THE UNIVERSITY OF MELBOURNE
SCHOOL OF COMPUTING AND INFORMATION SYSTEMS
COMP90038 ALGORITHMS AND COMPLEXITY

Assignment 2, Semester 2, 2020

Released: Wednesday the 14th of October. Deadline: Sunday the 1st of November 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. To develop problem-solving and design skills. To develop skills in analysis and formal reasoning about complex concepts. To improve written communication skills; in particular the ability to use pseudo-code and present algorithms clearly, precisely and unambiguously.

Problems

1. [4 Marks] Consider two sets of integers represented in arrays, $X = [x_1,\ x_2,\ \ldots,\ x_n]$ and $Y = [y_1,\ y_2,\ \ldots,\ y_n]$. Write two versions of a FINDSETUNION(X, Y) algorithm to find the union of X and Y as an array. An element is in the union of X and Y if it appears in at least one of X and Y.

   You may make use any algorithm introduced in the lectures to help you develop your solution. That is, you do not have to write the 'standard' algorithms – just use them. Therefore, you should be able to write each algorithm in about 10 lines of code. **You must include appropriate comments in your pseudocode.**

   (a) [2 Marks] Write a pre-sorting based algorithm of FINDSETUNION(X, Y). Your algorithm should strictly run in $\mathcal{O}(n \log n)$.

   R/
   The approach presented below will result in the set not being sorted.

```
function FINDSETUNION($X, Y, n$)
    MERGESORT($Y$)                                  ▷ Sorting is $\mathcal{O}(n \log n)$
    $Z[0, \cdots, 2n-1] \leftarrow [0, \cdots, 0]$
    for $i \leftarrow 0$ to $n-1$ do                ▷ The complete loop is $\mathcal{O}(n)$
        $Z[i] \leftarrow Y[i]$
    $j \leftarrow 0$
    for $i \leftarrow 0$ to $n-1$ do                ▷ The complete loop is $\mathcal{O}(n \log n)$
        if BINSEARCH($Y, X[i]$) $==$ $FALSE$ then
            $Z[n+j] \leftarrow X[i]$
            $j \leftarrow j+1$
    $W[0, \cdots, n+j-1] \leftarrow [0, \cdots, 0]$
    for $i \leftarrow 0$ to $n+j-1$ do              ▷ The complete loop is $\mathcal{O}(n)$
        $W[i] \leftarrow Z[i]$
    return $W$
```

   (b) [2 Marks] Write a Hashing based algorithm of FINDSETUNION(X, Y). Your algorithm should run in $\mathcal{O}(n)$.

R/

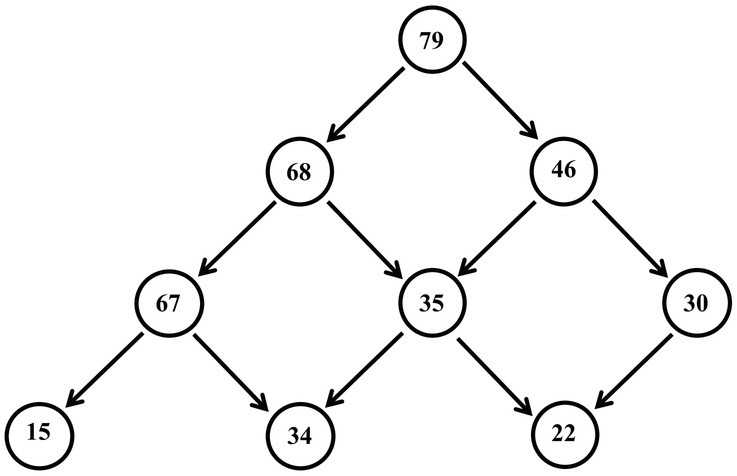
Assuming that the Hash table is correctly implemented, both loops are linear scans that should be completed in $\mathcal{O}(n)$.

```
function FindSetUnion($X, Y, n$)
    $H \leftarrow$ InitializeHashTable()
    for $i \leftarrow 0$ to $n-1$ do                              ▷ The complete loop is $\mathcal{O}(n)$
        Insert($H, Y[i]$)
    $Z[0, \cdots, n-1] \leftarrow [0, \cdots, 0]$
    for $i \leftarrow 0$ to $n-1$ do                              ▷ The complete loop is $\mathcal{O}(n)$
        $Z[i] \leftarrow Y[i]$
    $j \leftarrow 0$
    for $i \leftarrow 0$ to $n-1$ do                              ▷ The complete loop is $\mathcal{O}(n)$
        if Search($H, X[i]$) $==$ $FALSE$ then
            $Z[n+j] \leftarrow X[i]$
            $j \leftarrow j+1$
    $W[0, \cdots, n+j-1] \leftarrow [0, \cdots, 0]$
    for $i \leftarrow 0$ to $n+j-1$ do                            ▷ The complete loop is $\mathcal{O}(n)$
        $W[i] \leftarrow Z[i]$
    return $W$
```

