Sorting Algorithms

Niklas Walter

June 30, 2023

1 Introduction to sorting

In this note we give a short introduction to the problem of sorting a container object and present four famous algorithms used to solve it. In general, every sorting problem is of the following form

Input: A container object (e.g. array, list) with n entries $[a_1, ..., a_n]$.

Output: A permutation of the input container $[a_{\sigma(1)}, ..., a_{\sigma(n)}]$ such that $a_{\sigma(1)} \leq ... \leq a_{\sigma(n)}$, where σ is the regarding permutation operator.

Sorting can be considered as one of the most traditional and fundamental problems in computer science. This has a variety of reasons. A simple one is the fact that sorting algorithms employ a broad set of techniques also used in many other applications and methods. Moreover, in many real world applications it is of great interest to have a sorted dataset. Therefore, many "higher-level" algorithms depend on a sorted input.

2 Selection sort

The fist algorithm we consider arises very naturally from a naive approach of sorting. Consider an possibly unsorted input array A. Then the mechanism seperates A such that $A = [S, U]$, where S is a sorted subarray and U is unsorted. In the beginning, we obviously have $A = U$. Then the algorithm parses through U, finds the minimum value, puts it in the last position of S and deletes the entry from U. Afterwards this procedure is applied again but now to the shortened U until it is empty.

Algorithm 1 Selection sort

Input: Array A
Output: Sorted version of A

```
index_max ← length(A) − 1
index_insert ← 0
while index_insert < index_max do
    pos_min ← index_insert
    for i = index_insert + 1 to index_max do
        if A[i] < A[pos_min] then
            pos_min ← i
    exchange A[pos_min] and A[index_insert]
    index_insert ← index_insert + 1
return A
```

Lastly, we study the complexity. Assume that A has length n. Then to determine the first minimum we need to do $(n-1)$ comparisons. For the second we still need $(n-2)$ many and so

on. Therefore, the the total number of comparisons is given by

$$(n-1)+(n-2)+...+2+1=\frac{(n-1)n}{2}=\frac{n^2}{2}-\frac{n}{2}.$$

Hence, the Selection sort is of complexity $\mathcal{O}(n^2)$.

3 Bubble sort

This algorithm is also solely based on comparing the values with each other. In particular, it walks through the array A and compares adjacent entries. If the ordering rule is violated the entries are switched. After having parsed through the entire array once, the largest entry is in last position. Therefore, by repeating this procedure we can stop one position early in the second run and so on until the array is sorted.

Algorithm 2 Bubble sort

Input: Array A
Output: Sorted version of A

```
1: for i = 0 to length(A) − 1 do
2:     for j = length(A) downto j = i + 1 do
3:         if A[j] < A[j − 1] then
4:             exchange A[j] and A[j − 1]
5: return A
```

In the best case, thus when the array is already sorted, we only need to do $n-1$ comparisons and no switchings. Therefore, in this case the algorithm has complexity $\mathcal{O}(n)$. However, this case is not expected. In the worst case, the highest entry is in first position, the second highest in the second and so on. Hence, we need to do $n-1$ switchings in the first run, $n-2$ in the second etc. Therefore, the total number of switchings is given by

$$(n-1)+(n-2)+...+2+1=\frac{(n-1)n}{2}=\frac{n^2}{2}-\frac{n}{2}.$$

So in the worst case scenario, the runtime complexity is $\mathcal{O}(n^2)$. In particular, this is also true for the average case.

4 Divide-and-conquer

In computer science, divide-and-conquer is a popular paradigm to design algorithms. Its mechanism is based on the idea of breaking the main problem down in subproblems which are of the same type but easier to solve. Then the algorithm can be applied recursively and in the end the solutions of the subproblems are combined forming the solution for the original problem. The mechansim can be summarised as follows for each iteration step

1. Divide: The original problem is divided into smaller subproblems of the same type.

2. Conquer: Solve the subproblems (if possible).

3. Combine: Merge the solutions of the subproblems to attain the solution to the original problem.

If the subproblems are still big enough to solve, we possbily need to divide them further until it becomes small enough. When this is achieved we speak of the base case. Recurrences are at the core of divide-and-conquer algorithms. In the next section, we give a short introduction to them and show how to solve them in three special cases.

5 Recurrences and the master theorem

Mathematically, a recurrence is an equation which expresses each element of a sequence as a function of the preceding ones. Consider a sequence $(T(n))_{n\geq 0} \subseteq X$ for some set X. If each element can be expressed by only its predecessor then we get the recurrent relation

$$T(n) = \varphi(n, T(n-1)),\ n > 0$$

for a function $\varphi : \mathbb{N} \times X \to X$. Given an initial condition $T_0 = t \in X$ this sequence is unique. In the more general case, where $T(n)$ depends on more preceding values of the sequence, we write

$$T(n) = \varphi(n, T(n-1), ..., T(n-k)),\ n \geq k$$

for $\varphi : \mathbb{N} \times X^k \to X$ and some initial conditions for $T(0), ..., T(k)$.

Example 5.1. A famous example of a recurrence in mathematics appears when one wants to solve a ordinary differential equation (ODE) numerically using the Euler's method. For some $T > 0$ consider the time interval $[0, T]$ and an ODE on this interval described by

$$y'(t) = f(t, y(t)),\ y(0) = y_0.$$

Then we can define a gird of discrete time points as

$$0 = t_0 < t_1 < ... < t_k < ... < t_N = T$$

for $t_k = kh$, where we define $h := T/N$. Then we can compute the values $y(t_k)$, $k = 1, ..., N$, following the recurrence

$$y(t_{k+1}) = y(t_k) + f(t_k, y(t_k))h$$

starting with $y(t_0) = y(0) = y_0$.

