Question 1 [10 marks]

This question is about MIPS programming and function calls. Translate the following Python code faithfully into MIPS assembly language. Make sure you follow the MIPS function calling and memory usage conventions as discussed in the lectures. Use only instructions in the MIPS reference sheet.

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 == 1:` (0.5)	`lw $t0, 8($fp)  # arg n`
	`addi $t1, $0, 1`
	(1) { `bne $t0, $t1, else  # goto else if n != 1`
	1.5 marks

`return 1`	`addi $v0, $0, 1  # return 1`
	0.5 marks
	`lw $fp, 0($sp)`
	`lw $ra, 4($sp)`
	`addi $sp, $sp, 8`
	`jr $ra`
	1 mark

`else:`
`return n * func(n//2)`
	`else:`
	(1) { `lw $t0, 8($fp)`
	`sra $t0, $t0, 1` `# n // 2`
	(1) { `addi $sp, $sp, -4` `# push arg n // 2`
	`sw $t0, 0($sp)`
	(0.5) `jal func` `# func (n // 2)`
	2.5 marks
	(0.5) `addi $sp, $sp, 4` `# deallocate arg`
	`lw $t0, 8($fp)`
	(1) { `mult $t0, $v0` `# n * func(n // 2)`
	`mflo $v0`
	1.5 marks
	`lw $fp, 0($sp)`
	`lw $ra, 4($sp)`
	`addi $sp, $sp, 8`
	`jr $ra`
	1 mark

-0.5 syntax (without comma).

Question 2 [7 marks]

This question is about Iterators. Using Python define a `NegativeIterator` iterator class. This Iterator should work with a standard Python list. An instance of this class iterates through all the negative elements of the list without modifying the list.
For example:

```
>>> itr = NegativeIterator([3,-8,-6,0,-11])
>>> next(itr)
-8
>>> next(itr)
-6
>>> next(itr)
-11
>>> next(itr)
Stop Iteration
```

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

```
class NegativeIterator:

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

    def __iter__(self):
        return self

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

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

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

        self.current += 1
        return item
```

1 mark for a correct `__init__`
1 mark for a correct `__iter__`
For `__next__` :
1 mark for checking whether you at the end of the list
2 marks for finding the next negative number
1 mark for dealing with there are no more negative numbers
0.5 marks for updating the pointer to next item
0.5 marks for returning the item

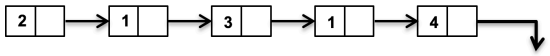
7

Question 3 [8 marks]

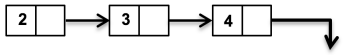
Consider the two classes `Node` and `List` as seen in the lectures, which define a **List ADT** implemented using a **linked structure**:

```
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
```

Define the method `delete_item(self, item)`, which deletes all the items in the list that have the value `item`. For example, assume `alist` is a List and `alist.head` points to the first node in the following structure:



After calling `alist.delete_item(1)`, `alist` should have the following structure with all the items of value `1` removed:




```
def delete_item(self, item):
    if not self.is_empty():
        while self.head.item == item:
            self.head = self.head.next

    current = self.head.next
    previous = self.head

    while current != None:
        if current.item == item:
            previous.next = current.next
            current = current.next
        else:
            previous = current
            current = current.next
```

1 mark for checking whether the list is empty
2 marks for removing all the matching items at the beginning of the list
1 mark for correct initialisation of current and previous
2 marks for deleting the subsequent matching items
1 mark for updating previous and current to the next item
1 mark for checking through all the items in the list

Question 4 [6 marks = 5 + 1]

This questions is about Heaps.

