Question 1 – Short questions [15 marks]

In this part you are required to answer the following short questions. Your answer should be concise. As a guideline, it should require no more space than the space that is provided.

(1) In MIPS, how many bits are required to store an array of 9 elements, where each element fits in one byte?

1*9 + 4 = 13 bytes or 104 bits (9 bytes for the content and 4 bytes for the size of the array)

1.5 mark for any valid expression of 13 bytes, no partial marks

(2) In MIPS, how many bits long is each of the temporary registers?

32 bits
1.5 mark for any valid expression of 32 bits

(3) Is it possible to translate all recursive solutions into iterative ones? Explain your answer.

Yes, but in some cases a Stack may be required to push the parameters that would normally be passed to the recursive function

1.5 marks iff Stack mentioned, 1 mark if reasoning without mentioning stack

(4) Is Quick-sort stable? Explain your answer.
Quick sort is not stable due to the partition. The swap with the pivot element can modify the relative order between two elements with the same key.
1.5 = 0.5 mark not stable + 1 mark evidence of stability and quicksort knowledge

(5) What is the role of the **PC** register in MIPS.
The program counter is a register that contains the memory address of the next instruction (or current instruction) **or** keeps track of program execution
1.5 marks, no partial marks

7.5

This page intentionally left blank, use if needed but it will not be marked.

(6) Consider a sorted linked list of integers, to which you are trying to add element X. What values of X will yield the best-case time complexity? And the worst?
Best when X is smallest (0.75 marks), worst when X is largest (0.75 marks)
1.5 marks

(7) List 3 different ways covered in the lectures to implement a Queue ADT

(a) Linear Array-based Queue
(b) Linked Queue
(c) Linear Circular Queue
(d) Priority Queue – based on time.

1.5 = 0.5 for each valid structure

(8) Tail recursion is ...
when the result of the recursive call is the result of the function **or** Nothing is done in the "way back". 1.5 marks

(9) What does get printed after executing the following code in Python?
`a = [1, 2, 3]; b = a; b.append(4); print(a)`
[1, 2, 3, 4]
1.5 marks

(10) The instance variables of a Node supporting a Binary Search Tree are:
the item and left, pointing to left subtree, and right pointing to right subtree
1.5 marks

7.5

This page intentionally left blank, use if needed but it will not be marked.

Question 2 – Array Containers [10 marks = 3 + 2 + 5]

Consider the partial implementation of an array-based stack given below:

```
1  class Stack:
2
3      def __init__():
4          self.array = 100 * [None]
5          self.top = -1
6
7      def __len__(self):
8          return self.top + 1
9
10     def is_empty(self):
11         return self.top == -1
12
13     def is_full(self):
14         return self.top == len(self.array)-1
15
16     def push(self, item):
17         if self.is_full():
18             self.resize()
19         self.array[self.top+1] = item
20         self.top += 1
```

(a) Implement the method `resize(self)` within the Stack class, which is called from the `push` method given above (line 18). This method should double the size of the underlying array, each time such array runs out of capacity. **Important:** `self.array` should be used like an array (as it has been done in the above code), not like a Python list (that is, do not use any additional operations like iterators, slices, etc).

```
1    def resize(self):
2          new_array = (len(self.array)*2)* [None]
3          for i in range(len(self.array)):
4              new_array[i] = self.array[i]
5          self.array = new_array
```

- 1 mark for creating new array twice the size
- 1 mark for correctly copying all elements
- 1 mark for correctly updating reference to `self.array`

3

This page intentionally left blank, use if needed but it will not be marked.

