mini_project

March 23, 2024

0.1 Small project

This small project aims to compare the real-life behavior of sorting algorithms with theoretical expectations. This requires measurements: real-life, physical experiments on computer code and data of repeated tests run on sorted data. In your paper, you shall implement and benchmark the following sorting algorithms, using the algorithms presented in pseudocode in the course:

- Insertion sort
- Merge sort
- Quicksort

In your benchmarks, you shall use test data suitable to test the the behavior of the algorithms under the worst-case, best case and average-case scenarios. In order to study the scaling behavior of algorithms with problem size, one usually increases problem size by a factor, e.g., 2, 10 or 16 instead of increasing the problem size linearly. You can limit your largest problem size so that the full set of all benchmarks do not execute long on your computer. Drop test cases that take a too long time (e.g., large sizes for quadratic algorithms)

[ ]:
```python
# imports
import random
import time
from prettytable import PrettyTable
import matplotlib.pyplot as plt
import sys
```

0.1.1 Insertion sort

[ ]:
```python
def insertion_sort(A):
    for j in range(1, len(A)):
        key = A[j]
        i = j - 1
        while i >= 0 and A[i] > key:
            A[i + 1] = A[i]
            i = i - 1
        A[i + 1] = key
```

0.1.2 Merge sort

[ ]:
```
# This function sorts an array using the merge sort algorithm
def merge_sort(A, p, r):
    if p < r:
        q = (p + r) // 2         # Splits the array into two approximately equal halves
        merge_sort(A, p, q)      # Recursively sorts the left half
        merge_sort(A, q + 1, r) # Recursively sorts the right half
        merge(A, p, q, r)        # Merges the two sorted halves


# This function merges the two sorted arrays
# A is the array and p, q, r are indices where p <= q < r
def merge(A, p, q, r):

    # Initialises the variables to represent the two halves
    n1 = q - p + 1
    n2 = r - q

    left = [0] * (n1)
    right = [0] * (n2)

    # Copies the two halves into the left and right arrays
    for i in range(0, n1):
        left[i] = A[p + i]

    for j in range(0, n2):
        right[j] = A[q + 1 + j]

    i = j = 0
    k = p

    # Merges the two halves into the original array
    while i < n1 and j < n2:
        if left[i] <= right[j]:
            A[k] = left[i]
            i += 1
        else:
            A[k] = right[j]
            j += 1
        k += 1

    while i < n1:
        A[k] = left[i]
        i += 1
        k += 1
```

```
    while j < n2:
        A[k] = right[j]
        j += 1
        k += 1
```

0.1.3 Quick sort

[ ]:
```
def quicksort(A, p, r):
    if p < r:
        q = partition(A, p, r)
        quicksort(A, p, q - 1)
        quicksort(A, q + 1, r)

def partition(A, p, r):
    x = A[p]  # Pivot is the first element
    i = p     #

    for j in range(p + 1, r + 1):
        if A[j] <= x:
            i = i + 1
            A[i], A[j] = A[j], A[i]

    A[p], A[i] = A[i], A[p]
    return i
```

