CSCI 4118/6105 Lab 1 (Part 1):

Analysis and Comparison of Sorting Algorithms.

Winter 2022

Objective

1. Understand asymptotic analysis of Sorting algorithms: Insertion Sort, Merge Sort, and Quick Sort.
2. Compare the above sorting algorithms with different inputs.

   Extension for Graduate students:
3. Explore and experiment with optimization techniques for improving the time and space complexities using Hybrid Sorting algorithms.

Pre-requisites

1. Understanding of Linear array-like data structures.
2. Knowledge to use basic git commands to clone, pull, push, commit and merge.
3. Java programming language: While it's not necessary to have proficiency in Java, the ability to read and understand any programming language is crucial.

Pre-read/Terminologies

1. Priori Analysis and Posteriori Testing.
2. Asymptotic Analysis: Time and Space Complexity in big O, big Theta (Θ), and big Omega (Ω).
3. Sorting: Ascending, Descending order; Lexical Order for Strings.
4. Stability of sorting algorithms and In-place algorithms.
5. Tail and Head Recursion.
6. Algorithm Design Paradigm: Divide and Conquer.

Resources

Gitlab Repository: https://git.cs.dal.ca/courses/2022-winter/csci-4118-6105/lab1/???? Where ???? is your CSID

Procedure

Note: Undergraduate students are encouraged to attempt additional questions meant for graduate students and get bonus points.

1. Clone the repository: *git clone https://git.cs.dal.ca/courses/2022-winter/csci-4118-6105/lab1/????.git* where *????* is your CSID.
2. Get the latest pull of the master branch: *git pull origin main*
3. Execute all three sorting algorithms for different input sizes and degree of sortedness.
4. Inputs: sorted, reverse sorted, and randomly sorted for an input size of 100, 1000, 100000 (For example, refer to the function: "*compare*" in "*Driver.java*" and "*Generator.java*" for utility functions).
5. Analyze and compare the performance of sorting algorithms (Order of growth and Watch time).
6. Report your findings in "*output_part_1.txt*" (refer to submission section for format).

Extension for Graduate students:

Perform further analysis for nearly sorted arrays by either rotating the array or changing the order of elements in a sorted array.

7. Example inputs (Sorted, sorted in reverse, left/right rotated by 1 to n - 1, random sorted, 1 to n - 1 elements out of order):
    a. [1, 4, 5, 7, 10, 13, 14, 22, 46]
    b. [46, 22, 14, 13, 10, 7, 5, 4, 1]
    c. [46, 1, 4, 5, 7, 10, 13, 14, 22]
    d. [13, 14, 22, 46, 1, 4, 5, 7, 10]
    e. [4, 5, 7, 10, 13, 14, 22, 46, 1]
    f. [13, 14, 22, 46, 1, 4, 5, 7, 10]
    g. [22, 1, 10, 4, 13, 5, 14, 7, 46]
    h. [1, 4, 5, 7, 46, 10, 13, 14, 22]
    i. [4, 5, 7, 10, 1, 13, 14, 22, 46]
8. Report your findings for sorted, unsorted, nearly sorted arrays for different input sizes.

Compute the runtime of the program (All Students)

1. Capture the time before and after running the program using java.time (Instant and Duration) or equivalent in order programming languages.
2. Executing the program:
    a. If you are using an IDE such as IntelliJ (Recommended), build the project and run Driver.main()
    b. Optionally, you can run the java program from the command line: *javac lab/sorting/*.java* (compile) and *java lab.sorting.Driver* (Execute).
3. Sample output

```
Random-order Array
Algorithm            | Length       | Duration
===============================================
Merge Sort           | 10           | PT0.000497S
Insertion Sort       | 10           | PT0.000491S
Hoare Quick Sort     | 10           | PT0.000241S
Lomuto Quick Sort    | 10           | PT0.000009S
-----------------------------------------------
Merge Sort           | 1000         | PT0.001029S
Insertion Sort       | 1000         | PT0.003203S
Hoare Quick Sort     | 1000         | PT0.001855S
Lomuto Quick Sort    | 1000         | PT0.000537S
-----------------------------------------------
Merge Sort           | 100000       | PT0.021066S
Insertion Sort       | 100000       | PT0.853875S
Hoare Quick Sort     | 100000       | PT0.008145S
Lomuto Quick Sort    | 100000       | PT0.02058S
```

Questions

1. Which sorting algorithm performs the best for a sorted/nearly sorted array?
2. Which algorithm is efficient for small and large datasets?
3. If you had to choose one sorting algorithm among the three for all input types, which one would you choose, despite the tradeoffs?