(b) What are the best-case and worst-case time complexity of a complete `push` implementation? Explain your answer.
Best case no resizing, is O(1), worst case when resizing O(n)

- 1 mark, best case with explanation
- 1 mark, worst case with explanation

(c) The following implements a mystery container based on two stacks.

```
1  class MysteryContainer:
2
3      def __init__(self):
4          self.stack_a = Stack()
5          self.stack_b = Stack()
6
7      def __len__(self):
8          return len(self.stack_a) + len(self.stack_b)
9
10     def shift(self):
11         if self.stack_b.is_empty():
12             while not self.stack_a.is_empty():
13                 self.stack_b.push(self.stack_a.pop())
14
15     def mystery_one(self, item):
16         self.stack_a.push(item)
17
18     def mystery_two(self):
19         if len(self) == 0:
20             raise ValueError('Cant do this when I am empty')
21         self.shift()
22         return self.stack_b.pop()
```

What are the two mystery methods doing? What kind of container Data Type, of the ones we have seen in class, is our `MysteryContainer` class trying to emulate? Explain your answer by means of an example.

This emulates a queue, where `mistery_one` is equivalent to appending to the Queue, and `mistery_two` is equivalent to serving from the Queue.

- 3 marks for explanation specifying method equivalence
- 2 marks for expository example or evidence of tracing

7

This page intentionally left blank, use if needed but it will not be marked.

Question 3 – Binary Trees [10 marks = 3 + 5 + 2]

Consider the partial implementation of a binary tree given below, which uses the tree node class:

```
1  class TreeNode:
2
3      def __init__(self, item=None, left=None, right=None):
4          self.item = item
5          self.left = left
6          self.right = right
7
8      def __str__(self):
9          return str(self.item)
10
11
12 class BinaryTree:
13
14     def __init__(self):
15         self.root = None
16
17     def is_empty(self):
18         return self.root is None
19
20     def height(self):
21         return self.height_aux(self.root)
22
23     def height_aux(self, current):
24         if current is None:
25             return 0
26         else:
27             max_val = max(self.height_aux(current.left),
28                           self.height_aux(current.right))
29             return max_val + 1
```

(a) Implement the method `__len__(self)` within the `BinaryTree` class, which determines the number of nodes in the binary tree.

```
1      def __len__(self):
2          return self._aux_len(self.root)
3
4      def _aux_len(self, current):
5          if current is None:
6              return 0
7          else:
8              return 1 + self._aux_len(current.left)
9                       + self._aux_len(current.right)
```

- 1 mark for correct initialisation on the root node, call to auxiliary
- 1 mark for correct base case
- 1 mark for correct logic on recursive case

3

This page intentionally left blank, use if needed but it will not be marked.

(b) Implement the method `is_balanced(self)` within the `BinaryTree` class, which returns `False` if the binary tree is not balanced, and `True` otherwise. A binary tree is balanced if the difference between the height of the left subtree and the right subtree is not larger than 1. An empty tree is assumed to be balanced.

```
1     def is_balanced(self):
2         return self.balanced_aux(self.root)
3
4      def balanced_aux(self, current):
5         if current is None:
6             return True
7         else:
8             dif = self.height_aux(current.left) -
9                   self.height_aux(current.right)
10            if abs(dif) > 1:
11                return False
12            else:
13                return self.balanced_aux(current.left) and
14                       self.balanced_aux(current.right)
```

- 1 mark for correct initialisation on the root node, call to auxiliary
- 1 mark for correct base case
- 3 mark for correct logic on recursive case

(c) What is the worst-case time complexity of printing a binary tree in pre-order? Explain your answer.
O(n), since each node is visited exactly once
2 marks = 1 mark on linear + 1 mark explanation

7

This page intentionally left blank, use if needed but it will not be marked.

