MONASH University

Office Use Only

Semester Two 2015
Examination Period

Faculty of Information Technology

EXAM CODES: **FIT1008**
TITLE OF PAPER: **COMPUTER SCIENCE – PAPER 1**
EXAM DURATION: 3 hours writing time
READING TIME: 10 minutes

THIS PAPER IS FOR STUDENTS STUDYING AT:(tick where applicable)

☐ Berwick ✓ Clayton ✓ Malaysia ☐ Off Campus Learning ☐ Open Learning
☐ Caulfield ☐ Gippsland ☐ Peninsula ☐ Enhancement Studies ☐ Sth Africa
☐ Parkville ☐ Other (specify)

During an exam, you must not have in your possession, a book, notes, paper, electronic device/s, calculator, pencil case, mobile phone, smart watch/device or other material/item which has not been authorised for the exam or specifically permitted as noted below. Any material or item on your desk, chair or person will be deemed to be in your possession. You are reminded that possession of unauthorised materials, or attempting to cheat or cheating in an exam is a discipline offence under Part 7 of the Monash University (Council) Regulations.

No exam paper or other exam materials are to be removed from the room.

AUTHORISED MATERIALS

OPEN BOOK	☐ **YES**	✓ **NO**
CALCULATORS	☐ **YES**	✓ **NO**
SPECIFICALLY PERMITTED ITEMS	☐ **YES**	✓ **NO**

if yes, items permitted are:

Candidates must complete this section if required to write answers within this paper

STUDENT ID: __ __ __ __ __ __ __ __ __ DESK NUMBER: __ __ __ __ __

Page		Mark
3		7
5		7
7		4
9		5
11		12
13		7

Page		Mark
15		8
17		5
19		4
21		8
23		7
25		6
Total:		80

Blank Page

Question 1 (6 + 1 = 7 marks)

This question is about *sorting algorithms*.

(a) What is the best and worst case complexity in Big O of the three simple sorting algorithms: *Bubble Sort*, *Selection Sort*, and *Insertion Sort*? (*No explanation means no marks*).

The worst complexity for the three simple sorting algorithms are the same of $O(N^2)$, given that each of these algorithms involves two nested loops each of which depends linearly on N. The situation is where the list is sorted in the reverse order.

As for the best complexity, both Bubble Sort and Insertion Sort could achieve O(N). In the case of BubbleSort, if no swaps had taken place during the first round of iteration through the list, i.e. the list is already sorted and no further iterations required. As for Insertion Sort, if the list is sorted, no shuffles would take place at each iteration.

However, Selection Sort will not have a best case, given that neither of the two nested loops can terminate early, i.e. both best and worst complexity are the same of $O(N^2)$.

For each algorithm:

1 mark for reasoning the best case complexity of each sorting algorithm.
1 mark for reasoning the worst case complexity of each sorting algorithm.
No marks if without explanation;
If the explanation is poorly reasoned but present; award 0.5 marks.

(b) Which of the three sorting algorithms as mentioned above is not a *stable* sorting method? (*No explanation means no marks*).

Selection Sort. The relative order of two elements which are of the same value is altered since elements are swapped at a distance. Consider the following list [2, 2, 1]. After the 1 has been swapped to the start of the lists, the order of the 2's have been changed.

0.5 marks for the correct algorithm mentioned and 0.5 marks for explaining why it is not stable.

Blank Page

Question 2 (5 + 2 = 7 marks)

This question is about *searching*.

(a) Write a Python function,

```
def find_sum(array, k)
```

which given a sorted array, **`array`**, and a value k, returns a tuple, **`(i,j)`**, of two distinct indexes i and j such that **`array[i] + array[j] = k`**. The algorithm should return **`None`** if no such indexes exist.

Examples:

`find_sum([1, 2, 3, 4, 5], 5)` returns either **`(0, 3)`** because 1 and 4 sum to 5, or **`(1, 2)`** because 2 and 3 sum to 5.

`find_sum([1, 2, 3, 4, 5], 8)` returns **`(2, 4)`** because 3 and 5 sum to 8.

