Objectives

* Sorting Algorithm Performance

Due Date

This assignment is due on Friday, March 8, 2024, by 11:00 pm.

**Remarks:**

* It is imperative you develop test cases for your code.
* When you are asked to hand in code, you cut-and-paste the definition(s) from your code and paste into this Word document immediately following the activity.
* This homework is to be individual work.

Value

This assignment is worth 20 points.

Activities

1. Implement the bubblesort (initially without any invariants), selectionsort, insertionsort, mergesort, and quicksort from the text in Chapters 12 and 13 in a file HW7.py.
2. Write unittests for each with a different class for each function in TestHW7.py. Within each class, conduct unit tests that include:
   * A list that is already sorted.
   * A list that is in reverse sorted order
   * A list that is randomly sorted
   * A variety of lengths of lists (including zero elements, a single element, two elements and then random lengths
   * Be sure to include lists with and without repeating elements
3. Modify each of the functions to include the ability to count how many instructions are being executed within each. Establish a *global* variable instructions = 0 and then increment the number of instructions for each instruction that is carried out with each of the functions, resetting instructions to zero before running each sorting algorithm (you can save some time and increment instructions with a value besides one for a block of code that is executed). Run a variety of tests to determine which sorting algorithm executes the fewest number of instructions for the same list being sorted. Paste your five functions (and other supporting functions) below.
4. def bubblesort(L):  
    global instructions  
    for iteration in range(len(L) - 1):  
    for i in range(len(L) - 1 - iteration):  
    instructions += 1 # Increment for comparison  
    if L[i] > L[i + 1]:  
    L[i], L[i + 1] = L[i + 1], L[i]  
     
   def selectionsort(L):  
    global instructions  
    n = len(L)  
    for i in range(n - 1):  
    min\_index = i  
    for j in range(i + 1, n):  
    instructions += 1 # Increment for comparison  
    if L[j] < L[min\_index]:  
    min\_index = j  
    L[i], L[min\_index] = L[min\_index], L[i]  
     
   def insertionsort(L):  
    global instructions  
    for i in range(1, len(L)):  
    key = L[i]  
    j = i - 1  
    while j >= 0:  
    instructions += 1 # Increment for comparison  
    if L[j] > key:  
    L[j + 1] = L[j]  
    j -= 1  
    else:  
    break  
    L[j + 1] = key  
     
     
   def merge(A, B):  
    global instructions  
    result = []  
    i = j = 0  
    while i < len(A) and j < len(B):  
    instructions += 1 # Increment for comparison  
    if A[i] < B[j]:  
    result.append(A[i])  
    i += 1  
    else:  
    result.append(B[j])  
    j += 1  
    # Extend without incrementing instructions, assuming extend isn't a set of instructions  
    result.extend(A[i:])  
    result.extend(B[j:])  
    return result  
     
     
   def mergesort(L):  
    if len(L) > 1:  
    mid = len(L) // 2  
    A, B = L[:mid], L[mid:]  
    mergesort(A)  
    mergesort(B)  
    L[:] = merge(A, B)  
     
   def partition(L, low, high):  
    global instructions  
    pivot = L[high]  
    i = low - 1  
    for j in range(low, high):  
    instructions += 1 # Increment for comparison  
    if L[j] <= pivot:  
    i += 1  
    L[i], L[j] = L[j], L[i]  
    L[i + 1], L[high] = L[high], L[i + 1]  
    return i + 1  
     
     
     
   def quicksort(L, low, high):  
    if low < high:  
    pi = partition(L, low, high)  
    quicksort(L, low, pi-1)  
    quicksort(L, pi+1, high)
5. Bubblesort works well with large numbers at the beginning (they bubble right up to the end of the list). Bubblesort does not work well with small numbers at the end (it takes multiple iterations to get the smaller item to be moved up to the front). Because you have not implemented any invariants in bubblesort (yet), there should be no difference in the number of instructions for a variety of list structures (ordered increasing, decreasing, random, etc.). Determine some test cases to verify this operation of bubblesort. Provide a listing of your test cases and the number of instructions needed below. Can you show that without invariants, the performance is the same regardless of the structure of the list?