(a) Suppose a `min-heap` is represented using an array. Write a Python function `def is_valid_heap(array)`, which given an `array` returns `True` if the `array` represents a valid `min-heap`, and returns `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
```

- comparing parents and children
  - 1 mark for verifying the left child with the parent
  - 1 mark for verifying the right child with the parent
  - if students instead compare sequential items (ex list in order)award 0.5 marks
- Looping over the array
  - 2 marks for looping (or recursing) until no more parents/children
  - 1 mark for looping over the whole array but with demonstration of understanding of parent/child relation elsewhere
  - 0 marks for looping over the whole array without this understanding
- 0.5 marks for returning True for a valid min-heap
- 0.5 marks for moving to the next parent item (incrementing k)

(b) Using Big-O notation, provide and 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 nodes need to be checked whether they satisfy the min-heap order condition.

1 mark for correct explanation and complexity, matching their algorithm - No half marks

6

Question 5 [10 marks = 2 + 3 + 3 + 2]

This question is about sorting algorithms.

(a) What is a stable sorting algorithm? A sorting algorithm is said to be stable if two elements with equal values appear in the same order in sorted output as they appear in the unsorted input.

or

A sorting algorithm is stable if it maintains the relative order among elements.

2 marks for correct explanation

(b) Is selection sort stable? Explain your answer **and** provide an example to make your case. At each iteration, selection sort find the minimum/ maximum element in the unsorted portion of the list and swap it into the sorted portion of the list. The swaps could change the relative order of elements in the list. Consider the example below:

- b_1, b_2, a, c with $a < b < c$ before sorting
- a, b_2, b_1, c after sorting

2 marks for correct explanation

1 mark for example as above or concrete values

(c) Using Python, write a function `def insertion_sort(a_list)`, which takes as input a list of numbers and sorts this list into increasing order using insertion sort.

```
def insertion_sort(the_list):
    n = len(the_list)
    for k in range(1, n):
        temp = the_list[k]
        i = k - 1
        while i >= 0 and the_list[i] > temp:
            the_list[i + 1] = the_list[i]
            the_list[i] = temp
            i -= 1
```

1 mark for correct syntax

1 mark for correctness of sorting

1 mark for implementing idea of insertion sort, moving elements into right position

(d) What are best and worst-case time complexity for Insertion Sort in Big-O notation. When do they occur? Explain your answer (no explanation means no marks). Best case of O(N) where N is the size of the list when the list is already in order. When the list is in order, only one comparison is required for each element with no swaps needed to move that element to its right position.

Worst case of O(N^2) where N is still the size of the list when the list is in reversed order. For each element, up to N comparisons and swaps are required to move the to its correct position.

0.5 mark correct best case value, explained

0.5 best case when **sorted**

0.5 mark correct worst case value, explained

0.5 worst case when **in decreasing order**

Question 6 [8 marks = 2 + 2 + 2 + 2]

This question is about Time complexity. For each of the given Python functions, state and explain the complexity in Big-O notation. If appropriate discuss best and worst cases. No explanation means no marks.

(a)
```
def func_a(the_list):
    total = 0
    for item in the_list:
        for i in range(5):
            total = total + i * item
```

Best case and worst case is O(N) where N is the size of the_list. The outer loops go through all elements in the list. The inner loop executes for a constant of 5 times per element in the list. It is not possible to exit the loops earlier, thus the best and worst case is the same.

1 mark O(N), 1 mark stating best = worst

(b)
```
def func_b(a, b):
    while a > b:
        print(str(a-b))
        a = a - 1
        b = b + 1
```

The best case is O(1) when $b <= a$, entering the loop at most one time. The worst case is O(N) where $N = a - b$. It will loop through N times where the values of a and b are moved closer together at each iteration.

1 mark correct best case, 1 mark correct worst case

(c)
```
def func_c(the_list):
    total = 0
    for i in range(len(the_list)):
        total = total + the_list[i]
        if i < 0:
            break
```

Best case and worst case is O(N) where N is the length of the list. The loop will go through N times from 0 to $n - 1$ and there is no way to exit the loop earlier.

1 mark correct complexity, 1 mark stating best = worst

(d)
```
def func_d(n):
    while n > 0:
        print(n)
        n = n // 2
```

Best and worst case is O($\log N$). At each iteration, the value of N is halved resulting in the loop iterating $\log N$ times. Best and worst case is the same because there is no way to exit the loop earlier.

1 mark correct complexity, 1 mark stating best = worst

Question 7 [10 marks]

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

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

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

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

Using Python, write a function `def magnitude(a_queue)`. This function takes as input a queue of numbers sorted in increasing order. The function then returns a new queue with the numbers sorted according to their absolute value, in increasing order. For example: given a queue [-322, -180, -5, 3, 7, 10, 180, 360]; the function would return a queue with values [3, -5, 7, 10, -180, 180, -322, 360]. You should only interact with the Queue and the Stack through the operations given above.

- 1 - includes all elements of the original Queue for the final queue
- 2 - reverses the negatives by some means (1 if reverses all elements)
- 1 - correct iteration over stacks and queues
- 1 - only uses ADT operations on stack/queue
- 1 - no additional structures
- 2 - compares negatives and positives together by absolute value
  - 2 if negs vs pos by absolute
  - 1 if compare negative and positive by non-absolute value
  - 1 if compare all values by absolute
  - 0 otherwise
- 1 maintains increasing order (absolute) in queue
- 1 - returns correct queue at end

See potential model answer on next page.

