Quick Sort

William Frid

Fall 2023

Introduction

Choosing what sorting algorithm to use when writing a program can be tricky. It generally boils down to what type of data will be sorted and the size of the data set. That being said, one algorithm stands out from the rest due to its name; Quick Sort. This sorting algorithm is an efficient general-purpose, divide-and-conquer algorithm, and works similarly to a merge sort.

In this report, you'll read about how to construct a quick sort for the two data structures arrays and singly-linked lists (with a tail pointer). The algorithm will also be analyzed and benchmarked to conclude its time complexity with the goal of finding its weak and strong points.

Quick Sort In a Nutshell

As mentioned in the introduction, quick sort is a divide-and-conquer algorithm. That means it divides the problem into smaller problems, in this case, sorts part by part. Let's say we got a data set looking like so $[8, 2, 4, 7, 1, 3, 9, 6, 5]$. The algorithm then selects the right-most element as a pivot. Then it goes through each element from the left and moves each element that's larger than the pivot element to the right, leading to $[2, 4, 1, 3, 8, 7, 9, 6, 5]$. *Note how the order of the smaller and larger remains the same (2 came before 4, 4 came before 1, and so on).* Next, the pivot element is swapped with the first of the larger elements (that is 8 in this case). Now we have the data set $[2, 4, 1, 3, 5, 7, 9, 6, 8]$, and it's time to divide the problem and make usage or recursion.

The elements on the left side of the pivot are sent to this same method, meaning $[2, 4, 1, 3]$. A pivot element is selected (3), and it places the smaller numbers on the left and the larger ones on the right: $[2, 1, 3, 4]$. Then, once again, it sends the smaller elements to the same method. Now there's only the data set $[2, 1]$ which can be easily sorted.

When the elements have been sorted two by two, the data is passed upwards until the first call to the recursive function. Note that the data passed further via recursion is both the smaller and larger numbers. In

other words, the sorting method received $[8, 2, 4, 7, 1, 3, 9, 6, 5]$, and called upon itself twice, once with the data set $[2, 4, 1, 3]$ and once with $[7, 9, 6, 8]$.

Constructing The Quick Sort

The two upcoming subsections cover the construction of the quick sort algorithm for both arrays and singly-linked lists in Java. Note how the complexity of these two solutions differ and might have an effect on the upcoming benchmarks.

Array Quick Sort

As seen in the snippet below displaying method `quickSort()`, we see that quick sort doesn't require that much logic. The method can be divided into three main parts: The first part consists of simple if-statements that check if it should cancel the run immediately. This is done since there's no need to compare if there's only one element. Second is the the one built on a while loop. This section sorts the elements so that the smaller elements are placed on the left side of the pivot, and larger elements on the right side. Finally, in the third section, the smaller elements are sent to the method, and larger as well, in two separate calls. This is where the recursion comes into play.

```
static void quickSort(int[] arr, int i1, int i2) {

    if (i1 == i2 || i1 > i2)
        return;

    int pivot = i2;
    int newPivot = pivot;

    int i = i1-1;
    int j = i1;
    while (arr[j] != arr[pivot]) {
        while (arr[j] > arr[pivot])
            j++;
        if (i != j)
            i++;
        else
            j++;
        if (arr[i] > arr[j] && i != j) {
            swap(arr, i, j);
            if (j == pivot)
                newPivot -= (j-i);
        }
    }
```

```
    partition(arr, i1, newPivot - 1);
    partition(arr, newPivot + 1, i2);
}
```

Singly Linked List Quick Sort

Since the algorithm remains the same for the two types of data, we can expect that the methods are somewhat similar. If we look at the snippet below we see that it is indeed fairly similar to the previous one. It too has three main parts: The first one is to check if it should run at all. The second part moves the nodes smaller and larger than the pivot element to two separate lists. These two lists are later (in part three) sent to the `quickSort()`, thus creating the recursion. Lastly is the additional logic in part three which assembles the whole list again as it was split up in section two.

```
public void quickSort() {

        if (head == null || head.next == null)
                return;

        Node pivot = tail;
        LinkedList smaller = new LinkedList();
        LinkedList larger = new LinkedList();

        Node t = head;

        while (t != pivot) {
                Node tt = t.next;
                if (t.item.compareTo(pivot.item) > 0) // Larger.
                        larger.add(t);
                else // Smaller.
                        smaller.add(t);
                t = tt;
        }

        if (smaller.head != null) {
                smaller.quickSort();
                head = smaller.head;
                smaller.tail.next = pivot;
        } else {
                head = pivot;
        }
```

```
        if (larger.head != null) {
                larger.quickSort();
                pivot.next = larger.head;
                tail = larger.tail;
        } else {
                tail = pivot;
        }
}
```

Benchmarking

Let's start off by analyzing the algorithm to find its big-O complexity for the average case. We know that for each recursion, it'll need to loop through each element, giving us n actions. Then it'll divide the data set into two (divide-and-conquer), and then repeat this process. Dividing the data set into two and sending those to the method again (recursion) gives us the behaviors of a logarithmic complexity $log(n)$. These two combined give us the complexity of $O(n * log(n))$.