Question 4 – MIPS [7 marks = 4 + 3]

Consider the the following MIPS program:

```
1  .data
2  a: .word 1071
3  c: .word 462
4  temp: .word 0
5
6  .text
7  loop:
8    lw $t0, c
9    beq $t0, $0, endloop
10   sw $t0, temp
11   lw $t1, a
12   lw $t2, c
13   div $t1, $t2
14   mfhi $t3
15   sw $t3, c
16   sw $t0, a
17   j loop
18
19 endloop:
20   lw $a0, a
21   addi $v0, $0, 1
22   syscall
```

(a) Translate the above MIPS code into python.

```
1  a = 1071
2  c = 462
3  temp = 0
4  while c != 0:
5      temp = c
6      c = a % c
7      a = temp
8  print(a)
```

- 0.5 mark for global variables correctly initialised
- 2 marks for loop logic/condition
- 1 mark logic/variables inside loop
- 0.5 marks for printing output

(b) Explain the purpose of the **fp** and **sp** registers.

sp always point to top of the stack frame, used to allocate/deallocate memory on stack. **fp** always points to saved copy of the frame pointer, which is also a copy of the **sp** , facilitating access to local variables

- 1.5 **sp** correct explanation with purpose even is just saying points to top of stack.
- 1.5 **fp** correct explanation with a purpose

7

This page intentionally left blank, use if needed but it will not be marked.

Question 5 – Iterators [9 marks = 6 + 3]

Consider an *iterable* `List` abstract data type, where the list iterator has the usual `next()` method to return the current item in the list (if any) and move the iterator to the next, and also method `has_next`, which returns `True` if the iterator has not yet reached the end of the list, and `False` otherwise.

Consider the following method:

```
1  def mystery(iterable_list):
2      it1 = iterable_list.iter()
3      it2 = iterable_list.iter()
4      while it2.has_next():
5          item1 = it1.next()
6          item2 = it2.next()
7          if it2.has_next():
8              it2.next()
9              /* HERE */
10     return item1 + item2
```

(a) Given an iterable list `my_list` with elements 9, 6, 7, 5, 3, 4, 2, 1, and 8, where 9 is the item at the head of the list, show the value of `item1` and `item2` every time the execution reaches the program point marked by `/** HERE **/` during the call to `mystery(my_list)`.

item1	item2	Pointing to.. (**not needed** for the answer, added to clarify)
9	9	At this point `it1` is pointing to node 6 and `it2` to node 7
6	7	At this point `it1` is pointing to node 7 and `it2` to node 3
7	3	At this point `it1` is pointing to node 5 and `it2` to node 2
5	2	At this point `it1` is pointing to node 3 and `it2` to node 8
3	8	At this point `it1` is pointing to node 4 and `it2` to None

The aim is to realise that both iterators traverse the list, but iterator it2 goes faster (in every iteration it advances one element more). Thus, when it2 reaches the end it1 is still in the middle of the list.

6 = 1 + 10*(0.5)

- 1 marks evidence of understanding iteration protocol
- 0.5 for each correct value of each iterator, applying consequential marks on any mistake

6

This page intentionally left blank, use if needed but it will not be marked.

(b) What are the best-case and worst-case time complexity for the method? Explain.

Best case is the same as worst case, since the number of steps performed by the method is the same, regardless of the characteristics of the list.

The Big O complexity is O(N), where N is the length of the list. This can be explained in different ways. One is that every element of the list is traversed by iterator it2 (so N steps), and the number of operations per element is constant. Another is that the number of iterations of the loop is N//2, the number of operations in each iteration is constant (well, accurately, it has a constant upper bound), and the complexity of these operations is O(1).

- 1 mark identifying best = worst.
- 2 marks for stating and correctly giving one explanation of linearity

3

This page intentionally left blank, use if needed but it will not be marked.

Question 6 – Namespaces [9 marks = 2 + 2 + 2 + 3]

Consider the following class, which is defined in file (also called module) `point.py`:

```
1  class Point:
2      def __init__(self,x, y):
3          self.xCoord = x
4          self.yCoord = y
5
6      def shift(self, x, y):
7          self.xCoord += x
8          self.yCoord += y
```

(a) Do identifiers `__init__` and `shift` belong to the same namespace? Explain.

