Question 1: [25 marks]

This question is about MIPS programming. Consider the following python code:

```
def min(x,y):
    result = y
    if x < y:
       result = x
    return result
```

which has been faithfully translated into MIPS (with a few mistakes) as follows:

```
1       .data
2
3       .text
4
5   min:
6       addi $sp, $sp, -8
7       sw $ra, 4($sp)
8       sw $fp, 0($sp)
9
10      addi $fp, $sp, 0
11
12      addi $sp, $sp, -8
13
14      lw $t0, 12($fp)
15      sw $t0, -4($fp)
16
17      lw $t0, 8($fp)
18      lw $t1, 12($fp)
19      slt $t0, $t1, $t0
20      beq $t0, $0, one
21
22      lw $t0, 8($fp)
23      sw $t0, -4($fp)
24
25      j two
26
27  one:
28      lw $v0, -4($fp)
29  two:
30      addi $sp, $sp, 8
31
32      lw $fp, 0($sp)
33      lw $ra, 4($sp)
34      addi $sp, $sp, 8
35
```

For the above code to be correct, one line is missing, two lines should disappear, and several lines have a correct instruction code (lw, slt, sw, etc) but incorrect arguments (registers, lables, addresses or immediate values).

In the next page, and for every line you think has a mistake, provide (a) the number of the line, (b) the correct code for the line, and (c) explain why this is correct (no explanation, no marks).

Important: the above code provides enough information to determine where the local variables and arguments are located in the stack without the need for a memory diagram.

25

Lines 5 to 10 are correct. They introduce the label for the function (line 5), make space in the stack for the \$ra and \$fp (line 6), save the \$ra (line 7) and \$fp (line 8) in the stack, and then copy the \$sp\$ onto the \$fp (line 10).

The first problem arises in line 12, which is making space for the local variables, but makes too much space (8 bytes, enough for two locals when there is only one). The correct instruction is addi \$sp, \$sp, −4.

Lines 14 and 15 are also correct. They load the value of argument y, and store it in local variable result, respectively.

Lines 17 and 18 are also correct. They load the value of arguments x and y, in registers\$t0 and \$t1, respectively.

The second problem arises in line 19. Which should test whether x is smaller than y and, instead, tests whether y is smaller than x. The correct instruction is slt \$t0, \$t0, \$t1.

Lines 20 to 23 are also correct. Line 20 jumps out of the "then" branch, by jumping to one if x is not smaller than y, line 22 loads x and line 23 stores it in result, completing the "then" branch.

The third problem arises in line 25, which should disappear, as there is no "else" to jump over.

Lines 27 and 28 are correct. The first one provides the label needed by the if-then to jump over the "then" part. The second one loads the result onto register \$v0 in preparation to return.

Line 29 is not needed either, as line 25 was not needd.

Line 30 should (like line 12) use 4 rather than 8.

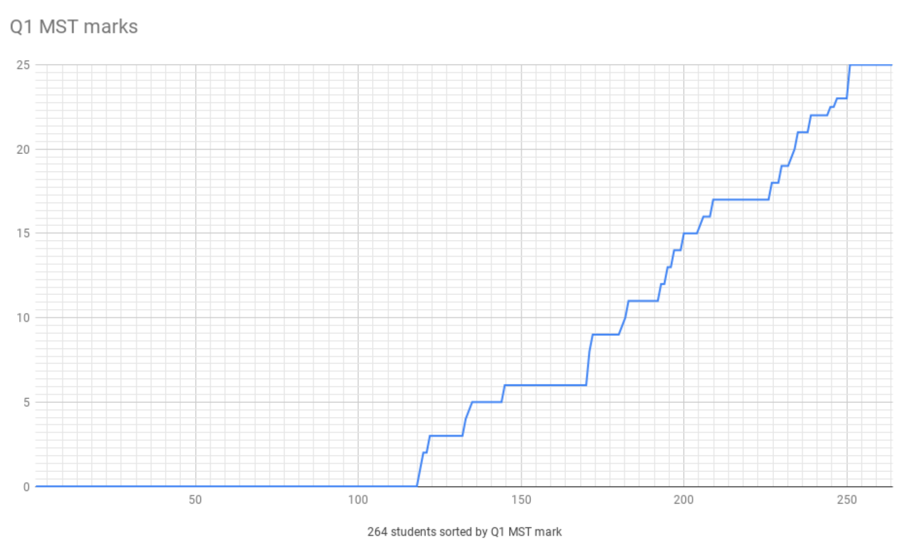
Lines 32 to 34 are also correct, the first two restore the \$fp and the \$ra, while the last one recovers the space needed by them.

