlock()...

index       : 6
procs       : [12,7,7]
real time   : 0.0000s
```

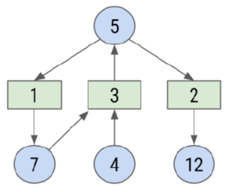
```
$ ./deadlock < test1.txt
Reading in lines from stdin...
Running find_deadlock()...

index       : -1
procs       : []
real time   : 0.0001s
```

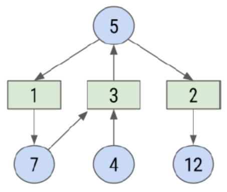
If you run the starter code (with an incomplete implementation), you will get the output on the left, which is obviously incorrect. Once implemented correctly, the output of your program will look like the one on the right, indicating no deadlock.

Few more examples:

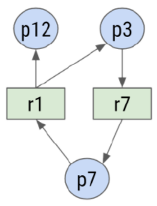
```
$ cat test2a.txt
5 -> 1
5 <- 3
5 -> 2
7 <- 1
12 <- 2
4 -> 3
7 -> 3
$ ./deadlock < test2a.txt
index: 6
procs: [4,7,5]
```



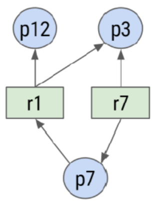
```
$ cat test2b.txt
5 -> 1
5 <- 3
5 -> 2
7 <- 1
12 <- 2
7 -> 3
4 -> 3
$ ./deadlock < test2b.txt
index: 5
procs: [7,5]
```



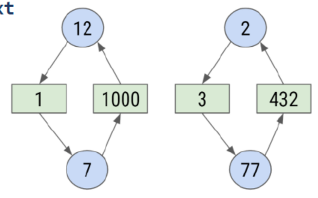
```
$ cat test3a.txt
p7 <- r7
p7 -> r1
p3 -> r7
p3 <- r1
p12 <- r1
$ ./deadlock < test3.txt
index: 3
procs: [p7,p3]
```



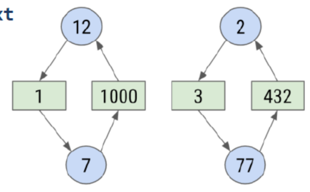
```
$ cat test3b.txt
p7 <- r7
p7 -> r1
p3 <- r7
p3 <- r1
p12 <- r1
$ ./deadlock < test3b.txt
index: -1
procs: []
```



```
$ cat test4.txt
12 -> 1
12 <- 1000
7 -> 1000
7 <- 1
2 -> 3
2 <- 432
77 -> 432
77 <- 3
$ ./deadlock < test4.txt
index: 3
procs: [12,7]
```



```
$ cat test5.txt
12 -> 1
12 <- 1000
7 -> 1000
2 -> 3
2 <- 432
77 -> 432
77 <- 3
7 <- 1
$ ./deadlock < test5.txt
index: 6
procs: [2,77]
```



Limits

You may assume the following limits on input:

- Both **process** and **resource names** will only contain alphanumeric characters and will be at most 40 characters long.
- Number of **edges** will be in the range [0 ... 30,000].

Your solution should be efficient enough to run on any input within the above limits in less than 10s, which means you should implement an efficient deadlock-detection algorithm (see appendix for hints). Remember, you are responsible for designing your own test cases.

Marking

Your submission will be marked both on correctness and speed for several test files. To get full marks, your program will need to finish under 10s on all inputs during marking. **You can not use threading on this assignment and thus must use one thread to achieve this speed requirement.**

About 80% of the marks will be based on tests with less than 10,000 edges, which should be easy to achieve. For example, your program should be able to finish **test6.txt** under 10s on **linuxlab** machines. The remaining 20% will be awarded only to submissions that can finish under 10s for ~30,000 edges, which is more difficult (e.g. **test7.txt** file). It is useful to note that the physical lab machines in the CPSC main lab are the machines with the 10s benchmark requirement. The ssh compute server will provide you with a less powerful CPU and thus a slower time.

Hints

I suggest using the following pseudocode for your implementation of find_deadlocks():