2. [12 Marks] A *web* is a data structure similar to a heap that can be used to implement the priority queue ADT. As with a heap, a web is defined by two properties:

   - **Structural property:** A web W consists of l levels, $l \geq 0$. The ith level, for $0 \leq i < l$, contains at most $i + 1$ entries, indicated as $W_{i,j}$ for $0 \leq j \leq i$. All levels but the last are completely filled, and the last level is left-justified.
   - **Ordering property:** Any node $W_{i,j}$ has at most two *children*: $W_{i+1,j}$ and $W_{i+1,j+1}$, if those nodes exits. The priority of a node is always greater than or equal to the priority of either child node.

   For example, the following diagram shows a web with 9 nodes and 4 levels. The arrows indicate "$\geq$" relationships.



   (a) [1+2+2+2 = 7 Marks] A web W with n nodes can be stored in an array A of size n, similarly to an array-based heap. So, for example, if $n \geq 1$, the top of the web $W_{0,0}$ will be stored at $A[0]$. Give formulas for the array index that the following web entires will have, justifying how you arrived at these conclusions. Assume the n, i and j are such that all indicated web nodes actually exist.

      i [1 Mark] $W_{i,j}$

R/ Consider a full web, with corresponding array A:

$$\begin{tikzcd}
 & & & W_{0,0} \arrow[dl] \arrow[dr] & & & \\
 & & W_{1,0} \arrow[dl] \arrow[dr] & & W_{1,1} \arrow[dl] \arrow[dr] & & \\
 & W_{2,0} \arrow[dl] \arrow[dr] & & W_{2,1} \arrow[dl] \arrow[dr] & & W_{2,2} \arrow[dl] \arrow[dr] & \\
W_{3,0} & & W_{3,1} & & W_{3,2} & & W_{3,3}
\end{tikzcd}$$

$$A = [\boldsymbol{W_{0,0}}, \boldsymbol{W_{1,0}}, \boldsymbol{W_{1,1}}, \boldsymbol{W_{2,0}}, \boldsymbol{W_{2,1}}, \boldsymbol{W_{2,2}}, \boldsymbol{W_{3,0}}, \boldsymbol{W_{3,1}}, \boldsymbol{W_{3,2}}, \boldsymbol{W_{3,3}}]$$

Assuming the ith level is full, the total number of nodes in the web is $(i+1) + i + (i-1) + \cdots + 1 = \frac{1}{2}(i+1)(i+2)$. We use this fact to determine how many positions in the array have been taken before the ith level.

Count the number of array positions that will be taken up by the level directly before level i. We can find this by looking at the total number of nodes before level i begins, making the replacement $i \to i-1$ in $\frac{1}{2}(i+1)(i+2)$. We then add the j value. This means the key value for the node $W_{i,j}$ will be at index $\frac{1}{2}i(i+1)+j$, if we assume the indexing begins at 0 ($A[0]$). That is: $W_{i,j} = A[\frac{1}{2}i(i+1)+j]$.

ii [2 Marks] Left and right children of $W_{i,j}$

R/ We build on the result from part i. Examine the general array $A = [W_{0,0}, W_{1,0}, W_{1,1}, W_{2,0}, W_{2,1}, W_{2,2}, W_{3,0}, W_{3,1}, W_{3,2}, W_{3,3}, \cdots]$ and notice the child's "i" value increases by 1, and the "j" for the left child is unaltered while the "j" value for the right increases by 1. So the left child of $W_{i,j}$ will be at index $\frac{1}{2}(i+1)(i+2)+j$ and the right will be at $\frac{1}{2}(i+1)(i+2)+j+1$.

