Before you turn this problem in, make sure everything runs as expected. First, **restart the kernel** (in the menubar, select Kernel→Restart) and then **run all cells** (in the menubar, select Cell→Run All).

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE", as well as your name and collaborators below:

In [1]:
```
NAME = "Pedro Haschelevici"
COLLABORATORS = "Barbara Machado, Fellipe Couto"
```

CS110 Fall 2020 - Assignment 1

Algorithm design & sorting - Problem set

Instead of a mini-project, which you will encounter in many of the upcoming CS110 assignments, this time you will be solving three independent problems. Use this opportunity to start your work early, finishing a question every week or so of class. Ideally, in the last week before the deadline, you will only have Q3 left to complete.

Fell free to add more cells to the ones always provided in each question to expand your answers, as needed. Make sure to refer to the CS110 course guide on the grading guidelines, namely how many HC identifications and applications you are expected to include in each assignment.

If you have any questions, do not hesitate to reach out to the TAs in the Slack channel "#cs110-algo-f20", or come to one of your instructors' OHs.

Submission Materials

Your assignment submission needs to include the following resources:

1. A PDF file must be the first resource and it will be created from the Jupyter notebook template provided in these instructions. Please make sure to use the same function names as the ones provided in the template. If your name is “Dumbledore”, your PDF should be named “Dumbledore.pdf”.
2. Your second resource must be a single Python/Jupyter Notebook named “Dumbledore.ipynb”. You can also submit a zip file that includes your Jupyter notebook, but please make sure to name it “Dumbledore.zip” (if your name is Dumbledore!).

Question 0 [#responsibility]

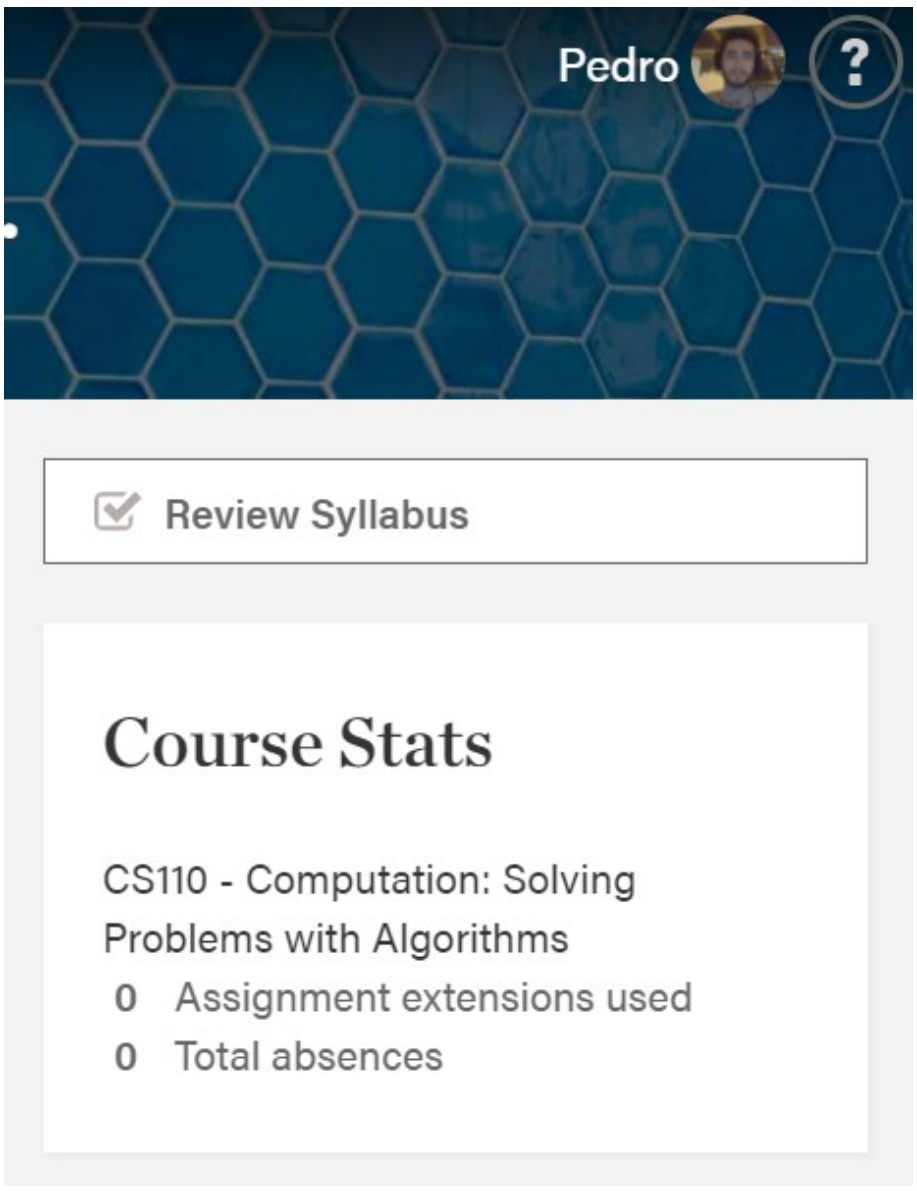
Take a screenshot of your CS110 dashboard on Forum where the following is visible:

- your name.
- your absences for the course have been set to excused up to the end of week 3 (inclusively).

This will be evidence that you have submitted acceptable pre-class and make-up work for a CS110 session you may have missed. Check the specific CS110 make-up and pre-class policies in the syllabus of the course.

In [2]:
```
from IPython.display import Image
Image(filename="proof.jpg")
```

Out[2]:



Question 1: Iteration vs. recursion [#ComputationalSolutions, #PythonProgramming]

A Fibonacci Sequence (https://en.wikipedia.org/wiki/Fibonacci_number) is a list of numbers, in which each number is the sum of the two previous ones (starting with 0 and 1). Mathematically, we can write it as:

- F(0) = 0
- F(1) = 1
- F(n) = F(n-1) + F(n-2) for n larger than 1.

Your task is to compute the **n**th number in the sequence, where **n** is an input to the function.

Use the code skeletons below to **provide two solutions** to the problem: an **iterative**, and a **recursive** one. For the recursive solution, feel free to use a helper function or add keyword arguments to the given function.

Explain how your solutions follow the iterative and recursive paradigms, respectively. **Discuss the pros and cons** of each approach as applied to this problem, and state which of your solutions you think is better, and why. Write approximately 100 words.

In [3]:
```
def fibonacci_iterative(n):
    a, b = 0, 1 #defining place holders for last and seccond last element
    for i in range(n): a, b = b, a+b;  #second last element = last element and current element = last + second last ele
    return a

def fibonacci_recursive(n):
    if n == 1 or n == 0: return n;  #base cases n=1 -> 1; n=0 -> 0
    return fibonacci_recursive(n-1)+fibonacci_recursive(n-2) #sum last + second last elements
```

In [4]:
```
assert fibonacci_iterative(0) == 0
assert fibonacci_recursive(0) == 0
assert fibonacci_iterative(4) == 3
assert fibonacci_recursive(4) == 3
```

The iterative approach works by simply iterating from 0 to n (positions in the fibonacci sequence) and stores the values of the last 2 elements, while calculating the next element, by simply adding the last 2. This function has a time complexity of $O(n)$, since it simply iterates from 0 to n.
The recusive approach works by running itself again times to get the last 2 elements and add them to get the result. The recursive relation for this approach is $T(n) = T(n-1) + T(n-2) + \Theta(1)$, solving this recurrence relation, we have that the time complexity for this approach is $O(2^n)$.
As we can see here, the iterative approach is far more eficient then the recursive approach.

Question 2: Understanding and documenting code [#CodeReadability, #SortingAlgorithms]

Imagine that you land your dream software engineering job, and among the first things you encounter is a previously written, poorly documented, and commented code.