The line jr \$ra is missing after line 34 to jump back to the resturn address

The most common mistakes were:

1. No explanation given, even if corrected code was provided (which meant 0 marks).
2. Use of syscall instead of jr \$ra
3. Identifying correct (rather than incorrect) code.
4. Providing an incorrect explanation for line 25.

This question was poorly answered, as seen in the following marks distribution (note that 35 students did not attend the test):

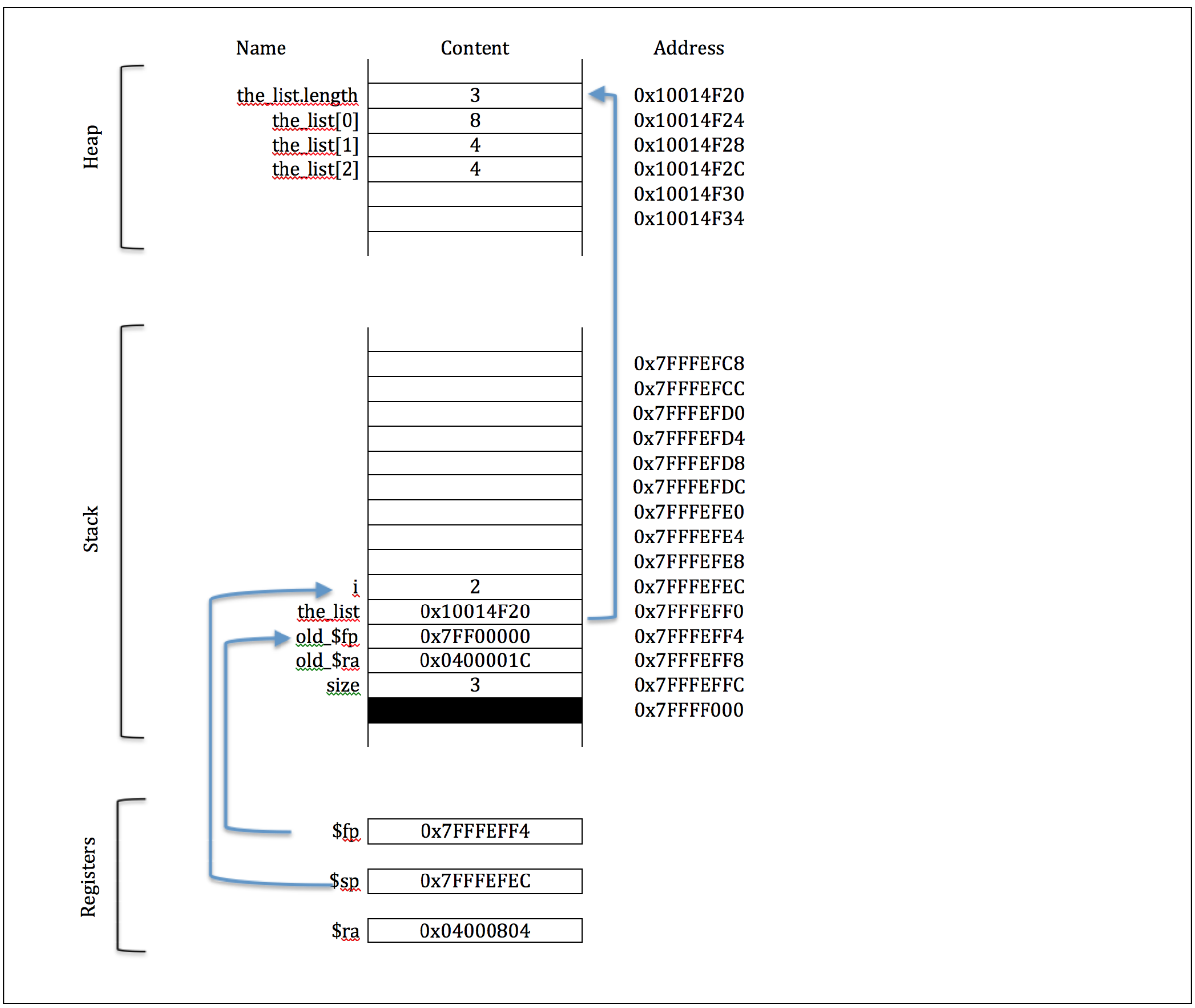


0

Question 2: [20 marks]

