Exploring Hybrid and Adaptive Sorting Algorithms

Author: Evan Green

Date: 10/27/2023

Table of Contents

Overview

I started this project with a simple goal in mind: to create a sorting algorithm that could get the job done a bit faster. So, I thought, why not take the best parts of two existing sorting techniques, but which ones should I choose? After deliberating with some peers I decided on Quick Sort and Insertion Sort.

Quick Sort is like the superstar when dealing with a big crowd of numbers, swiftly breaking them down into smaller groups, making the sorting job way less daunting. On the flip side, Insertion Sort is the quiet genius that shines with smaller groups or almost sorted ones, handling them with a straightforward, no-nonsense approach.

So, the idea was to create a buddy system, crafting a new algorithm I called Quick-Insertion Sort. This algorithm would dynamically decide who takes the stage, Quick Sort or Insertion Sort, based on the size of the data it's dealing with at the moment. If it's a large pool of unsorted data, Quick Sort steps up to break it down, and if it's a smaller or nearly sorted group, Insertion Sort takes over to wrap things up neatly.

The whole process was not just about speeding things up. It was also about diving deep into the world of algorithm design, exploring how mixing different sorting strategies could potentially tackle sorting chores more efficiently in various scenarios.

Project Components

Algorithm Design

1. Importing Necessary Modules
    - import time: Imports the time module to allow for timing the execution of the algorithm.
    - import random: Imports the random module for generating a random sample of numbers in one of the test cases.
    - import sys: This was the most important one in debugging the partitions in quicksort and the hybrid algorithm

2. Function Definitions

   a. insertion_sort(arr, low, high)
    - This function implements the Insertion Sort algorithm for a specified segment of the array arr from index low to high.
    - A loop iterates through the segment, and within the loop, another while-loop shifts elements to the right until the correct position for the current element is found.

   b. partition(arr, low, high)
    - This function implements the partitioning step of the Quick Sort algorithm.
    - It chooses the last element of the segment as the pivot, then rearranges the segment so that all elements less than the pivot come before it and all elements greater come after.

   c. quick_insertion_sort(arr, low, high, threshold)
    - This is the main function implementing the hybrid Quick-Insertion Sort algorithm.
    - It checks the size of the current segment against a threshold value. If the segment size is below the threshold, it calls insertion_sort; if above, it proceeds with a Quick Sort approach.
    - The Quick Sort approach involves partitioning the segment, then recursively calling `quick_insertion_sort` on the two resulting sub-segments.

d. run_test_case(arr, threshold)

- This function executes a test case, timing the execution of the quick_insertion_sort function on the input array.

3. Test Case Definitions
   - Six test cases are defined to evaluate the performance of the quick_insertion_sort function.

4. Setting The Threshold
   - The threshold value at which the algorithm switches from Quicksort to Insertion Sort is set to 10.

5. Executing Test Cases
   - The run_test_case function is called for each test case, and the execution time is printed to the console.

6. Output
   - The program prints the execution time for each test case, allowing for an analysis of the performance of the Quick-Insertion Sort algorithm under different data conditions.

Testing and Benchmarking

- For testing I used six test cases for each algorithm. This included taking the results and comparing the times of the quick-insertion sort to a traditional quick sort and insertion sort with the same benchmarks.

	Insertion Sort	Quick Sort	Hybrid Sort
1000 Unique	0.018 seconds	0.003 seconds	0.0005 seconds
1000 Reverse Sorted	0.036 seconds	0.003 seconds	0.00099 seconds
10000 Unique	1.720 seconds	0.038 seconds	0.0139 seconds
10000 Reverse Sorted	3.220 seconds	0.036 seconds	0.0109 seconds
100000 Unique	163.080 seconds	0.467 seconds	0.160 seconds
100000 Reverse Sorted	319.052 seconds	0.390 seconds	0.128 seconds

Performance Analysis

- The environment decided on after some testing was pycharm. I originally was running these cases in a replit environment but the internet speed became a factor to account for. So I spun up a local environment and did it there.
- The only other notable performance issue was setting a recursion limit on the quicksort due to the larger test cases.

Skills Gained

This project helped me gain not only a more in depth understanding of these algorithms, but also on debugging with memory errors. There was a portion of my partition that I couldn't fix because of one error. Then when I would increase the recursion limit it would error out somewhere else. This let me take a deeper dive in why my code was the way it was and made me research further on my partition function. Once I got it to work everything went smoothly.

Deliverables

- Repository: https://github.com/ThePalad1n/quick-insertion-sort

Conclusion

The Quick-Insertion Sort algorithm was both challenging and enlightening. It began with a simple yet significant ambition: to accelerate the sorting process by fusing the robustness of Quick Sort with the finesse of Insertion Sort. This endeavor not only aimed at achieving better performance but also served as a deep dive into the intricacies of algorithm design and optimization.

Achievements:
Algorithm Conception and Design:

- Successfully conceptualized and crafted a hybrid algorithm, Quick-Insertion Sort, that dynamically delegates sorting tasks to either Quick Sort or Insertion Sort based on data size and distribution.
- Established a thoughtful threshold to switch between the two sorting strategies, which was set at 10, striking a balance between optimality and complexity.

Code Structure and Implementation:

- Built a well-structured and modular code with distinct functions for each algorithm, a partitioning routine, and a mechanism to execute and time test cases.
- Leveraged Python for its readability, ease of use, and robust library support, which significantly eased the debugging and testing process.

Testing and Benchmarking:

- Designed six test cases to rigorously evaluate the performance of the Quick-Insertion Sort against traditional Quick Sort and Insertion Sort.
- The benchmarking results showcased a notable performance improvement with the hybrid sort, especially as the data size scaled up.

Performance Analysis:

- Transitioning to a local environment from a replit environment eradicated network latency, ensuring more accurate timing results.
- Encountered, analyzed, and resolve performance issues related to recursion limits and memory errors, which required a deeper understanding of the underlying algorithm and the Python environment.

Skills and Knowledge Acquired:

- Gained a richer understanding of sorting algorithms, their performance characteristics, and their behavior under different data conditions.
- Acquired valuable experience in debugging memory errors and managing recursion, which unveiled the importance of a well-designed partition function in ensuring algorithm efficiency and stability.

Reflections:

The challenges encountered, particularly around debugging memory errors and adjusting recursion limits, were not mere hurdles but learning stepping stones. They unveiled the delicate interplay between algorithm design, memory management, and system configurations. The project affirmed that while algorithmic optimization can significantly boost performance, a thorough understanding of the underlying system and environment is crucial for diagnosing and resolving issues that may arise.

The success of the Quick-Insertion Sort algorithm in achieving better performance across various scenarios reflects the potential of hybrid algorithms in tackling computational problems more efficiently. This project was not merely an exercise in coding but an exploration of how blending different algorithmic strategies could lead to more efficient solutions, providing a solid foundation for tackling more complex algorithmic challenges in the future.