Yes they do. They belong to the namespace of class `Point`, as they are bound to their method definition within that class.
2 marks for correct answer and explanation

(b) Do the `self` identifiers appearing in lines 2 and 6 belong to the same namespace? Explain.

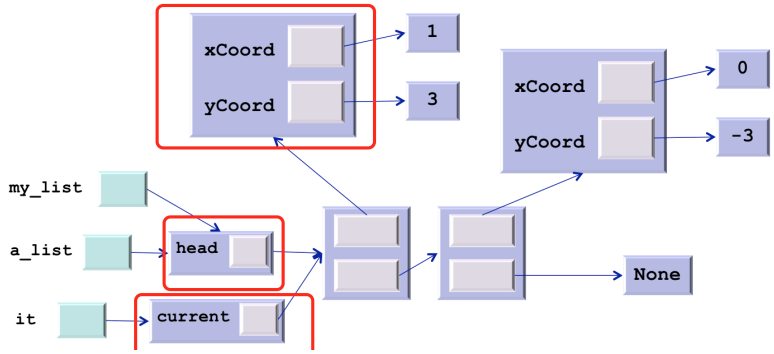
No, each identifier belongs to the namespace of the method they are defined in (thus, the self defined in line 2 belongs to the namespace of `__init__`, while the one defined in line 6 belongs to the namespace of `shift`. This is the same as arguments `x` and `y`.
2 marks for correct answer and explanation

(c) Where is the `x` identifier appearing in line 7 bound? Explain. It is bound in line 6, when the method is called with particular values for all its arguments.
2 marks for correct answer and explanation

6

This page intentionally left blank, use if needed but it will not be marked.

(d) Consider the following diagram:



which corresponds to the memory once the execution of call to `diagram(my_list)` with an iterable, linked list `my_list` that contains two point elements with coordinates (1,3) and (0,-3), has reached point `/*HERE*/` in the following code.

```
1  def diagram(a_list):
2      it = a_list.iter()
3      /*HERE*/
4      ...
```

Circle the objects corresponding to a list, to a point and to an iterator. Explain your choice.

There is one list object (red), two point objects (blue) and one iterator object (black). The rest are four integer objects (1,3,0 and -3), one None object, two node objects, and three variables (`my_list`, `a_list`, `it`), that point to the list and iterator objects.
3 marks = 1 mark for each correct circle (list, point, iterator), circle should enclose instance variables of object in question.

3

This page intentionally left blank, use if needed but it will not be marked.

Question 7 – Linked Lists [10 marks = 7 + 3]

Consider the following linked `List` class, which you are in the process of defining:

```
1  class Node:
2
3      def __init__(self, item=None, link=None):
4          self.item = item
5          self.link = link
6
7  class List:
8
9      def __init__(self):
10         self.head = None
11
12     def is_empty(self):
13         return self.head is None
```

(a) Define a method `periodic_delete(self,n)` **within the** `List` **class** that deletes the first n nodes of the list, leaves the next n, erases the next n, and so on. For example, given `a_list` with elements 1,2,3,4,5,6,7,8,and 9 where 1 is the item at the head node, the call to `a_list.periodic_delete(2)` will leave `a_list` as 3,4,7,8.

One possible way of doing is as follows:

```
1      def periodic_delete(self,n):
2          # Move the head to the nth node
3          self.head = self.find(self.head,n)
4          current = self.head
5          # keeps n nodes and deletes n in every iteration
6          while current is not None:
7              # returns the last node to keep
8              last_keep = self.find(current,n-1)
9              if last_keep is not None:
10                 # jumps n, returns first node to keep
11                 current = self.find(last_keep.link,n)
12                 # bypases the n nodes before current
13                 last_keep.link = current
14             else:
15                 current = None
16
17     # returns the nth node if any (might be None)
18     def find(self, current,n):
19         while current is not None and n > 0:
20             current = current.link
21             n -= 1
22         return current
```

- 1 mark head ends up in the right position
- 1 mark for readable code
- 2 marks, correct list iteration
- 3 marks, logic of bypassing nodes

This page intentionally left blank, use if needed but it will not be marked.