if \_\_name\_\_ == "\_\_main\_\_":  
 # Test cases  
 test\_cases = [  
 ([], []),  
 ([1], [1]),  
 ([4, 2], [2, 4]),  
 ([1, 2, 3, 4, 5], [1, 2, 3, 4, 5]),  
 ([5, 4, 3, 2, 1], [1, 2, 3, 4, 5]),  
 (random.sample(range(10), 10), sorted(random.sample(range(10), 10))),  
 ([1, 3, 2, 5, 4], [1, 2, 3, 4, 5]),  
 ([5, 3, 1, 2, 3, 1], [1, 1, 2, 3, 3, 5]),  
 ]  
  
 for arr, description in test\_cases:  
 instructions = 0 # Reset the instruction count  
 bubblesort(arr)  
 print(f"{description} list instructions: {instructions}")

"C:\Users\artjs\OneDrive - University of Connecticut\Documents\Spring 24\CSE 2050\cse2050\hw7\.venv\Scripts\python.exe" "C:\Users\artjs\OneDrive - University of Connecticut\Documents\Spring 24\CSE 2050\cse2050\hw7\HW7.py"

[] list instructions: 0

[1] list instructions: 0

[2, 4] list instructions: 1

[1, 2, 3, 4, 5] list instructions: 10

[1, 2, 3, 4, 5] list instructions: 10

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9] list instructions: 45

[1, 2, 3, 4, 5] list instructions: 10

[1, 1, 2, 3, 3, 5] list instructions: 15

Process finished with exit code 0

1. Now, implement the invariants in bubblesort as outlined in the text and provided on page 112. Repeat the tests from above. Can you confirm now that the number of instructions does change based on the initial structure of the list? Provide your code, test cases, and results below.
2. def bubblesort(L):  
    global instructions  
    n = len(L)  
    keepgoing = True  
    while keepgoing:  
    keepgoing = False  
    for i in range(n-1):  
    instructions += 1 # Increment for comparison  
    if L[i] > L[i+1]:  
    L[i], L[i+1] = L[i+1], L[i]  
    instructions += 3 # Increment for swap  
    keepgoing = True  
    n -= 1
3. if \_\_name\_\_ == "\_\_main\_\_":  
    # Test cases  
    test\_cases = [  
    ([], []),  
    ([1], [1]),  
    ([4, 2], [2, 4]),  
    ([1, 2, 3, 4, 5], [1, 2, 3, 4, 5]),  
    ([5, 4, 3, 2, 1], [1, 2, 3, 4, 5]),  
    (random.sample(range(10), 10), sorted(random.sample(range(10), 10))),  
    ([1, 3, 2, 5, 4], [1, 2, 3, 4, 5]),  
    ([5, 3, 1, 2, 3, 1], [1, 1, 2, 3, 3, 5]),  
    ]  
     
    for arr, description in test\_cases:  
    instructions = 0 # Reset the instruction count  
    bubblesort(arr)  
    print(f"{description} list instructions: {instructions}")
4. [] list instructions: 0
5. [1] list instructions: 0
6. [2, 4] list instructions: 4
7. [1, 2, 3, 4, 5] list instructions: 4
8. [1, 2, 3, 4, 5] list instructions: 40
9. [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] list instructions: 135
10. [1, 2, 3, 4, 5] list instructions: 13
11. [1, 1, 2, 3, 3, 5] list instructions: 45
12. Modify the bubblesort program with invariants so that it is optimized for both large numbers at the beginning and small numbers at the end. The function is called cocktailsort. It is the bubblesort algorithm but alternates the direction of the bubble – moving larger numbers to the end of the list in one direction and then smaller numbers to the beginning of the list. Can you confirm now that the number of instructions improves for the same set of test cases? Provide your code, test cases, and results below.