Consider the following python code, which calls the (corrected) min function defined in Question 1, and which you wish to translate into MIPS. Complete the memory diagram when the execution of call read_list(3) reaches the line with the comment #Here for the last time, assuming the three integers read into the list are 8, 9, and 4, in that order. Include names and contents of everything stored in the stack and heap. Assume the PC for the `jal read_list` instruction is `0x04000018` and $fp is `0x7FF00000` right before jumping to `read_list` from its caller, and the PC for the `jal min` instruction is `0x04000800` and $fp is `0x7FEF0000` right before jumping to `min` from `read_list`.

```python
def read_list(size):
    the_list = [0]*size
    for i in range(size):
        the_list[i] = int(input())
        if i != 0:
            the_list[i-1] = min(the_list[i-1],the_list[i])
       # Here
    return the_list
```



20

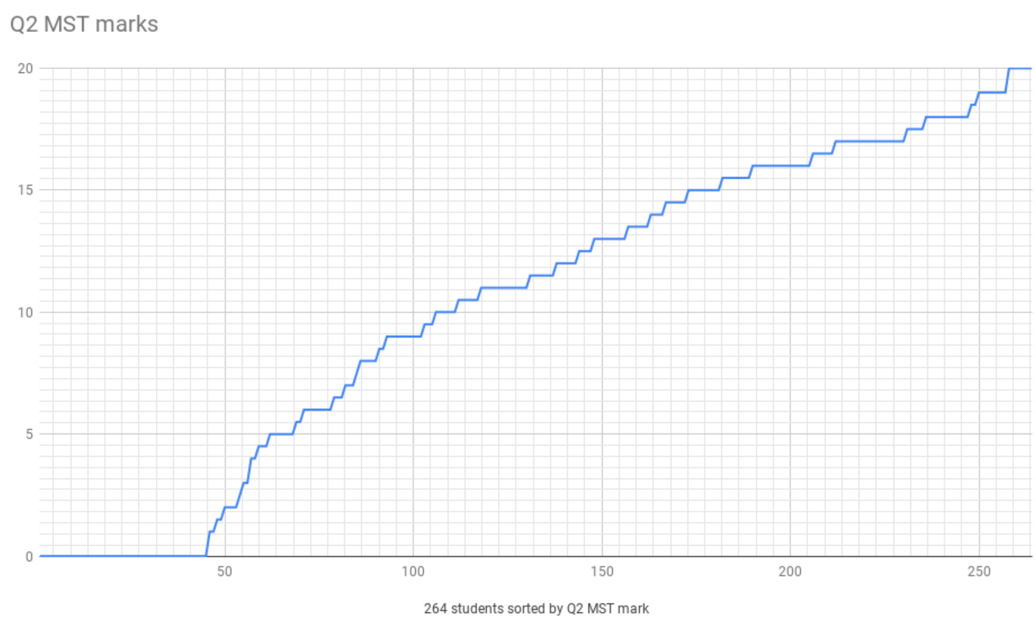
Note that the list might start one or two positions later in the heap. Also, the order between `the_list` and `i` is not important.

The most common mistakes were:

1. Heap: the_list [1] - using 9 rather than 4 as its value
2. Stack: old \$fp - using 0x7FFFF000 rather than 0x7FF00000
3. Stack: old \$ra - using a heap or stack address, or forgetting to add 4 to the PC
4. Registers: \$fp - using the `$fp` from the question, rather than pointing to the old `$fp`
5. Registers: \$sp - pointing to the location above the top of the stack
6. Registers: \$ra - using a heap or stack address or forgetting to add 4 to the PC

Note: Some students included deallocated items on the stack. Whilst technically this is correct, it wasn't required for the stack diagram, so it is not penalised but neither it was marked positively

This question was answered better, as seen in the following marks distribution:



0

Question 3: [15 marks]

Consider the following sorting algorithm implementation:

```
1  def sorting_algo(alist):
2
3      n = len(alist)
4      i=1
5      while i < n:
6          t = alist[i]
7          j = i - 1
8          while j>=0:
9              if alist[j] >= t:
10                 alist[j+1] = alist[j]
11                 j -= 1
12             else:
13                 break
14         alist[j+1] = t
15         i += 1
```

(a) (10 marks) What is the best case and worst case time complexity of the sorting algorithm implementation above? Explain each case (no explanation, no marks).