To verify this calculation of the big-O complexity, a benchmark is needed. The benchmark run used different sizes of randomly generated data sets (with unique elements) that were sorted 1000 times to find the best results with the least interference. The results shown in Table 1 clearly prove in both the column `Arr.(us)` and `LL(us)` (LL standing for Linked List) that the algorithm is indeed logarithmic like the calculation said. Furthermore, if we look at Figure 1 we can easily compare the behavior of the results to the calculated big-O complexity (marked as green). The figure clearly shows how the curves are running in parallel thus confirming the results found the Table 1.

Size	**Arr.(us)**	**Size/Arr.**	**LL(us)**	**Size/LL**
32	1	35.13	2	16.73
64	2	27.41	3	24.57
128	60	23.27	6	20.44
256	10	19.55	15	17.10
512	30	17.28	34	14.96
1024	65	15.64	72	14.13
2048	160	13.18	181	11.30
4096	310	13.04	392	10.45
8192	760	10.77	988	8.29
16384	1680	9.76	2192	7.47

Table 1: Benchmark comparing the average case.

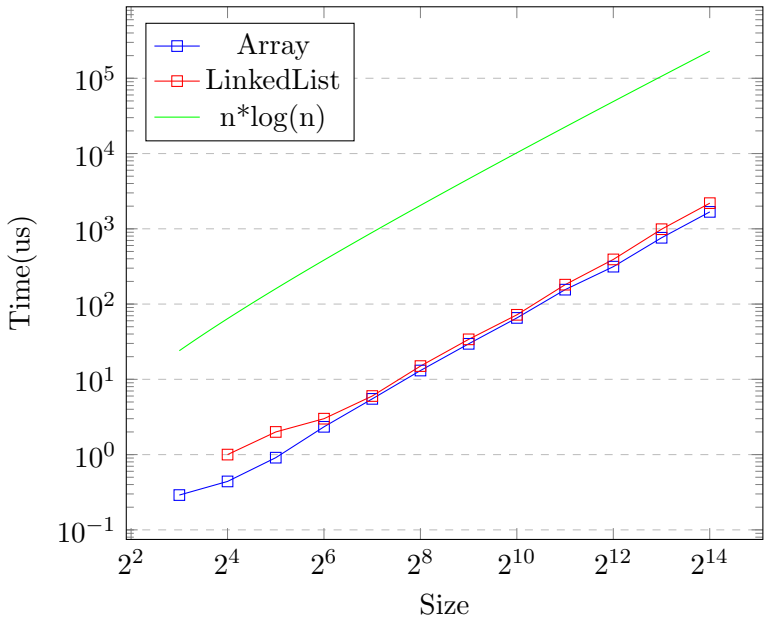

Figure 1: Benchmark results visualized next to time complexity function.

Discussion

But why is the algorithm faster when using an array? This can be due to different reasons. Some are how the code is written, and the data is being read/written. If we look at how the code is written, it's clear that the method for the linked list is longer and might thus contribute to the slower speed. This could probably be prevented by splitting up the code somewhat. Secondly, when allocating memory for an array, the memory allocated is close to each other, giving a great advantage when it comes to cashing thus speeding up the quick sort of arrays. On the other hand, nodes (used in the linked list) can be spread out more across the memory and thus lead to slower reads.

An additional benchmark was run for the worst-case scenario. Its results aren't included in the report since the big-O complexity for quick sort is then different. But it's worth pointing out that this benchmark proved that the linked list performs faster than the array when using quick sort in the worst-case scenario.

Looking at the benchmarking itself there were some issues with memory. Every three to four changes of size it seemed to halter and slow down leading to inconsistent results. This behavior was traced to the garbage collector which proved hard to solve. The simplest and quickest solution was to switch machines, which solved the issue and suddenly gave clear benchmark results.

Conclusion

Quick sort is a great algorithm for certain cases with some of its advantages being that it doesn't copy any data making it memory-efficient, high speeds for large data sets (as we saw in the big-O complexity), and the facts that it's general purpose and can be used to sort most types of data. The drawbacks are that it doesn't perform as well with almost ordered data and smaller data sets. It turns out to be better suited with arrays than linked lists, likely due to the speed of reading the data upon comparison.