```
def magnitude(a_queue):
    # temporary storage
    temp_stack = Stack()
    temp_queue = Queue()
    # separate the negative and positive items
    # serve and pop produce the right smallest magnitude
    while not a_queue.is_empty():
        item = a_queue.serve()
        if item < 0:
            temp_stack.push(item)
        else:
            temp_queue.append(item)
    # if no negative
    if temp_stack.is_empty():
        while not temp_queue.is_empty():
            a_queue.append(temp_queue.serve())
    # if no positive
    elif temp_queue.is_empty():
        while not temp_stack.is_empty():
            a_queue.append(temp_stack.pop())
    # if got both
    if not (temp_stack.is_empty() and temp_queue.is_empty()):
        # get the item
        item_a = -temp_stack.pop()
        item_b = temp_queue.serve()
        # loop till one is empty
        while True:
            # negative have lower magnitude or same
            if item_a <= item_b:
                a_queue.append(-item_a)
                if temp_stack.is_empty():
                    while not temp_queue.is_empty():
                        a_queue.append(temp_queue.serve())
                    break
                item_a = -temp_stack.pop()
            # positive have lower magnitude
            else:
                a_queue.append(item_b)
                if temp_queue.is_empty():
                    while not temp_stack.is_empty():
                        a_queue.append(temp_stack.pop())
                    break
                item_b = temp_queue.serve()
    # return results
    return a_queue
```

Question 8 [10 marks = 3 + 5 + 2]

(a) How can we address collisions in Hash Tables? List at least 3 different approaches, covered in the lectures, and explain how they differ from each other. 3 of the following:

- Linear probing: searches for an item with hash value N at position table[N], and performs linear search from then on until found
- Quadratic probing: performs search at position $N + S^2$ on step S after a collision occurs
- Double hashing: Determines the step with a second hash function
- Separate chaining: Uses a linked list in each position, on collision appends to the list

3 marks: 0.5 for mentioning each approach from the list, 0.5 from explanations for each

(b) Consider the class `Hash` which has the instance variables `array`, `table_size`, and `count`, and the following methods: `__init__()`, `hash(the_key)` and `rehash()`.

Using Linear Probing, define a method `__setitem__(self, key, data)` which inserts the `data` into the Hash Table at the **position** calculated using the key. If a collision occurs, the method should resolve it using linear probing. If the Hash Table is full, rehash the Hash Table and proceed to insert as above.

