Sorting algorithms

Gilles Englebert

March 28, 2024

Performance of algorithms

Measurements of performance

- Computation time
- Memory used
- Function calls
- Disk reads
- Network access

But this may depend on

- The machine the code is run on
- The input data

How to abstract away from this?

Performance of algorithms

Count instructions/memory allocation/function calls
However should do this in such a way that the platform does not matter, and that is gives us a valid estimate for a wide range of inputs. Here we get to **Big-O** notation (which ignores everything except the most important contribution, up to constants).
Examples for $n \to \infty$:

- $2n^2 - 5n + 3 = O(n^2)$.
- $n \log n + 2n = O(n \log n)$.
- $n! = O(n^n)$.

Let's sort the following functions from small to large.

$$n^2 + 5, n2^n, n!, (\log n)!, 1, \log n, 2^{n+1}, 2^n, 2^{2^n}.$$

Sorting by comparing

Task: Sort n objects $a_1, a_2, \ldots, a_n$ using pairwise comparison, i.e. asking questions "$a_i < a_j$?" for $1 \leq i, j \leq n$. We get **comparison-based** algorithms:

- $O(n^2)$: Bubble sort, Insertion sort, ...
- $O(n \log n)$: Quicksort, Merge sort, Heapsort, ...

There are other ways of sorting, that we will see later. We cannot do better than $O(n \log n)$ with comparison based algorithms! It is the best possible worst case asymptotic behaviour. For any comparison based algorithm there is at least one testcase that takes time $O(n \log n)$.

Excursion: information theory

Information can be quantified in **bits**. Knowing whether a coin lands heads or tails is 1 bit. More generally, if we know that something with k possible outcomes happens that gives us $\log_2(k)$ bits of information. There are $n!$ different ways that n objects could be arranged. Determining the order gives $\log_2(n!)$ bits of information. But:

$$\begin{aligned}\log_2(n!) &= \log_2(1) + \log_2(2) + \cdots + \log_2(n) \\ &\geq \log_2\left(\frac{n}{2}\right) + \log_2\left(\frac{n}{2}+1\right) + \cdots + \log(n) \\ &\geq \frac{n}{2}\log_2\left(\frac{n}{2}\right) = O(n \log n).\end{aligned}$$

But asking one comparison question gives 1 bit, so we need at least $O(n \log n)$ questions to determine the order.
More information based tasks: IOI'11 parrots, IOI'13 cave, EGOI '22 Chika wants to cheat.