Asking others how it works proved fruitless, as the original developer left. You are left with no choice but to understand the code's inner mechanisms, and document it properly for both yourself and others. The previous developer also left behind several tests that show the code working correctly, but you have a hunch that there might be some problems there, too. Your tasks are listed below. Here is the code:

In [5]:
```
def my_sort(array):
    """
    YOUR DOCSTRING HERE
    """

    # ...
    for i in range(len(array)):

        # ...
        item = array[i]

        # ...
        intended_position = 0
        for num in array:
            if num < item:
                intended_position += 1

        # ...
        if i == intended_position:
            continue

        # ...
        array[intended_position], item = item, array[intended_position]

        # ...
        while i != intended_position:

            # ...
            intended_position = 0
            for num in array:
                if num < item:
                    intended_position += 1

            # ...
            array[intended_position], item = item, array[intended_position]

    return array



assert my_sort([8, 5, 7]) == [5, 7, 8]
assert my_sort([10, 9, 8, 7, 6, 5, 4, 3, 2, 1]) == [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
input_array = [43, 99, 85, 45, 21, 58, 24, 12, 14, 64, 19, 94, 56, 13, 51, 2, 37, 11, 8, 66, 3, 95, 93, 53, 35, 81, 97,
assert my_sort(input_array) == sorted(input_array)
```

Question 2a: Understanding code

Explain, in your own words, **what the code is doing** (it's sorting an array, yes, but how?). Feel free to use diagrams, play around with the code in other cells, print test cases or partially sorted arrays, draw step by step images. In the end, you should produce an approximately 150-word write-up of how the code is processing the input array.

This code sorts the array by getting every element of it and comparing it to every other element in order to find its position in the sorted list. It starts by getting the first element and comparing it to everyone. For every element bigger than it, its position in the sorted list (given by the variable "intended_position") is increased by 1. After it is done comparing, the code check if the element is already in the right place, and if it is not, it puts the element in the right place in the array while getting the previous element that was there to be the next one to be analyzed. It then repeats the process with this element, until it finds an element in place. After that, the code repeats the whole process with the second, third, fourth... element of the array, until it does it with all of them.

Question 2b: Documenting code

Explain the difference between docstrings and comments. **Add** both a proper **docstring** and several **in-line comments** to the code (there is an editable copy below). You can follow the empty comments to guide you, but you can deviate, within reason. If this topic seems new to you, here (https://realpython.com/documenting-python-code/), here (https://www.datacamp.com/community/tutorials/docstrings-python) and here (https://sphinxcontrib-napoleon.readthedocs.io/en/latest/example_numpy.html) are some resources to get you started. Anyone from your section should be able to understand the code from your documentation. Remember, however, that brevity is also a desirable feature.

A docstring is general documentation for the code, it explains very briefly what the code does and provides details regarding the input parameters and outputs of the code. A comment is a more specific description of what that line or that section of the code is doing.

In [6]:
```
def my_sort(array):
    """
    Sorts array.


    Parameters
    ----------
    array : Python list or numpy array


    Returns
    -------
    array: a sorted Python list
    """

    # iterate through the whoule array from the first index to last index
    for i in range(len(array)):

        # defines the item to be analyzed to be equal to the element of index i in the array
        item = array[i]

        # find the position of this item by comparing it to every element in the array
        intended_position = 0
        for num in array:
            if num < item:
                intended_position += 1

        # if the item is already in place, skip the rest of this iteration of the loop
        if i == intended_position:
            continue

        # place the item in the position and get the previous element in this position to be the next item
        array[intended_position], item = item, array[intended_position]

        # skips the inside of the loop once the item is in the right position
        while i != intended_position:

            # find the position of this item by comparing it to every element in the array
            intended_position = 0
            for num in array:
                if num < item:
                    intended_position += 1

            # place the item in the position and get the previous element in this position to be the next item
            array[intended_position], item = item, array[intended_position]


    return array

#test case where the code did not pass. Left it commented, because it runs forever
#input_array = [3,2,3,1]
#assert my_sort(input_array) == sorted(input_array)
```

Question 2c: Testing code

Why are the tests that you are presented with insufficient? **Provide at least one** reasonable test case that you would include that the code doesn't pass (but you think it should).
You can also fix the code to pass your new tests, but this is an optional challenge.