0.1.4 Benchmarking

[ ]:
```
sys.setrecursionlimit(70000)

def sorted_list(n):
    return list(range(n))

def reverse_sorted_list(n):
    return list(range(n, 0, -1))

def random_sorted_list(n):
    lst = list(range(n))
    random.shuffle(lst)
    return lst

# Function to measure the time of a sorting algorithm
def measure_sort_time(func, data):
    start_time = time.time()
    if func == insertion_sort:
        func(data)
    elif func == merge_sort:
```

```
        func(data, 0, len(data) - 1)
    elif func == quicksort:
        func(data, 0, len(data) - 1)
    else:
        ValueError("Invalid function")
    return time.time() - start_time

# The test data sizes
sizes = [2**2, 2**3, 2**4, 2**5, 2**6, 2**7, 2**8, 2**9, 2**10, 2**11, 2**12,␣
 ↪2**13, 2**14]

# Empty dictionaries to store the times
sorted_times = {"Insertion sort": [], "Merge sort": [], "Quicksort": []}
reverse_times = {"Insertion sort": [], "Merge sort": [], "Quicksort": []}
random_times = {"Insertion sort": [], "Merge sort": [], "Quicksort": []}

insertion_table = PrettyTable()
insertion_table.field_names = ["Size", "Sorted (sec)", "Reverse sorted (sec)",␣
 ↪"Randomly sorted (sec)"]

merge_table = PrettyTable()
merge_table.field_names = ["Size", "Sorted (sec)", "Reverse sorted (sec)",␣
 ↪"Randomly sorted (sec)"]

quicksort_table = PrettyTable()
quicksort_table.field_names = ["Size", "Sorted (sec)", "Reverse sorted (sec)",␣
 ↪"Randomly sorted (sec)"]

for size in sizes:
    # Insertion sort
    data_best = sorted_list(size)
    time_best = measure_sort_time(insertion_sort, data_best[:])
    sorted_times["Insertion sort"].append(time_best)

    data_worst = reverse_sorted_list(size)
    time_worst = measure_sort_time(insertion_sort, data_worst[:])
    reverse_times["Insertion sort"].append(time_worst)

    data_average = random_sorted_list(size)
    time_average = measure_sort_time(insertion_sort, data_average[:])
    random_times["Insertion sort"].append(time_average)

    insertion_table.add_row([size, f"{time_best:.8f}", f"{time_worst:.8f}",␣
 ↪f"{time_average:.8f}"])


    # Merge sort
```

```
    sorted = sorted_list(size)
    time_sorted = measure_sort_time(merge_sort, sorted[:])
    sorted_times["Merge sort"].append(time_sorted)

    reversed = reverse_sorted_list(size)
    time_reversed = measure_sort_time(merge_sort, reversed[:])
    reverse_times["Merge sort"].append(time_reversed)

    random_merge = random_sorted_list(size)
    time_random = measure_sort_time(merge_sort, random_merge[:])
    random_times["Merge sort"].append(time_random)

    merge_table.add_row([size, f"{time_sorted:.8f}", f"{time_reversed:.8f}",␣
↪f"{time_random:.8f}"])


    # Quicksort
    quick_sorted = sorted_list(size)
    time_quick_sorted = measure_sort_time(quicksort, quick_sorted[:])
    sorted_times["Quicksort"].append(time_quick_sorted)

    quick_reversed = reverse_sorted_list(size)
    time_quick_reversed = measure_sort_time(quicksort, quick_reversed[:])
    reverse_times["Quicksort"].append(time_quick_reversed)

    quick_random = random_sorted_list(size)
    time_quick_random = measure_sort_time(quicksort, quick_random[:])
    random_times["Quicksort"].append(time_quick_random)

    quicksort_table.add_row([size, f"{time_quick_sorted:.8f}",␣
↪f"{time_quick_reversed:.8f}", f"{time_quick_random:.8f}"])

print("Insertion sort:\n", insertion_table)
print("Merge sort:\n", merge_table)
print("Quicksort:\n", quicksort_table)
```

```
Insertion sort:
 +--------+--------------+----------------------+-----------------------+
|  Size | Sorted (sec) | Reverse sorted (sec) | Randomly sorted (sec) |
+--------+--------------+----------------------+-----------------------+
|   4   |  0.00000215  |      0.00000095      |       0.00000191      |
|   8   |  0.00000119  |      0.00000310      |       0.00000286      |
|   16  |  0.00000119  |      0.00001097      |       0.00000572      |
|   32  |  0.00000286  |      0.00004101      |       0.00002313      |
|   64  |  0.00000477  |      0.00014210      |       0.00007391      |
|  128  |  0.00001001  |      0.00056195      |       0.00031996      |
|  256  |  0.00001884  |      0.00251389      |       0.00138879      |
```

512	0.00004387	0.00976419	0.00484800
1024	0.00008607	0.03853178	0.01967406
2048	0.00018501	0.15388513	0.08011127
4096	0.00035977	0.62755895	0.31652975
8192	0.00076008	2.49529910	1.26163411
16384	0.00143600	10.11675215	5.12041402

Merge sort:

