**Sorting Algorithm Experiment**

Sorting Algorithms and their Efficiencies

**Authors**:

Aaron Jeromsky

Ratanachat Saelee

Lekendrick Cage

Mark Buster

**North Seattle College**

**Course**:

Analysis of Algorithms (CSB302)

**Instructor**:

Eric Lloyd

**Date**:

1/31/2024

Table of contents

1. Experiment design
   1. Input data
   2. UML diagram
2. Algorithm comparison
3. Algorithm Descriptions and Results
   1. Algorithm: (same sub-sections for each algorithm)
      1. Best use cases regarding data set size
      2. Implementation choice and advantages/drawbacks
   2. Insertion Sort
   3. Selection Sort
   4. Bubble Sort
   5. Shell Sort
   6. Merge Sort
   7. Quick Sort (pivot selected as first element)
   8. Quick Sort (pivot selected as random element)
   9. Quick Sort (pivot selected with Median-of-three)
   10. Heap Sort
   11. Counting Sort
   12. Bucket Sort
   13. Radix Sort

Experiment Design

Input Data

This study aims to evaluate the operational efficiencies of twelve commonly used sorting algorithms, ranging from simpler but time-consuming ones to more complex yet faster alternatives. To achieve this, we created a comprehensive array dataset, exclusively consisting of integers, including occasional duplicates. The dataset involves five distinct pre-ordering patterns, each applied to fourteen array sizes ranging from 2^2 to 2^15. These arrays are organized randomly, in ascending and descending order, and with fifty percent or seventy-five percent in ascending order. Consequently, seventy arrays are processed through each algorithm to assess their respective efficiencies.

Various methods exist to measure algorithmic complexity. One approach involves developing software to track the number of operations during array processing, adjusting for array size to derive conclusive metrics. Alternatively, a simpler method involves measuring efficiency by recording the program's runtime. In this study, we adopted the latter approach, leveraging the runtime tracking functionality of the Integrated Development Environment (IDE). Subsequently, code was developed to extract recorded runtime data, enabling subsequent analysis of algorithmic efficiency.

UML Diagram

A diagram of a computer

Description automatically generated

This UML diagram displays the design structure of this program. In our program you can see the Main class is linked to two others which control how the program performs. The DataManager is where all the arrays are created and stored in a file. From there they are fed from that file into the TimeManager, and subsequently fed into each respective algorithm class. All the algorithm classes are separate from one another and contain only enough methods to let the algorithm function properly. When the algorithms run, they are timed by code in the TimeManager class and those times and written to another file to later be interpolated into a graph format.

This specific organization design enabled the greatest simplicity between classes. The experiment did not need to be complex in any way. That's why we had the DataManager class manage the data, and the TimeManager class handle the time. Sequentially, Main calls DataManager, and DataManager writes data to a file. Then TimeManager reads data from the file that DataManager wrote to. TimeManager takes that data and inputs it into each algorithm, located between the “startTime” and “endTime” method calls to track the runtime. Lastly, TimeManager compiles the calculated runtimes and writes those to another file, one for each algorithm. This is where we then access the times to paste into our spreadsheets to subsequently create line graphs.

Microsoft Excel is the software used to plot our time data into tables and from those tables we created line graphs. At this point in the experiment, enough data has been gathered and manipulated to where we can determine for our various sized sets of arrays what the empirical time complexity is for each algorithm and how that compares to the theoretical time complexities.

Algorithm Comparison

This experiment entails the comparative assessment of twelve prevalent sorting algorithms, examining their respective performances across diverse data sets. The algorithms under scrutiny exhibit varying capabilities concerning data set sizes, with some excelling in the handling of smaller data sets, while others demonstrate efficiency in managing considerably larger data sets. Specific algorithms exhibit optimal performance within defined data ranges, indicating their contextual suitability.

Categorization of these algorithms is best approached by delineating their compatibility with distinct data set sizes. Notably, algorithms performing optimally with larger data sets typically demonstrate efficiency across both small and large sets. Conversely, algorithms designed for smaller sets often manifest high time complexity, rendering them progressively inefficient as data set sizes increase.

Bubble sort, selection sort, insertion sort, and shell sort emerge as comparatively less time-efficient algorithms, with insertion sort and shell sort exhibiting superior performance within this subset. Despite their relative slowness, these algorithms possess a crucial tradeoff in the form of minimal memory requirements. As "in-place" algorithms, they demand minimal memory allocation, relying solely on space for a few extra variables and eschewing auxiliary data structures. Remarkably, insertion sort outperforms expectations by demonstrating superior performance with partially sorted lists within specific larger data sets.