iii [2 Marks] Left and right parents of $W_{i,j}$

R/ We use the same process as in part ii to find the index for the left and right parents of $W_{i,j}$. However, note that nodes that are on the surface of the web only have one parent.

if $j = 0$:	no left parent	
	right parent	$\frac{1}{2}(i-1)i+j$
if $0 < j < i$:	left parent	$\frac{1}{2}(i-1)i+j-1$
	right parent	$\frac{1}{2}(i-1)i+j$
if $j = i$:	left parent	$\frac{1}{2}(i-1)i+j-1$
	no right parent	

iv [2 Marks] Left and right children of the node corresponding to $A[k]$

R/ Left and right children of the node corresponding to $A[k]$. From part i we know that $k = \frac{1}{2}i(i+1) + j$, but this is not quite enough information to solve this question. Consider the $j = 0$ case ($k = \frac{1}{2}i(i+1)$) and solve for i: $i = \frac{1}{2}\left(-1 \pm \sqrt{8k+1}\right)$, i is a positive integer, so take the positive solution $i = \frac{1}{2}\left(-1 + \sqrt{8k+1}\right)$. To take the non-zero j cases into account, use the FLOOR function:

$$i(k) = \text{FLOOR}\left[\frac{1}{2}\left(-1 + \sqrt{8k+1}\right)\right].$$

We can then find j as a function of array index k:

$$j(k) = k - \frac{1}{2}i(k)(i(k)+1).$$

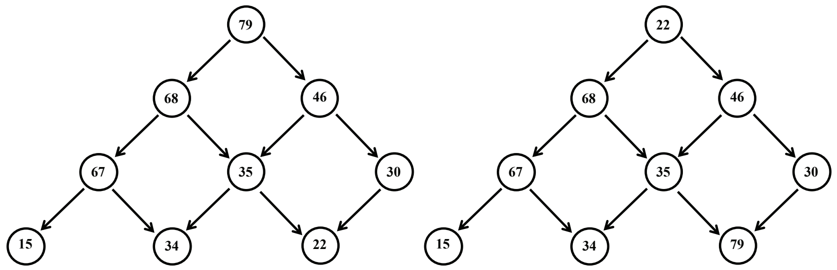
To find the children for array element $A[k]$, just use the results from part ii: left: $\frac{1}{2}(i(k)+1)(i(k)+2) + j(k)$; right : $\frac{1}{2}(i(k)+1)(i(k)+2) + j(k) + 1$.

(b) [2 Marks] Give upper and lower bounds for the number of nodes n in a web W with l levels, where $l \geq 1$. For example, a web with 2 levels has at least 2 and at most 3 nodes. Make your bounds as tight as possible. Briefly justify your answer.

R/ The upper bound will be when there is a full tree: $n = \frac{1}{2}l(l-1) = \frac{1}{2}l^2 + \frac{1}{2}l$.
The lower bound will be when the bottom level has only one node: $n = \underbrace{\frac{1}{2}(l-1)l}_{\text{nodes for } l-1 \text{ levels}} + 1 = \frac{1}{2}l^2 - \frac{1}{2}l + 1$.

(c) [3 marks] Briefly describe an efficient algorithm to eject the maximal element from the web W that is stored in an array A of size n. Find the complexity of this algorithm and justify your conclusion Note: you do not have to write this algorithm in pseudocode. We are expecting that you write a short paragraph or a short list of bullet points describing the important steps of the algorithm and explaining the time complexity.

R/ Consider the case that we wish to eject the maximal element 79 from the web. We swap 79 with the last element in the array A, 22 in this example.



Now we need to use the "sift down" type of operations we saw for ejecting maximal elements from a heap. Counting these sift down operations will give the time complexity. We need to swap the new key at $W_{0,0}$ down $l-1$ levels. From part (b), we know that the total number of nodes n has an upper and lower bound: $\frac{1}{2}l^2 - \frac{1}{2}l + 1 \leq n \leq \frac{1}{2}l^2 + \frac{1}{2}l$. If l is large, $n \approx \frac{1}{2}l^2$. So swapping l levels corresponds to $\mathcal{O}(\sqrt{n})$ operations.

