The University of Melbourne
School of Computing and Information Systems
COMP90038 Algorithms and Complexity

Assignment 1, Semester 2 2020

Released: Friday August 28 2020. Deadline: Sunday September 13 2020 23:59

This assignment is marked out of 30 and is worth 20% of your grade for COMP90038.

Objectives

To improve your understanding of the time complexity of algorithms and recurrence relations. To develop problem-solving and design skills. To improve written communication skills; in particular the ability to present algorithms clearly, precisely and unambiguously.

Scenario

In this assignment we will consider the following scenario. Suppose you have been tasked with writing some algorithms whose purpose is to manage *event logs* in a computer program. It is very common for programs to generate messages describing what is occurring during the program. These messages are called *events*, and are typically saved into one or more files called *log files*. Each such file is called an event log.

As an example, the following is a sample of a few log messages relating to the wireless network adapter on my MacBook Pro that were generated in the minutes before I typed this sentence.

```
Mon Aug 24 14:21:35.791 <kernel> en0: Received EAPOL packet (length = 193)
Mon Aug 24 14:21:35.791 <kernel> inputEAPOLFrame: 0 extra bytes present in EAPOL frame.
Mon Aug 24 14:21:35.791 <kernel> inputEAPOLFrame: decrypting key data
Mon Aug 24 14:21:35.791 <kernel> inputEAPOLFrame: Received message 3 of 4
Mon Aug 24 14:21:35.791 <kernel> process3of4: Unknown Elem ID 221
Mon Aug 24 14:21:35.791 <kernel> process3of4: Unknown Elem ID 0
Mon Aug 24 14:21:35.791 <kernel> process3of4: Performing IE check.
Mon Aug 24 14:21:35.791 <kernel> process3of4: sending replyPacket ( len = 113 ).
Mon Aug 24 14:21:35.791 <kernel> process3of4: received pairwise GTK
Mon Aug 24 14:21:35.791 <kernel> ptkThread: Sleeping!
```

The content of these messages is not important. But notice that each message begins with a *timestamp*, indicating the date and time at which the log event was generated.

In certain applications, such as the Linux kernel, log events are stored in memory rather than being immediately written to a log file. We will consider a common data structure for storing such events, called a *ring buffer* (also known as a *circular queue*). Importantly, this data structure uses a fixed amount of memory. However it behaves like a queue: items are accessed in a first-in-first-out fashion.

Ring Buffer

The style of ring buffer we consider for this application is implemented as an array, in which log entries are stored. That is, each array element holds one log entry. A log entry includes a timestamp and a message. The array is of a fixed size C, which is called its *capacity*, and it is indexed from $0, \ldots, C-1$.

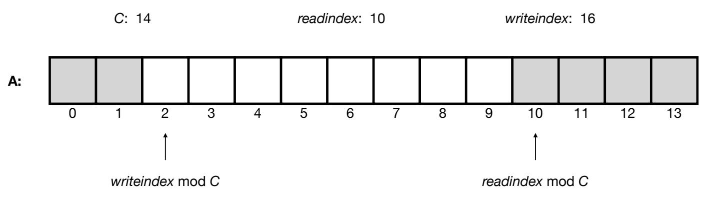


Figure 1: A graphical depiction of the ring buffer data structure. Shaded array elements represent data stored in the buffer, while white ones represent free space. Here the depicted ring buffer has capacity 14. 16 items have been written to the ring buffer in total so far, while 10 have been read, leaving 6 items remaining in the buffer to be read at present.

Read and Write Indices There are two index variables called the *write index* and the *read index* respectively. The write index is used to calculate the location that will be used to hold the next item that will be added to the ring buffer. The read index is used to calculate the index of the oldest item in the ring buffer, i.e. which item will be the first to be read from the buffer.

The write index counts the number of items that have been added to the buffer. The read index counts the number of items that have been read from the buffer. For the purposes of this assignment, you can assume that these indices are unbounded, i.e. you don't need to worry about what might happen if they get too large and overflow.

If the value of the write index is w, then the position used to store the next item to be added to the buffer is $w \mod C$. Likewise, if the read index has value is r, then the oldest item in the buffer (if it exists) is at position $r \mod C$.

Initial State Initially, both the read index and write index are set to 0.

Invariant An *invariant* is a property that talks about the internals of a data structure. Functions and procedures that operate on the data structure can assume that the invariant holds when they are called. However they must ensure that when they finish the invariant still holds.