`find_sum([1, 2, 3, 4, 5], 2)` returns **`None`**

```
def find_sum_k(sorted_array, k):
    low = 0
    high = len(sorted_array) -1
    while(low != high):
        if (sorted_array[low] + sorted_array[high] == k):
            return low, high
        elif (sorted_array[low] + sorted_array[high] > k):
            high = high - 1
        else:
            low = low + 1
    return None
```

1 mark for returning None when there are no indexes.
2 marks for considering possible pairs
1 mark for making sure the indexes satisfy array[i]+array[j]=k
1 mark for making sure the indexes are distinct.

(b) Explain the worst time complexity for the function you defined in part **(a)**. (*No explanation no marks*).

The worst time complexity is linear O(N). This would occur if k was less than twice the minimum value of the list. In this case you would go through the list, then return None.

2 marks for explanation and correct complexity.
1 mark for poor explanation.

Blank Page

Question 3 (1 + 1 + 1 + 1 = 4 marks)

This question is about *time complexity*. For each of the given Python functions, explain the worst time complexity. (*No explanation means no marks.*)

(a)
```
def total_func(n):
    total = 0
    for k in range(n):
        for num in range(k):
            total += num
```

O(n^2), as there are two nested loops. The outer loop is executed n times, and the number of times the inner loop is executed depends upon n.

(b)
```
def division_func(n):
    product = 1
    while n > 1:
        product *= 2
        n //= 2
```

O(logN). The loop is executed k times, where $n = 2^k$. So, $k = \log_2 n$.

(c)
```
def another_total_func(n):
    total = 0
    for num in range(n):
        total += num
    for num in range(2*n):
        total += num
```

O(n) .There are two loops. The first loop is iterated n times. The second loops is iterated 2*n times. Therefore the number of steps during execution is linear in terms of n.

(d)
```
def mid_func(n):
    low = 0
    high = n
    while low < high:
        mid = (low + high) // 2
        low = mid + 1
```

O(logN). In the worst case the loop is executed k times, where $n = 2^k$. So, $k = \log_2 n$.

1 mark for each correct answer, must have an explanation.

4

Blank Page

Question 4 (5 marks)

This question is about *searching*. Write a Python function,

`def max_repetitions(a_list)`

which given a list, **`a_list`**, sorted in ascending order, finds an element that appears the maximum number of times. *Note there can be more than one element that appears the maximum number of times.*

Examples:

`max_repetitions([1, 1, 2, 2, 2, 3, 2])` returns **`2`**.
`max_repetitions([1, 1, 1, 1, 1, 1, 1])` returns **`1`**.

`max_repetitions([])` returns **`None`**

`max_repetitions([1, 2, 3, 4, 5, 6])` returns either **`1`**, **`2`**, **`3`**, **`4`**, **`5`**, or **`6`**.

```
def max_repetitions(array):
    n = len(array)
    if n ==0:
        return None

    count = 0
    maximum = 0
    for i in range(n):
        count = 1
        for j in range(n):
            if (i!=j and array[i]== array[j]):
                count +=1
        if maximum < count:
            maximum = count
            max_repeated = array[i]

    return max_repeated
```

1 mark for dealing with an empty list
3 marks for finding the maximum
1 mark for returning the maximum.

5

Blank Page

Question 5 (10 + 2 = 12 marks)

(a) This question is about *MIPS programming and understanding function calls*. Translate the following Python code faithfully into MIPS assembly language. Make sure you follow the MIPS function calling and memory usage conventions.

Python Code	MIPS Code

`def func(n):`	func:
	addi $sp, $sp, -8
	sw $ra, 4($sp)
	sw $fp, 0($sp)
	addi $fp, $sp, 0
	
	2 marks

`if n == 0:`	lw $t0, 8($fp) # arg n bne $t0, $0, else 1 mark

`return 0`	addi $v0, $0, 0 # return 0
	lw $fp, 0($sp)
	lw $ra, 4($sp)
	addi $sp, $sp, 8
	jr $ra
	
	2 marks

`else:`
`return n + func(n-1)`
	else:
	lw $t0, 8($fp)
	addi $t0, $t0, -1 # n -1
	addi $sp, $sp, -4
	sw $t0, 0($sp)
	jal func
	2 marks
	addi $sp, $sp, 4 # deallocate arg
	lw $t0, 8($fp)
	add $t0, $t0, $v0 # n + func(n-1)
	addi $v0, $t0, 0
	2 marks
	lw $fp, 0($sp)
	lw $ra, 4($sp)
	addi $sp, $sp, 8
	jr $ra
	1 mark

(b) Explain why a recursive function may use more memory than an iterative function.