In contrast, merge sort and heap sort are identified as robust, versatile, and reliable algorithms suitable for general-purpose applications. Merge sort, though somewhat space-intensive during the merging process, maintains time efficiency. Heap sort, despite the potential complexity associated with binary heap integration, proves adept at rapid and reliable sorting, with the added advantage of being an in-place algorithm, thereby conserving memory.

Remaining algorithms, such as quicksort variants, exhibit prompt execution, each characterized by distinct attributes. The selection of an optimal algorithm necessitates careful consideration of specific use-case requirements. Quicksort implementations, including "pivot with a median-of-three" and "pivot as a random element," consistently operate in-place and align with theoretical expectations. However, the variant using "pivot as the first element" experiences degraded time efficiency for larger data sets with in-order or reverse-ordered patterns, excelling in other pre-sort configurations.

Counting sort, bucket sort, and radix sort present specialized strengths warranting consideration in specific scenarios. Counting sort, for instance, is unsuitable for datasets containing negative integers and requires additional space equivalent to the range of values to be sorted. Bucket sort performs admirably with well-balanced data distributions but falters when confronted with left or right-leaning distributions. Radix sort, while generally applicable, performs optimally when incoming values share similar character lengths, demonstrating proficiency in sorting strings.

Ultimately, consideration of memory constraints, input data characteristics, and the desired time complexity should guide the selection of a sorting algorithm for a given application.

Algorithm Descriptions and Results

Insertion Sort

A screenshot of a computer

Description automatically generated

Insertion sort is most practical for small data set, it is not very efficient for large since the worst-case time complexity of insertion sort is O(n^2). It performs better on partially sorted or small datasets. The result mostly meets theoretical expectations. We can see that it is much faster to sort when the input is partially sorted, when compared to random for small size array. Insertion sort's low space complexity makes it suitable for embedded systems with limited memory. If the dataset is small or memory is a critical constraint, insertion sort might be a reasonable choice. Since insertion sort is simple, there is one standard implementation of the algorithm.

Selection Sort

A screenshot of a computer

Description automatically generated

Selection Sort is characterized by its simplicity and its quadratic time complexity, (O(n^2)), regardless of the initial order of the elements. This inherent property of always performing (n(n-1)/2) comparisons makes it less efficient for larger datasets, as confirmed by our empirical results which depicted a quadratic increase in operation time with respect to the array size.

We chose to measure the Time of Operations as the second metric for Selection Sort because it directly reflects the algorithm’s performance regardless of the input state (random, sorted, or reverse sorted). The experimentally observed number of operations closely adhered to the theoretical quadratic growth, demonstrating that Selection Sort operates with consistent performance across different datasets.

The implementation of Selection Sort was chosen for its educational value and ease of understanding, which is beneficial for demonstration purposes and when working with smaller datasets. We rejected more complex or optimized variants such as bidirectional selection sort because the simplicity of the standard implementation better served the pedagogical goals of our study, and the optimization did not significantly improve the algorithm's worst-case time complexity.

Bubble Sort

A screenshot of a computer

Description automatically generated

Bubble sort is well-suited for small datasets due to its average and worst-case time complexity of O(n^2). Despite its quadratic time complexity, it has minimal space complexity (O(1)), making it suitable for scenarios with memory constraints and smaller datasets. An example of where the bubble sort algorithm could be applied is in embedded systems, such as medical devices or kitchen appliances, which typically have limited functions and compact spatial configurations. These systems often require sorting operations within restricted memory and that are most effective with relatively small datasets.

Our experimental results align consistently with the expected outcomes based on the theoretical analysis of the algorithm. It is important to highlight that our dataset did not exhibit any unusual patterns or outliers. The graphical representations consistently reflected the expected quadratic increase in time complexity as the size of the dataset increased.

Because the bubble sort algorithm has a simple design, there aren't many significantly different ways to implement it. This limits the options for alternative approaches in our study. The way we implemented it is in line with common practices in the industry wherever bubble sort is used.