```
result = empty Result
g = initialize empty graph
for i = 0 to edges.size():
  e = parse edge in edges[i]
  insert e into g
  run toposort on g
  if toposort failed to finish:
     result.procs = proc. nodes that toposort did not remove
     result.index = i
     return result
result.index = -1
return result
```

The above uses topological sort to detect whether a graph has cycles. Please note that toposort identifies any nodes that are directly or indirectly involved in a cycle, which is perfect for this assignment, as you need to report any processes that are involved in a deadlock.

I suggest you start with the following data structures to represent a graph. These data structures should be good enough to finish under 10s on medium sized files, such as test6.txt.

```
class Graph {
      std::unordered_map<std::string, std::vector<std::string>> adj_list;
      std::unordered_map<std::string, int> out_counts;
      ...
} graph;
```

The field **adj_list** is a hash table of dynamic arrays, representing an adjacency list. Insert nodes into it so that **adj_list["node"]** will contain a list of all nodes with edges pointing towards "node". The **out_counts** field is a hash table of integers, representing the number of outgoing edges for every node (outdegrees). Populate it so that **out_counts["node"]** contains the number of edges pointing out from "node".

With these data structures you can implement efficient topological sort (pseudo-code):

```
out = out_counts # copy out_counts so that we can modify it
zeros[] = find all nodes in graph with outdegree == 0
while zeros is not empty:
      n = remove one entry from zeros[]
      for every n2 in adj_list[n]:
            out[n2] --
            if out[n2] == 0:
                  append n2 to zeros[]
processes involved in deadlock are proc. nodes n with out[n]>0
```

To get run times under 10s on large files, such as **test7.txt**, you can use the same algorithm as above, but you will need to switch to data structures that are more efficient than hash tables. The problem with hash tables is that they spend considerable time on calculating hashes on strings, and also on resolving collisions. To avoid this overhead, you can pre-convert all strings (process and resource names) into consecutive integers, and then use fast dynamic arrays instead of hash tables to store the adjacency list and outdegree counts:

```
class FastGraph {
    std::vector<std::vector<int>> adj_list;
    std::vector<int> out_counts;
    ...
```

You can use the provided **`Word2Int`** class to help you with converting strings to unique consecutive integers. The topological sort algorithm would remain the same as above. Do not forget to convert the integers back to strings at the end, to correctly populate **`procs[]`**.

Q2. Programming question – CPU scheduling [50 marks]

For this question you will implement a shortest-job-first and round-robin CPU scheduling simulator. The input to your simulator will be a set of processes and a time slice. Each process will be described by an id, arrival time and CPU burst. Your simulator will simulate SJF/RR scheduling on these processes and for each process it will calculate its start time and finish time. Your simulator will also compute a condensed execution sequence of all processes. You will implement your simulator as functions simulate_sjf(),simulate_rr() with the following signatures:

```
void simulate_sjf(
      int64_t max_seq_len,
      std::vector<Process> & processes,
      std::vector<int> & seq);

void simulate_rr(
      int64_t quantum,
      int64_t max_seq_len,
      std::vector<Process> & processes,
      std::vector<int> & seq);
```

The parameter **quantum** will contain a positive integer describing the length of the time slice and **max_seq_len** will contain the maximum length of execution order to be reported. The array processes will contain the description of processes, where struct **Process** is defined in **scheduler.h** as:

```
struct Process {
      int id;
      int64_t arrival, burst;
      int64_t start_time, finish_time;
};
```

The fields **id**, **arrival** and **burst** for each process are the inputs to your simulator, and you should not modify these. However, you must populate the **start_time** and **finish_time** for each process with computed values. You must also report the condensed execution sequence of the processes via the output parameter **seq[]**. You need to make sure the reported order contains at most the first **max_seq_len** entries. The entries in **seq[]** will contain either process ids, or **-1** to denote **idle CPU**. You will also be required to report a sequence chart as will be shown later as the simulation progresses step by step.

A condensed execution sequence is similar to a regular execution sequence, except consecutive repeated numbers are condensed to a single value. For example, if a regular non-condensed sequence was **[-1,-1,-1,1,1,2,1,2,2,2]**, then the condensed equivalent would be **[-1,1,2,1,2]**.

Starter code

```
$ git clone https://csgit.ucalgary.ca/jwhudson/cpsc457w24.git
$ cd cpsc457w24/sched

$ make
```

You need to implement the simulate_sjf()simulate_rr() functions in scheduler.cpp. Do not modify any files except scheduler.cpp.

Using the driver

The starter code includes a driver (**main.cpp**) that parses command lines arguments to obtain the time slice and the maximum execution sequence length. It then parses standard input for the description of processes, where each process is specified on a separate line. Each input line contains 2 integers: the first one denotes the **arrival** time of the process, and the second one denotes the CPU **burst** length. For example, the file **test1.txt** contains information about **3** processes: **P0**, **P1** and **P2**:

```
$ cat test1.txt
1 10
3 5
5 3
```

The 2nd line "3 5" means that process **P1** arrives at time **3** and it has a CPU burst of **5** seconds.

After parsing the inputs, the driver calls your **simulatesjf()** or **simulate_rr()** based on if you've chosen the SJF or RR flags, and afterwards prints out the results. For example, to run your simulator with **quantum=3** and **max_seq_len=20** on a file **test1.txt**, you would invoke the driver like this (obviously these are the incorrect outputs in the default code):

```
bash-5.2$ ./scheduler 3 20 SJF < test1.txt
Reading in lines from stdin...
Running simulate_sjf(maxs=20,procs=[3])
Elapsed time  : 0.0000s

seq = [0,1,2]
+---------------------------+----------------------+----------------------+----------------------+
| Id |                Arrival |                Burst |                Start |               Finish |
+---------------------------+----------------------+----------------------+----------------------+
|  0 |                      1 |                   10 |                    1 |                   11 |
|  1 |                      3 |                    5 |                    3 |                    8 |
|  2 |                      5 |                    3 |                    5 |                    8 |
+---------------------------+----------------------+----------------------+----------------------+
bash-5.2$ ./scheduler 3 20 RR < test1.txt
Reading in lines from stdin...
Running simulate_rr(q=3,maxs=20,procs=[3])
Elapsed time  : 0.0000s

seq = [0,1,2]
+---------------------------+----------------------+----------------------+----------------------+
| Id |                Arrival |                Burst |                Start |               Finish |
+---------------------------+----------------------+----------------------+----------------------+
|  0 |                      1 |                   10 |                    1 |                   11 |
|  1 |                      3 |                    5 |                    3 |                    8 |
|  2 |                      5 |                    3 |                    5 |                    8 |
+---------------------------+----------------------+----------------------+----------------------+
```

Please note that the output above is incorrect, as the starter code contains an incomplete implementation of the scheduling algorithm. The correct results should look like this:

```
bash-5.2$ ./scheduler 3 20 SJF < test1.txt
Reading in lines from stdin...
Running simulate_sjf(q=3,maxs=20,procs=[3])
   0
   1 #
   2 #
   3 #  .
   4 #  .
   5 #  .  .
   6 #  .  .
   7 #  .  .
   8 #  .  .
   9 #  .  .
  10 #  .  .
  11    .  #
  12    .  #
  13    .  #
  14    #
  15    #
  16    #
  17    #
  18    #
  19
P0 waited 0 sec.
P1 waited 11 sec.
P2 waited 6 sec.
Average waiting time = 5.66667 sec.
Elapsed time  : 0.0002s

