CSE101

DUE MAY. 19

23:59 KST

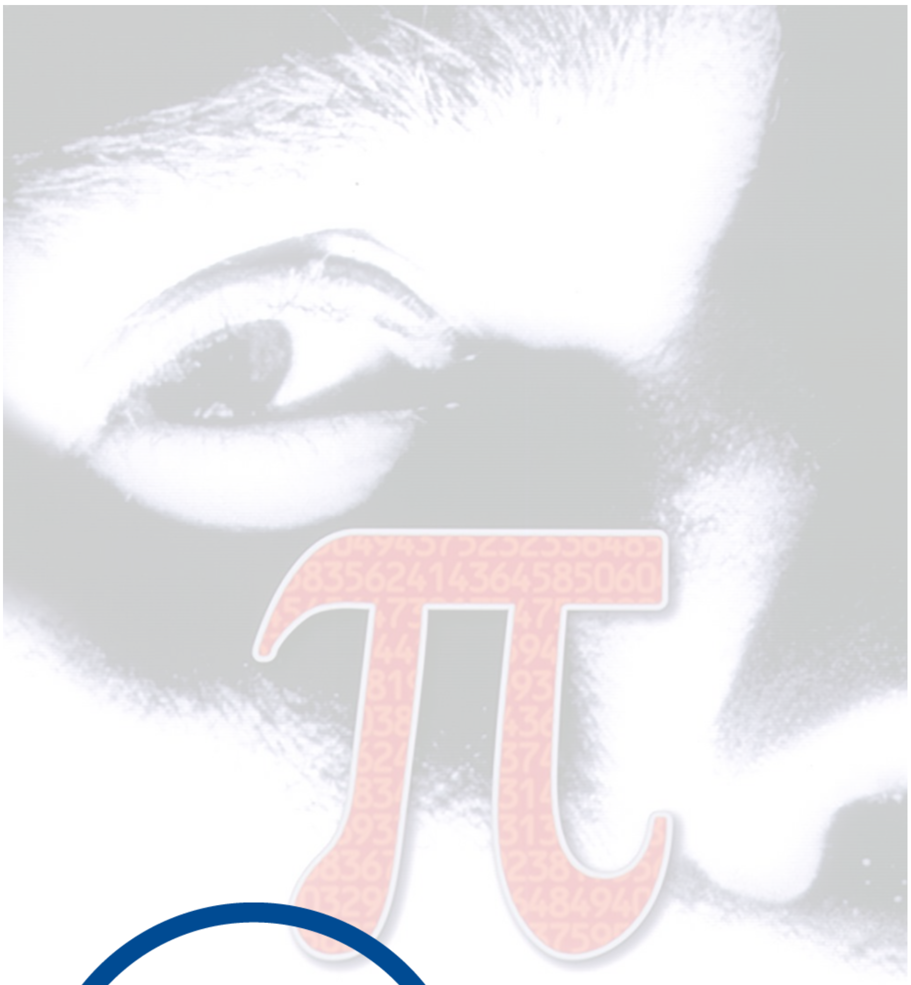
ASSIGNMENT IV

PROFESSOR

FRANCOIS

RAMEAU

RECURSION



SUNY Korea

The State University of New York

General instructions

- Please add your name and email in the dedicated location at the top of your code
- Do not use any external library, except when explicitly requested
- Try to follow the naming convention proposed by PEP-8 seen in class
- Use ONLY what we have already seen in class. If not, you will get ZERO points
- Use meaningful names for your variables and functions
- If you face problem submitting your code via GitHub please contact the professor and the TA by email
- Note that the received code will be tested on a classifier to detect potential usage of Large Language Model. We will also pay particular attention to plagiarism
- Leave comments in your code to explain your code and describe the difficulties you faced

INVITATION LINK

https://classroom.github.com/a/lOaAVRtn

Exercise 1: Estimation of pi (2 points)

The use of recursion to estimate pi has its roots in the development of infinite series and mathematical techniques throughout history. Ancient mathematicians like Archimedes and Madhava laid the foundation for using recursive methods by working with iterative series and algorithms. With the advent of calculus, infinite series such as the Leibniz formula and Gregory formula emerged, which can be elegantly expressed using recursion. While recursive approaches for estimating pi might not be the most efficient in modern times, they serve as valuable examples of applying recursion to solve mathematical problems and illustrate the continuous evolution of computational techniques for calculating pi.

In this exercise, you will implement 3 of these approaches with recursion.

1. The Wallis product (0.5 point)

The first approach you will implement is the Wallis product to approximate pi:

$$\frac{\pi}{2} = \prod_{n=1}^{\infty} \frac{4n^2}{4n^2-1} = \prod_{n=1}^{\infty} \left(\frac{2n}{2n-1} \cdot \frac{2n}{2n+1}\right) = \left(\frac{2}{1} \cdot \frac{2}{3}\right) \cdot \left(\frac{4}{3} \cdot \frac{4}{5}\right) \cdot \left(\frac{6}{5} \cdot \frac{6}{7}\right) \cdot \left(\frac{8}{7} \cdot \frac{8}{9}\right) \cdot \cdots$$

You will implement the Wallis pi approximation in the function `wallisPi(n)`. Note that n is the number of recursive calls you will use to approximate pi. The more iterations, the most accurate approximation you will get.

