INFO6205 Assignment Project
Literature Review

Zhilue Wang - 001522973 Shashwat Shrey - 002128122

Abstract. *In this project, our task is to implement different sorting algorithms for sorting Chinese names in pinyin order. The following report reviewed several existing sorting algorithms, such as QuickSort, TimSort, HuskySort, RadixSort and its improvement, and analyze their time and space complexity. We also had a breif discussion on these algorithms when sorting Chinese names.*

1. Sorting Methods

There are many sorting methods which can rearrange elements in order. Commonly used algorithms such as QuickSort and MergeSort can sort an large array efficiently compared to elementary sort like insertion sort. In general, sorting algorithms can be categorized into two groups, comparison based sorting and non-comparison based sorting. In the following sections, we will be introducing several sorting algorithms from both groups, with more focus on RadixSort particularly about its performance and possible improvements.

2. RadixSort

RadixSort is a sorting algorithm only for integers and its theory is to sort the elements by separating the integer into digits and distributing them into buckets according to the digit value. Because integers can also be used to present strings (like names or dates) and floating number with specific format, its application is not limited to integer sorting in practice. One of the RadixSort's characteristics is it does not involve comparison operations which are fundamental in other sorting methods such as Quicksort or Timsort.

Time complexity - RadixSort is usually considered a very fast algorithm. In theory, its time performance has constant growth rate $O(n)$. For each depth in RadixSort, it will iterate through both the original array with size n and the bucket array with size b. So the complexity for each depth is $O(n+b)$ and thus the total time complexity of the algorithm will be $O(d * (n + b))$ where d is the depth of each element. Usually the number of buckets b is designed to be very small and when n gets larger, the total complexity can be estimated as $O(d * n)$.

Space complexity - Depends on the implementation of RadixSort, it might use or not use an extra array to store counting result. [McIlroy et al., 1993] described two implementations of counting RadixSort (which performs distribution twice and the first time only counting the bucket size) in his paper. One of the implementations requires an extra array to store the result temporarily when moving objects. The space complexity of this implementation is $O(n)$. The other implementation will sort the array in-place without stability. It does not require any extra space so its space complexity is $O(1)$.

2.1. MSD and LSD RadixSort

There are two versions of RadixSort based on which end of the elements the algorithm starts sorting from. MSD RadixSort starts sorting from the most significant digit (usually

left to right), and LSD RadixSort starts from the least significant digit (right to left). MSD version is usually faster than LSD version as LSD starts at the digits that make the least difference and will require iterating through all digits, where MSD by nature does not require that. This property makes MSD RadixSort faster and more useful.

2.2. Number of buckets

One factor of the RadixSort's performance is the number of buckets b. Like most of the commonly used sorting methods, when the array size is small, RadixSort can be cut off to elementary sorts like insertion sort which have better performance on small arrays. The number of buckets b then will have impacts on which part of the time RadixSort will spend mostly on – insertion sort for small buckets or iterating through all elements and buckets. As we can imagine, when b becomes larger, we will spend more time on insertion sorts. But when b is very small, we will have much longer time to iterate through all digits. In our experiments, we tested two different values for b and checked their performance.

2.3. Cache usage improvement

For MSD RadixSort we used the classic two array implementation (which is taught in our lecture). A simple code example is here (copied from lecture slides):

```
int N = a.length;
int[] count = new int[R+1];
for (int i = 0; i < N; i++) // for loop A
    count[a[i]+1]++;
for (int r = 0; r < R; r++)
    count[r+1] += count[r];
for (int i = 0; i < N; i++) // for loop B
    aux[count[a[i]]++] = a[i];
for (int i = 0; i < N; i++)
    a[i] = aux[i];
```

[Kärkkäinen and Rantala, 2009] introduced a new improvement that will have less cache misses. Their new methods are based on the observation that both loop A and loop B are doing the same slow character access `a[i]`. They created a new array called oracle and store all `a[i]` into the oracle array:

```
int[] oracle = new int[N];
for (int i = 0; i < N; i++)
oracle[i] = a[i];
for (int i = 0; i < N; i++) // for loop A
    count[oracle[i]+1]++;
for (int i = 0; i < N; i++) // for loop B
    aux[count[oracle[i]]++] = oracle[i];
```

Because oracle array is stored sequentially in memory, this new code will generate fewer cache misses. We also tested this method in our experiment and the results showed that this method will indeed have better time performance.

3. Comparison based sorting

Comparison sorts are algorithms that make swaps or perform reordering based on comparison-based operations such as less than, greater than or equal to, to decide the correct position of an individual key in an array.

The requirement is that the operator forms a total order over the data with:

```
1. if a <= b and b <= c then a <= c (transitivity)
2. for all a and b, a <= b or b <= a (connexity)
```

A comparison sort must have an average case lower bound of $n \log n$ comparison operations, which is linearithmic.