The invariant for the ring buffer is that the write index must be greater than or equal to the read index, and the maximum difference between the read and write indices cannot exceed the buffer's capacity C.

Empty and Non-Empty Buffer The ring buffer is empty when the read and write index are equal to each other. Otherwise it is non-empty and an attempt to read an item from the buffer must succeed.

Reading an Item Reading an item from the buffer removes that item from the buffer. Suppose the array that implements the buffer is A, whose capacity is C, and the read and write indices are stored respectively in the *readindex* and *writeindex* variables. Then the following function is used to read an item from the ring buffer.

```
function GetItem($A, C, readindex, writeindex$)
    if $readindex \neq writeindex$ then
        $index \leftarrow readindex \mod C$
        $readindex \leftarrow readindex + 1$
        return $A[index]$
    else
        return null
```

Notice how this function updates the read index, by incrementing it. This has the effect of ensuring that the read item cannot be read again, and so simulates the idea that it has been removed from the buffer (without having to explicitly zero out that array element).

Your Tasks

1. [2 marks] Notice that the GetItem function above increments the *readindex* parameter, when reading an item from the ring buffer. For this to work, this parameter must be passed *by reference.* What would happen instead if it were passed *by value*? Explain in no more than 2–3 sentences.

   Solution:
   The update to *readindex* would not persist after GetItem is called. Hence, if we were to call GetItem a second time, we would end up reading the same value we read last time (and indeed we would forever keep reading this value).

   Marking Rubric:
   For the two marks we ideally want to see the student demonstrate knowledge of both of these: (1) *readindex* is not updated. Hence, (2) subsequent reads will re-read the old data. Award 1 out of 2 marks if it is not clear that student understands both of these points. Award 0 if student demonstrates no understanding of the distinction between pass-by-value and pass-by-reference.

2. [3 marks] Implement the following $\Theta(1)$ procedure that adds an item x to the ring buffer. Your function should always succeed: if the ring buffer is already full, the oldest item should be discarded. You can assume that all arguments (parameters) to the procedure are passed by reference. You can also assume that the invariant holds when your procedure is called. You should ensure that it holds again when your procedure finishes (returns).

   procedure PutItem($A, C, \mathit{readindex}, \mathit{writeindex}, x$) ...

   Solution:

```
procedure PutItem($A, C, readindex, writeindex, x$)
    if $writeindex = readindex + C$ then          ▷ buffer is full, discard the oldest item
        $A[writeindex \mod C] \leftarrow x$          ▷ $writeindex \mod C = readindex \mod C$
        $readindex \leftarrow readindex + 1$
        $writeindex \leftarrow writeindex + 1$
    else
        $A[writeindex \mod C] \leftarrow x$
        $writeindex \leftarrow writeindex + 1$
```

Marking Rubric:

- 3 marks if fully correct and no problems with formatting, function header, etc.
- 2 marks if fully correct but small problems with formatting and/or incorrect function header.
- 1 mark if small correctness issue plus issues with formatting, etc.
- 2 marks for small correctness issues (but only if no major issues with formatting or the function header, arguments, etc.).
- 1 mark if the function has a major correctness issue (e.g. forgetting to update *readindex* when the buffer is full).

3. [4 marks] Consider the following, very inefficient, *decrease and conquer* algorithm. Its job is to remove from the ring buffer all log entries that are older than the given timestamp t. Note that each item in the ring buffer stores two pieces of data: a timestamp and a message. Given an item x, x.time denotes x's timestamp and x.msg denotes x's message.

```
procedure RemoveOlder($A, C, readindex, writeindex, t$)
    if $readindex \neq writeindex$ then
        if $A[readindex \mod C]$.time $< t$ then
            $readindex \leftarrow readindex + 1$
            RemoveOlder($A, C, readindex, writeindex, t$)
```

Write down a recurrence relation for the worst-case complexity of this algorithm, that counts the number of basic operations it performs. The size of the input n is the number of items in the ring buffer: $writeindex - readindex$. The algorithm has two basic operations (1) the comparison $A[readindex]$.time $< t$ and (2) the update of *readindex*: $readindex \leftarrow readindex + 1$.

Use telescoping (i.e. backward substitution) to find a closed form for your recurrence relation and, thus, determine the worst-case complexity of this algorithm.