(b) How would the linked implementation compare in terms of efficiency and in terms of Big O complexity, to an implementation for an array-based lists? Explain.

A linked list solution needs to traverse every element in the list (as done above), and modify the link of one node every $2 * n$ nodes (the one that bypases the subsequent n nodes). This will result in O(N) big O complexity, where N is the lenght of the list, since we traverse N nodes and perform constant time operations for each node (accessing the link and subtracting to the counter plus sometimes modifying the link). Regarding efficiency, it is quite efficient (as it only needs to modify one link every $2 * n$.

The array based implementation is a bit more complex, as we need to shuffle every element that we need to keep. This still means O(N) big complexity (we still end up depending on N, as for every n elements deleted, we need to shuffle another n). But it will be slightly less efficient, as copying is required for every element we keep, and also we need to keep track of at least two counters (one for the first n elements that need to be shuffled, and another for the n elements that need to be overwritten).

No need to be this detailed. Just to say that they are same big O (with a clear reason) and similar efficiency.

3 marks: 1 mark stating big O equivalence, 2 marks for reasoning. Some students are comparing structures without reference to part A, we have decided to award full marks and deduct accordingly since question could have been clearer

3

This page intentionally left blank, use if needed but it will not be marked.

Question 8 – Recursive sorts [8 marks = 4 + 4]

This question is about recursive sorting algorithms. One day, while you are demonstrating your sorting algorithm on an array of integers, your friend decides to trick you and changes exactly one of the numbers to a value higher than any other value in the array. Use an array of size 6 to explain how would this affect the result of your sorting (that is, the order of the resulting array) if the change happened:

(a) To some element during the partition phase in mergesort. For example, to the first element in the list after the first call to partition has finished.

There would be no effect, as in, the list would still be correctly sorted but with one element less and a new highest element. This is because mergesort does the actual sorting during the merge, not during the partition. In fact, during the partition the elements are not accessed at all, so their value is irrelevant.
1 mark stating no effect, 3 marks for reasoning.

(b) To the pivot element, right after the pivot was chosen and you were about to perform a partition in quicksort. For example, the pivot was chosen as the last element, and that was the element changed before calling partition at some point during quicksort.

Again, the list would still be correctly sorted (with one element less and one element biggest). This is because the pivot is modified BEFORE calling partition and, thus, the list would be correctly split according to whichever value the pivot has. However, in this case the efficiency of that particular step might be altered. For example, if the original pivot was the median, the original partition would have split the list in two lists of roughly the same length. In the new one, one list would be empty and the other contain all elements except the pivot (since the new pivot is greater than any other). Still, this is a single step which might or might not affect the efficiency rest of the algorithm (depending on how the pivot is chosen).
1 mark stating no effect, 3 marks for reasoning.

Comment: again, no need to be that detailed since we only asked for whether they are correctly sorted

8

This page intentionally left blank, use if needed but it will not be marked.

Question 9 – Heaps [10 marks = 3 + 3 + 4]

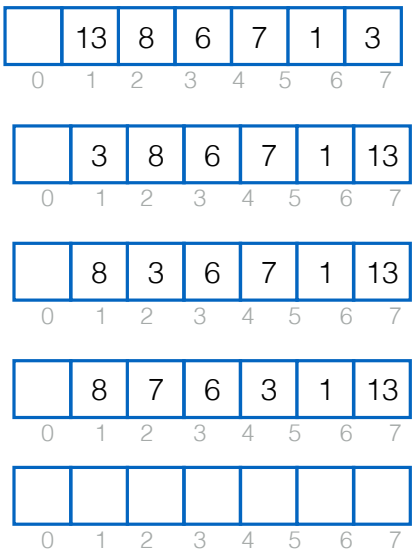
(a) What is the worst-case time complexity of Heap Sort? Explain your answer. Heapsort proceeds by adding the elements of the list into a heap, and then removing them. Adding implies $\log(1) + \log(2) + \log(3) \cdots + \log(n)$, and removing implies $\log(n) + \log(n-1) + \log(n-2) \cdots + \log(1)$. This is bounded by $O(n \log n)$.