3. [14 Marks] Researchers from the School of BioSciences have requested our help with one of their experiments. They are performing behavioural experiments with zebrafish. At any one instance in

time there are a large number of zebrafish in the aquarium. For their particular experiment, the biologist take a snapshot of the aquarium and then need to find the longest series of zebrafish such the length of each fish along the horizontal direction in the aquarium is increasing. They also need to know the number of zebra fish in this series.

For example, the snapshot of the aquarium resulted in fish lengths of $[2, 5, 3, 7, 11, 1, 12, 4, 15, 14, 6, 16]$. One possible longest series of increasing lengths in this case is $[2, 3, 7, 11, 12, 14, 16]$ with 7 zebrafish. We say one possible longest series of increasing lengths here because it is not necessarily unique. For example, the length 14 in the output could be replaced with 15: $[2, 3, 7, 11, 12, 15, 16]$ and also be valid.

In this question you will consider algorithms for finding the longest series of increasing lengths via the function LONGESTINCREASINGLENGTHS($A[0, \cdots, n-1]$), as well as the size of this output array.

(a) [1+2+1 = 4 Marks] Consider a recursive algorithm:

i [1 Mark] Write down a recurrence relation for the function LONGESTINCREASINGLENGTHS.

R/ The input array is given by $A[0, \cdots, n-1]$. Denote the array size of the longest series of increasing lengths at element $A[i]$ by sizeLIL(i), where $A[i]$ must be included. sizeLIL(i) can be written recursively as:

$$\text{sizeLIL}(i) = \begin{cases} \text{MAXIMUM}\,[\text{sizeLIL}(j)] + 1, & \text{where } 0 \le j < i \text{ and } A[j] < A[i] \\ 1, & \text{if no such } j \text{ exists,} \end{cases}$$

with base case: sizeLIL($j = 0$) = 1. To find the longest series of increasing lengths, we consider MAXIMUM(sizeLIL(i)) over $0 \le i < n$.

ii [2 Marks] Using this recurrence relation, write a recursive algorithm in pseudocode for LONGESTINCREASINGLENGTHS that only calculates the array size of the longest series of increasing lengths. You do not need to output the actual array containing the longest series of increasing lengths in this part of the question. For the example above with input $A = [2, 5, 3, 7, 11, 1, 12, 4, 15, 14, 6, 16]$, the output should just be 7. The pseudocode should be about 10 lines of code.

R/
Gives the array size of the longest series of increasing lengths: sizeLIL(A, i, n, prev). Call this function via: sizeLIL$(A, 0, n, -\infty)$ for input array A of size n.

```
function sizeLIL($A, i, n$, prev)

    if $i$ == $n$ then                          ▷ Base Case: nothing is remaining
        return 0

    exclude ← sizeLIL($A, i+1, n$, prev)        ▷ Case 1: exclude the current element
                                                ▷ and process the remaining elements

    include ← 0                                 ▷ Case 2: include the current element if
    if $A[i]$ > prev then                       ▷ it is greater than previous sizeLIL
        include ← 1+ sizeLIL($A, i+1, n, A[i]$)

    return maximum(include, exclude)
```

Another equally valid top-down solution (which also has exponential time complexity) is given below :

```
function LongestIncreasingLengths($A, n$)
    result ← 0
    for $i$ ← 0 to $n-1$ do
        result ← maximum(result, sizeLIL($A, i+1$))
    return result
function sizeLIL($A, n$)        ▷ This function finds the longest increasing sequence
                                ▷ from the first $n$ elements, the sequence must
                                ▷ include $A[n-1]$.
    if $n$ ≤ 1 then             ▷ Base Case for an array containing 0 or 1 element
        return n
    result ← 1                                  ▷ Variable to store final result
    for $i$ ← 0 to $n-2$ do
        if $A[i]$ < $A[n-1]$ then       ▷ If the last element is larger than the $i$th
                                        ▷ element, we can extend the sequence
                                        ▷ ending at $A[i-1]$.
            result ← maximum(result, sizeLIL($A, i+1$) + 1)
    return result
```

iii [1 Mark] What is the time complexity of this recursive algorithm? Justify your answer.

