DOKUZ EYLÜL UNIVERSITY

ENGINEERING FACULTY

DEPARTMENT OF COMPUTER ENGINEERING

CME 2204

Algorithm Analysis

Comparing Dual Pivot Quick Sort and Shell Sort

By

KAAN YILMAZ 2021510072

INTRODUCTION

Sorting algorithms play a fundamental role in computer science and are extensively used in various applications. Understanding their performance characteristics is crucial for selecting the most suitable algorithm for specific scenarios. In this report, we compare the performance of two popular sorting algorithms, dualPivotQuickSort and shellSort, across different input scenarios: equal integers, random integers, increasing integers, and decreasing integers.

METHODS

1. SortingClass: This class contains two different sorting methods, namely dualPivotQuickSort and shellSort.
   - dualPivotQuickSort(int[] array, int low, int high): A helper method to sort the array within a specified range (low - high).
   - sort(int[] array, int low, int high): A helper method used to sort elements within a particular range (low - high). This method determines pivot elements and sorts the array based on these pivot elements.
   - swap(int[] array, int i, int j): Swaps the positions of two elements in the array.
   - shellSort(int[] arrayToSort): Implements the shell sort algorithm. This algorithm works by partially sorting the array and then reducing the gap gradually.

2. Main: This class facilitates interactive usage of the program.
   - It prompts the user to choose the type of array to create, the sorting method to use, and the size of the array.
   - Based on user selections, it creates an array and applies the chosen sorting method.
   - Finally, it displays the sorting time.

INFORMATIONS

1. Dual Pivot Quick Sort:

Dual pivot quicksort is an improvement over the traditional single pivot quicksort algorithm. It employs two pivots to partition the array into three parts instead of just one, potentially reducing the number of comparisons and swaps required to sort the array.

Dual Pivot Quicksort Algorithm:

1. **Selection of Pivots**: Initially, two pivot elements are chosen from both ends of the array. The left pivot is denoted by LP and the right pivot by RP. If LP is greater than RP, they are swapped.
2. **Partitioning**: The array is partitioned into three parts:
    - The first part contains elements less than LP.
    - The second part contains elements greater than or equal to LP and less than or equal to RP.
    - The third part contains elements greater than RP.

$< LP$	LP	$LP \leq \ \& \ \leq RP$	RP	$RP <$

3. **Shifting Pivots**: After partitioning, the pivots might not be in their final positions. So, we iterate through the array and swap elements to ensure that LP is at its correct position, followed by RP.
4. **Recursion**: The array is recursively partitioned and sorted using the same dual pivot quicksort algorithm on the three parts created in the previous step.
5. **Base Case**: The recursion stops when the size of the partitioned array becomes sufficiently small, typically when it contains fewer elements than a certain threshold. At this point, a sorting algorithm sort the remaining elements efficiently.

Runtime Complexity:

- **Best Case**: In the best case scenario, each partitioning step divides the array into three nearly equal-sized partitions. This results in a balanced partitioning tree. The time complexity in this case is O(n log n), where n is the number of elements in the array.
- **Average Case**: On average, dual pivot quicksort performs better than single pivot quicksort due to its ability to create more balanced partitions. The time complexity in this case is also Θ (n log n).

Each recursive call processes a subarray of size roughly n/3.

The algorithm makes 3 recursive calls.

T(n) = 3T(n/3) + Ω (n)

Solve this equation by Master Method

$n^{\log_b a}$

I know that our b is: n/3 → 3

Also my a is: 3T → 3

$n^{\log_3 3} \rightarrow n^1$

Then I look at the d. D is the Ω (n) degree. → d's degree is 1.

Then I look at the cases. This calculation is the case 2.

Case 2 → $\log_b a$ = d → 1 = 1

So that Θ(n^dlog n) → Θ(n log n)

- **Worst Case**: The worst-case time complexity occurs when the array is already sorted in increasing or decreasing order. In such scenarios, the partitions formed after each partitioning step are highly unbalanced, leading to a partitioning tree resembling a linear chain. This results in a time complexity of O(n^2).

In the worst case, the array is divided into two subarrays of sizes (n-1) and 1, or vice versa, at each recursive call.

T(n) = T(n-1) + O(n) → T(n) = T(n-1) + T(n-2) + T(n-3) + ... + T(1) + O(n)

We understand that T(n – k) = 1, n = k so that O(level*cost) → O(n*n) → O(n^2)

Space Complexity:

- **Average and Best Cases**: The space complexity is O(log n) due to the recursive calls made on partitions. This is because the recursive calls are done on smaller partitions, and thus the stack space required for recursion is proportional to the logarithm of the input size.
- **Worst Case**: In the worst case, the space complexity increases to O(n) due to the unbalanced partitioning, which leads to a deep recursion tree, consuming more stack space.

2. Shell Sort:

Shell sort, also known as Shell's method, is an algorithm that builds on the insertion sort by allowing the exchange of elements that are far apart. It was introduced by Donald Shell in 1959 and is a generalized version of insertion sort.

Shell Sort Algorithm:

1. **Initialization:** Start by defining a gap sequence, often called the increment sequence. This sequence determines the gap between elements to be compared during each pass of the algorithm. Commonly used sequences include the Knuth sequence (3k - 1), where k is an integer, or simply dividing the array size by 2 repeatedly.
2. **Gap Sorting:** Divide the array into sublists, each sublist having elements that are 'h' positions apart, where 'h' is the gap size. Perform insertion sort on each sublist. This step makes the array 'h-sorted'.
3. **Reduce Gap:** Repeat step 2 with a smaller gap size. The gap size is often reduced by half in each iteration. This step continues until the gap size becomes 1.
4. **Final Pass:** Finally, perform a normal insertion sort with a gap size of 1. This step ensures that the array is fully sorted.