seq = [-1,0,2,1]
+---------------------------+----------------------+----------------------+----------------------+
| Id |              Arrival |                Burst |                Start |               Finish |
+---------------------------+----------------------+----------------------+----------------------+
|  0 |                    1 |                   10 |                    1 |                   11 |
|  1 |                    3 |                    5 |                   14 |                   19 |
|  2 |                    5 |                    3 |                   11 |                   14 |
+---------------------------+----------------------+----------------------+----------------------+
bash-5.2$ ./scheduler 3 20 RR < test1.txt
Reading in lines from stdin...
Running simulate_rr(q=3,maxs=20,procs=[3])
   0
   1 #
   2 #
   3 #  .
   4 .  #
   5 .  #  .
   6 .  #  .
   7 #  .  .
   8 #  .  .
   9 #  .  .
  10 .  .  #
  11 .  .  #
  12 .  .  #
  13 .  #
  14 .  #
  15 #
  16 #
  17 #
  18 #
  19
P0 waited 8 sec.
P1 waited 7 sec.
P2 waited 5 sec.
Average waiting time = 6.66667 sec.
Elapsed time  : 0.0003s

seq = [-1,0,1,0,2,1,0]
+----------------------------+----------------------+----------------------+----------------------+
| Id |              Arrival |                Burst |                Start |               Finish |
+----------------------------+----------------------+----------------------+----------------------+
|  0 |                    1 |                   10 |                    1 |                   19 |
|  1 |                    3 |                    5 |                    4 |                   15 |
|  2 |                    5 |                    3 |                   10 |                   13 |
+----------------------------+----------------------+----------------------+----------------------+
```

You should note that quantum will be ignored in SJF scheduling but will be used by RR. Only capital RR triggers round-robin, lower case or anything else will default to SJF in main.cpp as it is implemented.

The completed output has two new parts you can see here:

1. a table describing the state of each process for every simulation time step,
2. followed by a summary, which includes the wait time for each process and the average wait time for all processes.

The first column in the table is the simulation time. There is also one column for each process to describe its state for the given simulation time. Use "." to denote READY state, "#" to denote RUNNING state, and a empty space " " to denote a process that has not yet arrived or a finished process. Make sure the output of your program is nicely aligned like the example above.

Important - If your simulation detects that an existing process exceeds its time slice at the same time as a new process arrives, you need to insert the existing process into the ready queue before inserting the newly arriving process.

Limits

You may make the following assumptions about the inputs:

The processes are sorted by their arrival time, in ascending order. Processes arriving at the same time must be inserted into the ready queue in the order they are listed.

- All arrival times will be non-negative.
- All burst times will be greater than 0.
- Process IDs will be consecutively numbered starting with 0.
- All processes are 100% CPU-bound, i.e., a process will never be in the waiting state.
- There will be between 0 and 30 processes.
- Time slice and CPU bursts will be integers in the range [1 … 262]
- Process arrival times will be integers in the range [0 … 262]
- finish time of every process will fit into int64_t.

The git repository includes some test files and the **README.md** contains several sample results. Please remember to also design your own test data to make sure your program works correctly and efficiently for all of the above limits.

Marking

Your submission will be marked both on correctness and speed for a number of different test files. To get full marks, your program will need to finish under 10s on all test cases. About half of the test cases will include inputs with small values for arrival times and CPU bursts. We do not plan to test anything that should be an efficiency concern. For example, we will **NOT** have very large arrival times, or very large burst times, or a quantum much smaller than process burst times. All of these would result in the step by step solution we are requested for simulation output to be drastically slowed down.

Start with a simulation loop that increments current time by 1. This should make your simulator work fast enough for small arrival times and bursts.

Hints for a simple solution

This simple solution increments the current time in the simulation loop by at most 1 time unit, and many students will find it the easiest to debug.

