OPAL
Overpowered Algorithm Library

Rafał Żelazko

April 14, 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, ..., LENGTH(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:

$$N_i = n - 1$$

$$N_j = \frac{1}{2}n$$

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.

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

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.

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

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   Merge Sort

3.1   Description

Merge sort is a more advanced sorting algorithm that takes advantage of the famous divide-and-conquer strategy. Because of this, merge sort is much more efficient than insertion sort and other simple algorithms. Merge sort provides several advantages:

- It is stable: equal elements will never be reordered.

- It is parallelizable: various parts of the algorithm can be run concurrently without data races or loss of stability.

- It is optimal: the average time complexity is $O(n \log n)$.

However, merge sort is not without its drawbacks. It is not in-place as it requires additional memory for merging. It is also not online and is way slower with small arrays than most $O(n^2)$ algorithms.

3.2   Procedure

Common implementation of the algorithm:

```
 1: A ← ...
 2: A ← MERGE-SORT(A, 1, LENGTH(A) + 1)
 3: function MERGE-SORT(A, l, u)
 4:     if u − l > 1 then
 5:         m ← ⌊(l + u) ÷ 2⌋
 6:         MERGE-SORT(A, l, m)
 7:         MERGE-SORT(A, m, u)
 8:         MERGE(A, l, m, u)
 9:     end if
10: end function
```

1. The algorithm begins by checking whether the array is long enough and quits instantly if the array is already sorted (i.e. contains a single element). In this implementation u is not included in the sorting range. (line 4)

2. If the array needs to be sorted, it is split into two sub-arrays of roughly the same length, each one of which is sorted recursively using merge sort itself. (lines 6 and 7)

3. Finally, the two sorted sub-arrays are merged into one. The merge procedure is the most important part of the algorithm as it is the major factor in its time complexity. Amazingly, merge can be carried out in $O(n)$ time, which is the reason of merge sort's great performance. (line 8)

```
1: function MERGE(A, l, m, u)
2:     B ← empty array
3:     i ← l
4:     j ← m
5:     while i < m  &  j < u do
6:         if A[i] ≤ A[j] then
7:             B ← B + A[i]
8:             i ← i + 1
9:         else
10:            B ← B + A[j]
11:            j ← j + 1
12:        end if
13:    end while
14:    B ← B + A[i ... m)
15:    B ← B + A[j ... u)
16:    A[l ... u) ← B
17: end function
```

1. The merge procedure begins by allocating an auxiliary buffer that will store the result of merging.

2. Next, two iterators, i and j, are initialized. They keep track of the current progress in two of the sub-arrays. (lines 3 and 4)

3. The two sub-arrays are then merged into one by comparing the elements at indices i and j. The smaller element is added to the buffer and the index of the sub-array it came from is incremented. (line 5)

4. If one of the sub-arrays is exhausted, the remaining elements of the other sub-array are added to the buffer. (lines 14 and 15)

5. Finally, the buffer is copied back to the right position in the original array. (line 16)

3.3 Computational Complexity

The computational complexity of the merge sort is a more complicated matter. The algorithm is based on the divide-and-conquer strategy, which means that it utilizes recursion. On each level of recursion, the problem of size n is divided into two sub-problems of roughly the same size $\frac{n}{2}$. Then, the results of the two sub-problems are joined together in $O(n)$ time. This results in the following recurrence relation:

$$T(n) = 2T(n/2) + O(n)$$

The master theorem can be utilized in order to calculate the complexity of this algorithm:

$$a, b = 2 = const$$
$$\log_b a = \log_2 2 = 1$$
$$f(n) = n = n^1 = n^{\log_b a}$$
$$\Rightarrow T(n) = \Theta(n \log n)$$

It follows that the worst, the best, and the average case complexity of merge sort all are $\Theta(n \log n)$ which is the best possible performance for any comparison-based sorting algorithm.

$$T(n) = \Theta(n \log n)$$

3.4 Optimizations

In order to push the performance of merge sort even further, two concurrent execution threads can be spawned at each level of recursion. This is possible because both sub-arrays do not overlap at all which in turn eliminates data dependencies. The two threads can then sort their respective sub-arrays independently and let the supervising thread merge the results.

For optimal performance the total number of bottom-level threads should be a little higher than the number of CPU cores as possible, but not too high as scheduling too many concurrent jobs might cause a significant overhead.

Thread count limiting can be implemented by checking whether the sub-array is long enough to be put on two extra threads. If it is not, the sub-array is sorted entirely on the current thread. The array length threshold below which the sub-array is not split can be calculated before the algorithm is executed:

$$N_{jobs} = 2^{\lceil \log_2 N_{threads} \rceil}$$

$$n_{threshold} = \max(n \div N_{jobs}, 1024)$$

Ideally, the worker threads should be stored in a thread pool and reused for subsequent merge sort calls. This way, the overhead of creating and destroying threads is eliminated.

Temporary merge buffers can also be pooled, but for most uses it is enough to just allocate a single buffer of size n at the beginning of the algorithm and reuse it for all merge calls. In this scenario each thread offsets its working area in the buffer to $[l, u)$ which guarantees that each merge operation will get its own isolated piece of the buffer.