R/ The recursive algorithm sizeLIL(A, i, n, prev) calls itself more than once. Therefore the complexity is exponential, since each following call to the function sizeLIL(A, i, n, prev) will also call itself more than once. In this case the recursive algorithm calls itself twice, so the time complexity is $\mathcal{O}(2^n)$.

(b) [5+1+1 = 7 Marks]

i [5 Marks] Building on from your recursive algorithm in part (a), write down a dynamic programming implementation in pseudocode for the function LongestIncreasingLengths$(A[0, \cdots, n-1])$ to find the longest series of increasing lengths. This should also output the size of the longest series of increasing lengths. The pseudocode should be about 20 lines of code.

R/
Gives both the array size and array containing the longest series of increasing lengths: LIL(A, n). Call this function via: sizeLIL, arrayLIL ←LIL(A, n) for input array A of size n.

```
function LIL($A, n$)

    LILarray ← $0 \times A + 1$   ▷ Initialise longest series of increasing lengths values
                                  ▷ to 1 for all indexes
    maxLength ← 1                 ▷ Setup initial array length of longest series of
                                  ▷ increasing lengths
    bestEnd ← 1                   ▷ Setup best final array position in case input
                                  ▷ array is reverse sorted

                                  ▷ Bottom up Dynamic Programming
    for $i \leftarrow 1$ to $n-1$ do         ▷ The complete nested loop is $\mathcal{O}\left(n^2\right)$
        for $j \leftarrow i-1$ to 0 do
            if $A[i] > A[j]$ and LILarray[$i$] $<$ LILarray[$j$] $+ 1$ then
                LILarray[$i$] ← LILarray[$j$]+1
        if LILarray[$i$] $>$ maxLength then
            bestEnd ← $i$
            maxLength ← LILarray[$i$]

    maximum ← LILarray[bestEnd]   ▷ Find maximum number of
                                  ▷ elements in LIL

    for $j \leftarrow 0$ to maxLength-1 do   ▷ Pick out LIL elements
        LILoutput[$i$] ← 0
    LILoutput[maxLength-1]← $A$[bestEnd]

    $j$ ←maxLength
    for $i \leftarrow$ bestEnd-1 to 0 do
        if LILarray[$i$] $==$ $j-1$ then
            LILoutput[$j-2$] ← $A[i]$
            $j \leftarrow j-1$

    return maximum, LILoutput
```

ii [1 Mark] Explain how the recurrence relation used for your dynamic programming implementation involves overlapping instances.

R/ The bottom up dynamic programming algorithm LIL(A, n) was built from the recursive algorithm in part (a). We can see this from the **if** loop within the nested **for** loop:

```
    if $A[i] > A[j]$ and LILarray$[i]$ < LILarray$[j] + 1$ then
            LILarray$[i]$ ← LILarray$[j]$+1
```

Here we can see that as we scan over the input array A: we ask the question for each element $A[i]$: should this element be included as part of the array containing the longest series of increasing lengths? We use a similar condition to the one we saw in the recursive algorithm in part (a) to decide this. For each element $A[i]$ we need to do a scan over all elements at index positions $i-1$ to 0 in order to decide this.

iii [1 Mark] What is the time complexity of your algorithm and how much auxiliary space was required. Justify your answer.

R/ The bottom up dynamic programming algorithm LIL(A, n) has one nested **for** loop, so the upper bound on the time complexity is $\mathcal{O}\left(n^2\right)$, at worst. This algorithm uses an array of size n as auxiliary space.

(c) [1+2 = 3 Marks] The time complexity of the recursive algorithm for LongestIncreasingLengths was exponential, while the dynamic programming algorithm lead to a polynomial time complexity (note, you need to determine that polynomial above). Here we will investigate an algorithm for the function LongestIncreasingLengths that has a time complexity of $\mathcal{O}(n \log n)$.

Consider building a set of arrays for the input array $A[0, \cdots, n-1]$. As we scan along A, we will compare $A[i]$ with the final element in each array in this set. This comparison will satisfy the following conditions:

(1) If $A[i]$ is smaller than the final element in each array, start a new array of size 1 with $A[i]$.

(2) If $A[i]$ is larger than the final element in each array, copy the longest array and append $A[i]$ to this new array.