Recursive functions may use more memory, as every function call will have its own stack frame associated with it. Each of these stack frames has its own copies of local variables, arguments, etc. Given that recursive functions are functions that call themselves as 'iterations', every recursive call is a new stack frame. On the other hand, an iterative function has iterations within the same function to perform the task rather than calling new functions; only one stack frame is needed.

2 marks for a complete explanation mentioning stack frames and why recursive functions have many while iterative functions have just one. (1 mark if reasoning is unclear.)

12

Question 6 (5 + 2 = 7 marks)

(a) This question is about *recursion* and *binary trees*.

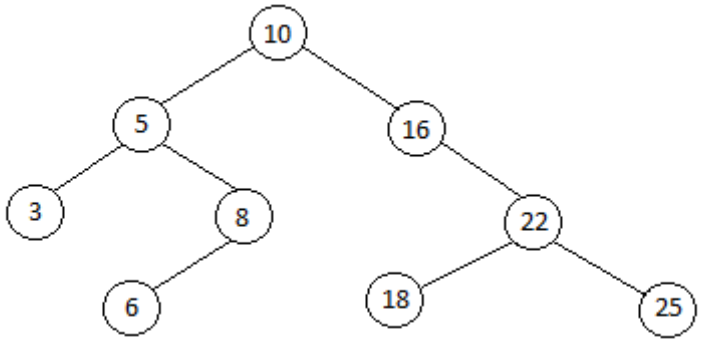
Consider the two classes **`BinaryTreeNode`** and **`BinaryTree`** which define a **Binary Data type** implemented using linked nodes, and which are defined as follows:

```
class BinaryTreeNode:
      def __init__ (self, value, left=None, right=None):
            self.value = value
            self.left = left
            self.right = right

class BinaryTree:
      def __init__ (self):
            self.root = None
```

Define a recursive method **`find_path_max_sum(self)`** inside the **`BinaryTree`** class that finds the maximum sum of values of the nodes within a path from the root to a leaf. The method returns the maximum sum.

For example, if the binary tree is:



there are 4 paths from the root to a leaf. The sum of the values in these paths are 18, 29, 66, and 73. So in this case the method should return **`73`**. If the binary tree is empty, the value return should be zero.

Write your answer of this part on the next page

Write your answer for Question 6(a) here

```
def find_path_max_sum(self):
    current = self.root
    if current is None:
        return 0
    return rec_find_path_max_sum(current)

def rec_find_path_max_sum(current)
    left_sum = None
    right_sum = None

    if not current.left is None:
        left_sum = rec_find_path_max_sum(current.left)

    if not current.right is None:
        right_sum = rec_find_path_max_sum(current.right)

   if left_sum is not None and right_sum is not None:
        if left_sum < right_sum:
            return current.value + right_sum

        return current.value + left_sum
   elif left_sum is not None:
        return current.value + left_sum
   else:
        return current.value + right_sum
```

1 mark for having a correct base case.
1 mark for reducing the problem every iteration.
1 mark for setting up the first call to the recursive function.
2 marks for correctness.
-0.5 if the possibility of negative values is not taken into account.

(b) Explain the best and worst time complexity of the function you defined in part **(a)**. (*No explanation means no marks*).

The worst and the best time complexity are both O(N), where N is the number of items in the binary tree. This because every node is visted once.

1 mark for best time complexity and explanation.
1 mark for worst time complexity and explanation.

