Sorting an array

Fredrik Berzins

Fall 2023

Introduction

In this assignment we will explore some search algorithms, all have their pros and cons and it's important to know when to use which. We will in this assignment only work with arrays of a given size and not add any new elements to the data set.

Sorting explanation

A sorting algorithm will in a structured way work thru the algorithm until all elements in the array are in order. When it comes to sorting most methods have a memory complexity of O(1) or O(n), this is important in some circumstances but for normal use the time it take is more important, and how the time complexity is different for best, average and worst case.

Stable

A Stable sorting algorithm keeps the previous order where possible. For example when sorting a deck of cards by value and not suite, the order of the fives will be the same as before. This is only relay useful if their was some order to the input array. It can also be useful when resorting by a new metric. A common examples of this is when looking at a list of product online and changing sorting from alphabetic to price this will put them in price order but items with the same price will be in alphabetical order.

Sorting implementation

When sorting a data set, their are multiple algorithms to choose from for example merge sort or the simplest method selection sort. To compare different algorithms Big O notation is used, this is a way of easily compare how long a search will take for different algorithms and data set sizes.

Selection sort

Selection sort is the simplest type of sorting where it looks fro the smallest and puts it first, then the second smallest and puts it in second place and so on. When sorting the algorithm only needs extra memory to store one value while moving values, this gives it a O(1).

Insert sort

Insert sort works by checking the next value in the array and comparing it to see if it's smaller then the current number, if so swap the numbers and redo until the value below the current are in order. This algorithm only uses one extra space to store a temporary value while swapping values so it has a memory complexity of O(1).

Merge sort

Merge sort works by splitting the array and sorting the halves then merging them together, this is shown below. The sorting is done recursively with the same algorithm so in principle it splits the array down to one value arrays and compares them when merging them. With how to final merging is done it needs a second array this gives it a memory complexity of O(n).

```java
private static void sort_merge(int[] array, int[] aux, int lo, int hi) {
    if (lo != hi) {
        int mid = (lo + hi)/2;
        sort_merge(array, aux, lo, mid);
        sort_merge(array, aux, mid + 1, hi);
        merge_merge(array, aux, lo, mid, hi);
    }
}
```

Low copy merge sort

This is a modified version of merge where it avoids some of the copying between original array and auxiliary array. This is done by making a copy of the input array as the first step and then sort the auxiliary array and return it to the original array, this means the auxiliary array and the original array have the same content. When that's is done the auxiliary array can be sorted and let the result be returned/stored in the original array, as seen below. We can also remove the code that copies the original array to the auxiliary in the merging function.

```java
private static void sort_lowCopyMerge(int[] array, int[] aux, int lo, int hi) {
    if (lo != hi) {
        int mid = (lo + hi)/2;
        sort_lowCopyMerge(aux, array, lo, mid);
        sort_lowCopyMerge(aux, array, mid + 1, hi);
        merge_lowCopyMerge(array, aux, lo, mid, hi);
    }
}
```

Benchmarks

The testing was done with data sets between 100 and 6400 values(doubling each time, 100,200,400...). Each time test was redone 1'000 times witch included sorting 1'000 different arrays to get as accurate results as possible. This gives a min time showing avg from the group with the 1'000 quickest runs.

Result

The results from table 1 shows that merges sort is significantly faster then both selection and insert sort, it also shows that insert sort is takes about 40%-80% of the time selection sort uses. Both selection and insert sort has the same time complexity of O(n^2) while the merge sort has a consistent O($n * log(n)$) these are the average/worst case while the best case doesn't differ with selection or merge sort but best case for insertion sort is O(n). When it comes to comparing merge and the modified low copy merge sort it shows that it takes about 90% of the time.

Array length	**Selection**	**Insert**	**Merge**	**Low copy merge**
100	3850	1523	2914	2608
200	11149	5145	6853	6047
400	34086	19060	14830	13570
800	112733	73633	33784	30506
1600	397232	290720	72204	67094
3200	1451600	1140080	157426	149836
6400	5516391	4587772	334439	322900

Table 1: Shows time to sort n elements in an arrays with different algorithms in ns.

Conclusion

In conclusion merge sort is a good choice when consistency and time is valued and their is an abundance of memory. It is not necessarily the best suited since quick sort for example uses less memory, but has a worse worst case of O(n^2) compared to merge sort O($n * log(n)$). Another thing to keep in mind with quick sort is that it's not stable. When comparing selection and insertion sort the big differences are the, best case time complexity of O(n^2) for selection and O(n) for insertion sort. The other difference is that insertions sort is stable and selection sort is not, This isn't necessarily a down side for selection sort but its good to know since in some cases a stable i more suited for the task. When comparing the merge with the modified version(Low copy merge). Sins the low copy merge sort is about faster then the normal merge sort and still has a Big O notation of O($n * log(n)$). Low copy merge still uses O(n) memory so it doesn't have a better memory complexity. So in conclusion the low copy merge is over all better since its faster and uses the same amount of memory.