Finally, other $O(n^2)$ sorting algorithms can be used as fallback for even smaller sub-arrays as they typically are much faster at solving smaller sorting problems. For this purpose, OPAL makes use of insertion sort with a threshold of $n = 64$ which decimates the running time of the algorithm by 20% on average.

4 Heap Sort

4.1 Description

Heap sort is a $O(n \log n)$ comparison-based sorting algorithm, based on the heap data structure. It is a bit more complicated than single-threaded merge sort and does not exceed in performance. However, heap sort does not require any additional memory and is therefore a good alternative to merge sort when memory is scarce. Some notable properties of heap sort are:

- It is in-place: only a constant amount of additional memory is needed.

- It is not stable: equal elements can be reordered.

- It is optimal: the average time complexity is $O(n \log n)$.

Finally, heap sort is an excellent improvement over $O(n^2)$ sorting algorithms such as insertion sort and selection sort.

4.2 Procedure

The algorithm works by maintaining a max-heap tree that consists of the values in the array. The heap always occupies the front portion of the array and it is used to progressively pick out the largest elements in the tree. At each step, the largest element is moved outside of the heap to the end of the array. Heap sort procedure is built around the `heapify` function that is used to restore the max-heap property of the tree if one of its nodes is misplaced.

```
 1: A ← ...
 2: n ← LENGTH(A)
 3: i ← ⌊n ÷ 2⌋
 4: while i ≥ 1 do
 5:     HEAPIFY(A, n, i)
 6:       i ← i − 1
 7: end while
 8: while n ≥ 2 do
 9:     SWAP(A[1], A[n])
10:       n ← n − 1
11:     HEAPIFY(A, n, 1)
12: end while
```

1. The algorithm begins by finding the index of the last non-leaf node in the tree. Tree is represented as an array containing the elements in level order. (line 3)

2. The tree is converted to a max-heap. This is done by calling `heapify` on each non-leaf node in the heap starting from the bottom of the tree. (line 4)

3. The heap is then sorted by repeatedly reducing the size of the heap by one element and moving the root of it (i.e. the largest element) outside of the heap. (line 9)

4. Since the root is replaced by another element, each time the heap property needs to be restored. This is done by calling `heapify` once on the root node. (line 11)

5. Every time before the heap is shrunk by one element, the largest element in the heap is moved to the end of it. This way, a sorted sequence is progressively built at the end of the array.

```
 1: function HEAPIFY(A, n, i)
 2:     l ← 2i
 3:     r ← 2i + 1
 4:     x ← i
 5:     if l ≤ n and A[l] > A[x] then
 6:         x ← l
 7:     end if
 8:     if r ≤ n and A[r] > A[x] then
 9:         x ← r
10:     end if
11:     if x ≠ i then
12:         SWAP(A[i], A[x])
13:         HEAPIFY(A, n, x)
14:     end if
15: end function
```

1. The function begins by finding the indices of the left and right child nodes of the current node. (lines 2 and 3)

2. The index of the largest element is initialized to the current node. Subsequently, the left and right child nodes are compared to the current node and the largest one of them is found. (line 4)

3. If the largest element is one of the children, the current node is swapped with it and the function is called recursively on the child node. (lines 11 and 13)

4. After the last recursive call finishes, the heap property is restored.

4.3   Computational Complexity

In order to find the total time complexity of heap sort, we shall first analyze the `heapify` sub-procedure. `heapify` descends one level down during each recursive call, so in the worst case, the total number of recursive calls is equal to the depth of the heap (i.e. a well balanced binary tree), which is $\lfloor \log_2 n \rfloor$. Other than that, `heapify` does not use any loops, so the final complexity of `heapify` is $O(\log n)$.

The loop used to build the max-heap is executed $\lfloor n \div 2 \rfloor$ times, so the time complexity of the heap building step is:

$$T(n) = \sum_{i=1}^{\lfloor n \div 2 \rfloor} O(\log n)$$
$$= O(n \log n)$$

The second loop used to shrink the tree and fix the heap property is executed $n - 1$ times, so the time complexity of the heap sorting step is:

$$a = const.$$
$$T(n) = \sum_{i=1}^{n-1} (O(\log n) + a)$$
$$= O(n \log n)$$

Finally, the overall worst-case computational complexity of heap sort is the maximum of the time complexities of the two steps, which is $O(n \log n)$.

Since the heap sorting step is always executed after the max-heap building step, it will always replace the largest element in the heap with one of the smallest elements in the tree. This means that calls to `heapify` during the heap sorting step will always have the same time complexity of $\Theta(\log n)$. This ensures that the total time complexity of heap sort is exactly $\Theta(n \log n)$.

$$T(n) = \Theta(n \log n)$$

Note: If the array only contained the same value repeated multiple times, `heapify` would never be called recursively, so the computational complexity of heap sort would be $\Omega(n)$. In some literature this edge case is taken care of by using inclusive bounds in the `heapify` function to ensure the exact time complexity of $\Theta(n \log n)$.

5   References

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