Shell Sort

A screenshot of a computer

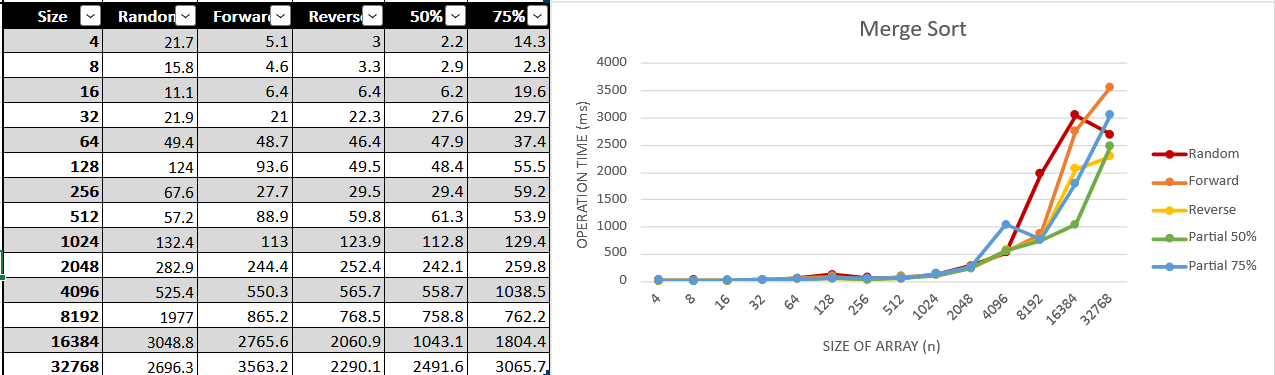
Description automatically generated

Shell Sort's design leverages interval comparisons to enhance efficiency over basic insertion sort, specifically aiming to minimize the number of shifting operations. Our team selected Time of Execution as the second metric to evaluate its performance because it reflects the actual speed improvements Shell Sort offers, especially as the size of the dataset increases.

Our empirical findings, in terms of Time of Execution, align with Shell Sort's theoretical time complexity, which is generally between (O(n)) and (O(n^2)), depending on the chosen gap sequence. The observed times show a less-than-quadratic growth for medium-sized datasets, substantiating the algorithm’s efficiency.

We implemented Shell Sort with a simple gap sequence, reducing it by half each time, for a balance of ease of understanding and execution efficiency. More complex gap sequences were considered but ultimately rejected due to the marginal performance gains that did not justify the increased complexity for our datasets.

Merge Sort



Merge sort follows the divide-and-conquer paradigm with O(n log n) time complexity for all cases. This sorting algorithm has a stable time complexity; however, it does use additional space for the temporary arrays during the merging process. This makes it less memory efficient, so it is generally suited for scenarios where stability, predictable performance, and the need for a reliable sorting algorithm are crucial such as large system where efficient sorting of large amounts of data is required. The result mostly meets theoretical expectations. We can see that the time algorithm taken for different sort input is generally with range of each other. There is typically one main implementation of the merge sort algorithm, however specific detail can vary such a data type being int array or how I divided the array. I decided to go with System.arraycopy instead of for loop since it is usually faster because it is a native method and can take advantage of low-level optimizations.

Quick Sort First

A screenshot of a computer

Description automatically generated

The quick sort algorithm is an efficient algorithm that strikes a balance between time and space complexity, with an average case of Θ(n log(n)) time complexity and only O(log(n)) space complexity.

Although the algorithm is usually efficient, it suffers from an O(n^2) time complexity in its worst-case scenario, which occurs when the input data is already sorted (forward or backward) resulting in an exponential number of calls.

My expectations for this sorting algorithm were largely met as the algorithm demonstrated some of the shortest operating times out of the set of algorithms we worked with. Although I was aware of the worst-case scenario being with a sorted array, I was taken aback by just how disastrous it was for the operation time. As the graph above shows, the operating time on the sorted arrays is so long that all other times are practically unreadable due to the sheer difference in scale.

Additionally, for this project, I went with a recursive implementation rather than an iterative one since the recursive approach since I found it to be easier to model and understand. An iterative approach wasn’t as interesting to me since it relies heavily upon while loops which make it difficult to keep track of what is occurring within the algorithm.

However, following this approach results in a stack overflow exception when sorting larger datasets, in this case the 32,768 element arrays. To prevent the exception, the stack calls needed to be reduced to accomplish this I forced the algorithm to only operate on one-half of the array at a time thereby reducing the memory needed at any one time.

Quick Sort Median

A screenshot of a graph

Description automatically generated

*Since each quick sort algorithm is similar, I’ll focus on what is different for each subsequent one.*

Using the element with a median value of the array offers a solution to the worst-case time complexity where sorted data is operated on. The algorithm achieves this through selecting the median element as a pivot rather than the first or a random one. This is to prevent the worst-case scenario from occurring by almost always having smaller and larger values to be sorted. This improves the worst-case time complexity from O(n^2) to O(log n).

Upon learning of the approach, I had expected the approach to significantly cut down on the operating time, which it did. It also had incredibly fast operating times on partially sorted arrays to my surprise. I assumed that the median pivot would only prevent the worst-case scenario, but it also appears to immensely speed up partially sorted arrays as well.

For this approach, I copied the recursive code I wrote from the previous quick sort algorithm, since following the same basic structure made it easier to compare the results and ensure the differences in operation time were merely the result of using a more efficient approach.

