Exercise 6 Report

David Malášek
Born: 13 May 2004

November 26, 2025

DVGA12 Programming and Data Structures
Computer Science
Faculty of Health, Science and Technology

I hereby affirm that this report represents my own independent work and that I have neither given nor received unauthorized assistance in its completion.

Contents

1 Introduction

This report represents the implementation of five common algorithms in the C programming language: bubble sort, insertion sort, quick sort, linear search and binary search. The purpose of this project is to analyze how these algorithms behave in the best, average and worst cases, in other words, to compare their time complexity.

The analysis is based on the results from a testing program provided with the assignment.

2 Description of the algorithms

This section briefly explains the function of the five algorithms and their running time.

2.1 Bubble sort

Bubble sort goes through the input array many times. In each pass, it compares neighbor elements and performs a swap of these elements if they are in the wrong order. In this implementation, there is also a flag that reflects if a swap occurred in the current pass. If there was no swap, the array is already sorted and the algorithm stops early, which makes the algorithm a little bit more efficient.

- **Best case:** $O(n)$
  This happens when the array is already sorted. Bubble sort goes through the array once, does not swap anything and finishes.
- **Average case:** $O(n^2)$
  For an unsorted array the algorithm peforms a lot of passes and comparisons, so the time grows rapidly when the array gets larger.
- **Worst case:** $O(n^2)$
  Example situation could be when the array is sorted in the opposite direction. Then many swaps are needed and bubble sort is slow.

Bubble sort is rather simple to implement, but it is not a good choice for large arrays because of the quadratic time.

2.2 Insertion sort

Insertion sort builds the sorted array step by step. It starts from the second element and considers the items on the left as already sorted. Then it takes the current element and moves it to the left until it finds the correct position.

- **Best case:** $O(n)$
  If the input array is already sorted, each new element is placed immediately and we only do one comparison per element.
- **Average case:** $O(n^2)$
  For random data we usually have to move each element a number of positions to the left, which gives a quadratic number of operations overall.
- **Worst case:** $O(n^2)$
  If the input is sorted in reverse order, almost every element has to move all the way to the beginning of the array.

Compared to bubble sort, insertion sort is still quadratic, however, it is faster especially when the array is already almost sorted.

2.3 Quick sort

Quick sort picks the middle element of the current range as a pivot. Then, it rearranges the elements so that everything smaller than the pivot goes to the left and everything larger goes to the right. After this, it calls itself recursively on the left part and on the right part.

- **Best case:** $O(n \log n)$
  This case happens when the pivot splits the array into two similar halves each time.
- **Average case:** $O(n \log n)$
  For random data and a middle pivot, the partitions are usually not too unbalanced, so in practice the running time also follows this behavior.
- **Worst case:** $O(n^2)$
  In theory, if the pivot is not ideal (e.g. always the smallest value), the partitions are extremely unbalanced and the algorithm can get quadratic time. With the middle pivot, this is unlikely but still possible on specially prepared inputs.

Quick sort is more complicated than bubble or insertion sort, however, it is much faster for large arrays.

2.4 Linear search

Linear search goes through the array from the beginning to the end and compares each element with the searched value. In case it finds it, it returns `true`, if it reaches the end, it returns `false`.

- **Best case:** $O(1)$
  The value is at the first position.
- **Average case:** $O(n)$
  On average we have to check about half of the array.
- **Worst case:** $O(n)$
  The value is at the very end or it does not appear at all, so we check all elements.

Linear search does not need a sorted array, but it is not fast on large arrays.

2.5 Binary search

Binary search requires a sorted array. Instead of checking all values one by one, it checks the middle element of the current range. If the middle is smaller than the target, it continues within the right half, otherwise it continues within the left half. It repeats this process until it finds the value or until the range becomes empty.

- **Best case:** $O(1)$
  The target value is exactly in the middle.
- **Average and worst case: about** $O(\log n)$
  In each step the search range becomes half as big as before, therefore the number of steps grows slowly with the size of the array.

Binary search is much faster than linear search on large sorted arrays, however, it requires a sorted array.

3 Experimental results

A program for measuring running times was provided along with the assignment. This section presents the results from this program.

3.1 Sorting algorithms

Bubble sort

Table 1 shows the running times of bubble sort. It is clear that the best case (already sorted input) is much faster than the average and worst cases, which grow very quickly with the input size.

Table 1: Running times of bubble sort for different input sizes n (in seconds).

n	Best case	Average case	Worst case
512	0.00000375	0.00119400	0.00132500
1024	0.00000500	0.00375225	0.00374025
2048	0.00000950	0.00856025	0.01000225
4096	0.00001875	0.02403550	0.02575450
8192	0.00003625	0.10291425	0.10419675
16384	0.00007425	0.46810700	0.39825800

Insertion sort

Table 2 displays the running times for insertion sort. Just like bubble sort, it is rapid in the best case (already sorted) and much slower in the average and worst cases. However, for the same n it is generally faster compared to bubble sort.