Extension for Graduate students:

4. Write a hybrid algorithm (Tim sort) with a combination of Insertion sort and Merge sort.
    a. Choose a chunk size that is efficient for insertion sort: 32
    b. Sort subarrays (in-place) of the input array using insertion sort.
    c. Merge subarrays using merge sort.

Submission

Note: For submission - git add, commit and push the answers to the lab1/???? repository and verify the submission in the GitLab web interface.

1. Answer the questions in "*questions_part_1.txt*"
2. Paste the output for all three algorithms for different inputs and summarize your findings from the experiment in "*output_part_1.txt*".

Extension for Graduate students:

3. Create a new file, "TimSort" (Example: "TimSort.java" for java program), under the lab/sorting package in the same repository clone earlier (refer to resources section). Write a program for Tim Sort (Merge sort + Insertion Sort). The reference code is in Java, but you can choose any other language you are comfortable with.

Sorting Algorithms

Optional read: A brief introduction on Insertion, Merge, and Quick sort algorithms.

Insertion Sort

Overview:

1. Time complexity
    a. Worst-case and Average-case: O(n^2)
    b. Best-case: O(n)
2. Space complexity: Constant.
3. In-place and Stable Algorithm.
4. Steps:
    a. Divide the arrays into a sorted and unsorted portion and expand the sorted portion - element by element.
    b. While expanding, insert the new element in its proper place within the sorted subarray until the sorted subarray length is equal to the input array length.

Merge Sort

Overview:

1. Divide and Conquer Algorithm.
2. Stable Algorithm.
3. Time complexity (worst, average, and best): O(n log n)
4. Space complexity: O(n) (Typical Implementation).
5. Merge is the key function.
6. Steps:
    a. Partition the array into two equal partitions.
    b. Recursively sort the left and right half and merge the solutions (Merging two sorted arrays).

Quick Sort

1. Divide and Conquer, Unstable (Typical Implementation) Algorithm.
2. Time complexity (T(n) is the time taken by quick sort for input size n):
    a. Worst-case: $T(n) = T(n - 1) + \Theta(n) = O(n^2)$
    b. Best-case: $T(n) = 2T(n/2) + \Theta(n) = O(n \log n)$
    c. Average-case (Average of all permutations): $O(n \log n)$
3. Space complexity:
    a. Worst-case: $O(n)$
    b. Best-case: $O(\log n)$
    c. Average-case: $O(\log n)$
4. Despite the worst-case time complexity of $O(n^2)$, quick sort is
    a. In-place algorithm (Does not require the auxiliary space to partition the array).
    b. Cache Friendly (Since the algorithm is in-place).
    c. Average Case time complexity: $O(n \log n)$.
    d. Tail Recursive (Recursive call is the last thing the recursive function does).
5. Partition is the key function: Stable: Naive, Unstable: Lomuto, and Hoare (Most Efficient).
6. Steps:
    a. An element is said to be in the sorted position if the elements on the left are lesser and those on the right are greater.
    b. Pick an element as a pivot and place it in a sorted position or in lesser/greater portion; the left and right sections of the pivot are unsorted and hence recursively repeat until all elements are in sorted positions.

Grading

Task	1 Point (x3)	0 Points
1	For each question in *questions_part_1.txt*, answered correctly.	Incorrect answer or not answered.

Task	2 Points (x1)	1 Point	0 Points
2	Thoroughly tested and reported findings in "*output_part_1.txt*" for all three sorting algorithms with different input array sizes and degree of sortedness.	Tested less than three sorting algorithms or only a few cases of array sizes and degree of sortedness and reported findings in "*output_part_1.txt*".	No evidence of testing any of the sorting algorithms.

Extension for Graduate students:

Task	5 Points (x1)	3 Point	0 Points
3	Tim Sort hybrid sorting algorithm implemented and tested.	Tim Sort hybrid sorting algorithm implemented, logically correct (pseudo-code), and/or evidence for testing and explanation.	Incorrect answer or not answered.