Quick Sort Random

A screenshot of a graph

Description automatically generated

*Since each quick sort algorithm is similar, I’ll focus on what is different for each subsequent one.*

Using a random element as the pivot is remarkably similar to using the median element. This largely meets my expectations as using a random element as the pivot would be somewhat similar to using the median, as most likely the random value will be somewhere around the middle. Though of course, since it’s random there is still a small chance that the worst pivot element is selected every time, which means its worst-case time complexity goes back to being O(n^2).

The algorithm performs less efficiently than the median pivot approach, this is to be expected since a random pivot will usually not be the optimal one, but not the worst one either. This results in times being slower overall but following a similar pattern.

I noticed there are some unusual spikes in the operating times. If I had to think of a reason, perhaps the pivots selected were simply less than optimal, and rerunning the program might produce different results?

Heap Sort

A screenshot of a computer

Description automatically generated

Heap sort is one of the overall most effective and reliably consistent algorithms. Theoretically it performs well with small and large data sets, regularly exhibiting a time complexity of O(nlogn). This remained true in our study as well. Heap sort often has extra memory requirements in its implementation, but we decided to use an implementation that did not need this. Ours uses an abstract data structure of a max heap over top of a single array. It sorts this array in-place by partitioning off the sorted portion of it as abstractly is removing the max values of the heap and swapping them with the last values of the array.  After all values of the max-heap are removed, the array is sorted.

In the end we chose this implementation because of its simplicity and its effective encapsulation of being contained in one class. The encapsulation was critical to the organization of our program.

Counting Sort

A screenshot of a graph

Description automatically generated

Counting Sort is distinct for its non-comparative nature and linear time complexity, (O(n+k)), where (n) is the number of elements and (k) is the range of input. This makes it exceptionally efficient for sorting integers when (k) is not significantly larger than (n).

Our team opted to measure the Time of Execution as the second metric for Counting Sort. Given the algorithm's linear complexity, the execution time is a transparent indicator of its performance across varying sizes and states of datasets. Our empirical results showed a consistent linear relationship between execution time and the number of elements, which matched the theoretical performance of Counting Sort.

The implementation of Counting Sort was chosen due to its simplicity and efficiency when dealing with a limited range of integer values. We recognized that its memory usage is directly tied to the maximum value in the dataset, which informed our decision to employ this algorithm specifically for datasets with a known, constrained range.

Alternative implementations that utilized more space-efficient data structures or allowed for sorting of negative integers were considered. However, they were dismissed in favor of maintaining a low complexity and avoiding the overhead that comes with handling a wider range of numbers, which was unnecessary for our dataset.

Bucket Sort

A screenshot of a computer

Description automatically generated

The theoretical expectations for a bucket sort algorithm entail a time complexity of O(n+k) in both the best and average cases, with a worst-case time complexity of O(n^2). Our empirical findings closely align with these theoretical expectations. An intriguing observation arose when processing a 50% sorted array, which unexpectedly led to our worst-case scenario. In contrast, for all other array patterns, the algorithm consistently demonstrated performance closer to the more expeditious end of the O(n log n) range. The overall performance of the algorithm aligned with anticipated outcomes.

The selected implementation prioritized simplicity. Adopting a straightforward strategy, we utilized an array to house the buckets, employing ArrayLists within it to function as the individual buckets. This approach was deemed viable, leveraging Java's Collections.sort() method to attain the desired order within the buckets. While an alternative approach involving priority queues as buckets could eliminate the necessity for the Collections.sort() method, this was intentionally eschewed to maintain code simplicity. However, it is worth contemplating the incorporation of priority queues in future implementations, given their suitability for this algorithm and their ability to uphold low memory overhead.

Radix sort

A screenshot of a computer

Description automatically generated

Radix sort is a linear time with the time complexity of O(n). It is best used when dealing with large datasets and the key size is fixed and known in advance. It may not be as efficient as other algorithms if the size of the dataset is small, or the range of keys is large. Radix sort is commonly used in applications where integer or string keys are involved, such as sorting strings based on their lexical order or sorting integers based on their binary representation. It can be useful for large-scale data processing systems where the dataset is significant, and the keys are fixed size. The result mostly meets theoretical expectations. We can see that the time algorithm taken for different sort input is generally with range of each other. There are a few ways to implement radix sort subroutine. Although bucket sort can be better used for when range of digits vary significantly, when distribution of elements uneven. Accessing elements in linked lists may result in less cache efficiency compared to a contiguous array. I use counting sort subroutine for radix instead of bucket sort. Since the data type we test are integer only and range of digits is known in advance, so having simpler array structure and may be more memory efficient as it directly uses an array of fixed size.