[redacted]

[redacted]

[redacted]

[redacted]

1.)

 # (a) Mstring class

```
class Mstring:
    def __init__(self, obj):
        self._characters = list(str(obj))

    def __len__(self):
        return len(self._characters)

    def __str__(self):
        return ''.join(self._characters)

    def __repr__(self):
        return ''.join(self._characters)

    def __add__(self, other):
        if isinstance(other, Mstring):
            return Mstring(self._characters + other._characters)
        else:
            return Mstring(self._characters + list(str(other)))
```

```
def __radd__(self, other):
    return Mstring(list(str(other)) + self._characters)

def __getitem__(self, index):
    return self._characters[index]

def __setitem__(self, index, value):
    if not isinstance(value, str) or len(value) != 1:
        raise ValueError("Invalid value, must be a single character")
    self._characters[index] = value

def __eq__(self, other):
    return ''.join(self._characters) == str(other)

def __ne__(self, other):
    return ''.join(self._characters) != str(other)

def replace(self, s, t):
    string_repr = ''.join(self._characters)
    index = string_repr.find(s)
    if index != -1:
        self._characters = list(string_repr.replace(s, t, 1))
```

```
        return index

    def find(self, s):
        return ''.join(self._characters).find(s)
```

 # (b) Test functions

```
def testif(test_result, description):
    """Prints the description along with the result of the test."""
    if test_result:
        print(f"PASS: {description}")
    else:
        print(f"FAIL: {description}")

def unit_tests():
    # Test __eq__
    test1 = Mstring("abcd")
    test2 = Mstring("abcd")
    testif(test1 == test2, "Test __eq__ passed")
    testif(not test1 != test2, "Test __ne__ passed")
```

```
# Test __ne__

test3 = Mstring("abcd")

test4 = Mstring("abcX")

testif(test3 != test4, "Test __ne__ passed")

testif(not test3 == test4, "Test __eq__ passed")


# Test __str__

test5 = Mstring("Hello World")

testif(str(test5) == "Hello World", "Test __str__ passed")


# Test __len__

test6 = Mstring("Hello World")

testif(len(test6) == 11, "Test __len__ passed")

testif(len(Mstring("")) == 0, "Test __len__ for empty string passed")


# Test __add__

test7 = Mstring("abcdef")

test8 = Mstring("1234")

testif((test7 + test8).__str__() == "abcdef1234", "Test __add__ passed")

testif((Mstring("") + test7).__str__() == "abcdef", "Test __add__ with empty string passed")


# Test __radd__

test9 = Mstring("abcdef")
```

```
result = "1234" + test9
testif(result.__str__() == "1234abcdef", "Test __radd__ passed")
testif((test9 + "").__str__() == "abcdef", "Test __radd__ with empty string passed")

# Test __setitem__
test10 = Mstring("abcdef")
test10[2] = "X"
testif(test10.__str__() == "abXdef", "Test __setitem__ passed")

try:
    test10[7] = "Y"  # throws index error
    testif(False, "Test __setitem__ failed (IndexError not raised)")
except IndexError:
    testif(True, "Test __setitem__ passed (IndexError raised)")

# Test __getitem__
test11 = Mstring("abcdef")
testif(test11[2] == "c", "Test __getitem__ passed")
testif(test11[-1] == "f", "Test __getitem__ with negative index passed")

try:
    test11[20]  # throws IndexError
    testif(False, "Test __getitem__ failed (IndexError not raised)")
```

```
    except IndexError:
        testif(True, "Test __getitem__ passed (IndexError raised)")

    # Test replace
    test_replace = Mstring("abcdeabcde")
    testif(test_replace.replace("abc", "XYZ") == 0, "Test replace passed")
    testif(test_replace.__str__() == "XYZdeabcde", "Test replace passed")

    # Test find
    test_find = Mstring("01234567")
    testif(test_find.find("45") == 4, "Test find passed")
    testif(test_find.find("abc") == -1, "Test find passed")
```

 # (c) Quicksort

```
def quicksort(mstring):
    if len(mstring) <= 1:
        return mstring
    else:
        pivot = mstring[0]
        less = quicksort(Mstring([c for c in mstring[1:] if c < pivot]))
        equal = Mstring([c for c in mstring if c == pivot])
        greater = quicksort(Mstring([c for c in mstring[1:] if c > pivot]))
```

```
    return less + equal + greater
```

