Better Sorting Algorithms

Josh Wilcox (jw14g24@soton.ac.uk)

March 8, 2025

Contents

1 Correctness of Sorting Algorithms

- A sorting algorithm is correct when:
  - Its output is in non-decreasing order
  - The items in the output are a permutation of the items in the input

1.1 Proving correctness of insertion sort

At the start of each iteration of the **for** loop, the sub-array consists of all items originally in the list but in sorted order. The for loop ends when the list is fully sorted

2 Comparison Based sorting algorithms

- A comparison based sorting algorithm can **only** gain information about the items in the input sequence by performing *pairwise comparisons*
  - **Pairwise Comparison** - A single query asking if $a_i < a_j$ (between exactly two elements)

3 Decision Trees

- Decision trees are a way to visualise **many algorithms**
- Shows a series of decisions made during an algorithm
- For sorting algorithms:
  - Shows what the algorithm does at every comparison

3.1 Time complexity with decision trees

- The time taken to complete a task is the *depth* of the decision tree
- **Worst Case** - depth of deepest leaf
- **Best case** - depth of shallowest leaf
- **Average Case** - average depth of leaves

4 Lower bound on comparison based sorting

- Any sorting algorithm with *pairwise comparisons* must have a leaf in the decision tree **for every permutation of sorting the list**
  - Each leaf gives a different possible ordering of elements
  - There are $n!$ possible permutations
- Remember the decision tree is binary (yes or no):

$$\text{Height of Decision Tree for Sorting} \geq log_2(n!)$$

4.1 Calculating the lower bound

$$\begin{aligned} log_2(n!) &= log_2(1 \cdot 2 \cdot \ldots \cdot \frac{n}{2} \cdot \ldots \cdot n) \\ &= log_2(1) + \ldots + log_2(\frac{n}{2}) + \ldots + log_2(n) \\ &\geq log_2(\frac{n}{2}) + \ldots + log_2(n) \\ &\geq \frac{n}{2} log_2(\frac{n}{2}) = \frac{n}{2} log_2(n) - \frac{n}{2} \end{aligned}$$

$$log_2(n!) = \Omega(nlog_2(n))$$

5 Merge Sort

5.1 Recursion Algorithm

```
public void MergeSort(a, start, end) {
    if (start < end){
        mid = floor((start + end) / 2) // Middle element index
        MergeSort(a, start, mid) // Left half
        MergeSort(a, mid+1, end) // Right half
        Merge(a, start, mid, end) // Merge the split up arrays
    }
}
```

5.2 Merging Sub Arrays

- The merge operation takes two sorted sub-arrays and combines them into a single sorted array.
- Assume the two sub-arrays are `left` and `right`.
- Initialize three pointers: `i` for the `left` sub-array, `j` for the `right` sub-array, and `k` for the position in the merged array.
- Compare the elements at `left[i]` and `right[j]`:
  - If `left[i]` is smaller, place `left[i]` in the merged array at position `k` and increment `i` and `k`.
  - Otherwise, place `right[j]` in the merged array at position `k` and increment `j` and `k`.
- Repeat the comparison until one of the sub-arrays is exhausted.
- Copy any remaining elements from the non-exhausted sub-array into the merged array.

5.3 Properties of Merge Sort

- It is **stable**
- It is **not** *in-place*
- Merging is quick
  - At most $n - 1$ comparisons to merge two subarrays
- Recurrence Relation:
$$T(n) = 2T(\frac{n}{2}) + \mathcal{O}(n)$$
  - Therefore **worst case** complexity: $\mathcal{O}(nlog(n))$

5.4 Improving Merge sort with Insertion Sort

- Insertion sort is very efficient for short arrays
- Therefore, if a sub-array size falls below a certain threshold, it is better to switch to insertion sort to sort the array instead of splitting even more
- These can then be merged in the same ray

6 Quick Sort

6.1 High level idea

- Separate the array into two parts depending on whether the elements are smaller or greater than some **pivot**
- Recurse on both parts until the array is sorted

6.2 Selecting a Pivot

- Pivot is the best when the partitions it creates are of the same length
- Possible options:
  - Choosing first element in array
  - Choose the median of first, middle, and last element of array
    - Pivot is somewhat more likely to be the median of the whole array
    - Therefore, split in two parts more often
  - Choose the pivot randomly

6.3 Time complexity

- Partitioning takes $\Theta(n)$ operations
- When the pivot element is the smallest in each partition, we would need $n-1$ partitioning rounds
  - This means partitioning is $\mathcal{O}(n)$
- **Worst-Case complexity** - $\mathcal{O}(n^2)$
- If the pivot is the median value, the array is split in half each time therefore we have $\Omega(log(n))$ partitions
  - Therefore on **average**, Quicksort is $\mathcal{O}(nlog(n))$