Solution:
We define a recurrence relation C. When the buffer is empty $readindex = writeindex$, no basic operations are performed. So

$$C(0) = 0.$$

Otherwise, in the worst case, we perform two basic operations. We then increment the read index. So if $writeindex - readindex = n$ before, in the recursive call $writeindex - readindex = n - 1$. Hence:

$$C(n) = 2 + C(n-1).$$

Using telescoping we have:

$$\begin{aligned} C(n) &= 2 + C(n-1) \\ &= 2 + (2 + C(n-2)) = 4 + C(n-2) \\ &= 4 + (2 + C(n-3)) = 6 + C(n-3) \\ &\dots \\ &= 2i + C(n-i) \\ &= 2n + C(n-0) = 2n + C(0) \\ &= 2n + 0 = 2n \end{aligned}$$

Hence, $C(n) = 2n$ is the closed form. Hence, the worst-case complexity of this algorithm is in $\Theta(n)$.

Marking Rubric:

- 2 marks for correct recurrence relation (one mark for each correct equation).
- 2 marks for correct telescoping. Award 1 out of 2 for small error during telesoping. Award 0 out of 2 for telescoping if multiple errors or if it seems clear the student does not understand what telescoping is. Award 2 out of 4 if steps are not shown.

4. [5 marks] Design a more efficient implementation of RemoveOlder that performs the same task but whose worse-case complexity is in $\Theta(\log n)$. For this question you are free to assume that timestamps never decrease over time. That is, if an item x_2 is added to the ring buffer after some item x_1 has been added, then $x_2.\mathsf{time} \geq x_1.\mathsf{time}$.

Solution:
Assuming that timestamps never decrease over time, the ring buffer is sorted by timestamp. Therefore, we can use binary search over the range $readindex, \ldots, writeindex - 1$ to find the smallest index k where $readindex \leq k < writeindex$ such that $A[k \mod C].\mathsf{time} >= t$. That requires at most $\Theta(\log(n))$ comparisons (where, recall, $n = writeindex - readindex$). Then we can remove all elements older than this one by simply setting $readindex \leftarrow k$, an $O(1)$ operation. Hence, the worst case complexity is in $\Theta(\log n)$.

```
    ▷ Precondition: smallest index $k$ such that $A[k \mod C].\mathsf{time} \geq t$ is in $lo \ldots hi$ if it exists
procedure RemoveOlderS($A, C, readindex, writeindex, t, lo, hi$)
    if $lo \leq hi$ then                    ▷ There is a non-empty part of the array to search
        $mid \leftarrow (lo + hi)/2$
        if $A[mid \mod C].\mathsf{time} < t$ then
            RemoveOlderS($A, C, readindex, writeindex, t, mid + 1, hi$)
        else
            if $mid = lo$ then    ▷ This is the smallest such $k$ by definition of the precondition
                $readindex \leftarrow lo$
            else                                                        ▷ $lo < mid$
                if $A[mid - 1 \mod C].\mathsf{time} \geq t$ then        ▷ $mid$ is not the smallest such $k$
                    RemoveOlderS($A, C, t, lo, mid - 1$)
                else                                          ▷ $mid$ is the smallest such $k$
                    $readindex \leftarrow mid$
    else                                   ▷ No such $k$, so the entire buffer must be emptied
        $readindex \leftarrow writeindex$


procedure RemoveOlder($A, C, readindex, writeindex, t$)
    if $readindex \neq writeindex$ then
        RemoveOlderS($A, C, readindex, writeindex, t, readindex, writeindex - 1$)
```

Marking Rubric:

- Award 5 marks for a fully correct solution with $\log n$ complexity.
- Award 4 out of 5 marks for a $\log n$ solution that has only small correctness issues (e.g. formatting, incorrect function headers, missing parameters to function calls, etc.).
- Award 3 out of 5 marks for a $\log n$ solution that has medium correctness issues (e.g. small logic issues but looks mostly correct).
- Award 2 out of 5 marks for a $\log n$ solution that has major correctness issues (e.g. major logic issues and looks like it would almost never work).
- Award 1 out of 5 marks for a solution that demonstrates no understanding of binary search, or the problem being solved.
- Award 3 marks at most if the solution does not have $\log n$ complexity, and only if that solution is otherwise correct (no correctness issues).
- Award 2 marks for a non-$\log n$ solution that has small correctness issues (as defined above).
- Award 1 mark for a non-$\log n$ solution that has medium correctness issues (as defined above).
- Award 0 marks for a non-$\log n$ solution that has major correctness issues.