(3) If $A[i]$ is in between, find the array with the **smallest**[1] final element that is greater than $A[i]$ and replace that element with $A[i]$.

i [1 Mark] Write down the set of arrays that satisfy these rules for the input array $A = [0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15]$.

[1]clarification from LMS

R/

$A[i] = 0$	[0]
$A[i] = 8$	[0]
	[0,8]
$A[i] = 4$	[0]
	[0,4]
$A[i] = 12$	[0]
	[0,4]
	[0,4,12]
$A[i] = 2$	[0]
	[0,2]
	[0,4,12]
$A[i] = 10$	[0]
	[0,2]
	[0,4,10]
$A[i] = 6$	[0]
	[0,2]
	[0,4,6]
$A[i] = 14$	[0]
	[0,2]
	[0,4,6]
	[0,4,6,14]
$A[i] = 1$	[0]
	[0,1]
	[0,4,6]
	[0,4,6,14]
$A[i] = 9$	[0]
	[0,1]
	[0,4,6]
	[0,4,6,9]

$A[i] = 5$	[0]
	[0,1]
	[0,4,5]
	[0,4,6,9]
$A[i] = 13$	[0]
	[0,1]
	[0,4,5]
	[0,4,6,9]
	[0,4,6,9,13]
$A[i] = 3$	[0]
	[0,1]
	[0,4,3]
	[0,4,6,9]
	[0,4,6,9,13]
$A[i] = 11$	[0]
	[0,1]
	[0,4,3]
	[0,4,6,9]
	[0,4,6,9,11]
$A[i] = 7$	[0]
	[0,1]
	[0,4,3]
	[0,4,6,7]
	[0,4,6,9,11]
$A[i] = 15$	[0]
	[0,1]
	[0,4,3]
	[0,4,6,7]
	[0,4,6,9,11]
	[0,4,6,9,11,15]

ii [2 Marks] Building from these conditions, explain how an algorithm for the function LongestIncreasingLengths could run with time complexity $\mathcal{O}(n \log n)$ **to find the array size of the longest series of increasing lengths, as we did in question (3a)**[2]. You may make use any algorithm introduced in the lectures to help you with your explanation. Note: you do not have to write this algorithm in pseudocode. We are expecting that you write a short paragraph or a short list of bullet points describing the important steps of the algorithm to explain the time complexity.
Hint: what if you only consider the final elements of this set of arrays as a single array?

[2]also clarification from LMS

R/ The number of arrays in the set after we apply the three conditions to an input A gives the array size of the longest series of increasing lengths. We could find this by using the hint, and only keeping track of the final element from each array in the set.

Given the three conditions above, we can see that the final element from each array in the set is in ascending order. This is clear when we use the hint and only keep track of the final elements. For the example, consider the input array A from part (a) above. The array of final elements after scanning through the input A is given by:

$A[i] = 0$	[0]
$A[i] = 8$	[0,8]
$A[i] = 4$	[0,4]
$A[i] = 12$	[0,4,12]
$A[i] = 2$	[0,2,12]
$A[i] = 10$	[0,2,10]
$A[i] = 6$	[0,2,6]
$A[i] = 14$	[0,2,6,14]
$A[i] = 1$	[0,1,6,14]
$A[i] = 9$	[0,1,6,9]
$A[i] = 5$	[0,1,5,9]
$A[i] = 13$	[0,1,5,9,13]
$A[i] = 3$	[0,1,3,9,13]
$A[i] = 11$	[0,1,3,9,11]
$A[i] = 7$	[0,1,3,7,11]
$A[i] = 15$	[0,1,3,7,11,15]

Scanning through the input array A requires a time complexity of $\mathcal{O}(n)$. However, after we consider each element $A[i]$, we need to apply the three conditions above. In order to decide where in the array of final set elements $A[i]$ should be put will require a search function. Since the final set elements are in sorted order, we can use BinarySearch, which has a time complexity of $\mathcal{O}(\log n)$. Combining this with the linear scan through each element in A, the overall time complexity would be $\mathcal{O}(n \log n)$.

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.

- Number of lines are given as an **indication only** and should not be considered as the actual length of the code. Correct solutions could have a few lines more or less depending on your notation, but not many more (if you see yourself writing ten more extra lines, then you are probably doing something wrong).

- 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 the Lecturer, Casey Myers (Casey.Myers@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.