OPAL

Overpowered Algorithm Library

Rafał Żelazko

March 11, 2023

Contents

1 Introduction

1.1 What OPAL is and What it is not

OPAL stands for Overpowered Algorithm Library. It is a collection of algorithms and data structures packaged as a header-only library for easy use with C++ 20 and higher. OPAL, however, is not a replacement for the standard library. It is not meant to replace any of the standard components as it is not optimized for speed in every way possible. It should rather be regarded as a learning tool and reference implementation for people who want to learn about algorithms and data structures.

1.2 Why You Should Choose OPAL

There are many other libraries that provide algorithms or data structures, but most of them are either too complicated or contain too few or no helpful comments at all. First and foremost, OPAL aims to provide an easy to understand implementation built on good coding practices. Every algorithm and structure in OPAL is implemented using modern C++ template techniques. This allows for vast flexibility in use and makes it easier to extend the library with new functionality.

1.3 The Purpose of OPAL

OPAL is a project that I started in order to build a framework that would help me easily create algorithm implementations for lectures at my university. I wanted to refresh my knowledge of modern C++, and this is why I decided to use it as the core programming language in OPAL. I hope that OPAL will be useful to other people as well, and I will be happy to receive any feedback or suggestions.

2 Insertion Sort

2.1 Description

Insertion sort is a simple sorting algorithm that builds the final sorted array linearly. It is much less efficient on large lists than more advanced algorithms such as quicksort or merge sort. Insertion sort, however, provides certain advantages:

- It is in-place: only a constant amount of additional memory is needed.
- It is stable: equal elements will never be reordered
- It is online: it can sort a list as it receives it.

Because of its stability and portability, insertion sort is often used as a subroutine in more advanced divide-and-conquer algorithms.

2.2 Procedure

The canonical implementation of the algorithm [1]:

```
1: A ← ...
2: for i = 2, 3, ..., len(A) do
3:     a ← A[i]
4:     j ← i − 1
5:     while j ≠ 0 and a < A[j] do
6:         A[j + 1] ← A[j]
7:         j ← j − 1
8:     end while
9:     A[j + 1] ← a
10: end for
```

1. The procedure starts by focusing each element starting from the second one up to the last one. (line 2)
2. The currently focused element is saved to a separate variable as it might be overwritten in subsequent steps. (line 3)
3. Previous elements are shifted one position to the right until the right spot for the focused value is found. (line 5)
4. Finally, the focused value is inserted in the old place of the most recently shifted element. If no elements were shifted, the focused value will be inserted in the same place where it has been. (line 9)

2.3 Computational Complexity

To analyze the computational complexity of insertion sort, one must count the number of times that both the outer for-loop and the inner while-loop are iterated. The number of outer-loop iterations is equal to the number of elements in the array minus one. The number of inner-loop iterations ranges from one to the number of elements in the array minus one, but it also depends on the order of the elements in the array. However, the average number of inner-loop iterations is approximately equal to the number of elements in the array divided by two. This results in the following formulae:

$$\begin{aligned} N_i &= n - 1 \\ N_j &= \frac{1}{2}n \end{aligned}$$

The best case scenario occurs when the array is already sorted. In this case, the algorithm will perform only N_i outer-loop iterations and no inner-loop iterations at all.

$$\begin{aligned} C(n) &= N_i + N_i \cdot 0 \\ &= n - 1 \\ &= \Omega(n) \end{aligned}$$

The worst case scenario is when the array is sorted in reverse order. In this case, the algorithm will perform N_i outer-loop iterations and each time around N_j more inner-loop iterations.

$$\begin{aligned} C(n) &= N_i + N_i \cdot N_j \\ &= (n-1) + \frac{1}{2}n(n-1) \\ &= \frac{1}{2}n^2 + \frac{1}{2}n - 1 \\ &= O(n^2) \end{aligned}$$

The algorithm performs similarly in the average case scenario as well. This produces the following table:

Computational Complexity		
Best	**Worst**	**Average**
$\Omega(n)$	$O(n^2)$	n^2

It is worth noting that such low computational complexity in the best case scenario for this algorithm makes it an excellent choice to use with nearly sorted arrays as the performance hit is negligible.

3 References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press, 2001.