5. [4 marks] Suppose you have to implement an algorithm to remove from the ring buffer all items whose timestamps are within a range $[t_1, t_2)$, that is remove all items x for which $t_1 \leq x.\mathsf{time} < t_2$. Explain why such an algorithm cannot be implemented with worst-case complexity $\Theta(\log n)$. Your explanation should be no more than a paragraph of text, i.e. 5 or 6 sentences.

Solution:
This is not possible because, (assuming a non-empty buffer with non-zero items that need to be removed) while we can use binary search to find the indices k_1 and k_2 where $readindex \leq k_1 \leq k_2 < writeindex$ and $A[k_1 \mod C]$ is the oldest item that needs to be removed and $A[k_2 \mod C]$ is the newest item that needs to be removed (i.e. we remove all items between these two indices inclusive), there is no way to remove just these items, if there are items older than k_1 in the buffer (i.e. when $k_1 > readindex$) and also newer items than k_2 (i.e. $k_2 < writeindex - 1$). Doing that requires copying data inside the buffer. In the worst case we might be removing only one item that is in the middle of a full buffer, meaning there would be approximately $n/2$ items that need to be copied, giving us a worst-case complexity in $\Theta(n)$.

Marking Rubric:
We are looking for four pieces of information:

- A correct description of the problematic scenario: having to remove items in the middle of a buffer.
- Mention that the worst-case complexity of doing that is linear: n.
- A clear explanation of *why* the worst-case complexity of doing that is linear (e.g. it involves *copying data* and the worst-case amount of such data is in $\Theta(n)$).
- Hence, since $n > \log n$, overall complexity cannot be $\log n$.

Give 1 mark out of 4 if the student's reasoning is incorrect but mentions we cannot solve the problem with logarithmic complexity.

6. [8 marks] Suppose now that instead of storing log entries, we used the ring buffer (specifically GetItem and PutItem) to store graph nodes, to implement breadth-first traversal of a graph. That is, imagine you implement the standard breadth-first traversal algorithm from lectures using the ring buffer as the queue of nodes to be visited.

   Suppose this hypothetical algorithm was run on a graph $\langle V, E\rangle$ for which the number of nodes $|V|$ in the graph was larger than C, the buffer's capacity. Answer each of the following questions:

   (a) [2 marks] Would every node be guaranteed to be traversed? Explain your answer in no more than a paragraph (i.e. 5 or 6 sentences). You might like to draw a picture of a graph to assist your explanation.

   Solution:
   This depends somewhat on one's definition of "traversed". If we define "traversed" as meaning that a node was marked non-zero and added to the queue (i.e. *enqueued*) then the answer is "yes": any node that doesn't get added to the queue when it should be (because its parent was discarded when the buffer got full) is guaranteed to be traversed eventually due to the outer loop in BFS. However the resulting traversal might not be a valid breadth-first one.
   If instead we define "traversed" as meaning that a node is *dequeued* from the queue, then it can be the case that not all nodes are traversed, since a node could be enqueued and later lost when the queue gets full, before it is dequeued. That node won't be enqueued a second time since it is now marked non-zero.

   Marking Rubric:
   To get full marks here the student should ideally make clear what their interpretation of "traversed" / "visited" is and their answer should be consistent with that.

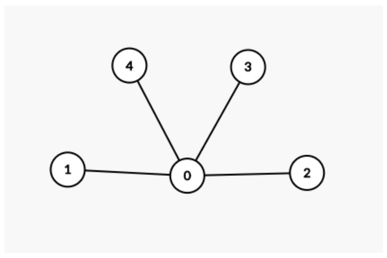
   - 2 out of 2: Clear and correct.
   - 1 out of 2: Vague / unclear justification of why some nodes may not be visited. Or, insufficient reasoning from "nodes overwritten in the queue" to "nodes not visited"
   - 0 out of 2: Demonstrate clear misunderstanding on either buffer (typical error: claiming latest node gets overwritten) or BFS (typical error: claiming nodes in subtree not visited)

   (b) [6 marks] Devise a recursive, depth-first search-based algorithm for determining whether breadth-first traversal of a graph will succeed (i.e. a valid breadth-first traversal will be performed). By "depth-first search-based" we mean that your algorithm should be a variant of the recursive depth-first traversal algorithm shown in lectures. For this question you can assume that the graph is *undirected*. Your algorithm is allowed to be conservative: it can say that traversal is not possible even if it is. However, it should never say that traversal is possible when it is not.