2. The Gregory's series (0.5 point)

Now that your Wallis product is working you will try to implement the Gregory's function in a recursive manner:

$$\frac{\pi}{4} = \sum_{n=1}^{\infty} \frac{(-1)^n}{2n+1} = 1 - \frac{1}{3} + \frac{1}{5} + \frac{1}{7} + \cdots$$

Implement this recursive pi approximation in the function `gregoryPi(n)`.

3. The Ramanujan's formula (1 point)

Now we will implement a slightly more complex (but faster converging) solution, the Ramanujan's formula. To implement this formula, you will first have to implement the factorial function as developed in the class. You will implement it in the function `factorial(n)`.
Now you can implement the following function in a recursive manner given this formula:

$$\pi = \frac{9801}{2\sqrt{2}\sum_{n=0}^{\infty} \frac{(4n)!}{(n)!} \cdot \frac{(1103 + 26390n)}{(4 \times 99)^{4n}}}$$

Note that only the second part of the denominator needs to be computed recursively. The rest of the code is provided in the main.

Exercise 2: Reimplement things with recursion (8 points)

In the second part of this homework, we will revisit some iterative functions we have studied before in this class (either in labs, lecture notes or assignments). For all these exercises, utilizing for loops is prohibited.

1. Find the minimum value in a list (2 points)

You all know the function `min()` in Python? This function returns the minimum value in a list. For instance if you consider the following list `[33, 29, 70, 1, 89]`, the `min()` function will return 1. You will reimplement this behavior using recursion only.

Your function will be implemented in the function `findMin(lst, min=None)`.

What is this "`min=None`"? In a function definition, variables can be initialized with default values, such as `min=None` in the `findMin` function. This allows the function to be called without explicitly providing a value for that argument, and the default value will be used instead. When you initialize default values, you can call your function without these arguments: `findMin(lst)` will, for instance, work perfectly well and will initialize the variable min to the value None. It is particularly useful when dealing with recursive functions, as it makes the call easier for the user. An example can be found here.

For reference, here is the original iterative function:

```python
def findMin(lst):
    min = lst[0]
    for i in range(1,len(lst)):
        if lst[i]<min:
            min = lst[i]
    return min
```

2. Check if a list is sorted in an ascending manner (2 points)

In the sorting lab, we have implemented bubblesort, this sorting algorithm stops when the list is sorted. This check can be implemented by looking every two adjacent elements in the list are in the right order. The iterative implementation is the following:

```python
def checkSort(lst):
    for i in range(len(lst)-1):
        if lst[i]>lst[i+1]:
            return False
    return True
```

Reimplement this function in `checkSortRec(lst)` using recursion only.

3. Count occurrences of a number in a list (2 points)

Write a recursive code to count the number of occurrences of the query integer in the list. This behavior will be implemented in the function `countOccurenceRec(lst, query, count=0).` Once again we have an initialized variable count therefore, the call of this function can be `countOccurenceRec(lst, query).`

For reference, here is the iterative version:

```python
def countOccurence(lst, query):
    count = 0
    for value in lst:
        if value == query:
            count += 1
    return count
```

4. Compute distances between a value and elements in a list (2 points)

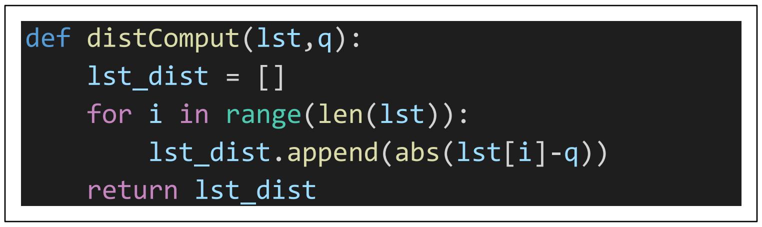
Finally, we will implement a function to compute the distances between a query value and a list containing integers. Your function, `distComputRec(lst, q, lst_dist)`, will populate `lst_dist` with all the distances between the query q and the elements of `lst`. Note that this function does not return anything; instead, it modifies the `lst_dist` in place. To implement this function, you should not use any index counter, but rather get inspiration from examples in the lecture notes (tip: think about slicing).

Usage example:

```
Lst_dist = []

distComputRec([1, 4, 6, 8, 12],5,lst_dist) → lst_dist: [4, 1, 1, 3, 7]
```

Iterative code for reference :



```
def distComput(lst,q):
    lst_dist = []
    for i in range(len(lst)):
        lst_dist.append(abs(lst[i]-q))
    return lst_dist
```