7

Blank Page

Question 7 (2 + 3 + 3 = 8 marks)

Consider the partial implementation of Circular Queue based on an array.

```
class CircularQueue:
    def __init__(self, size):
        assert size > 0, "Size should be positive"
        self.the_array = size*[None]
        self.count = 0
        self.rear = 0
        self.front = 0

    def is_empty(self):
        return self.count == 0

    def is_full(self):
        return self.count >= len(self.the_array)
```

(a) Define method reset, and state its worst case complexity

```
def reset(self):
        self.front = 0
        self.rear = 0
        self.count = 0
```

Worst time complexity is O(1).

1 mark for code. 1 mark for time complexity.

(b) Define method append, and state its worst case complexity

```
def append(self, new_item):
        assert not self.is_full(), "Queue is full"
        self.the_array[self.rear] = new_item
        self.rear = (self.rear+1)% len(self.the_array)
        self.count += 1
```

Worst time complexity is O(1).

2 marks for code. 1 mark for time complexity.

(c) Define method serve, and state its worst case complexity

```
def serve(self):
        assert not self.is_empty(), "Queue is empty"
        item = self.the_array[self.front]
        self.front +=1
        if self.front == len(self.the_array):
            self.front = 0
        self.count -=1
        return item
```

Worst time complexity is O(1).

2 marks for code. 1 mark for time complexity

8

Blank Page

Question 8 (5 marks)

This question is intended to test your skills at programming with *queues* and *stacks.*

Suppose you have a **`Queue`** class which implements a queue using some data structure (you do not need to know which one) and defines the following methods:

```
__init__()
append(item)
serve()
is_empty()
```

and a **`Stack`** class which implements a stack using some data structure (you do not need to know which one) and defines the following methods:

```
__init__()
push(item)
pop()
is_empty()
```

Define a function:

`def reverse_k(a_queue, k)`

which is given a queue, **`a_queue`**, and a non-negative integer **k**, reverses the first **k** elements of **`a_queue`** while keeping the rest of the elements in the given order. If **k = 0**, then **`a_queue`** should remain unchanged.

You must only use the methods defined for the above classes.

```
def reverse_k(a_queue, k):
    tmp_stack = Stack()
    for _ in range(k);
       if not a_queue.is_empty():
           tmp_stack.push(a_queue_serve())

    tmp_queue = Queue()
    while not stack.is_empty():
        tmp_queue.append(stack.pop())

    while not a_queue.is_empty():
        tmp_queue.append(a_queue.serve())

    while not tmp_queue.is_empty():
        a_queue.append(tmp_queue.serve())
```

2 marks for revering the first k elements
2 marks for appending the rest of the elements to the queue
1 mark for making sure at the end the correct queue is in a_queue

-0.5 marks if they do not initialize a stack or queue.
-0.5 marks Minor mistakes (e.g., initialising data structures with size, major syntax issues)
- 1 marks for not properly using the interface given for either data structure. Or using other data structures.
Students are **not** penalised if they put the elements at the end of the queue as long as the elements are put back in the queue.

Blank Page

5

Question 9 (2 + 2 = 4 marks)

This question is about *recursive programming* and *linked structures*. Consider the two classes **`Node`** and **`List`** as seen in the lectures, which define a **list data type** implemented using a linked structure:

```
class Node:
    def __init__ (self, item = None, link = None):
        self.item = item
        self.next = link

class List:
    def __init__ (self):
        self.head = None

    def is_empty (self):
        return self.head is None
```

(a) Now consider the following code:

```
def mystery(a_list, node):
    if node is not None:
        right = node.next
        if a_list.head is not node:
            node.next = a_list.head
            a_list.head = node
        else:
            node.next = None
        mystery(a_list, right)

def my_function(a_list):
    mystery(a_list, a_list.head)
```

Describe what **`my_function`** does to a linked list?

It reverses a linked list.

2 marks for a correct answer.

(b) What is the worst-case complexity for **`my_function`**? (*No explanation means no marks*)

The worst time complexity is O(N)

2 marks for explanation.

4

Question 10 (1 + 2 + 5 = 8 marks)

This question is about *hash tables*.

(a) Why should the table size and the constants defined in a hash function be *prime*? Explain (*no explanation means no marks*).

Both the table size and the constants defined in a hash function should be prime in order to avoid common factors that could result in collisions.

