1. Write a function `is_sorted` that takes in an integer `n` and returns true if the digits of that number are increasing from right to left.

```python
def is_sorted(n):
    """
    >>> is_sorted(2)
    True
    >>> is_sorted(22222)
    True
    >>> is_sorted(9876543210)
    True
    >>> is_sorted(9087654321)
    False
    """
```
Tree Recursion

1. Mario needs to jump over a series of Piranha plants, represented as a string of 0’s and 1’s. Mario only moves forward and can either *step* (move forward one space) or *jump* (move forward two spaces) from each position. How many different ways can Mario traverse a level without stepping or jumping into a Piranha plant? Assume that every level begins with a 1 (where Mario starts) and ends with a 1 (where Mario must end up).

```python
def mario_number(level):
    """
    Return the number of ways that Mario can traverse the level, where Mario can either hop by one digit or two digits each turn. A level is defined as being an integer with digits where a 1 is something Mario can step on and 0 is something Mario cannot step on.
    >>> mario_number(10101)
    1
    >>> mario_number(11101)
    2
    >>> mario_number(100101)
    0
    """
    if __________:
        ________
    elif ______________:
        ________
    else:
        _______________________
```
2. Implement the function `make_change`. You may not need to use all the lines.

```python
def make_change(n):
    """Write a function, make_change that takes in an integer amount, n, and returns the minimum number of coins we can use to make change for that n, using 1-cent, 3-cent, and 4-cent coins. Look at the doctests for more examples.
    >>> make_change(5)
    2
    >>> make_change(6) # tricky! Not 4 + 1 + 1 but 3 + 3
    2
    """
    if ______________:
        return 0
    elif ____________:
        return _________________________
    elif ____________:
        ______________________________
        ______________________________
        return __________________________
    else:
        ______________________________
        ______________________________
        ______________________________
        return __________________________
```

1. The following is an Abstract Data Type (ADT) for elephants. Each elephant keeps track of its name, age, and whether or not it can fly. Given our provided constructor, fill out the selectors:

```python
def elephant(name, age, can_fly):
    """
    Takes in a string name, an int age, and a boolean can_fly. Constructs an elephant with these attributes.
    >>> dumbo = elephant("Dumbo", 10, True)
    >>> elephant_name(dumbo)
    "Dumbo"
    >>> elephant_age(dumbo)
    10
    >>> elephant_can_fly(dumbo)
    True
    """
    return [name, age, can_fly]
def elephant_name(e):
    #
    def elephant_age(e):
    #
    def elephant_can_fly(e):
```


2. This function returns the correct result, but there’s something wrong about its implementation. How do we fix it?

```python
def elephant_roster(elephants):
    """
    Takes in a list of elephants and returns a list of their names.
    """
    return [elephant[0] for elephant in elephants]
```

3. Fill out the following constructor for the given selectors.

```python
def elephant(name, age, can_fly):
    """
```

```python
def elephant_name(e):
    return e[0][0]
def elephant_age(e):
    return e[0][1]
def elephant_can_fly(e):
    return e[1]
```

4. How can we write the fixed `elephant_roster` function for the constructors and selectors in the previous question?
5. (Optional) Fill out the following constructor for the given selectors.

def elephant(name, age, can_fly):
    """
    >>> chris = elephant("Chris Martin", 38, False)
    >>> elephant_name(chris)
    "Chris Martin"
    >>> elephant_age(chris)
    38
    >>> elephant_can_fly(chris)
    False
    """

def select(command):
    return select

def elephant_name(e):
    return e("name")

def elephant_age(e):
    return e("age")

def elephant_can_fly(e):
    return e("can_fly")