Weekly Challenge 01: Comparison

CS/CE 412/471 Algorithms: Design and Analysis

Spring 2025

1. Sort on Top

We are going to compare the running times of 3 sorting algorithms: *insertion sort*, *merge sort*, and python's built-in `sorted`. For a given sequence, a, of size, n, we will note the time, t, it takes to sort a using each of the algorithms. We will obtain values of t for various values of n and plot them. We will then use the plot to estimate the size of n at which merge sort becomes faster than insertion sort.

Task

Write a program that

- iterates a variable, n, over the values $10, 20, 30, 40, \ldots, 500$,
- for each n, generates a random sequence (`list`), a, of n integers,
- measures the time taken to sort each a by each of the following algorithms: *insertion sort*, *merge sort*, and python's built-in `sorted`,
- plots the time taken by each algorithm for each of the values of n.

Use the plot to estimate, n_0, the value of n at which merge sort becomes faster than insertion sort.

Submission

You will submit an image file containing the plot and, overlaid on it, the string, $n_0 =?$, where "?" is replaced by your value of n_0. Also submit your `py` file.

Hint

- Use `step` with `range` to iterate over values of n.
- A random sequence of size n can be generated as

```
import random
a = list(range(n))
random.shuffle(a)
```

- Here is a collection of timing functions in python. You may use any one.
- Insertion- and merge- sort are standard sorting algorithms. You may use third-party implementations.

- Remember to make a copy of a for each sorting algorithm, so that each algorithm sorts the same sequence of values.
- You may use the `matplotlib` module for plotting.

```
import random
import time
import matplotlib.pyplot as plt
from scipy.interpolate import make_interp_spline
import numpy as np

# implementation of insertion sort
def insertion_sort(arr):
   n = len(arr)
   if n <= 1:
       return
   for i in range(1, n):
       key = arr[i]
       j = i-1
       while j >= 0 and key < arr[j]:
           arr[j+1] = arr[j]
           j -= 1
       arr[j+1] = key

# implementation of merge sort
def merge_sort(arr):
   if len(arr) > 1:
       mid = len(arr) // 2
       left_arr = arr[:mid]
       right_arr = arr[mid:]

       merge_sort(left_arr)
       merge_sort(right_arr)

       i = 0
       j = 0
       k = 0

       while i < len(left_arr) and j < len(right_arr):
           if left_arr[i] < right_arr[j]:
               arr[k] = left_arr[i]
               i += 1
           else:
               arr[k] = right_arr[j]
               j += 1
           k += 1

       while i < len(left_arr):
           arr[k] = left_arr[i]
           i += 1
```

```
            k += 1

        while j < len(right_arr):
            arr[k] = right_arr[j]
            j += 1
            k += 1

# implementation of measuring time
def measure_time(func, arr):
    start_time = time.perf_counter()
    func(arr)
    end_time = time.perf_counter()
    return end_time - start_time

# estimating n0 the very first occurence at n when the Merge Sort
becomes 'faster' than Insertion Sort which means the point where
Merge Sort time is less than that of Insertion Sort
def estimate_n0(n_values, merge_times, insertion_times):
    for i in range(len(n_values)):
        if merge_times[i] < insertion_times[i]:
            return n_values[i]
    return None

def main():
    n_values = range(10, 501, 10)
    # n_values = range(10, 1000, 10)

    insertion_times = []
    merge_times = []
    builtin_times = []

    for n in n_values:
        a = list(range(n))
        random.shuffle(a)

        # measuring time for insertion, merge and python's built-
in sorting algortihms
        insertion_times.append(measure_time(insertion_sort,
a.copy()))
        merge_times.append(measure_time(merge_sort, a.copy()))
        builtin_times.append(measure_time(sorted, a.copy()))

    #smoothing the curves using interpolation
    n_values_smooth = np.linspace(min(n_values), max(n_values),
500)
```

```
   insertion_smooth = make_interp_spline(n_values,
insertion_times)(n_values_smooth)
   merge_smooth = make_interp_spline(n_values, merge_times)
(n_values_smooth)
   builtin_smooth = make_interp_spline(n_values, builtin_times)
(n_values_smooth)

   plt.figure(figsize=(10, 6))
   plt.plot(n_values_smooth, insertion_smooth, label='Insertion
Sort', color='palevioletred')
   plt.plot(n_values_smooth, merge_smooth, label='Merge
Sort',color='cadetblue')
   plt.plot(n_values_smooth, builtin_smooth, label="Python's
Built-in Sort",color='darkslateblue')
   plt.xlabel('Size of List (n)')
   plt.ylabel('Time Taken (seconds)')
   plt.title('Comparison of Sorting Algorithms')
   plt.legend()
   plt.grid()

   n0 = estimate_n0(n_values, merge_times, insertion_times)

   if n0:
       plt.axvline(x=n0, color='r', linestyle='--', label=f"n =
{n0}", xy=(n0, merge_times[n_values.index(n0)]),
                   xytext=(n0 + 10,
merge_times[n_values.index(n0)] + 0.002),
                   arrowprops=dict(arrowstyle="->", color='r',
lw=1.5), fontsize=12, color='red')

   plt.savefig('WC01_bq08283.png')
   plt.show()

   print(f'Merge Sort faster than Insertion Sort at point n =
{n0}')

if __name__ == "__main__":
   main()

"""
References
https://www.geeksforgeeks.org/python-program-for-insertion-sort/
https://www.askpython.com/python/examples/merge-sort-in-python
https://builtin.com/articles/timing-functions-python
```

" " "