0.5 marks for mentioning common factors and 0.5 marks for collisions.

(b) Under which situation is Linear Probing the preferred solution for addressing collisions as compared to Double Hashing and Quadratic Probing? Explain (*no explanation means no marks*).

If sufficient memory is available and the hash table is sufficiently sparse, Linear Probing is a preferred solution. With the hash table being kept under the load factor < 0.5, the chances of collision and clustering are lesser. Thus, the linear probe length is ensured to be low (as low as 1 step) since there is no need to perform a second hash function or go through a quadratic step.

1 mark for mentioning the right situation where Linear Probing is better; and 1 mark for reasoning.

(c) Consider the class **`Hash`** which has the instance variables **`array`**, **`table_size`**, and **`count`**, and the following methods:

```
__init__()
hash(the_key)
```

Using Linear Probing, define a method **`__getitem__(self, the_key)`** which does the following:

- If there is an entry in the hash table with the key, **`the_key`**, then it returns the value associated with **`the_key`**.
- If there is no entry with the key, **`the_key`**, it raises a **`KeyError`** exception.

Write your answer of this part on the next page

Write your answer for Question 10(c) here

Assuming each location in the array stores (key, value).

```
def __getitem__(self, the_key):

    position = self.hash(the_key)

    for i in range(self.table_size):

        if self.array[position] is None:          # key not found
            raise KeyError(the_key)

        elif self.array[position][0] == the_key:  # key found
            return self.array[position][1]

        else:
            position = (position + 1) % self.table_size

    raise KeyError(the_key)
```

5 marks for Question 4(c):

- 1 mark for using the hash function appropriately for the starting position to search.
- 1 mark for iterating through the hash table.
- 1 mark for updating the position as linear search if the key has not been found.
- 1 mark for returning the associated value if the key is found.
- 1 mark for handling the situation where the key is not found (i.e. raise KeyError exception).

8

Blank Page

Question 11 (1 + 1 + 5 = 7 marks)

This question is about *iterators*. Define a **`PositiveIterator`** iterator class which is defined on a list. An instance of this class iterates through all the positive elements of the list.

For example:

```
>>> itr = PositiveIterator([3,-2,-1,5,11])
>>> next(itr)
3
>>> next(itr)
5
>>> next(itr)
11
```

Your class must have the three methods: **`__init__`**, **`__iter__`** and **`__next__`**.

```
class PositiveIterator:

    def __init__(self, list):
        self.list = list
        self.current = 0                                  1 mark

    def __iter__(self):
        return self                                       1 mark

    def __next__(self):
        if self.current == len(lef.list)
             raise StopIteration                          1 mark

        item = self.list[self.current]
        while item < 0 and self.current < len(self.list):
            self.current += 1
            item = self.list[self.current]                2 marks

        if self.current == len(self.list):
            raise Stop Iteration                          1 mark

        self.current += 1
        return item                                       1 mark
```

1 mark for a correct __init__
1 mark for a correct __iter__

In __next__ : 1 mark for checking whether you at the end of the list
2 marks for finding the next positive number
1 mark for dealing with there are no more positive numbers
0.5 marks for updating the pointer to next item
0.5 marks for returning the item

7

Blank Page

Question 12 (5 + 1 = 6 marks)

This question is about *heaps.*

(a) Suppose a **max-heap** is represented using an array. Write a Python function,

`def is_valid_heap(array)`

which given an **`array`** and returns **True** if the **array** represents a **max-heap**, and **False** otherwise.

```
def is_valid_heap(array):
    n = len(array)
    k = 1

    while 2*k + 1 < n:
       if array[2*k] > array[k]:
            return False

       if array[2*k+1] > array[k]:
            return False

       k += 1

   if 2*k == n-1 and array[2*k] > array[k]:
       return False

   return True
```

(b) Explain the worst time complexity of the function you defined in part **(a)**. (*No explanation means no marks*).

The worst time complexity is O(N), where N is the number of items in the heap (or array). This occurs when the array is a heap and all the items need to be checked whether they satisfy the order condition.

1 mark for correct explanation and complexity.

End of Exam