The best case is when the list is already sorted. In this case, the inner loop will break $\forall i$, and the outer loop will run n-1 times, where n is the length of alist. Since the comparison of two list elements is $O(m)$ where m is the maximum size of the elements, the best-case complexity is $O(n * m)$.

The worst case is when the list is inversely sorted (i.e., sorted in decreased order). In this case, the outer loop runs the same number of times (n-1), but the inner loop runs $i - 1$ times $\forall i \in 1..n - 1$. Thus, the worst-case complexity is $O(n^2 * m)$.

The most common mistakes for this part of the question were:

1. Forgetting to include m, the size of each element in the complexities.
2. Not recognising that the algorithm can terminate early.
3. Assuming that the break statement breaks out of both of while loops.

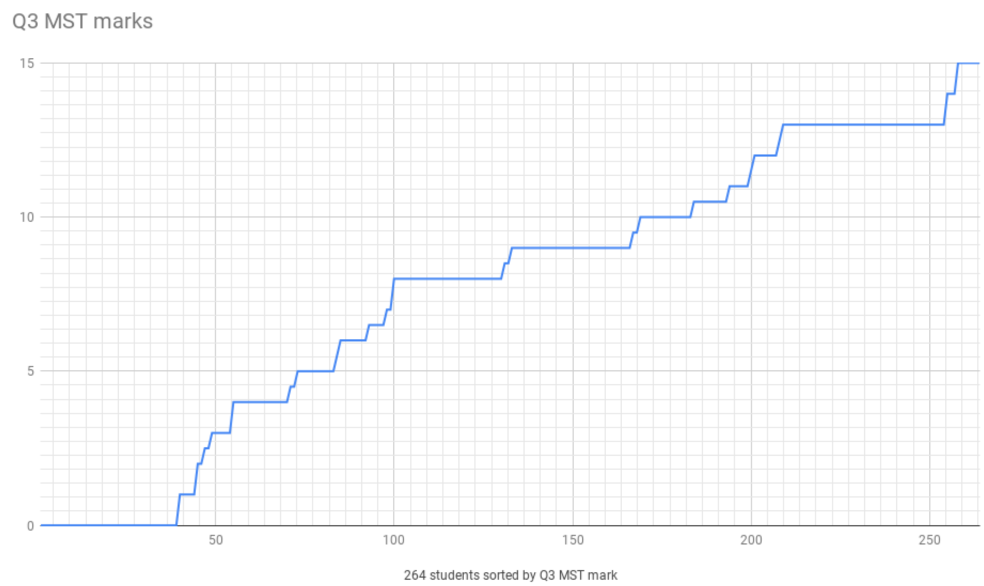
(b) (5 marks) Is the algorithm above stable? Explain (no explanation, no marks).

The >= comparison in line 9 will result in elements with the same value being swapped. Thus, the relative order of the initial elements with the same value is altered, which means the sorting algorithm is not stable.

The most common mistake for this part of the question was not to provide the mechanism in the code that causes the instability.

15

This two-part question was answered as seen in the following marks distribution:



0

Question 4: [20 marks]