Comparison based sorting algorithms offer the notable particle advantage that control over the comparison function allows sorting of many different datatypes and fine control over how the list is sorted.

3.1. TimSort

TimSort is a hybrid sorting algorithm, which is a combination of merge sort and insertion sort. This algorithm makes use of the pre-ordered data found in the real world sorting problems, and avoid sorting them again, by inserting them in the right positions, hence making use of insertion and merge sort to work as the fastest sorting algorithm for best case, and is being used as Python's default sorting algorithm.

Time complexity - The Time Complexity of TimSort is a result of the joint contribution of Merge sort and Insertion sort, making use of the naturally ordered elements in the real-world examples of random arrays.

Best Case: $\Omega(n)$ Worst Case: $\Theta(n \log n)$ Average Case: $O(n \log n)$

Space complexity - TimSort requires an extra array to store a sorted elements so the space complexity for TimSort is $O(n)$

3.2. QuickSort

Quick sort works on a divide and conquer principal, but it is not the same as merge sort, it works very well for a variety of different kinds of input data, and is substantially faster than any other sorting methods in typical applications. The desirable feature is that it's an in-place algorithm and requires only $n \log n$ on the average to sort an array of length n.

Time complexity - QuickSort works on a pivot-based comparison system, making an average of $2n \ln n$ compares to sort an array of n distinct elements. Also making $n^2/2$ compares in the worst case, that is when the array is sorted in an increasing or decreasing order.

Best Case: $\Omega(n \log n)$ Worst Case: $\Theta(n \log n)$ Average Case: $O(n^2)$

Space complexity - Quick Sort algorithm makes recursive calls which makes its space complexity $O(n)$.

3.3. Dual-Pivot QuickSort

The way dual-pivot QuickSort stands different from quick sort can almost be understood from its name, as the QuickSort uses one pivot , the dual pivot QuickSort actually uses two pivots, one on the left end of the array and one in the right end of the array. Then begins the partitioning into three parts. This is a little bit faster than QuickSort although the worst-case time complexity remains similar to that of QuickSort.

Time complexity - Dual Pivot QuickSort algorithm, because of its multi-partitioning operation, performs a little better than Quicksort

$$\text{Best Case: } \Omega(n \log n) \quad \text{Worst Case: } \Theta(n \log n) \quad \text{Average Case: } O(n^2)$$

Space complexity - Dual Pivot Quick Sort algorithm makes recursive calls which makes its space complexity $O(n)$.

3.4. HuskySort [Hillyard et al., 2020]

HuskySort works on the less noted factor of a sorting algorithm's performance, that is the array accesses. Most divide and conquer algorithms will divide works into two parts – linear part and linearithmic part. By moving works from linearithmic part to linear part, HuskySort reduces the number of array access and hence the processing time. HuskySort is a hybrid algorithm. It can combine dual pivot quick sort and TimSort to come up with an algorithm that outperforms both these techniques when used individually.

Complexity - According to the paper, Husky Sort has time complexity as $O(n \log n)$ and space complexity as $O(n)$. In our experiments, we found that HuskySort will indeed have around 40% improvement compared to TimSort, which is aligned with the paper's statistics.

4. Discussion

Above we introduced several commonly used sorting algorithms and several algorithms that have better performance than classic ones. When sorting Chinese strings, we will use Java Collator class for string comparison so that they can be sorted in pinyin order. For HuskySort, because it will encode the object into a 64 bits hash code, we will use CollationKey class to convert Chinese string to integers then it can be used to encode husky code. Same conversion is done for RadixSort.

For sorting Chinese names using HuskySort, although the bytes array converted form Chinese string might have size larger than 8 (which will make the husky encoding imperfect), we found out that most of name strings can still be distinguishable using the generated husky code. Thus, in theory, the performance of HuskySort will not lose a lot due to the imperfect encoding.

RadixSort is considered the fastest one in above algorithms, as it has linear run time. But due to the limitation that it can only sort integers, it cannot be used in most use cases. Here as we can use Collator to convert the Chinese names into a comparable bytes array, we will be able to use RadixSort and expect it to have the best performance among the algorithms.

References

Hillyard, R. C., Liaozheng, Y., and R, S. V. K. (2020). Huskysort.

Kärkkäinen, J. and Rantala, T. (2009). Engineering radix sort for strings. In Amir, A., Turpin, A., and Moffat, A., editors, *String Processing and Information Retrieval*, pages 3–14, Berlin, Heidelberg. Springer Berlin Heidelberg.

McIlroy, P. M., Bostic, K., and Mcilroy, M. D. (1993). Engineering radix sort. *COMPUTING SYSTEMS*, 6:5–27.