NSYSU-MATH Data Structure – Spring 2024

Homework 4

Design: Exploring the sorting algorithms

Data Preparation

In this assignment, you're tasked with exploring and implementing various sorting algorithms. We will delve into the intricacies of sorting techniques that were introduced during our lectures, applying them to structured data. You are free to choose between Python and C++ for your implementations. This document outlines the resources provided to you in the form of a zip file named `HW4.zip` and guides you on how to structure your work effectively. Below is an overview of the directory structure and the contents in the zip file:

1. Python Implementation (`Py/` directory):
   - ✓ `ds_sort.py`: This file is your primary workspace where you will develop the sorting functions.
   - ✓ `benchmark.py`: A template for conducting benchmark analysis.
   - ✓ `test.py`: You can test the speed and functionality of your program here.
2. C++ Implementation (`Cpp/` directory):
   - ✓ `ds_sort.cpp`: This file is your primary workspace where you will develop the sorting functions.
   - ✓ `Benchmark.cpp`: A template for conducting benchmark analysis.
   - ✓ `main.cpp`: You can test the speed and functionality of your program here.
   - ✓ `ds_sort`.h: Header files for the sorting algorithms.

Description

You need to choose one of the variants from each category (Totally 3 algorithms should be chosen). Implementations should be done in `ds_sort.py` for Python or `ds_sort.cpp` for C++, with tests conducted in `test.py` or `main.cpp`, respectively.

A. Extend the bubble sort or the shell sort
   1. A bubble sort can be modified to "bubble" in both directions. The first rightward pass will shift the largest element to its correct place at the end, and the following leftward pass will shift the smallest element to its correct place at the beginning. The second complete pass will shift the second largest and second smallest elements to their correct places, and so on. Implement this in `bubble_sort_bidirection()/bubbleSortBidirectional()`, ensuring it supports **sorting in ascending and descending order**.

2. Improve shell sort by allowing for custom gap sequences, diverging from the traditional halving strategy. This adaptation permits the exploration of different gap sequences (a list or vector in descending oreder) to optimize sorting efficiency for specific data sets. Implement this in `shell_sort_gap()/shellSortGap();`, which should also **support both ascending and descending sorting**.

B. Improve the selection sort or the merge sort
   1. Modify selection sort to ensure stability, meaning it preserves the order of duplicate elements as they appear in the original list. Additionally, enable the sort to prioritize either the first or second element in a tuple. Implement this stable version in `select_sort_stable()/selectSortStable().` You **need to demonstrate its stability** compared to the traditional selection sort in the report.
   2. Recall that the slicing operator is $O(k)$ where k is the size of the slice. We will need to remove the slice operator in order to make merge sort truly $O(n \log n)$. Implement this as `merge_sort_noslice()/ mergeSortNoSlice();`. You need to compare its performance to the merge sort mentioned in our class **and show your implementation is faster** in the report.

C. Optimize the quick sort
   1. Implement the median-of-three method for selecting a pivot value as a modification to quicksort. Implement your variant in `quick_sort_median()/quickSortMedian()`. Demonstrate that it functions correctly **when handling a large sorted array, compared with the original quicksort**, which may encounter a recursion limit in the repot.
   2. One way to improve quicksort is to use insertion sort for lists of short length, referred to as the "partition limit." Implement your variant in `quick_sort_limit()/quickSortLimit()`. Specifically, when the sublist's length exceeds the limit, continue the recursion; if it is less than the limit, switch to using insertion sort. Additionally, adjust your implementation to select the pivot value from the middle element of the array. Show that this method operates correctly **when managing a large sorted array, unlike the original quicksort, which could reach the recursion limit** in the repot.

Finally, Conduct a comprehensive benchmark analysis for all implemented sorting algorithms (bubble sort, shell sort, selection sort, merge sort, and quick sort) across arrays of sizes 10, 50, 100, 200, 500, 1000, 2000, 5000, and 10000. Include tests with **sorted, inverse-sorted, and randomly sorted** lists (vectors) of integers to thoroughly evaluate performance.

Specifications

1. Be sure to follow the input and output formats specified in the template file.
2. You are allowed to use only the standard libraries of Python or C++. In addition, feel free to refer or use the sorting algorithms from our provided code base.
3. For benchmarking purposes, you can use the sorting algorithms from our provided code base or replace them with your implementations. Specifically, for quicksort, **you must use your own implementation developed in Category C**.
4. We have supplied basic benchmark code. You can use this directly but must also write code to generate the three types of lists (sorted, inverse-sorted, and random) in various sizes, as well as the main program.

Deliverables

1. Deadline: 2024/5/19 (Sun.), 11:59 PM. Hand in the following two items to the cyber universities. Please see our Facebook group for the late policy and rules.
2. Report:
   - ✓ Outline your test design and observation for Categories B and C.
   - ✓ Present detailed benchmark results for the five sorting algorithms, including observations and comparisons to the Big O complexities discussed in class.
   - ✓ Conclude with insights gained from this homework.
3. Program Source Files:
   - ✓ Submit your source files and report according to instructions stated here. **Ensure that you follow the provided template files**.
   - ✓ Source File Comments: Each file must begin with three lines of comments indicating the Author, Date, and Purpose of the program.

Grading Policy

- Function correctness: 45% (15% for each algorithms)
- Benchmark analysis: 25%
- Report and discussion: 30%.

Reference

1. https://realpython.com/sorting-algorithms-python/#the-timsort-algorithm-in-python
2. https://github.com/diptangsu/Sorting-Algorithms
3. https://github.com/tahoe01/Sorting-Algorithms
4. https://www.angela1c.com/projects/cta_benchmarking/ctabenchmarkingproject