Assignment 2

Problem 1 - Convert recursion to iteration

Rewrite the following recursive function in [recursion_to_iteration.py] using iteration instead of recursion. Your iterative function should do exactly the same task as the given recursive function.

```
def recur(n):
    if n < 0:
        return -1
    elif n < 10:
        return 1
    else:
        return 1 + recur(n // 10)
```

Important: this problem will be graded manually.

Problem 2 - All Possible Combinations

The *knapsack problem* or rucksack problem is a classic in combinatorial optimization. Given a set of items, each with a mass and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit, and the total value is as large as possible.

We're going to work on a simpler version of this problem. Given a set of items, each associated with a value, we're going to determine which combinations of items sum up to a given value: the capacity of the knapsack.

Implement a solution for this simpler problem in file `knapsack.py`

Function `knapsack(capacity, weights)` enumerates and returns a list of sublists of all possible combinations of items whose values sum up to a given `capacity`.

Usage examples:

```
In [1]: knapsack(14, [1, 2, 8, 4, 9, 1, 4, 5])
Out[1]: [[9, 5], [9, 1, 4], [4, 1, 4, 5], [4, 9, 1], [8, 1, 5], [2, 8, 4], [2, 8,
4], [1, 9, 4], [1, 4,
4, 5], [1, 4, 9], [1, 8, 5], [1, 8, 1, 4], [1, 8, 4, 1]] #(Order for inner values,
outer lists don’t matter.)
```

Important: function `knapsack(capacity, weights)` is not well suited for the recursion you need, so you may create an extra function called from `knapsack` to enable recursion.

Problem 3 - Element Uniqueness Problem

Consider the recursive solution for the *element uniqueness problem* reproduced below from p.165 of the textbook. Its worst case runtime is a horrid exponential: O(2^N)

```
def unique3(S, start, stop):
    """Return True if there are no duplicate elements in the slice S[start:stop]"""
    if stop - start <= 1: return True                # at most one item
    elif not unique3(S, start, stop-1): return False # first part has duplicate
    elif not unique3(S, start+1, stop): return False # second part has duplicate
    else: return S[start] != S[stop-1]               # do first and last differ?
```

In file unique.py, implement an efficient recursive function `unique(S)` for solving the element uniqueness problem. Your implementation's worst case runtime must be quadratic, in O(N^2), without requiring the input set to be sorted.

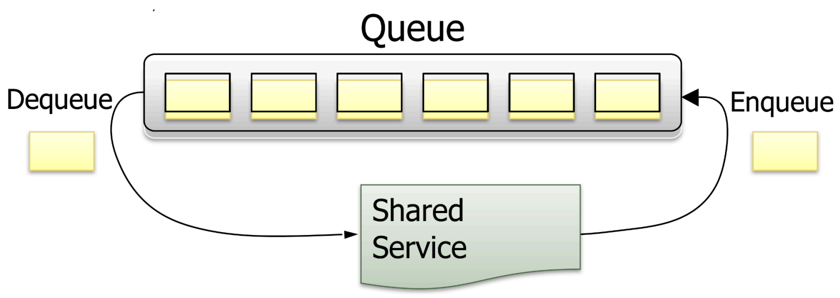
Usage examples:

```
In [1]: unique([1, 7, 6, 5, 4, 3, 1])
Out[1]: False

In [2]: unique([9, 4, 'a', [], 0]) # All elements are unique
Out[2]: True
```

Problem 4 - Round Robin Strategy

In certain applications such as turn-based games, the queue ADT facilitates the implementation of rotations. Each object gets dequeued when its turn comes, and then enqueued back when its turn is over; this ensures fair access to resources.



In file RoundRobin.py, modify the circular array implementation of `ArrayQueue` to include a `rotate()` method that has semantics identical to the combination `Q.enqueue(Q.dequeue())`.
However, your implementation should be more efficient than making two separate calls (*note: this will be checked manually*). In particular, there is no need to increment the size of the array, and then decrement it.

Problem 5 Token checker

Token checking is an important part of the syntax check when compiling the code of a program. The objective is to verify that all symbols (values, variable names, parentheses, operators, statement terminators, …) are in legal positions.

In this problem, we will focus on parentheses. Your goal is to check that the position of every closing bracket matches legally with the position of its opening bracket: `"["` with `"]"`, `"{"` with `"}"`, and `"("` with `")"`. A legal match occurs when the closing bracket follows immediately the most recent opening bracket that hasn't been matched yet.

In file [`TokenChecker.py`](), implement function `check_tokens(filename)`

`check_tokens` reads the contents of file `filename`, and then checks that all the bracket tokens match correctly. It should return `True` for a file whose matches are all legal, `False` otherwise.

A correct matching sequence could be:

```
"(", "[", "]", "{", "(", ")", "}", ")"
```

Examples of incorrect matching sequences include:

```
"[", "{", "(", "(", ")", "}", ")", "]"

"{", "(", ")", "}", "[", "]", "{", "}", "}"

"{", "(", ")", "}", "(", "{", "[", "]", "}"
```

Usage example with the code of [`test.c`]() below:

```
int main()
{
    int i, sum = 0;
    int n[10];

    for ( i = 0; i < 10; i++ ) {
        n[ i ] = i + 100;
    }

    for ( i = 1; i <= LAST; i++ ) {
        sum += i;
    }
    printf("sum = %d\n", sum);

    return 0;
}
```

```
In [1]: check_tokens("test.c")
Out[1]: True
```

Hint: use the power of the stack!

Submission format

The files below constitute your coding canvas for this assignment.

1. `recursion_to_iteration.py`
2. `knapsack.py`
3. `unique.py`
4. `RoundRobin.py`
5. `TokenChecker.py`

You need to complete them, and then submit them directly (do not put them in a directory or zip) to gradescope:

https://www.gradescope.com/courses/399287/assignments/2080895