Please refer to the lecture slides for ideas on how to structure your simulation loop. Here are some suggestions for data structures for keeping track of the current state:

- The current time, e.g. int64_t curr_time

- The remaining time slice of the currently executing process, e.g. int64_t remaining_slice
- Currently executing process, e.g. int cpu, so that cpu is an index into processes[], and cpu=-1 represents idle CPU
- Ready Queue (RQ) and Job Queue (JQ), e.g. std::vector<int> rq, jq
    - the integers stored in jq and rq would be indices into processes[], just like cpu
    - initialize JQ with all processes, and remove them from JQ as they 'arrive'
- You will need to keep track of the remaining bursts for all processes
    - Since you cannot modify processes[], you need to keep track of this in your own data structure, e.g. std::vector<int64_t> remaining_bursts;

Submission

Submit 2 files for this assignment to D2L:

- Your solution to Q1 in a file called **find_deadlock.cpp.**
- Your solution to Q2 in a file called **scheduler.cpp.**

Please note – you need to **submit all files every time** you make a submission, as the previous submission will be overwritten.

Submit this as a separate file. **Do not** submit an archive, such as ZIP or TAR. If you submit an archive, you will receive a penalty.

Submit the web address of your Gitlab repository you used for your assignment (I recommend one repository for all 6 of your assignments that you can re-use). Your TA must be added as a Developer. There are penalties for submissions without the ability to access the corresponding students Gitlab by the TA.

While the starter code contains many different files, the only file you are allowed to modify is **find_deadlock.cpp/ scheduler.cpp.** Do not modify any other files. All code you write must go in the **find_deadlock.cpp/ scheduler.cpp** files, and that should be the only file you will submit for grading. We will test your code by supplying our own **main()** function, which will be different from the **main()** function in the starter code. It is therefore vital that you maintain the same function signature as declared in **find_deadlock.h/ scheduler.h.** Before you submit **find_deadlock.cpp/ scheduler.cpp** to D2L, make sure it works with unmodified files from the starter code!

General information about all assignments:

All assignments are due on the date listed on D2L. Late submissions without remaining late days banked will not be marked.

1. Extensions beyond the late day policy can be discussed more than 5 business days in advance and are granted only by the course instructor.
2. After you submit your work to D2L, verify your submission by re-downloading it.

3. You can submit many times before the due date. D2L will simply overwrite previous submissions with newer ones. It is better to submit incomplete work for a chance of getting partial marks, than not to submit anything. Please bear in mind that you cannot re-submit a single file if you have already submitted other files. Your new submission would delete the previous files you submitted. So please keep a copy of all files you intend to submit and resubmit all of them every time.
4. Assignments are likely going to be marked by your TAs. If you have questions about assignment marking, contact your TA first. If you still have questions after you have talked to your TA, then you can contact your instructor.
5. All programs you submit must run on **linux lab** or **cslinux.ucalgary.ca.** If your TA is unable to run your code on these, you will receive 0 marks.
6. Unless specified otherwise, you must submit code that can finish on any valid input under 10s on **linux lab** or **cslinux.ucalgary.ca (will be slower)**, when compiled with **-O2** optimization. Any code that runs longer than this may receive a deduction, and code that runs for too long (about 30s) will receive 0 marks.
7. **Assignments must reflect individual work.** Here are some examples of what you are not allowed to do for individual assignments: you are not allowed to copy code or written answers (in part, or in whole) from anyone else; you are not allowed to collaborate with anyone; you are not allowed to share your solutions (code or pseudocode) with anyone; you are not allowed to sell or purchase a solution; you are not allowed to make your code available publicly (e.g. via public git repositories). This list is not exclusive. For further information on plagiarism, cheating and other academic misconduct, check the information at this link: http://www.ucalgary.ca/pubs/calendar/current/k-5.html .
8. We will use automated similarity detection software to check for plagiarism. Your submission will be compared to other students (current and previous), as well as to any known online sources. Any cases of detected plagiarism or any other academic misconduct will be investigated and reported.