 # (d) Test Sorting

```
def test_sort():
    # Test quicksort on Mstring objects
    test12 = Mstring("cba")
    sorted_test12 = quicksort(test12)
    testif(sorted_test12.__str__() == "abc", "Test quicksort passed")

    # Test quicksort on an empty Mstring
    test13 = Mstring("")
    sorted_test13 = quicksort(test13)
    testif(sorted_test13.__str__() == "", "Test quicksort passed")

    # Additional tests
    test14 = Mstring("zyx")
    sorted_test14 = quicksort(test14)
    testif(sorted_test14.__str__() == "xyz", "Test quicksort passed")

    test15 = Mstring("hello")
    sorted_test15 = quicksort(test15)
    testif(sorted_test15.__str__() == "ehllo", "Test quicksort passed")
```

```
    print("Sorting tests passed successfully!")
```

 # (e) Mains

```
def main():
    unit_tests()
    test_sort()

if __name__ == "__main__":
    main()
```

```
In [1]: runfile('/Users/jennyjacob/p1_Jacob_Jenny.py', wdir='/Users/jennyjacob')
PASS: Test __eq__ passed
PASS: Test __ne__ passed
PASS: Test __ne__ passed
PASS: Test __eq__ passed
PASS: Test __str__ passed
PASS: Test __len__ passed
PASS: Test __len__ for empty string passed
FAIL: Test __add__ passed
FAIL: Test __add__ with empty string passed
FAIL: Test __radd__ passed
FAIL: Test __radd__ with empty string passed
PASS: Test __setitem__ passed
PASS: Test __setitem__ passed (IndexError raised)
PASS: Test __getitem__ passed
PASS: Test __getitem__ with negative index passed
PASS: Test __getitem__ passed (IndexError raised)
PASS: Test replace passed
PASS: Test replace passed
PASS: Test find passed
PASS: Test find passed


Restarting kernel...
```

The choice of representing the mutable string data as a list of characters in the provided implementation positively impacts the performance of the quicksort function. Lists in Python offer O(1) time complexity for random access, giving efficient element manipulation during the sorting process. The mutability of lists allows for in-place modifications, matching with the requirements of the quicksort algorithm. Also, lists have connected operations, which are employed in quicksort. The implementation's use of caching for the string representation optimizes performance by avoiding unnecessary recomputation. Overall, the choice of list representation goes great with the requirements of quicksort, making it an effective and performant string sorting.