Table 2: Running times of insertion sort for different input sizes n (in seconds).

n	Best case	Average case	Worst case
512	0.00000675	0.00037825	0.00076525
1024	0.00000950	0.00141775	0.00247025
2048	0.00002025	0.00367250	0.00561300
4096	0.00003550	0.00835800	0.01333425
8192	0.00007100	0.02137225	0.04168950
16384	0.00014200	0.07975900	0.16811750

Quick sort

Table 3 reflects the running times for quick sort. Here the best, average and worst cases are all in a similar range and the growth with n is much slower than with bubble sort or insertion sort.

Table 3: Running times of quick sort for different input sizes n (in seconds).

n	Best case	Average case	Worst case
512	0.00003650	0.00007675	0.00002075
1024	0.00004725	0.00015100	0.00004275
2048	0.00009550	0.00032125	0.00009075
4096	0.00018950	0.00052250	0.00019950
8192	0.00039225	0.00130175	0.00042325
16384	0.00089450	0.00289625	0.00083700

Looking at all three tables together, we can see that for large n quick sort is much faster than bubble or insertion sort algorithms.

3.2 Searching algorithms

Linear search

Table 4 shows the running times for linear search. Even in the worst case, the absolute values are very small, but the time still grows with n, which corresponds with the $O(n)$ behavior.

Table 4: Running times of linear search for different input sizes n (in seconds).

n	Best case	Average case	Worst case
512	0.00000225	0.00000375	0.00000575
1024	0.00000025	0.00000250	0.00000500
2048	0.00000050	0.00000575	0.00001050
4096	0.00000100	0.00001275	0.00001675
8192	0.00000100	0.00001875	0.00003450
16384	0.00000075	0.00004325	0.00006950

Binary search

Table 5 displays the times for binary search. The running times are not only very small, but they also change very little as n increases, which matches the expected behavior of an $O(\log n)$ algorithm.

Table 5: Running times of binary search for different input sizes n (in seconds).

n	Best case	Average case	Worst case
512	0.00000250	0.00000275	0.00000250
1024	0.00000125	0.00000100	0.00000075
2048	0.00000075	0.00000175	0.00000100
4096	0.00000175	0.00000075	0.00000100
8192	0.00000025	0.00000075	0.00000075
16384	0.00000175	0.00000175	0.00000175

4 Discussion

For the sorting algorithms, the tables clearly show the difference between quadratic and $n \log n$ time complexity. Bubble sort and insertion sort both have very small times in the best case (already sorted input), and the growth of their best case times looks almost linear. However, as soon as we look at the average and worst cases, the running times increase much faster. For example, for $n = 16384$ bubble sort needs about 0.47 seconds on average, while insertion sort needs about 0.08 seconds. This matches the $O(n^2)$ behavior.

Quick sort, on the other hand, behaves very differently. Its running times in the best, average and worst cases are all in the same small range, and the growth with n is much slower. For $n = 16384$ the average time is only about 0.0029 seconds, which is far below the times of bubble sort and insertion sort for the same size. This corresponds with the expected $O(n \log n)$ complexity.

Regarding the searching algorithms, linear search times grow with n in all three cases, especially in the worst case where the algorithm has to scan the whole array. This lines up with the $O(n)$ complexity.

Binary search, on the other hand, stays extremely fast even for the largest input size. The times in Table 5 change very little as n increases, which is what is expected from an $O(\log n)$ algorithm.

5 Summary and conclusions

This report presents the implementation and running time analysis of five algorithms:

- sorting algorithms: bubble sort, insertion sort, quick sort
- searching algorithms: linear search, binary search

All of the algorithms were tested on arrays of several sizes and for different input cases (best, average and worst), and their running time was measured in seconds.

Here is a conclusion regarding the testing of sorting algorithms.

- Bubble sort and insertion sort are rapid only in the best case (already sorted input). In the average and worst cases, their running times grow roughly quadratically with the input size and become very large for $n = 16384$.
- Insertion sort is usually faster than bubble sort for the same n, but its running time still grows with n^2 and has the same $O(n^2)$ complexity.
- Quick sort is much faster compared to both bubble sort and insertion sort on large arrays. Its running time grows roughly like $n \log n$ and stays below a few milliseconds even for the largest input.

A conclusion for the testing of searching algorithms is presented below:

- Linear search is simple and works on any array, but its running time increases with n in all three cases, which matches the $O(n)$ complexity.
- Binary search is extremely fast on sorted arrays. The measured times stay tiny and almost flat as n grows, which is consistent with an $O(\log n)$ algorithm.

Overall, the tests confirmed the theoretical complexity results in practice. Quadratic algorithms quickly become slow for large inputs, while $n \log n$ and $\log n$ algorithms remain efficient.

6 Lessons learned

While working on this project I learned several useful things about algorithms. I already knew the theory of Big-O complexities, but seeing the actual running times on real data helped me better understand what $O(n^2)$ or $O(n \log n)$ mean in practice. I also experienced how strongly the behavior of the algorithms depends on the input data: whether the array is sorted, random or reversed, it makes a clear difference. Simple algorithms like bubble sort or linear search are not difficult to implement and are fine for small arrays, but for larger inputs more advanced algorithms such as quick sort and binary search are much more efficient.

Overall, the project gave me a more concrete feeling for algorithm efficiency and useful practice with implementing and testing algorithms in C.