Runtime Complexity:

- **Best Case**: When the array is already sorted, the total number of comparisons made during each interval is equal to the size of the array. In such cases, the best-case time complexity is Ω(n log(n)). This is because, with smaller gaps, the algorithm behaves similarly to insertion sort.
- **Average Case**: The average-case time complexity of Shell sort is often expressed as Θ(n log n). The exact time complexity is difficult to determine precisely due to the dependence on the chosen gap sequence. Empirical studies and analyses have shown that Shell sort tends to perform better than algorithms on average.
- **Worst Case**: The worst-case time complexity of Shell sort is O(n^2). This occurs when the gap sequence is not carefully chosen and results in a similar performance to insertion sort.

Space Complexity:

- **Best, Average and Worst Cases**: The space complexity of Shell sort is O(1), meaning it operates in constant space. This is because the sorting is done in place, without requiring additional memory allocation proportional to the size of the input array.

Runtime Analysis:

	EQUAL INTEGERS			RANDOM INTEGERS			INCREASING INTEGERS			DECREASING INTEGERS		
	1.000	10.000	100.000	1.000	10.000	100.000	1.000	10.000	100.000	1.000	10.000	100.000
dualPivot QuickSort	2.7873 ms	15.0411 ms	714.9408 ms	0.3496 ms	2.0935 ms	12.1577 ms	3.2595 ms	13.1614 ms	703.3942 ms	2.3198 ms	15.4150 ms	741.6190 ms
shellSort	0.2233 ms	2.4428 ms	4.7543 ms	0.4645 ms	3.9154 ms	12.3660 ms	0.2234 ms	2.9639 ms	4.9356 ms	0.3064 ms	2.2096 ms	5.8912 ms

DualPivotSort and Shell Sort algorithms perform differently in different data scenarios. For example, in cases of equal numbers, DualPivotSort works quickly with low processing costs, while Shell Sort offers a more balanced performance. While DualPivotSort's performance is variable and sensitive to data structure when working with random numbers, Shell Sort is generally efficient and runs fast. With increasing numbers, Shell Sort is effective on nearly sequential data, while DualPivotSort performs poorly because it has to rotate the data almost in reverse order. While Shell Sort generally runs fast with decreasing numbers, DualPivotSort's performance may be low because it has difficulty dealing with such arrangements. In conclusion, both algorithms have their strengths and weaknesses, and which algorithm to prefer depends on the data scenario.

Scenario A:

In this scenario, ranking algorithms play an important role to place students in universities because this process has to be done based on the exam scores and preferences of 3 million students. Ideally, an algorithm that can quickly sort such a large data set should be chosen. Considering the size of the data set, a sorting algorithm that provides the best performance in terms of time complexity should be preferred.

DualPivotQuickSort is an algorithm that generally works effectively on a large data set. However, since the worst-case complexity of this algorithm is O(n^2), its performance may vary depending on some characteristics of the dataset. Shell Sort, on the other hand, is generally known as a fast sorting algorithm and can work effectively on medium-sized data sets. However, it may not perform best for large data sets.

Based on this information, we can think that in the scenario of placing 3 million students, DualPivotQuickSort may be a faster and more effective option. Because this algorithm generally performs better on large data sets and has a better average case complexity. However, when making this decision, it is important to consider the characteristics of the data set and the worst-case performance of the algorithm to be used.

Scenario B:

In the given scenario, there is a Turkish-English dictionary data and we want to translate this data into English-Turkish. We consider a case where the data is already sorted alphabetically and there are thousands of words. In this case, we need to choose a suitable sorting algorithm to quickly sort and translate the data.

DualPivotQuicksort generally performs well on large data sets, while Shell Sort is especially effective on nearly ordinal data. While at first glance, DualPivotQuicksort's performance advantage on large data sets is striking, Shell Sort's advantage on nearly ordinal data stands out, given that the data is already sorted. There are several reasons why Shell Sort was chosen as a particular sorting algorithm. First, Shell Sort is a fast sorting algorithm in practice and performs especially well on nearly ordinal data. Since the data set is already sorted alphabetically, Shell Sort's suitability for this situation is very important.

Finally, looking at the given runtime table, it can be seen that Shell Sort performs well in certain scenarios. In particular, Shell Sort appears to run fast in the case of "Increasing Integers".

REFERENCES

- https://www.geeksforgeeks.org/dual-pivot-quicksort/
- https://stackoverflow.com/questions/20917617/whats-the-difference-of-dual-pivot-quick-sort-and-quick-sort
- http://codeblab.com/wp-content/uploads/2009/09/DualPivotQuicksort.pdf
- https://awdesh.medium.com/dual-pivot-quick-sort-javas-default-sorting-algorithm-for-primitive-types-77342e1df5e5
- https://www.youtube.com/watch?v=r3a25XPf2A8
- https://www.geeksforgeeks.org/shellsort/
- https://www.simplilearn.com/tutorials/data-structure-tutorial/shell-sort#:~:text=Shell%20sort%20is%20a%20sorting,moved%20to%20the%20far%20left.
- https://en.wikipedia.org/wiki/Shellsort
- https://www.youtube.com/watch?v=ddeLSDsYVp8&t=224s