Consider an Abstract Data Type, called `Set`, that can store unique objects without a particular order. Part of the `Set` class has been written and is provided to you below. Given this code, the methods and their documentation, we ask that you finish the implementation of this ADT by writing the methods `add` and `remove`, making sure you follow the documentation provided.

```python
class Set():
    def __init__(self,capacity):
        """Creates a Set with the given capacity."""
        self.the_array = [None] * capacity
        self.array_size = 0

    def size(self):
        """Returns the number of elements in the set. Time complexity O(1)."""
        return self.array_size

    def is_full(self):
        """Returns True if and only if the set is full (i.e. the capacity is met). Time complexity O(1)."""
        return self.size() == len(self.the_array)

    def is_empty(self):
        """Returns True if and only if the set is full (i.e. the capacity is met). Time complexity O(1)."""
        return self.size() == 0

    def search(self, item):
        """If the given item is in the set, return its index, otherwise returns -1. Time complexity O(n), where n is the capacity of the set."""
        for i in range(len(self.the_array)):
            if self.the_array[i] == item:
                return i
        return -1

    def add(self, item):
        """Adds this item to the set.
           Raises an Exception if the item is already in the set.
           Raises an Exception if the set is full.
           Time complexity O(n) where n is the capacity of the set."""
```


20

```python
    def remove(self, item):
        """Removes the given item from the set.
           Raises an Exception if the item is not in the set.
           Time complexity O(n) where n is the capacity of the set."""
```


0

There are many solutions possible. An efficient solution consists in storing items contiguously, and when removing an item, “plugging the hole” with the last item in the list:

```python
    def add(self, item):
        """Adds this item to the set.
           Raises an Exception if the item is already in the set.
           Raises an Exception if the set is full.
           Time complexity O(n) where n is the capacity of the set."""
        if self.search(item) != -1:
            raise Exception("The item is already in the set.")
        if self.is_full():
            raise Exception("The set is full.")
        #we look for the next available slot
        self.the_array[self.size()] = item
        self.array_size += 1

    def remove(self, item):
        """Removes the given item from the set.
           Raises an Exception if the item is not in the set.
           Time complexity O(n) where n is the capacity of the set."""
        i = self.search(item)
        if i == -1:
            raise Exception("The item is not in the set")
        self.the_array[i] = self.the_array[self.size()-1]
        self.the_array[self.size()-1] = None #optional
        self.array_size -= 1
```

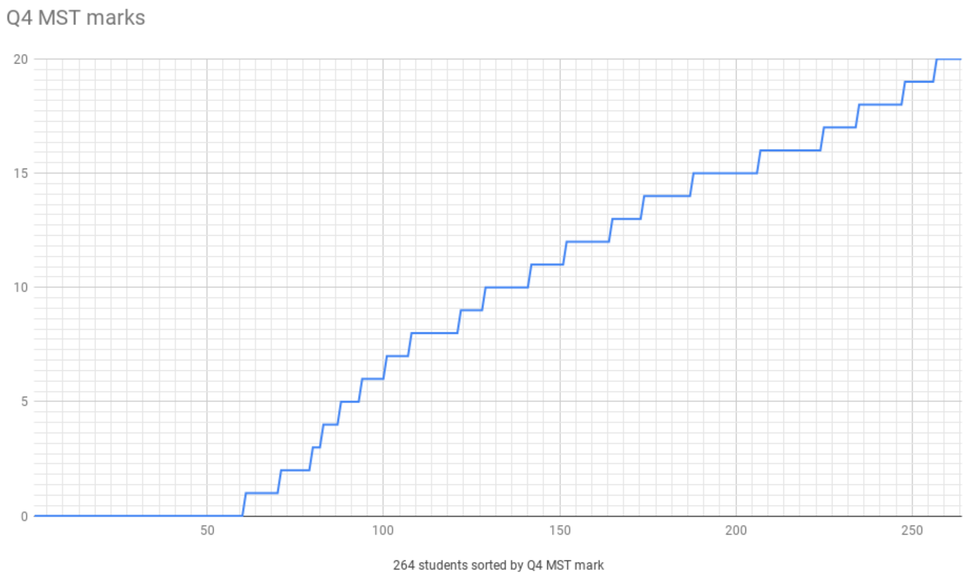
A less efficient solution that relies on storing items non-contiguously is:

```python
     def add(self, item):
        """Adds this item to the set.
           Raises an Exception if the item is already in the set.
           Raises an Exception if the set is full.
           Time complexity O(n) where n is the capacity of the set."""
        if self.search(item) != -1:
            raise Exception("The item is already in the set.")
        if self.is_full():
            raise Exception("The set is full.")
        i = self.search(None)#we look for the next available slot
        assert(i != -1)
        self.the_array[i] = item
        self.array_size += 1

    def remove(self, item):
        """Removes the given item from the set.
           Raises an Exception if the item is not in the set.
           Time complexity O(n) where n is the capacity of the set."""
        i = self.search(item)
        if i == -1:
            raise Exception("The item is not in the set")
        self.the_array[i] = None
        self.array_size -= 1
```

It stores items in a non-contiguous manner, relying on `None` to mark free slots, but requires the search to traverse the entire array (in the first solution the search could be modified to iterate over the number of elements in the set, rather than over the capacity of the array in which the elements are stored).