The code works for all tests presented, however they are not sufficient, since they do not represent all possible types of lists that could be given as inputs. As seen in the test case I put in the code cell above, the code does not work when there are repeated numbers in the array.

Question 3: New and mixed sorting approaches [#SortingAlgorithms, #ComputationalCritique, #PythonProgramming, #ComplexityAnalysis]

In this question, you will implement and critique a previously unseen sorting algorithm. You will then combine it with another, known sorting algorithm, to see whether you can reach better behavior. This is the most difficult of the assignment questions, so schedule enough time for it.

Question 3a: Implementation from pseudocode

Use the following pseudocode to **implement merge_3_sort()**. It is similar to merge sort — only that instead of splitting the initial array into halves, you will now split it into thirds, call the function recursively on each sublist, and then merge the triplets together. You might want to refer to this beautiful resource (https://drive.google.com/file/d/1XH5bNiHhVchVWoCDLBtvjsCFkL9cU0do/view?pli=1) written by Prof. Drummond for details about the regular merge sort algorithm.

Code in the second cell; the first cell is locked so that you always have access to the original pseudocode.

In [7]:

```
def merge_3_sort(array, p, q):
    """
    Sorts array[p] to array[q] in place.


    Parameters
    ----------
    array : Python list or numpy array
    p : int
        index of array element to start sorting from
    q : int
        index of last array element to be sorted

    Returns
    -------
    array: a sorted Python list

    """
    # check if more than one elements remain between array[p] and array[q]
        # otherwise return the single element remaining

    # calculate where one-third and two-thirds of the input array are
    # recursively call merge_3_sort() on each of the three sublists

    # call merge_3() on the three sorted sublists to combine them
        # this can be either done as a return or in-place

    # return the sorted array
```

```
def merge_3(array, p, q, r, s):
    """
    Merges 3 sorted sublists (array[p] to array[q], array[q+1] to array[r] and array[r+1] to array[s]) in place.

    Parameters
    ----------
    array : Python list or numpy array
    p : int
        index of first element of first sublist
    q : int
        index of last element of first sublist
    r : int
        index of last element of second sublist
    s : int
        index of last element of third sublist

    """

    # copy each sublist to a placeholder
    # append infinity to the end of each placeholder list

    # fill each position in array[p] to array[q]
        # use the smallest of the front elements of the placeholder lists
```

In [8]:
```
def merge_3_sort(array, p, q):
    """
    Sorts array[p] to array[q] (inclusive) in place.


    Parameters
    ----------
    array : Python list or numpy array
    p : int
        index of array element to start sorting from
    q : int
        index of last array element to be sorted

    Returns
    -------
    array: a sorted Python list

    """
    if p < q:  # check if more than one elements remain between array[p] and array[q]

        # calculate where one-third and two-thirds of the input array are
        first = p + ((q-p)//3)
        second = p + 2*((q-p)//3) + 1

        # recursively call merge_3_sort() on each of the three sublists
        merge_3_sort(array, p, first)
        merge_3_sort(array, first+1, second)
        merge_3_sort(array, second+1, q)

        # call merge_3() on the three sorted sublists to combine them
        merge_3(array, p, first, second, q)

    # return the sorted array
    return array


def merge_3(array, p, q, r, s):
    """
    Merges 3 sorted sublists (array[p] to array[q], array[q+1] to array[r] and array[r+1] to array[s]) in place.
    """

    # copy each sublist to a placeholder
    L = array[p:q+1]
    M = array[q+1:r+1]
    R = array[r+1:s+1]

    # append infinity to the end of each placeholder list
    L.append (float('inf'))
    M.append (float('inf'))
    R.append (float('inf'))

    # fill each position in array[p] to array[q]
    # use the smallest of the front elements of the placeholder lists
    l = 0
    m = 0
    r = 0
    for i in range(p, s+1):
        if L[l] <= R[r] and L[l] <= M[m]:
            array[i] = L[l]
            l += 1
        elif R[r] <= L[l] and R[r] <= M[m]:
            array[i] = R[r]
            r += 1
        else:
            array[i] = M[m]
            m += 1

    return array
```

Question 3b: Testing your code

To the singular test provided below, **add** at least **5 more assert statements**, which showcase that your code works as intended. In a few sentences, **justify** why your set of tests is appropriate and possibly sufficient.