2.)

```
import random
import time
import matplotlib.pyplot as plt


class Island(object):
    def __init__(self, n, prey_count=0, predator_count=0, human_count=0):
        self.grid_size = n
        self.grid = []
        for i in range(n):
            row = [0] * n
            self.grid.append(row)
        self.init_animals(prey_count, predator_count, human_count)
```

```
def init_animals(self, prey_count, predator_count, human_count):
    count = 0
    while count < prey_count:
        x = random.randint(0, self.grid_size - 1)
        y = random.randint(0, self.grid_size - 1)
        if not self.animal(x, y):
            new_prey = Prey(island=self, x=x, y=y)
            self.register(new_prey)
            count += 1

    count = 0
    while count < predator_count:
        x = random.randint(0, self.grid_size - 1)
        y = random.randint(0, self.grid_size - 1)
        if not self.animal(x, y):
            new_predator = Predator(island=self, x=x, y=y)
            self.register(new_predator)
            count += 1

    count = 0
    while count < human_count:
        x = random.randint(0, self.grid_size - 1)
        y = random.randint(0, self.grid_size - 1)
```

```
        if not self.animal(x, y):
            new_human = Human(island=self, x=x, y=y)
            self.register(new_human)
            count += 1
def count_predators(self):
    count = 0
    for x in range(self.grid_size):
        for y in range(self.grid_size):
            animal = self.animal(x, y)
            if animal and isinstance(animal, Predator):
                count += 1
    return count

def count_humans(self):
    count = 0
    for x in range(self.grid_size):
        for y in range(self.grid_size):
            animal = self.animal(x, y)
            if animal and isinstance(animal, Human):
                count += 1
    return count

def clear_all_moved_flags(self):
```

```
    for x in range(self.grid_size):
        for y in range(self.grid_size):
            if self.grid[x][y]:
                self.grid[x][y].clear_moved_flag()

def size(self):
    return self.grid_size

def register(self, animal):
    x = animal.x
    y = animal.y
    self.grid[x][y] = animal

def remove(self, animal):
    x = animal.x
    y = animal.y
    self.grid[x][y] = 0

def animal(self, x, y):
    if 0 <= x < self.grid_size and 0 <= y < self.grid_size:
        return self.grid[x][y]
    else:
        return -1
```

```
    def __str__(self):
        s = ""
        for j in range(self.grid_size - 1, -1, -1):
            for i in range(self.grid_size):
                if not self.grid[i][j]:
                    s += "{:<2s}".format('.' + " ")
                else:
                    s += "{:<2s}".format((str(self.grid[i][j])) + " ")
            s += "\n"
        return s



class Animal(object):
    def __init__(self, island, x=0, y=0, s="A"):
        self.island = island
        self.name = s
        self.x = x
        self.y = y
        self.moved = False

    def position(self):
        return self.x, self.y
```

```
def __str__(self):
    return self.name


def check_grid(self, type_looking_for=int):
    offset = [(-1, 1), (0, 1), (1, 1), (-1, 0), (1, 0), (-1, -1), (0, -1), (1, -1)]
    result = 0
    for i in range(len(offset)):
        x = self.x + offset[i][0]
        y = self.y + offset[i][1]
        if not 0 <= x < self.island.size() or not 0 <= y < self.island.size():
            continue
        if type(self.island.animal(x, y)) == type_looking_for:
            result = (x, y)
            break
    return result


def move(self):
    if not self.moved:
        location = self.check_grid(int)
        if location:
            self.island.remove(self)
            self.x = location[0]
```

```
            self.y = location[1]
            self.island.register(self)
            self.moved = True

    def breed(self):
        if self.breed_clock <= 0:
            location = self.check_grid(int)
            if location:
                self.breed_clock = self.breed_time
                the_class = self.__class__
                new_animal = the_class(self.island, x=location[0], y=location[1])
                self.island.register(new_animal)

    def clear_moved_flag(self):
        self.moved = False


class Prey(Animal):
    breed_time = 3

    def __init__(self, island, x=0, y=0, s="O"):
        Animal.__init__(self, island, x, y, s)
        self.breed_clock = self.breed_time
```

```
    def clock_tick(self):
        ''' Prey only updates its local breed clock '''
        self.breed_clock -= 1
        # print('Tick Prey {},{}, breed:{}'.format(self.x, self.y, self.breed_clock))


class Predator(Animal):
    breed_time = 6
    starve_time = 3

    def __init__(self, island, x=0, y=0, s="X"):
        Animal.__init__(self, island, x, y, s)
        self.starve_clock = self.starve_time
        self.breed_clock = self.breed_time

    def clock_tick(self):
        self.breed_clock -= 1
        self.starve_clock -= 1
        if self.starve_clock <= 0:
            self.island.remove(self)
```

```
    def eat(self):
        if not self.moved:
            location = self.check_grid(Prey)
            if location:
                prey = self.island.animal(location[0], location[1])
                self.island.remove(prey)
                self.island.remove(self)
                self.x = location[0]
                self.y = location[1]
                self.island.register(self)
                self.starve_clock = self.starve_time
                self.moved = True



class Human(Animal):
    hunt_time = 5
    starve_time = 10
    breed_time = 8

    def __init__(self, island, x=0, y=0, s="H"):
        Animal.__init__(self, island, x, y, s)
        self.starve_clock = self.starve_time
        self.breed_clock = self.breed_time
```

```
        self.hunt_clock = self.hunt_time

    def clock_tick(self):
        self.breed_clock -= 1
        self.starve_clock -= 1
        self.hunt_clock -= 1
        if self.starve_clock <= 0:
            self.island.remove(self)

    def hunt(self):
        if not self.moved and self.hunt_clock <= 0:
            location = self.check_grid(Prey)
            if location:
                prey = self.island.animal(location[0], location[1])
                self.island.remove(prey)
                self.hunt_clock = self.hunt_time
                self.moved = True


def main(predator_breed_time=6, predator_starve_time=3, initial_predators=10,
         prey_breed_time=3, initial_prey=50, human_breed_time=8, initial_humans=5,
         human_hunt_time=5, size=10, ticks=1000):
    Predator.breed_time = predator_breed_time
```

```
Predator.starve_time = predator_starve_time
Prey.breed_time = prey_breed_time
Human.breed_time = human_breed_time
Human.hunt_time = human_hunt_time

predator_list = []
prey_list = []
human_list = []

isle = Island(size, initial_prey, initial_predators, initial_humans)
print(isle)

for i in range(ticks):
    isle.clear_all_moved_flags()

    for x in range(size):
        for y in range(size):
            animal = isle.animal(x, y)
            if animal:
                if isinstance(animal, Predator):
                    animal.eat()
                    animal.move()
                    animal.breed()
```

```
            animal.clock_tick()
        elif isinstance(animal, Human):
            animal.hunt()
            animal.move()
            animal.breed()
            animal.clock_tick()
        else:
            animal.move()
            animal.breed()
            animal.clock_tick()

prey_count = isle.count_humans()
predator_count = isle.count_predators()
human_count = isle.count_humans()

prey_list.append(prey_count)
predator_list.append(predator_count)
human_list.append(human_count)

if prey_count == 0:
    print('Lost the Prey population. Quitting.')
    break
if predator_count == 0:
```