```
def __setitem__(self, key, data):
    position = self.hash(key)
    for _ in range(self.table_size):
        # found empty slot
        if self.array[position] is None:
            self.array[position] = (key, data)
            self.count += 1
            return
        # found key
        elif self.array[position][0] == key:
            self.array[position] = (key, data)
            return
        else:  # not found, try next
            position = (position + 1) % self.table_size
    self.rehash()
    self.__setitem__(key, data)
```

5 marks: 1.5 mark for syntax, 1.5 mark for correctness, 2 marks for applying linear probing concept

(c) List one advantage and one disadvantage of Separate Chaining over Linear Probing? Explain your answer.

One of the following advantages or similar

- Items can be inserted and deleted easily with low time complexity.
- Size of the hash table is dynamic, allowing varying amount of items in the table.

One of the following disadvantages or similar

- Uses more memory as the linked-structure of the chains require extra space to store the links (also the key to overcome collision).
- Searching through the chain have a linear complexity due to its linked structure.

2 marks: 1 mark for one advantage, 1 mark for one disadvantage

Question 9 [12 marks = 2 + 2 + 3 + 3 + 2]

A Dequeue is a Queue that supports operations to add and serve items to and from the front and the rear of the Queue. This question is about implementing a Dequeue ADT based on an array. **All the method implementations should be circular to avoid wasting space in the underlying array. Use assertions to deal with potential errors.** The constructor and two additional methods are defined as follows:

```
class CircularDeQueue:
    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) Implement the method `def append_rear(self, new_item)`, which appends a `new_item` at the rear of the Queue.

```
def append_rear(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
```

0.5 appends to the queue

0.5 handling circularity

0.5 full queue special case

0.5 syntax correct

(b) Implement the method `def serve_front(self)`, which removes from the Queue and returns the object at the front of the Queue.

```
def serve_front(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
```

0.5 serves from the queue

0.5 handles circularity

0.5 empty queue special case

0.5 syntax correct

4

(c) Implement the method `def append_front(self)`, which appends a `new_item` at the front of the Queue.

```
def insert_front(self, new_item):
        assert not self.is_full(), "Queue is full"
        if self.front >0:
            self.front-=1
        else:
            self.front = len(self.the_array) -1
        self.the_array[self.front] = new_item
        self.count += 1
```

1.5 appends to the queue to the front correctly

0.5 handling circularity

0.5 full queue special case

0.5 syntax correct

(d) Implement the method `def serve_rear(self)`, which takes the object at the rear of the Queue, removing it from the queue and returning it.

```
def serve_back(self):
        assert not self.is_empty(), "Queue is empty"
        if self.rear >0:
            self.rear -=1
        else:
            self.rear = len(the_array) -1
        item = self.the_array[self.rear]
        self.count -=1
        return item
```

1.5 serves from the queue from the rear correctly

0.5 handling circularity

0.5 empty queue special case

0.5 syntax correct

(e) Implement the method `def print_items(self)`, which prints all the objects in the Queue from the front to the rear. Each element should be printed in one line.

```
def print_items(self):
    index = self.front
    for _ in range(self.count):
        print(str(self.the_array[index]))
        index = (index+1) % len(self.the_array)
```

0.5 prints all elements

0.5 syntax correct

1 handles circularity

Question 10 [7 marks = 3 + 2 + 2]

Consider the ADT SortedList and the class defined below:

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

    def __len__(self):
        return self.count
```

(a) Implement the class method `def _binary_search(self, item)`, which returns the index of `item` if it is in the list, or `-1` if the item is not in the list. Your implementation should have worst-case time complexity $O(\log n)$.

```
def _binary_search(self, item):
    lower = 0
    upper = len(self) -1
    while lower <= upper:
        mid = (lower + upper)//2
        if self.the_array[mid] == item:
            return mid
        elif self.the_array[mid] > item:
            upper = mid - 1
        else:
            lower = mid + 1
    return -1
```

0.5 syntax

1.5 correctness

1.0 binary search concept

(b) Calling the method `_binary_search` defined in part *a*, implement `def index(self, item)`. This method should return the index of the first occurrence of `item` in the list, or raise a `ValueError` exception if `item` is not in the list.