In [9]:
```
input_array = [8, 5, 4, 6, 7, 2, 9, 1, 3]
assert merge_3_sort(input_array, 0, 8) == sorted(input_array)

input_array = [9,8,7,6,5,4,3,2,1]
assert merge_3_sort(input_array, 0, 8) == sorted(input_array)

input_array = [1,2,3,4,5,6,7,8,9]
assert merge_3_sort(input_array, 0, 8) == sorted(input_array)

input_array = [1,2]
assert merge_3_sort(input_array, 0, 1) == sorted(input_array)

input_array = [2,1]
assert merge_3_sort(input_array, 0, 1) == sorted(input_array)

input_array = [1]
assert merge_3_sort(input_array, 0, 0) == sorted(input_array)

input_array = []
assert merge_3_sort(input_array, 0, 0) == sorted(input_array)

input_array = [1,1,1,1,1,1]
assert merge_3_sort(input_array, 0, 5) == sorted(input_array)

input_array = [1,2,3,3,2,1]
assert merge_3_sort(input_array, 0, 5) == sorted(input_array)
```

In [ ]:
```
```

I think these tests are sufficient, but I can't be 100% sure. I tried to get all the possible types of inputs: large random arrays; reverse sorted arrays; sorted arrays; size 2 sorted arrays; size 2 unsorted arrays; size 1 arrays; empty arrays; all equal elements arrays; repeated numbers array;
It passed all the tests I tried, however, it is impossible to try every single case. So, I would say it is suficient for now, until I find a different case to test it.

Question 3c: Mixing algorithms

The algorithm becomes unnecessarily complicated when it tries to sort a really short piece of the original array, continuing the splits into single-element arrays. To work around this, **implement the following**: if the input sublist length is below a certain threshold (which you decide), sort it by bubble sort instead of continuing to recurse. To ensure you won't break your old code, first copy it to the cell below, and then create the new version here.

Justify on the basis of analytical or even experimental arguments what might be the optimal threshold for switching to bubble sort.

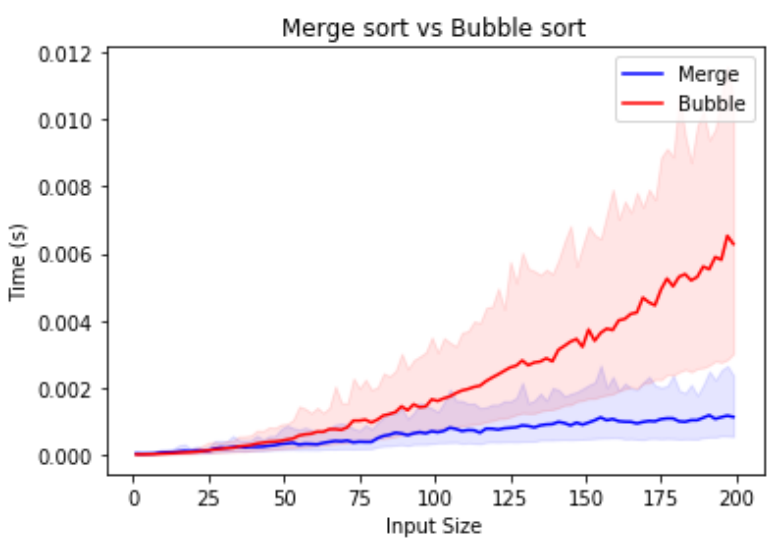
Remember that **tests** are important.