- 1 mark for evidence of understanding heapsort.
- 2 marks for correct complexity with correct reasoning.

(b) The elements in the list `[5, 8, 16, 1, 3, 7]` are added into a min-heap. If the min-heap is implemented using an array, draw the state of the array after all the elements have been added.

`[None, 1, 3, 7, 8, 5, 16]`.
3 marks for **any order reflecting a min-heap**, 1 mark if None forgotten.

(c) A max-heap is implemented using an array, which contains `[None, 13, 8, 6, 7, 1, 3]`. Complete the diagram below showing how the array changes as the method `get_max` is executed. The diagram should depict what happens every time an element swaps positions in the array, and should finish with the state of the array once the maximum element has been retrieved.



4 marks for correct solution, not skipping any swaps. -1 mark for each mistake. **Figure has changed from v0.**

10

This page intentionally left blank, use if needed but it will not be marked.

Question 10 – Hash Tables [12 marks = 4 + 2 + 2 + 4]

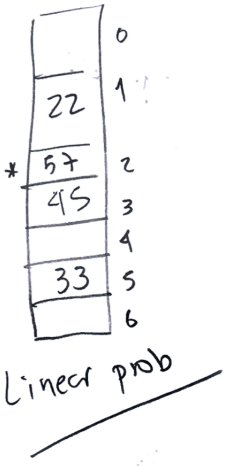
This question is about hash tables and dictionaries.

(a) List at least 3 data types covered in the lectures than can be used to implement a Dictionary ADT. In each case explain how insertions and deletions would perform in terms of complexity.

- HashTable, constant time insertion, worst case is $O(n)$ but does not arise often with a good hashing function; deletes are $O(n)$.
- Binary Search Tree, deletes and inserts are worst case $O(n)$, but may perform $O(\log n)$
- Array Sorted List, insert and delete $O(n)$

1.33 for each reasonable structure, with correct complexity for delete and insert.

(b) A hash table is supported by an array of size 7. Draw the final state of the array after inserting the numbers, 45, 22, 57 and 33. Collisions are resolved using linear probing with hash function $h_1(k) = k\%7$.

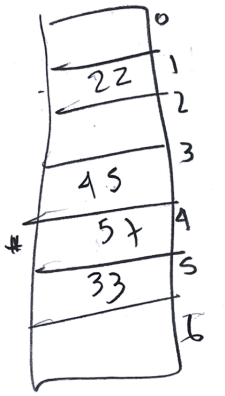


- 2 marks correct solution, 1 mark minor mistakes

8

This page intentionally left blank, use if needed but it will not be marked.

(c) Repeat the exercise above when collisions are resolved using double hashing. The first hash function is the same, i.e., $h_1(k) = k\%7$; the secondary hash function is $h_2(k) = 5 - (k\%5)$



- 2 marks correct solution, 1 mark minor mistakes

(d) Consider the following code, where the class `HashTable` implements a hash table with a universal hash function.

```
1 list = [1, 2, 3, 4, 5, 1, 2]
2 table = HashTable(capacity=365)
3 for item in list:
4     table.insert(key=item, value=str(item))
```

After the above is executed, a user attempts `table.insert(6, '6')`. What is the chance of having a collision arising from this insertion? Explain your answer.

There are 5 unique (out of 7) keys, thus the chance is therefore P = 5/365.

2 marks for 7/356, i.e., ignoring unique elements **or** 4 marks correct solution accounting for duplicates

END OF EXAM.

4