```
def index(self, item):
    position = self._binary_search(item)
    if position == -1:
        # item is not in the list
        raise ValueError("Element " + str(item) + " is not in the list")
    # there is at least a copy and is in position position
    while position >=0 and self.the_array[position] == item:
        # search back until we find the first appearance
        position -= 1
    return position + 1
```

0.5 not found handled with correct exception

0.5 finds a value

1 correctly finds first appearance

5

(c) What is the best-case time complexity of the method `index(self, item)`, and when does it occur? Explain your answer.

O(1), item is unique in the middle of the list.

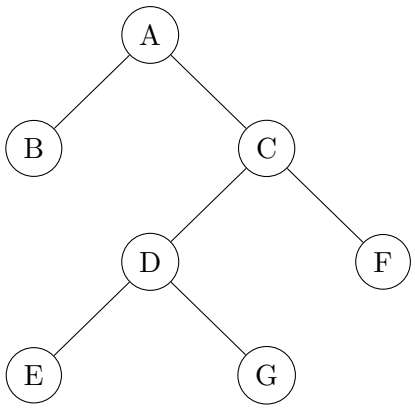
1 mark correct complexity

1 mark explanation of when it happens (no half marks)

Question 11 [6 marks = 3 + 1 + 1 +1]

This question is about Binary Trees and Binary Search Trees.

(a) State the outcomes of printing the elements of the Binary Tree below in pre-order, in-order and post-order.

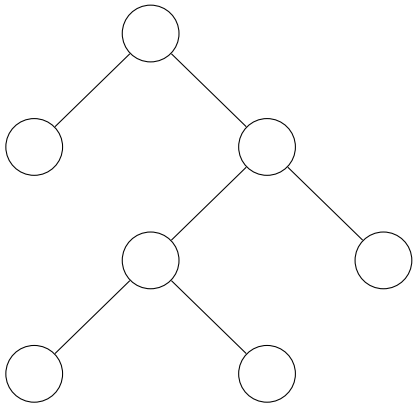


- **Pre-order:** A, B, C, D, E, G, F
- **In-order:** B, A, E, D, G, C, F
- **Pos-order:** B, E, G, D, F, C, A

3 marks: 1 mark per each correct, no partial marks.

(b) What are the best-case and worst-case time complexities in terms of the number of nodes, for an algorithm that prints the elements of a Binary Tree in pre-order. Explain your answer. No explanation means no marks.
Any printing should print each node once, best = worst case is O(N).
1 mark, correct complexity with explanation

(c) Fill in the nodes in the Binary Tree below with the elements from the list $[1, 5, 6, 7, 9, 10, 4]$ so that it is a valid Binary Search Tree.



1 mark for any valid BST. No partial marks.

(d) What is the worst-case time complexity for an algorithm that searchs for an item in a Binary Search Tree. Explain your answer. No explanation means no marks.
O(N), when you have a degenerate linear tree 1 mark for correct complexity with explanation. No half marks.

6

Question 12 [6 marks = 4 + 2]

The next two questions are about recursion.

(a) Consider the following partial implementation of a Binary Tree.

```
class TreeNode:
    def __init__(self, item=None, left=None, right=None):
        self.item = item
        self.left = left
        self.right = right

    def __str__(self):
        return str(self.item)


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

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

Using Python write a method `def height(self)` for the class `BinaryTree` that computes the height of a given tree using **recursion**.

```
    def height(self):
        return self.height_aux(self.root)

    def height_aux(self, current):
        if current is None:
            return 0
        else:
            return max(self.height_aux(current.left),
                       self.height_aux(current.right)) + 1
```

0.5 mark for correctly setting up the initial recursion in a separate method

0.5 syntax

0.5 mark correct base case

0.5 recursive case acts on sub-trees

2 marks for correctness

(b) Quick-sort is a recursive sorting algorithm that relies on a partition method with linear time complexity. If you had a magic algorithm to do the partition in $O(1)$ time, what would be the best and worst case time complexity of Quick-sort? Explain your answer. No explanation means no marks.

- **Best case**: The list is divided in halves, which happens O(log N) times. Because partitioning is constant, the best case complexity is O(log N).
- **Worst case**: The list is partition N times, reducing the size of the problem one element at the time. Thus, complexity is O(N) given a constant partition.

1 mark correct best case, with explanation.

1 mark correct worst case, with explanation.

No partial marks.