def cocktailsort(L):  
 global instructions  
 n = len(L)  
 swapped = True  
 start = 0  
 end = n - 1  
 while swapped:  
 swapped = False  
 # Move larger elements to the end  
 for i in range(start, end):  
 instructions += 1 # Increment for comparison  
 if L[i] > L[i + 1]:  
 L[i], L[i + 1] = L[i + 1], L[i]  
 instructions += 3 # Increment for swap  
 swapped = True  
 if not swapped:  
 break  
 swapped = False  
 end -= 1  
 for i in range(end - 1, start - 1, -1):  
 instructions += 1 # Increment for comparison  
 if L[i] > L[i + 1]:  
 L[i], L[i + 1] = L[i + 1], L[i]  
 instructions += 3 # Increment for swap  
 swapped = True  
 start += 1

if \_\_name\_\_ == "\_\_main\_\_":  
 # Test cases  
 test\_cases = [  
 ([], []),  
 ([1], [1]),  
 ([4, 2], [2, 4]),  
 ([1, 2, 3, 4, 5], [1, 2, 3, 4, 5]),  
 ([5, 4, 3, 2, 1], [1, 2, 3, 4, 5]),  
 (random.sample(range(10), 10), sorted(random.sample(range(10), 10))),  
 ([1, 3, 2, 5, 4], [1, 2, 3, 4, 5]),  
 ([5, 3, 1, 2, 3, 1], [1, 1, 2, 3, 3, 5]),  
 ]  
  
 for arr, description in test\_cases:  
 instructions = 0 # Reset the instruction count  
 # bubblesort(arr)  
 # print(f"{description} list instructions: {instructions}")  
 cocktailsort(arr)  
 print(f"{description} list instructions: {instructions}")

"C:\Users\artjs\OneDrive - University of Connecticut\Documents\Spring 24\CSE 2050\cse2050\hw7\.venv\Scripts\python.exe" "C:\Users\artjs\OneDrive - University of Connecticut\Documents\Spring 24\CSE 2050\cse2050\hw7\invariant.py"

1. [] list instructions: 0
2. [1] list instructions: 0
3. [2, 4] list instructions: 4
4. [1, 2, 3, 4, 5] list instructions: 4
5. [1, 2, 3, 4, 5] list instructions: 40
6. [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] list instructions: 95
7. [1, 2, 3, 4, 5] list instructions: 13
8. [1, 1, 2, 3, 3, 5] list instructions: 44
9. Process finished with exit code 0

import unittest  
import random  
from HW7 import bubblesort, selectionsort, insertionsort, mergesort, quicksort  
  
class TestSortingAlgorithms(unittest.TestCase):  
 global instructions  
  
 def setUp(self):  
 # Reset instructions count before each test  
 self.instructions = 0  
  
 def test\_bubblesort(self):  
 self.run\_sort\_tests(bubblesort)  
  
 def test\_selectionsort(self):  
 self.run\_sort\_tests(selectionsort)  
  
 def test\_insertionsort(self):  
 self.run\_sort\_tests(insertionsort)  
  
 def test\_mergesort(self):  
 self.run\_sort\_tests(mergesort)  
  
 def test\_quicksort(self):  
 # Test cases for quicksort  
 test\_cases = [  
 ([], []),  
 ([1], [1]),  
 ([4, 2], [2, 4]),  
 ([1, 2, 3, 4, 5], [1, 2, 3, 4, 5]),  
 ([5, 4, 3, 2, 1], [1, 2, 3, 4, 5]),  
 (random.sample(range(10), 10), sorted(random.sample(range(10), 10))),  
 ([1, 3, 2, 5, 4], [1, 2, 3, 4, 5]),  
 ([5, 3, 1, 2, 3, 1], [1, 1, 2, 3, 3, 5]),  
 ]  
 for arr, expected in test\_cases:  
 self.instructions = 0  
 quicksort(arr, 0, len(arr) - 1)  
 self.assertEqual(arr, expected)  
  
 def run\_sort\_tests(self, sort\_func):  
 test\_cases = [  
 ([], []),  
 ([1], [1]),  
 ([4, 2], [2, 4]),  
 ([1, 2, 3, 4, 5], [1, 2, 3, 4, 5]),  
 ([5, 4, 3, 2, 1], [1, 2, 3, 4, 5]),  
 (random.sample(range(10), 10), sorted(random.sample(range(10), 10))),  
 ([1, 3, 2, 5, 4], [1, 2, 3, 4, 5]),  
 ([5, 3, 1, 2, 3, 1], [1, 1, 2, 3, 3, 5]),  
 ]  
  
 for original, expected in test\_cases:  
 arr = original[:] # Make a copy of the array to sort  
 if sort\_func == quicksort:  
 sort\_func(arr, 0, len(arr) - 1)  
 else:  
 sort\_func(arr)  
 self.assertEqual(arr, expected)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 unittest.main()