## Question 1. [8 MARKS]

Remeber that `type(v)` returns `v`'s type, and you can compare that result to `int`, `str`, or any other type using `==`. For example, `type(3) == int` evaluates to `True`.

### Part (a) [6 MARKS]

Implement this function recursively:

```
def first_int(L):
    '''Return the first int in L, or None if there are no ints.'''

    if not L:
        return None
    elif type(L[0]) == int:
        return L[0]
    else:
        return first_int(L[1:])
```

### Part (b) [2 MARKS]

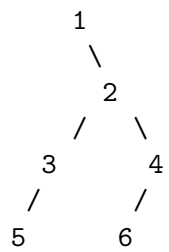
Given this call, does your solution look at `148`? Circle the correct answer.

```
first_int(['a', 'b', 150, 'c', 'd', 148, 'e'])
```

NO

## Question 2. [6 MARKS]

Consider this tree:




Write the various traversals:

```
preorder: 1 2 3 5 4 6

inorder: 1 5 3 2 6 4

postorder: 5 3 6 4 2 1
```

## Question 3. [8 MARKS]

This question has you trace a new traversal algorithm. Remember that "ADT" stands for "Abstract Data Type"; stacks and queues are both ADTs.

Consider this tree and this `Node` class (which we have seen a lot of in lecture):


```
class Node(object):
    '''A node in a tree, with a key, left subtree, and right subtree.'''

    def __init__(self, k):
        '''A node containing key k with no children.'''
        self.key = k
        self.left = None
        self.right = None
```

Consider this algorithm (written mostly in English), where `adt` is either a `Stack` or a `Queue`. When the algorithm says "add", if `adt` is a `Stack` it means `push`, and if `adt` is a `Queue` it means `enqueue`.

Similarly, when we say "remove" we mean `pop` or `dequeue` as appropriate.

Note that this traversal algorithm is putting `Node` objects into the ADT.

```
def traverse(r, adt):
    add Node r to adt
    while adt is not empty:
        remove the next Node n from adt and print n.key
        if n.left is not None, add n.left to adt
        if n.right is not None, add n.right to adt
```

**Part (a)** [4 MARKS] If `adt` refers to a `Queue`, what is printed for the tree at the top of the page?

```
1 2 3 4 5 6
```

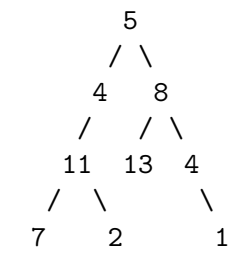
**Part (b)** [4 MARKS] If `adt` refers to a `Stack`, what is printed for the tree at the top of the page?

```
1 2 4 6 3 5
```

If you like, you can use this page for rough work for the traversal question.

## Question 4. [10 MARKS]

Consider this tree:



The path from the 5 to the 7 includes the values 5, 4, 11, and 7, whose sum is 27.
The path from the 5 to the 2 includes the values 5, 4, 11, and 2, whose sum is 22.
The path from the 5 to the 13 includes the values 5, 8, and 13, whose sum is 26.
The path from the 5 to the 1 includes the values 5, 8, 4, and 1, whose sum is 18.

Write the following function recursively. Notice that the base case is when `r` is a leaf. This won't work if you call it on an empty tree, so make sure you don't when you recurse.

No helper functions are allowed. You need to recurse on the left and right subtrees; what will you pass into these calls for the `total` parameter?

```
def has_path_sum(r, total):
    '''Given Node r, return a boolean indicating whether there is a path
    from r to any leaf whose values add up to int total. Precondition: r
    is not None.'''

if r.left == None and r.right == None:
    return total == r.key
else:
    left = r.left and has_path_sum(r.left, total - r.key)
    right = r.right and has_path_sum(r.right, total - r.key)
    return left or right
```

## Question 5. [10 MARKS]

The parentheses below line up for each boolean subexpression. Note that the entire body is a single `return` statement.

Fill in the blanks with `or`, `and`, `subL`, `seqL`, `subL[1:]`, and `seqL[1:]` so that the code works as advertised. You may use each of them as often as you need to.

```
def is_sub_sequence(subL, seqL):
    '''Return whether the items in list subL appear in list seqL in the same order.
    A list is a sub-sequence of itself, and the empty list if a sub-sequence of
    every list.

    Examples:
        is_sub_sequence([1, 3], [1, 2, 3, 4]) should evaluate to True.
        is_sub_sequence([3, 1], [1, 2, 3, 4]) should evaluate to False.
    '''

    return (
            len(subL) == 0 ____________

                (len(seqL) > 0 ____________

                    (is_sub_sequence(____________, ____________) ____________

                        (subL[0] == seqL[0] ____________

                            is_sub_sequence(____________, ____________)

                        )

                    )
                )
           )

    return (
            len(subL) == 0 or
                (len(seqL) > 0 and
                    (is_sub_sequence(subL, seqL[1:]) or
                        (subL[0] == seqL[0] and
                            is_sub_sequence(subL[1:], seqL[1:])
                        )
                    )
                )
           )
```