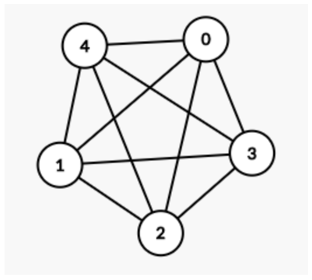
The worst-case running time of your algorithm should not exceed $\Theta(|V|+|E|)$, the complexity of depth-first traversal. Your algorithm is *not* permitted to attempt to run breadth-first traversal over the graph.

You should explain how your algorithm works in no more than a paragraph of text (i.e. 5 or 6 sentences).

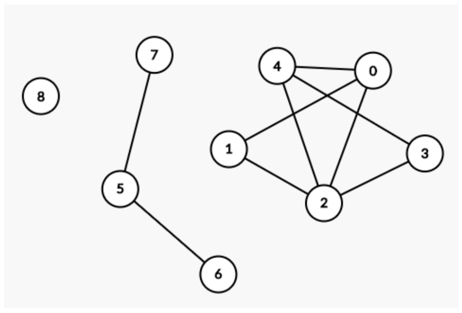
The less conservative your algorithm is, the higher your score will be for this question. Suppose C was 4. Then your algorithm should answer "yes" for each of the following three graphs, while being general (i.e. you should not specialise your algorithm to these cases and it should work for arbitrary positive C).



Graph 1



Graph 2



Graph 3

Solution:
Clearly the number of nodes is an upper bound for how large the queue can grow. However can do a bit better, by recognising that nodes from distinct graph components can never be in the queue at the same time. Therefore the maximum size of the queue will be bounded by the size of the graph's largest component, minus one. Therefore, we can run DFS to calculate the size of each component of the undirected graph. Keep a running maximum. If that exceeds $C+1$, say that traversal is possible. Otherwise, say that it is not.

```
function DFS($\langle V, E \rangle$)
    mark each node in $V$ with 0
    $count \leftarrow 0$
    for each $v$ in $V$ do
        if $v$ is marked with 0 then
            DfsExplore($v$)
            if $count > C + 1$ then                ▷ $count$ is now the size of the component
                return false
            $count \leftarrow 0$
    return true

function DfsExplore($v$)
    $count \leftarrow count + 1$
    mark $v$ with $count$
    for each edge $(v, w)$ do                      ▷ $w$ is $v$'s neighbour
        if $w$ is marked with 0 then
            DfsExplore($w$)
```

Marking Rubric:
To get all 6 marks the solution must:

- Have the required complexity
- Make no assumptions about the order in which nodes and edges are visited
- Not be too conservative, i.e. it should be able to handle at least the three examples in the handout (while also being general, i.e. is not tailored specifically to those three cases).
- Be clear and elegant
- Marks awarded:
  - 6 out of 6: Clear and correct solution.
  - 5 out of 6: Correct solution with minor error (indentation, typo), which does not affect the logic.
  - 3 out of 6: Correct but over complicated. Correct logic with multiple issues. Too trivial (total number of nodes $< C$).
  - 2 out of 6: Attempting to calculate "level" of each node. Over long solution (≥ 2 pages) or logic is extremely unclear. Produces false positive (typical case: compare degree of nodes with C).
  - 0 out of 6: Using idea of BFS.

Otherwise award marks in line with the degree of understanding shown by the student.

7. [4 marks] For this question, now assume we are using the ring buffer data structure to store characters, i.e. each array element of the ring buffer holds a single character, and so a ring buffer stores a sequence of characters (also known as a string) of length $\mathit{writeindex} - \mathit{readindex}$. For a non-empty ring buffer, the first character in the string is at position $\mathit{readindex} \mod C$, the second is at $\mathit{readindex} + 1 \mod C$, and so on up to the final character at position $\mathit{writeindex} - 1 \mod C$.

   We discussed brute force string matching in lectures, to find the first occurrence of a patten p in a string s (the algorithm returns -1 if the pattern is not found and otherwise returns the position in the string at which p appears). It has $O(|p| \cdot |s|)$ time complexity in the worst case. However, there are algorithms for this task which have better worst-case time complexity of $O(|p| + |s|)$. Suppose you have been given such an algorithm.