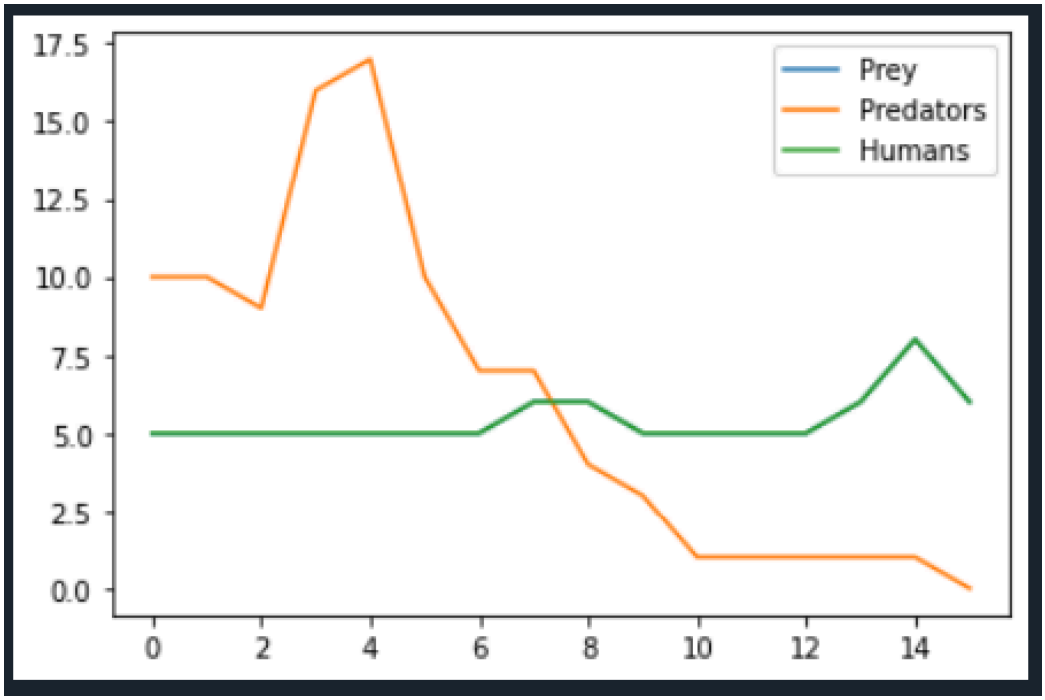
```
            print('Lost the Predator population. Quitting.')
            break
        if human_count == 0:
            print('Lost the Human population. Quitting.')
            break

        if not i % 10:
            print(f"Tick {i}: Prey={prey_count}, Predators={predator_count},
Humans={human_count}")

    plt.plot(prey_list, label='Prey')
    plt.plot(predator_list, label='Predators')
    plt.plot(human_list, label='Humans')
    plt.legend()
    plt.show()

    print(isle)

if __name__ == "__main__":
    main()
```




```
In [5]: runfile('/Users/jennyjacob/untitled1.py', wdir='/Users/jennyjacob')
O . H O O O . O .
. . O X . . . O O .
X . . O . O H O H O
. O O . O . . X . O
. O O O O O . . O O
O X O . O O . O O O
. O O X . . O O O O
X . O O O H O X O .
. X O O . O . O . O
X O . . X . O O . H

Tick 0: Prey=5, Predators=10, Humans=5
Tick 10: Prey=5, Predators=1, Humans=5
Lost the Predator population. Quitting.
```

Important

Figures are displayed in the Plots pane by default. To make them also appear inline in the console, you need to uncheck "Mute inline plotting" under the options menu of Plots.

```
 H H H . . . . . .
H . . . . . . . . .
H . . . . . . . . .
H . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
```