In [10]:
```
def bubble_sort(A):
    for i in range(0, len(A)):
        for j in range(len(A)-1, i, -1):
            if A[j] < A[j-1]:
                A[j], A[j-1] = A[j-1], A[j]
    return A
```

In [11]:
```
import time
import matplotlib.pyplot as plt
import random
import numpy as np

timess = []  #list of lists of times for merge
timessb = []  #list of lists of times for bubble

for j in range(100):   #repeat 100 times
    times = []   #list of times for merge
    timesb = []   #list of times for bubble

    for i in range(1,101):
        l = random.sample(list(range(10000)), 2*i)  #random list of size 2*i
        s = time.perf_counter()   #start timer
        merge_3_sort(l, 0, len(l)-1)
        e = time.perf_counter()  #end timer
        times.append(e-s)

    for i in range(1,101):
        l = random.sample(list(range(10000)), 2*i)  #random list of size 2*i
        s = time.perf_counter()   #start timer
        bubble_sort(l)
        e = time.perf_counter()   #end timer
        timesb.append(e-s)

    timess.append(times)
    timessb.append(timesb)

#make it into an np array
timess = np.array(timess)
timessb = np.array(timessb)

#take the average of the times for each size
averageM = np.average(timess, axis = 0)
averageB = np.average(timessb, axis = 0)

#lower and uper-bound using confidence intervals of 95%
lowM = np.quantile(timess, 0.025, axis=0)
upM = np.quantile(timess, 0.975, axis=0)

lowB = np.quantile(timessb, 0.025, axis=0)
upB = np.quantile(timessb, 0.975, axis=0)
```

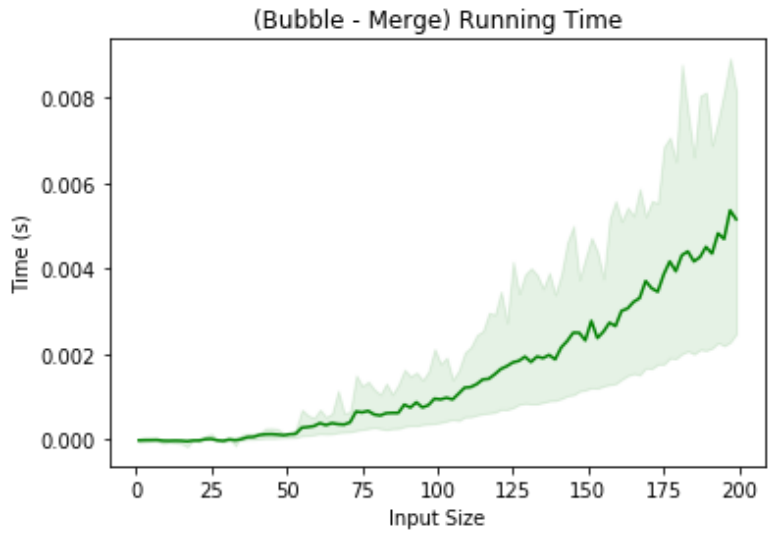
In [30]:
```
plt.plot(list(range(1,201,2)),averageM, c='blue')
plt.plot(list(range(1,201,2)),averageB, c='red')

plt.fill_between(list(range(1,201,2)), lowM, upM, color='blue', alpha = 0.1)
plt.fill_between(list(range(1,201,2)), lowB, upB, color='red', alpha = 0.1)

plt.legend(['Merge','Bubble'])
plt.xlabel('Input Size')
plt.ylabel('Time (s)')
plt.title('Merge sort vs Bubble sort')
```

Out[30]:
```
Text(0.5, 1.0, 'Merge sort vs Bubble sort')
```



In [31]:
```
#get the diference between the times
diference = averageB - averageM
dUp = upB - upM
dLow = lowB - lowM

plt.plot(list(range(1,201,2)), diference, c='green')
plt.fill_between(list(range(1,201,2)), dLow, dUp, color='green', alpha = 0.1)

plt.xlabel('Input Size')
plt.ylabel('Time (s)')
plt.title('(Bubble - Merge) Running Time')
```

Out[31]: Text(0.5, 1.0, '(Bubble - Merge) Running Time')



