Quicksort

Hannes Mann

2022-10-16

Introduction

This is a report for the second higher grade assignment in the course ID1021 Algorithms and Data Structures. In this assignment, the quicksort algorithm's performance characteristics should be explored with arrays.

The implementation was done in C++ and was compiled with GCC using the `-O0` parameter to ensure the compiler doesn't optimize away any part of the code.

The source code can be found at: GitHub.

Implementing quicksort

Quicksort is a recursive sorting algorithm that works by repeatedly dividing a list into two, with an arbitrarily chosen "pivot" element as the point where the list splits. This means a "partition" method is needed that can split a list in-place and keep track of where the pivot element ends up.

For this report the first element in the list is always chosen as the pivot element, but these methods should work with any pivot element.

With an array

The quicksort implementation for arrays closely follows the suggestions in the assignment. We define two indices to hold the first element before the pivot and the first element after or equal to the pivot. These are called `before_pivot` and `after_or_equal_pivot` in the code.

The implementation works by repeatedly scanning the array from both ends and, if there is still a valid range where items are not yet sorted, moving elements from one "side" to the another by swapping them. This of course doesn't guarantee the each side of the array is sorted but they will all follow the rule that they are either smaller or bigger than the pivot element.

Once this is done the pivot element is inserted at `after_or_equal_pivot` which guarantees this is now the new pivot position and the quicksort method can split the array up from this point.

With a linked list

A linked list makes quicksort easier to implement since the positions in the array are flexible. Rather than swapping elements we can simply reprogram the existing nodes to make new lists without additional allocations or copying.

The algorithm works by leaving the first element in the list untouched and starting from the second element. Elements are pushed "backwards" into one of two lists depending if they are smaller, bigger or equal to the pivot element. This means that, for example, the first element encountered that is smaller than the pivot element becomes the *tail* of `before_pivot`, not the *head*.

```
while(next) {
    LinkedList<int>* node_to_advance_to = next->next;

    bool smaller_than_pivot = next->item < list->item;
    LinkedListReference<int>* list_to_push_into =
        smaller_than_pivot ? &before_pivot : &after_or_equal_pivot;

    next->next = list_to_push_into->begin;
    list_to_push_into->begin = next;

    if(!list_to_push_into->end) {
        list_to_push_into->end = list_to_push_into->begin;
    }

    next = node_to_advance_to;
}
```

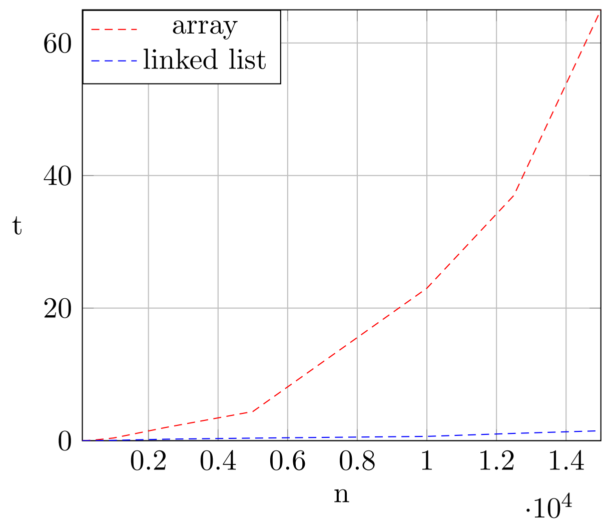
After the list has been partitioned it needs to be reassembled. This list is reassembled from any of the lists that exist in the order (`before_pivot`, `list`, `after_or_equal_pivot`). All lists obviously don't need to exist if, for example, there are no elements smaller than the pivot. The implementation keeps track of tail pointers for all lists so appending happens in $O(1)$ time.

Benchmarks

Benchmarking was done by generating n random numbers and comparing the time it takes to sort a list containing these numbers. The same sequence is used for the array and linked list implementation.

n	**array**	**linked list**
100	0.0076 ms	0.0037 ms
250	0.038 ms	0.011 ms
500	0.13 ms	0.024 ms
1000	0.41 ms	0.055 ms
2500	2.0 ms	0.23 ms
5000	4.4 ms	0.39 ms
10000	23 ms	0.65 ms
12500	37 ms	1.1 ms
15000	65 ms	1.5 ms

Table 1: Execution time for sorting a random list with n elements.



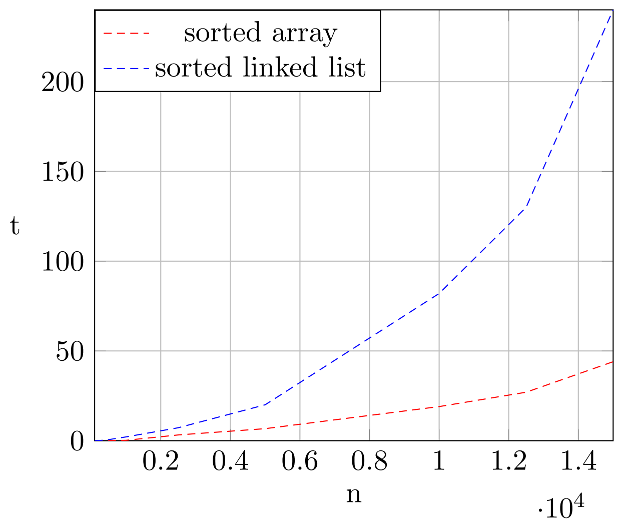
As can be seen from the results the linked list implementation ends up a lot faster when sorting random arrays. The linked list quicksort scaled roughly linearly ($O(n)$) but the array implementation needs many loop iterations for every element and this will scale with n as the search starts in both ends of the array. The expected average performance of quicksort is $O(n \log n)$ but the partitioning is clearly the dominating factor here for both implementations. Recursion only needs to happen a few times on average.

Sorted data

There is one scenario where the linked list implementation falls short and that's when sorting data that has already been sorted before.

n	sorted array	sorted linked list
100	0.0037 ms	0.0088 ms
250	0.016 ms	0.11 ms
500	0.060 ms	0.58 ms
1000	0.21 ms	2.1 ms
2500	3.2 ms	7.2 ms
5000	6.7 ms	20 ms
10000	19 ms	82 ms
12500	27 ms	130 ms
15000	44 ms	240 ms

Table 2: Execution time for sorting a sorted list with n elements.



The array implementation performs as expected but the linked list sort scales *horribly* as n increases (this benchmark took around 2 minutes to run on a modern processor from 2020, and that's only with 15000 elements). This function becomes $O(n^2)$ when data is sorted.

The reason for this is the choice of the pivot element. When the data is already sorted, the pivot element will always be the smallest element and what the linked list ends up doing is repeatedly creating smaller and smaller lists of size $(n-1)$ until the data is sorted. The array implementation doesn't suffer from this issue because it moves the pivot element around. The first element isn't necessarily the "smallest" after the first iteration.

One way to solve this is to pick a pivot element at random instead of always using the element at index 0 as the pivot.