   Now show how we can design an algorithm to determine whether a given ring buffer is a *rotation* of another ring buffer (both of capacity C), in $O(C)$ time. Suppose ring buffer A stores the string a and ring buffer B stores the string b, then A and B are rotations of each other, if there exist strings u and v such that $a = uv$ and $b = vu$ (uv means u and v concatenated). Note that u or v could be empty, so by our definition, every ring buffer is a rotation of itself.

Solution:
Here is an algorithm sketch, with the upper bound complexity of each step annotated, thus justifying $O(C)$ complexity.

(a) If $writeindex_A - readindex_A \neq writeindex_B - readindex_B$, return **false** ($O(1)$).

(b) Create two arrays X and Y, both of length $writeindex_A - readindex_A$ ($O(C)$).

(c) Copy all items (from $readindex_A$ to $writeindex_A$) from ring buffer A, to array X; ($O(C)$) // eg. X=[0,1,2,3,4,5]

(d) Copy all items (from $readindex_B$ to $writeindex_B$) from ring buffer B, to array Y; ($O(C)$) // eg. Y=[4,5,0,1,2,3]

(e) Create an array Z whose length is twice that of X. ($O(C)$)

(f) Concatenate X with itself, store the result in Z; ($O(C)$) // eg. Z=[0,1,2,3,4,5,0,1,2,3,4,5]

(g) Search Y in Z (using the $O(|p|+|s|)$ algorithm for string matching); ($O(C+2C) = O(C)$)

(h) If the searching result is -1, return **false**; Otherwise, return **true**. ($O(1)$)

Marking Rubric:
Solutions that are not pseudocode here are acceptable (i.e. should not be penalised) *only if* the description is clear enough to show that the student fully understands the algorithm they are proposing. Otherwise deduct 1 to 2 marks, depending on the level of non-clarity.

- 4 out of 4: No mistakes and correct solution
- 3 out of 4: 1 minor mistake
- 2 out of 4:
  - 2 (or more) minor mistakes (e.g. not checking lengths of the array, plus mistake handling arrays/ using uninitialised variables etc.), or
  - 1 major mistake, but solution is on the right track or has wrong complexity but is otherwise correct.
- 1 out of 4: multiple issues, including at least 1 major mistake or incorrect solution or partial (incorrect) attempt.

Do not deduct marks if the student's solution operates directly on the strings, rather than on ring buffers (see Lianglu's comment on the discussion board).
Otherwise award marks in line with the degree of understanding shown by the student.

Submission and Evaluation

- You must submit a PDF document via the LMS. Note: handwritten, scanned images, are acceptable *only if* they are clearly legible. Write very neatly, and if you photograph your submission be sure to use an app that auto crops and rotates images, such as OfficeLens, to ensure the resulting submission is easy to mark. Convert images to PDF before submission. Do not submit Microsoft Word documents — if you use Word, create a PDF version for submission.

- Marks are primarily allocated for correctness, but elegance of algorithms and how clearly you communicate your thinking will also be taken into account. Where indicated, the complexity of algorithms also matters.
- We expect your work to be neat – parts of your submission that are difficult to read or decipher will be deemed incorrect. Make sure that you have enough time towards the end of the assignment to present your solutions carefully. Time you put in early will usually turn out to be more productive than a last-minute effort.
- You are reminded that your submission for this assignment is to be your own individual work. For many students, discussions with friends will form a natural part of the undertaking of the assignment work. However, it is still an individual task. You should not share your answers (even draft solutions) with other students. Do not post solutions (or even partial solutions) on social media or the discussion board. It is University policy that cheating by students in any form is not permitted, and that work submitted for assessment purposes must be the independent work of the student concerned.

  Please see `https://academicintegrity.unimelb.edu.au`

If you have any questions, you are welcome to post them on the LMS discussion board *so long as you do not reveal details about your own solutions.* You can also email Toby Murray (toby.murray@unimelb.edu.au). In your message, make sure you include COMP90038 in the subject line. In the body of your message, include a precise description of the problem.

Late Submission and Extension

Late submission will be possible, but a late submission penalty will apply of 3 marks (10% of the assignment) per day.

Extensions will only be awarded in extreme/emergency cases, assuming appropriate documentation is provided simply submitting a medical certificate on the due date will not result in an extension.