In [34]:
```
threshold = 40

def merge_3_sort_threshold(array, p, q):
    """
    Sorts array[p] to array[q] (inclusive) in place.


    Parameters
    ----------
    array : Python list or numpy array
    p : int
        index of array element to start sorting from
    q : int
        index of last array element to be sorted

    Returns
    -------
    array: a sorted Python list

    """
    if q-p > threshold: # check if the interval is bigger than the threshold

        # calculate where one-third and two-thirds of the input array are
        first = p + ((q-p)//3)
        second = p + 2*((q-p)//3) + 1

        # recursively call merge_3_sort_threshold() on each of the three sublists
        merge_3_sort_threshold(array, p, first)
        merge_3_sort_threshold(array, first+1, second)
        merge_3_sort_threshold(array, second+1, q)

        # return merge_3() on the three sorted sublists to combine them
        return merge_3(array, p, first, second, q)

    else:
        #call bubble sort
        array[p:q+1] = bubble_sort(array[p:q+1])
        return array
```

In [ ]:

Based on the tests did on the cells above, until n is arround 40, bubble sort is slightly faster than the merge sort. Therefore, I decided to put my threshold as 40. I'll evaluate further in the next question the result of this.

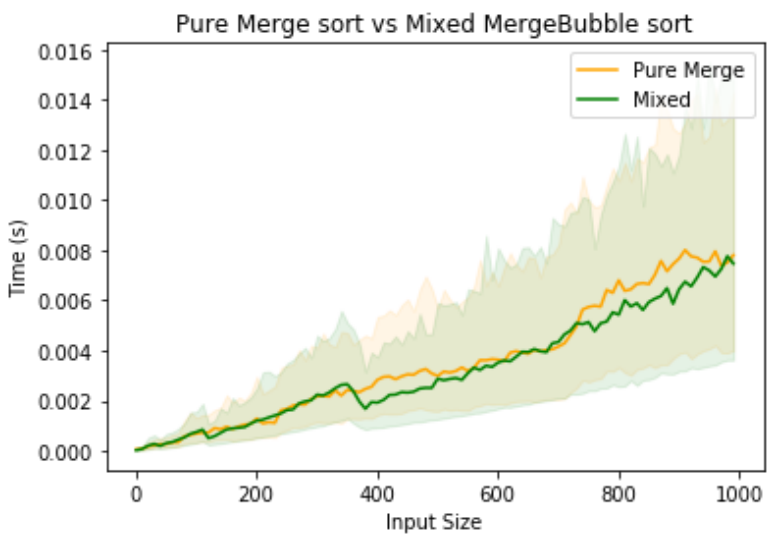
Question 3d: Algorithmic comparison

Finally, **assess** taking the mixed approach versus a strictly recursive algorithm. Make this comparison as complete as possible. This should include both an analytical BigO complexity definition, as well as graphical experimental evidence. Finally, you need to include a write-up summarizing the discovered information, comparing and contrasting the two algorithms in terms of any metrics you deem important (up to 200 words).

In [26]:
```
#list of lists of times
timesM = []
timesT = []

for j in range(100): #repeat 100 times
    #list of times
    times = []
    timesb = []

    for i in range(1,101):
        l = random.sample(list(range(10000)), 10*i)  #random list of size 10*i
        s = time.perf_counter()  #start timer
        merge_3_sort(l, 0, len(l)-1)
        e = time.perf_counter()  #end timer
        times.append(e-s)

    for i in range(1,101):
        l = random.sample(list(range(10000)), 10*i)  #random list of size 10*i
        s = time.perf_counter()  #start timer
        merge_3_sort_threshold(l, 0, len(l)-1)
        e = time.perf_counter()  #end timer
        timesb.append(e-s)

    timesM.append(times)
    timesT.append(timesb)

#make it into an np array
timesM = np.array(timesM)
timesT = np.array(timesT)

#take the average of the times for each size
averageMe = np.average(timesM, axis = 0)
averageT = np.average(timesT, axis = 0)

#lower and uper-bound using confidence intervals of 95%
lowMe = np.quantile(timesM, 0.025, axis=0)
upMe = np.quantile(timesM, 0.975, axis=0)

lowT = np.quantile(timesT, 0.025, axis=0)
upT = np.quantile(timesT, 0.975, axis=0)
```