In the application of algorithms or especially divide-and-conquer algorithms we can think of $T(n)$ describing the number of operations needed to perform for an input of size n - hence the runtime of the algorithm. In the case of divide-and-conquer methods we can express it as

$$T(n) = aT(n/b) + f(n),$$

where $a \geq 1$ is the number of subproblems created and $b > 1$ is the factor by which the size of the original problem is reduced by in each iteration step. Here, $f(n)$ denotes the cost produced by dividing the problem and merging the solutions in the end. We want to derive asymptotic bounds for $T(n)$ to get an idea about the behaviour of the runtime depending on n. Therefore, we state the master theorem for a more general case of recurrences defined below.

Theorem 5.1 (Master Theorem)**.** Consider the functional $T : \mathbb{N}_0 \to \mathbb{N}_0$ which is of the form

$$T(n) = \sum_{i=1}^{m} T(n/b_i) + f(n),$$

where $m \in \mathbb{N}$, $b_i > 1$ and $f(n) \in \mathcal{O}(n^k)$ with $k \in \mathbb{N}_0$. Then it holds

$$T(n) \in \begin{cases} \mathcal{O}(n^k) & \text{if } \sum_{i=1}^{m} b_i^{-k} < 1, \\ \mathcal{O}(n^k \log n) & \text{if } \sum_{i=1}^{m} b_i^{-k} = 1, \\ \mathcal{O}(n^c) \text{ with } \sum_{i=1}^{m} b_i^{-c} = 1 & \text{if } \sum_{i=1}^{m} b_i^{-k} > 1. \end{cases}$$

Remark 5.1. Recall that for two function f and g the Landau symbol $\mathcal{O}$ is defined by

$$f \in \mathcal{O}(g) :\Leftrightarrow \exists C > 0 \exists x_0 > 0 \forall x > x_0 : |f(x)| \leq C|g(x)|,$$

which just means that asymptotically the growth of function f is bounded from above by the one of g. Therefore, if we have $T(n) \in \mathcal{O}(n^2)$ then the runtime growths at most quadratically with the size of the input.

6 Merge sort

This sorting algorithm follows the divide-and-conquer paradigm in the following way

1. Divide: Divide the input array A of length n into two subarrays of length $n/2$.

2. Conquer: Sort the two subarrays using merge sort.

3. Combine: Merge the two sorted subarrays to produce the sorted version of A.

Here the base case is reached when A is divided into atoms. Therefore, we first divide until the subarrays have length 1 and then start merging them in a way that the regarding result is sorted in every step. So the main focus here lies in the **Combine** step.

Algorithm 3 Mergesort

Input: Array A
Output: Sorted version of A

```
1:  if length(A) > 1 then
2:      middle ← length(A)//2
3:      left ← A[: middle]
4:      right ← A[middle :]
5:      Mergesort(left)
6:      Mergesort(right)
7:
8:      i ← 0
9:      j ← 0
10:     k ← 0
11:
12:     while i < length(left) and j < length(right) do
13:         if left[i] <= right[j] then
14:             A[k] ← left[i]
15:             i ← i + 1
16:         else
17:             A[k] ← right[j]
18:             j ← j + 1
19:
20:     while i < length(left) do
21:         A[k] ← left[i]
22:         i ← i + 1
23:         k ← k + 1
24:
25:     while j < length(right) do
26:         A[k] ← left[j]
27:         j ← j + 1
28:         k ← k + 1
```

Lastly, we study the complexity of this algorithm. We observe the recurrence

$$T(n) = T(\left\lfloor \frac{n}{2} \right\rfloor) + T(\left\lceil \frac{n}{2} \right\rceil) + \mathcal{O}(n),$$

where the first term corresponds to sorting the first subarray, the second to sorting the second subarray and the last on to combining them. To obtain the asymptotic order of growth for the runtime we can use the master theorem with values $m = 2$, $b_1 = b_2 = 2$ and $k = 1$. So we are in case 2 and see that $T(n) \in \mathcal{O}(n \log n)$.

7 Quicksort

This idea is also based on the divide-and-conquer paradigm in the following way

1. Divide: Divide the input array A into two subarrays L and R such that all elements of L are smaller than a pivot element $A[p]$ and the other way around for R.

2. Conquer: Sort the two subarrays using quicksort.

3. Combine: Since the subarrays are already sorted we can combine them with no work done.

The most complex step here is the *Divide* step which builds the partition of the input array in the way described above.

Algorithm 4 Quicksort

Input: Array A
Output: Sorted version of A

For the complexity analysis we consider again the worst-, best- and average-case. In the worst-case is attained when the chosen pivot element is the smallest or largest element of the input array. In this case the subarrays have length $n - 1$ and 0. Therefore, the runtime follows

$$T(n) = T(n-1) + T(0) + \mathcal{O}(n) = T(n-1) + \mathcal{O}(n).$$

In this case one can show that the recurrence has a solution such that $T(n) \in \mathcal{O}(n^2)$. On the other hand, the best case is attained if the subarrays have no more than $n/2$ elements (hence the original array is split perfectly). In this case, the runtime is described by

$$T(n) = T(\left\lfloor \frac{n}{2} \right\rfloor) + T(\left\lceil \frac{n}{2} \right\rceil - 1) + \mathcal{O}(n).$$

This also corresponds to case 2 of the master theorem and hence we obtain $T(n) \in \mathcal{O}(n \log n)$. In fact, it can be shown that the average-case is closer to the best-case than the worst-case and has to same asymptotic runtime.