This question was answered as seen in the following marks distribution:



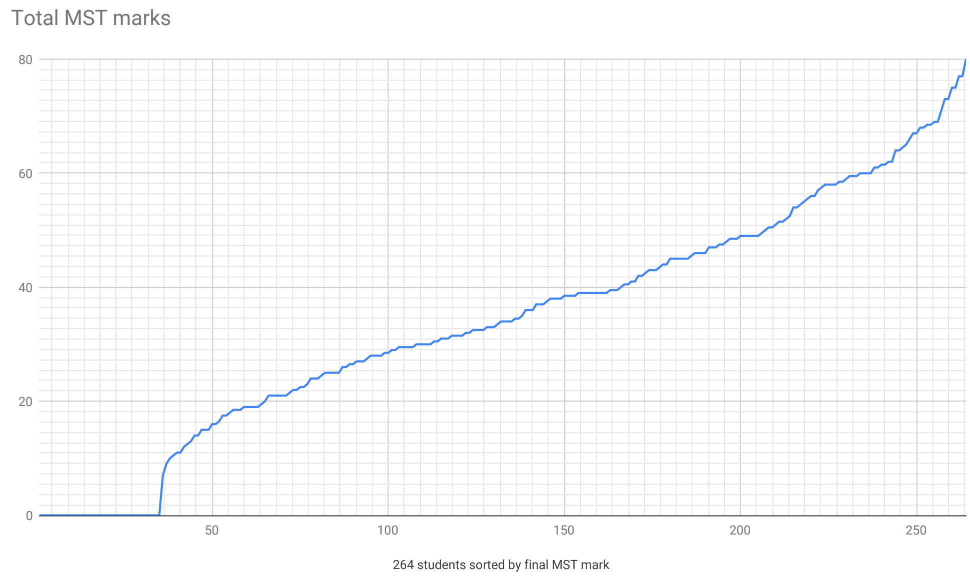
Common mistakes (and reference codes).
QUESTION 4 AS A WHOLE
(Q1) 'add' and 'remove' do not work together
(Q2) student chooses an Exception type that is not the one given, namely 'Exception', or does not properly write exception statements. We must follow the ADT!
(Q3) the error messages given in the exception are not all meaningful
(Q4) student does not use the method 'search' when they could
(Q5) student does (other) things that are unnecessary
(Q6) class attributes not used with 'self.'

ADD
(A1) no attempt to check if the item is in the set (assertions count as attempts)
(A2) student does not raise that exception exactly when and how it should be
(A3) student does not attempt to check if the set is full (assertions count as attempts)
(A4) student does not raise that exception exactly when and how it should be
(A5) student does not check those two exceptions in that exact order
(A6) the complexity is not O(capacity) or the method is not finished
(A7) the method does not add an element to the_array when and where it should
(A8) the instance variable array_size is not incremented iff the item is added

REMOVE
(R1) student does not attempt to check if the item is in the set (assertions count as attempts)
(R2) student does not raise that exception exactly when and how it should be
(R3) the complexity is not O(capacity) or the method is not finished
(R4) the method does not remove an element to the_array when it should
(R5) the instance variable array_size is not decremented iff the item is removed

The distribution of total marks (that is, added over the four questions) is as follows:

0



Recall that 35 students did not attend the test at Clayton and, thus, got a 0 overall. From the 229 students who did attend, 130 (56.5%) got N, 30 (13%) P, 24 (10.5%) C, 24 (10.5%) D and 21 (9%)HD. Given the similarities between the Mid Semester Test and the final exam, and the fact that getting 50% in the exam is a hurdle for the unit, we recommend all students who got less than 60% to significantly increase their study effort for this unit.

End of Mid Semester Test

0

This page intentionally left blank, use if needed but it will not be marked *unless* you explicitly ask us to do so

0