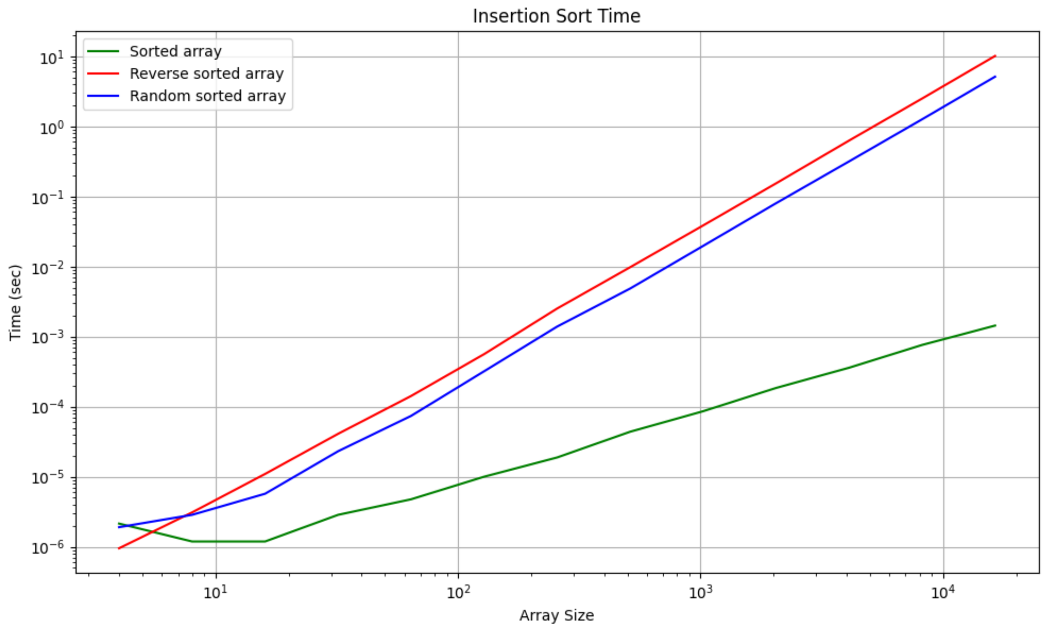
Size	Sorted (sec)	Reverse sorted (sec)	Randomly sorted (sec)
4	0.00000691	0.00000501	0.00000381
8	0.00001001	0.00000882	0.00000906
16	0.00001979	0.00001979	0.00002098
32	0.00004625	0.00003886	0.00004196
64	0.00008702	0.00008392	0.00009084
128	0.00018883	0.00018406	0.00020409
256	0.00046396	0.00043511	0.00047898
512	0.00088072	0.00086284	0.00094700
1024	0.00193000	0.00196600	0.00217700
2048	0.00404596	0.00400400	0.00455999
4096	0.00854516	0.00851393	0.00957179
8192	0.01809621	0.01827002	0.02043009
16384	0.03842092	0.03829908	0.04409504

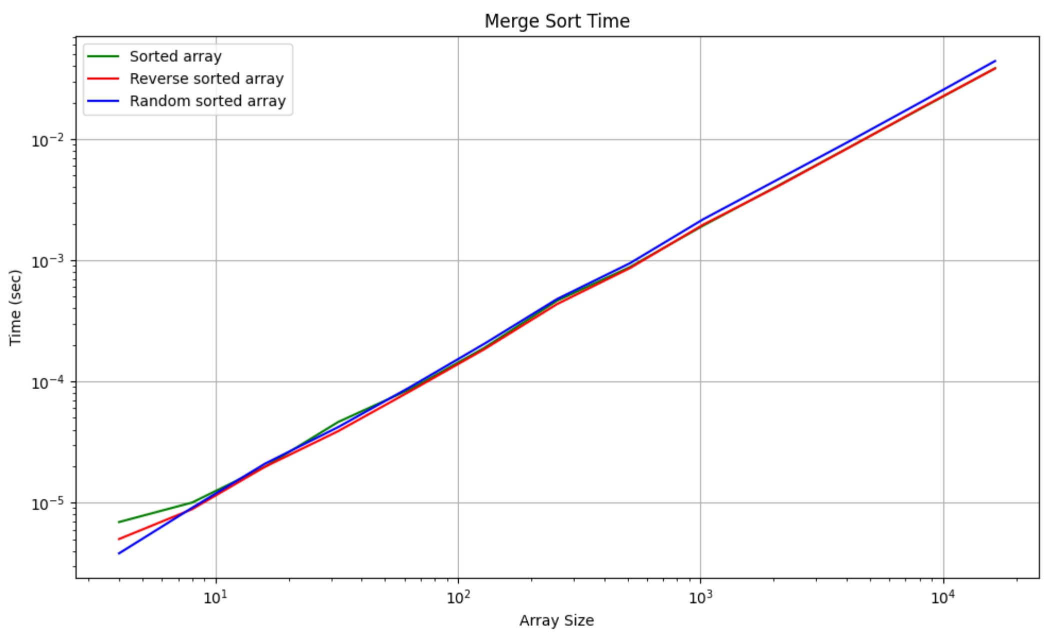
Quicksort:

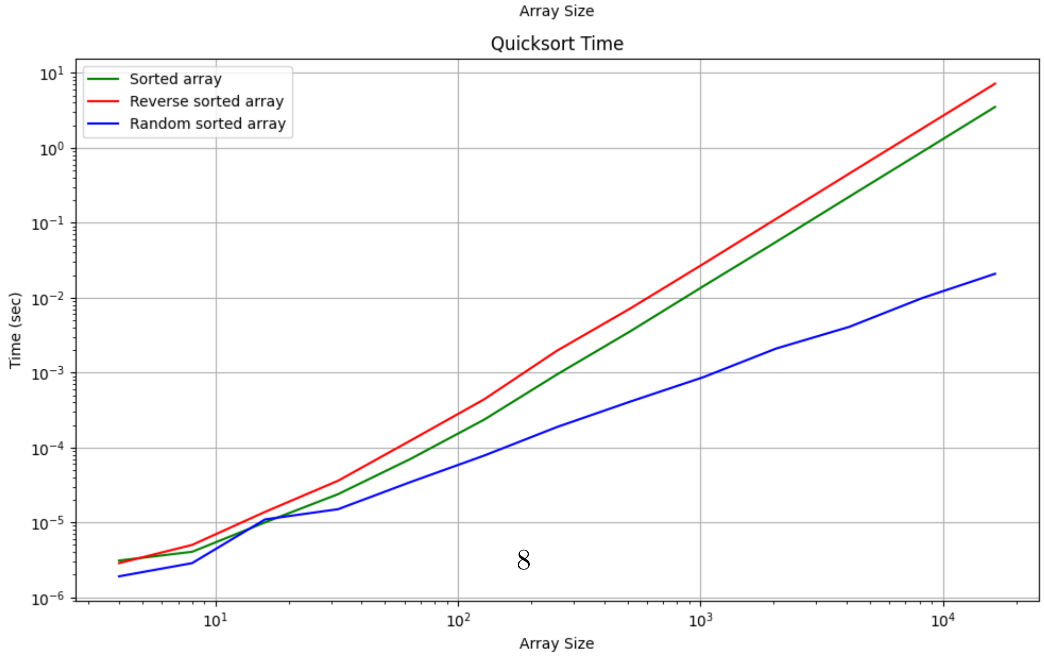
Size	Sorted (sec)	Reverse sorted (sec)	Randomly sorted (sec)
4	0.00000310	0.00000286	0.00000191
8	0.00000405	0.00000501	0.00000286
16	0.00001001	0.00001383	0.00001097
32	0.00002384	0.00003600	0.00001502
64	0.00007105	0.00012493	0.00003481
128	0.00023508	0.00043893	0.00007796
256	0.00094604	0.00196314	0.00018787
512	0.00354409	0.00714684	0.00040913
1024	0.01415491	0.02809906	0.00086594
2048	0.05551910	0.11278367	0.00209308
4096	0.22205997	0.45222807	0.00407100
8192	0.88166285	1.80404711	0.00986600
16384	3.51866508	7.19084287	0.02084494

0.1.5 Vizualization

[ ]:
```
plt.figure(figsize=(10, 18))
plt.subplot(3, 1, 1)
plt.plot(sizes, sorted_times["Insertion sort"], 'g-', label='Sorted array')
plt.plot(sizes, reverse_times["Insertion sort"], 'r-', label='Reverse sorted␣
↪array')
plt.plot(sizes, random_times["Insertion sort"], 'b-', label='Random sorted␣
↪array')
plt.xlabel('Array Size')
plt.ylabel('Time (sec)')
plt.title('Insertion Sort Time')
plt.legend()
plt.grid(True)
plt.xscale('log')
plt.yscale('log')

plt.subplot(3, 1, 2)
plt.plot(sizes, sorted_times["Merge sort"], 'g-', label='Sorted array')
plt.plot(sizes, reverse_times["Merge sort"], 'r-', label='Reverse sorted array')
plt.plot(sizes, random_times["Merge sort"], 'b-', label='Random sorted array')
plt.xlabel('Array Size')
plt.ylabel('Time (sec)')
plt.title('Merge Sort Time')
plt.legend()
plt.grid(True)
plt.xscale('log')
plt.yscale('log')

plt.subplot(3, 1, 3)
plt.plot(sizes, sorted_times["Quicksort"], 'g-', label='Sorted array')
plt.plot(sizes, reverse_times["Quicksort"], 'r-', label='Reverse sorted array')
plt.plot(sizes, random_times["Quicksort"], 'b-', label='Random sorted array')
plt.xlabel('Array Size')
plt.ylabel('Time (sec)')
plt.title('Quicksort Time')
plt.legend()
plt.grid(True)
plt.xscale('log')
plt.yscale('log')

plt.tight_layout()
plt.show()
```


Insertion Sort Time
Sorted array
Reverse sorted array
Random sorted array
Time (sec)
Array Size


Merge Sort Time
Sorted array
Reverse sorted array
Random sorted array
Time (sec)
Array Size


Quicksort Time
Sorted array
Reverse sorted array
Random sorted array
Time (sec)
Array Size