Advanced sort algorithms implementation

Due May 6th, 11:59 pm

Task1(20pts) Merge Sort implementation

Based on the slides of Advanced sorting algorithm slides

* Write a Python script with the following functions:
  + def shell\_sort(a\_list):
  + def gap\_insertion\_sort(a\_list, start, gap):
* Run 3 test cases with random lists in size 20, 100 and 10000
* Ensure the program sorts three lists in ascending order

Task2(20pts) Quick sort Implementation

Based on the slides of Advanced sorting algorithm slides

* Write a Python script with the following three functions:
  + quick\_sort(a\_list)
  + quick\_sort\_helper(a\_list, first, last)
  + partition(a\_list, first, last)
  + call the functions properly

Task3(20pts) Merge Sort Implementation

Based on the slides of Advanced sorting algorithm slides

* Write a Python script with the following three functions:
  + merge\_sort(a\_list)
  + merge(left, right)
  + Call merge function in the merge\_sort

Task 4(20pts) Test() function

* create 3 random lists with size 20, 100 and 10000, value range from 1 to 100
* Use the above sorting functions to sort the 3 lists:
  + shell\_sort()
  + quick\_sort()
  + merge\_sort()
* output instruction:
  + Original Random List:
    - Display the randomly generated list before sorting.
  + Sort Function Used:
    - Clearly indicate which sorting algorithm is being tested
  + Ordered List:
    - Print the resulting list after sorting.
  + Running Time:
    - Print the time taken to sort the list, in seconds, with at least 6 decimal places of precision.

Sample output for size 20

Testing list size: 20

Original List: [34, 87, 12, 55, 76, 23, 45, 67, 89, 3, 90, 21, 8, 44, 66, 10, 5, 39, 71, 18]

Sorted List: [3, 5, 8, 10, 12, 18, 21, 23, 34, 39, 44, 45, 55, 66, 67, 71, 76, 87, 89, 90]

Sort Function: Shell Sort

Time Taken: 0.000024 seconds

Sort Function: Quick Sort

Time Taken: 0.000011 seconds

Sort Function: Merge Sort

Time Taken: 0.000015 seconds

Task 5(20pts) Analyze and Modify the Quicksort Algorithm

* Define a new partition function, partition\_median. The pivot is selected as the median of the first, middle, and last elements of the sublist.
* Define the quick\_sort\_median
* Use the new partition\_median function in the quick\_sort\_median implementation.

Answers

import random

import time

# Task 1: Shell Sort Implementation

def shell\_sort(a\_list):

sublist\_count = len(a\_list) // 2

while sublist\_count > 0:

for start\_position in range(sublist\_count):

gap\_insertion\_sort(a\_list, start\_position, sublist\_count)

sublist\_count = sublist\_count // 2

def gap\_insertion\_sort(a\_list, start, gap):

for i in range(start + gap, len(a\_list), gap):

current\_value = a\_list[i]

position = i

while position >= gap and a\_list[position - gap] > current\_value:

a\_list[position] = a\_list[position - gap]

position -= gap

a\_list[position] = current\_value

# Task 2: Quick Sort Implementation

def quick\_sort(a\_list):

quick\_sort\_helper(a\_list, 0, len(a\_list) - 1)

def quick\_sort\_helper(a\_list, first, last):

if first < last:

split\_point = partition(a\_list, first, last)

quick\_sort\_helper(a\_list, first, split\_point - 1)

quick\_sort\_helper(a\_list, split\_point + 1, last)

def partition(a\_list, first, last):

pivot\_value = a\_list[first]

left\_mark = first + 1

right\_mark = last

done = False

while not done:

while left\_mark <= right\_mark and a\_list[left\_mark] <= pivot\_value:

left\_mark += 1

while a\_list[right\_mark] >= pivot\_value and right\_mark >= left\_mark:

right\_mark -= 1

if right\_mark < left\_mark:

done = True

else:

a\_list[left\_mark], a\_list[right\_mark] = a\_list[right\_mark], a\_list[left\_mark]

a\_list[first], a\_list[right\_mark] = a\_list[right\_mark], a\_list[first]

return right\_mark

# Task 3: Merge Sort Implementation

def merge\_sort(a\_list):

if len(a\_list) <= 1:

return a\_list

mid = len(a\_list) // 2

left\_half = merge\_sort(a\_list[:mid])

right\_half = merge\_sort(a\_list[mid:])

return merge(left\_half, right\_half)

def merge(left, right):

result = []

i = j = 0

while i < len(left) and j < len(right):

if left[i] < right[j]:

result.append(left[i])

i += 1

else:

result.append(right[j])

j += 1

result.extend(left[i:])

result.extend(right[j:])

return result

# Task 5: Quicksort with Median-of-Three Pivot

def partition\_median(a\_list, first, last):

mid = (first + last) // 2

pivot\_candidates = [(a\_list[first], first), (a\_list[mid], mid), (a\_list[last], last)]

pivot\_candidates.sort()

pivot\_value, pivot\_index = pivot\_candidates[1]

a\_list[first], a\_list[pivot\_index] = a\_list[pivot\_index], a\_list[first]

return partition(a\_list, first, last)

def quick\_sort\_median(a\_list):

def helper(lst, first, last):

if first < last:

split\_point = partition\_median(lst, first, last)

helper(lst, first, split\_point - 1)

helper(lst, split\_point + 1, last)

helper(a\_list, 0, len(a\_list) - 1)

# Task 4: Testing Function

def Test():

sizes = [20, 100, 10000]

for size in sizes:

print(f"\nTesting list size: {size}\n")

rand\_list = [random.randint(1, 100) for \_ in range(size)]

print("Original List:", rand\_list if size == 20 else "[List too large to display]")

for sort\_name, sort\_func in [

("Shell Sort", shell\_sort),

("Quick Sort", quick\_sort),

("Merge Sort", lambda lst: merge\_sort(lst))

]:

list\_copy = rand\_list[:]

start\_time = time.time()

if sort\_name == "Merge Sort":

list\_copy = sort\_func(list\_copy)

else:

sort\_func(list\_copy)

end\_time = time.time()

print(f"\nSort Function: {sort\_name}")

if size == 20:

print("Sorted List:", list\_copy)

print("Time Taken: {:.6f} seconds".format(end\_time - start\_time))

Test()