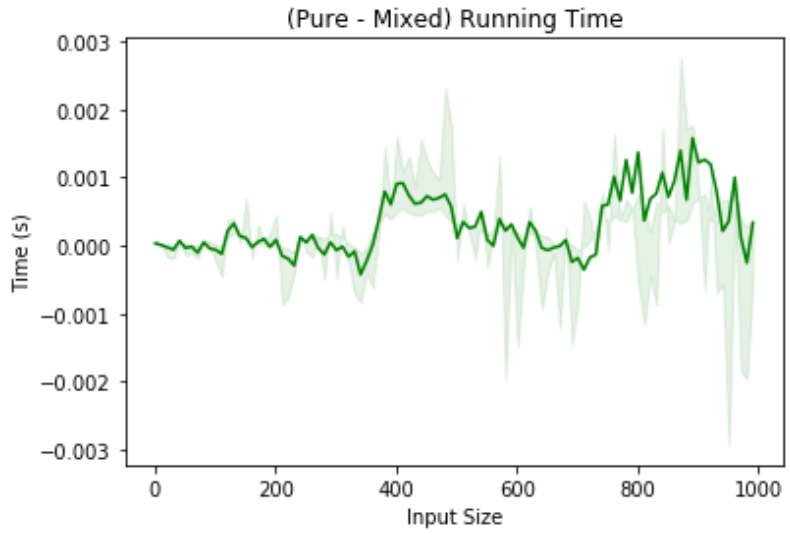
In [32]:
```
plt.plot(list(range(1,1001,10)),averageMe, c='orange')
plt.plot(list(range(1,1001,10)),averageT, c='green')

plt.fill_between(list(range(1,1001,10)), lowMe, upMe, color='orange', alpha = 0.1)
plt.fill_between(list(range(1,1001,10)), lowT, upT, color='green', alpha = 0.1)

plt.legend(['Pure Merge','Mixed'])
plt.xlabel('Input Size')
plt.ylabel('Time (s)')
plt.title('Pure Merge sort vs Mixed MergeBubble sort')
```

Out[32]:
```
Text(0.5, 1.0, 'Pure Merge sort vs Mixed MergeBubble sort')
```



In [33]:
```
#get the diference between the times
diference = averageMe - averageT
dUp = upMe - upT
dLow = lowMe - lowT

plt.plot(list(range(1,1001,10)), diference, c='green')
plt.fill_between(list(range(1,1001,10)), dLow, dUp, color='green', alpha = 0.1)

plt.xlabel('Input Size')
plt.ylabel('Time (s)')
plt.title('(Pure - Mixed) Running Time')
```

Out[33]: Text(0.5, 1.0, '(Pure - Mixed) Running Time')



Using the experimental evidence above, we can see that the average case of the mixed approach starts to get slghtly faster when n starts getting bigger than 400. However, by looking at the confidence interval, we can also see that the diference between both approaches is almost non-existent.
The recurrence relation for the merge_3_sort is $T(n) = 3T(\frac{n}{3}) + O(n)$, solving this relation, we get $O(nlog_3 n)$.
For the mixed approach, we combined it with the bubble sort, which has a time complexity of $O(n^2)$. While the 3 way merge sort is not very eficient for small inputs, the bubble sort is not a lot better. This, addind to the fact that we are working with small input sizes as threshold (small running time and small running time difference), makes the difference of the running times of the purelly recursive and mixed approach not different enough for the mixed approach to be worth using.

HCs:

1- #algorithms - Throughout the whole assingment I used this HC to either create my codes, analyze other codes or understand how a code worked.

2- #confidenceintervals - Used this HC on questions 3c and 3d to get a better understanding and insights of the tests for running time of the different algorithms.

3- #dataviz - Used this HC on questions 3c and 3d to get a better vizualization of the tests for running time of the different algorithms.

In [ ]:
```
```