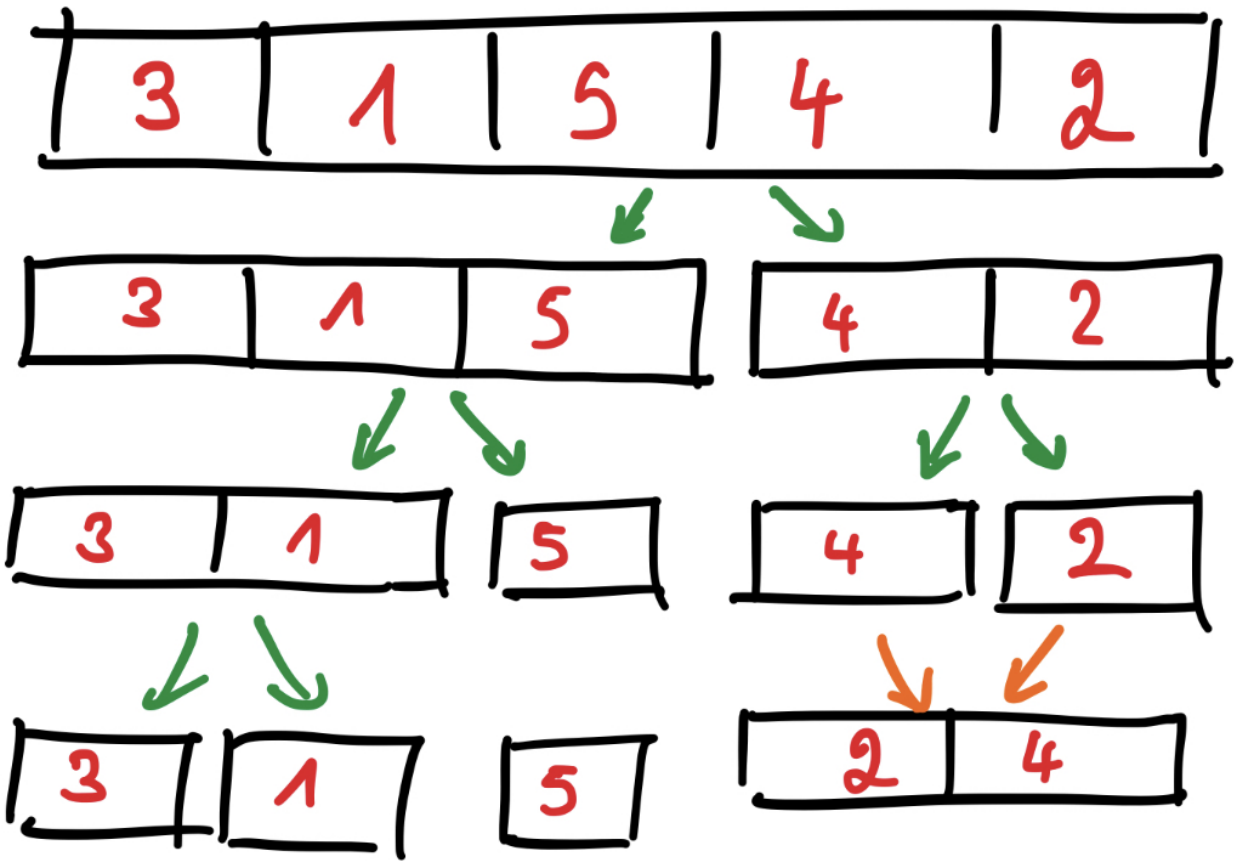
Quick Sort

Idea: Choose a pivot somewhere among the elements. Now partition the list in elements bigger and elements smaller than the pivot in time $O(n)$. Sort the 2 new lists recursively.
This is an example of **divide and conquer**, reducing a problem to smaller problems until they become trivial (e.g. list with one element).
Note: The choice of pivot is critical here, since we want the two new lists to be roughly half the size. This is when the algorithm is most efficient. The average case performance of quicksort is $O(n \log n)$, but it could be $O(n^2)$ for the worst case.

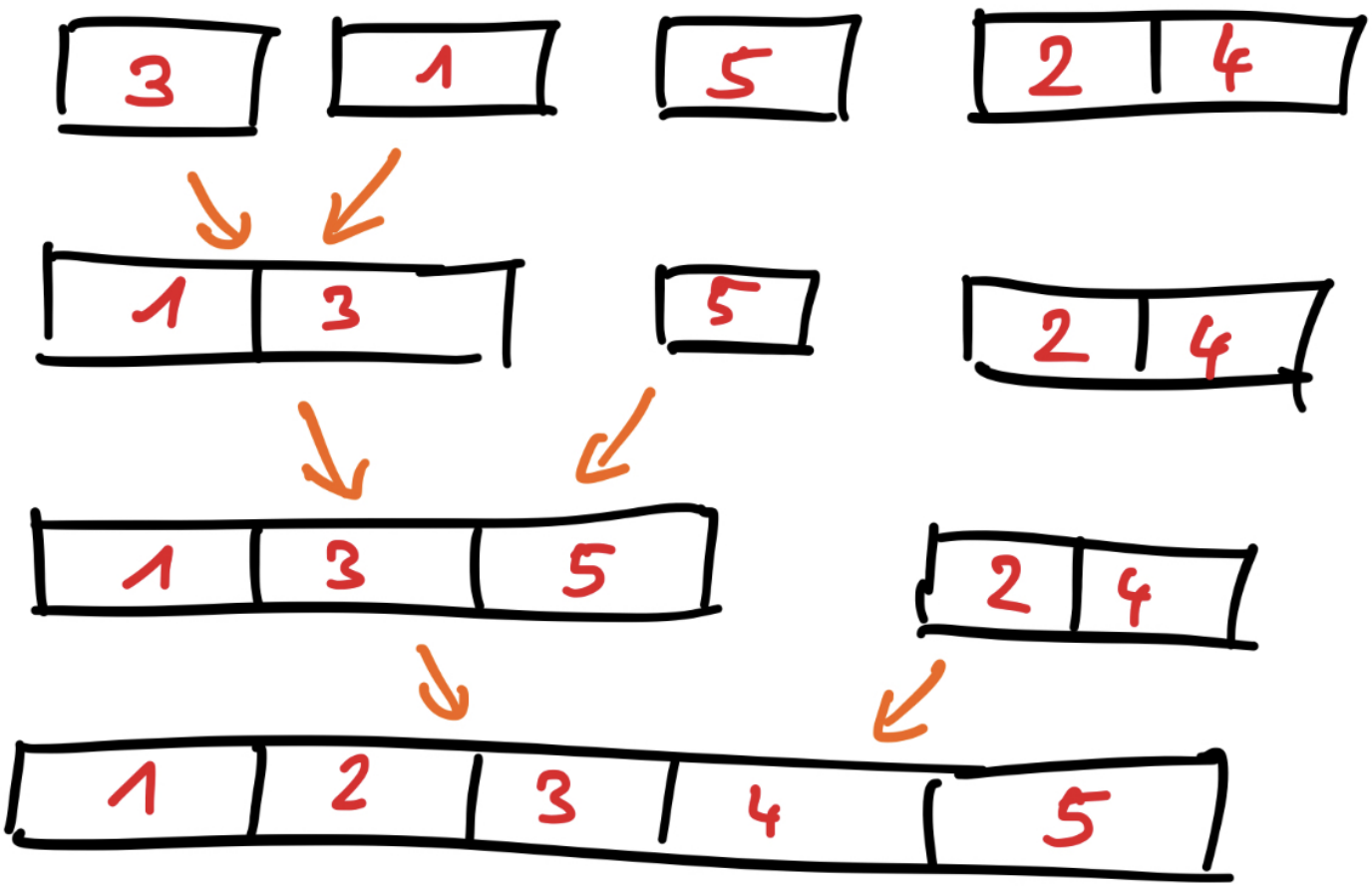
Merge Sort

Idea: Split list in two in the middle. Sort each part recursively (base case: list with one element). Create new list by repeatedly taking the smallest element of either list (there are only two choices now), remove that element from the list and append it to a new list. If $f(n)$ is the execution time, then $f(n) = 2f(\frac{n}{2}) + O(n)$, thus by plugging in we see that $O(n \log n)$ solves it. Thus same asymptotic runtime as Quicksort. But sometimes there can be a difference in performance (depending on if comparisons or movements of the data are more important)

Example



Example



Heap sort

A **heap** is a data structure that can do the following operations on a collection of integers efficiently, here n is the current size of the heap:

- Insert a new element in time $O(\log n)$.
- Retrieve the largest element in time $O(1)$.
- Remove the largest element in time $O(1)$.

In C++, the priority queue type is a heap!

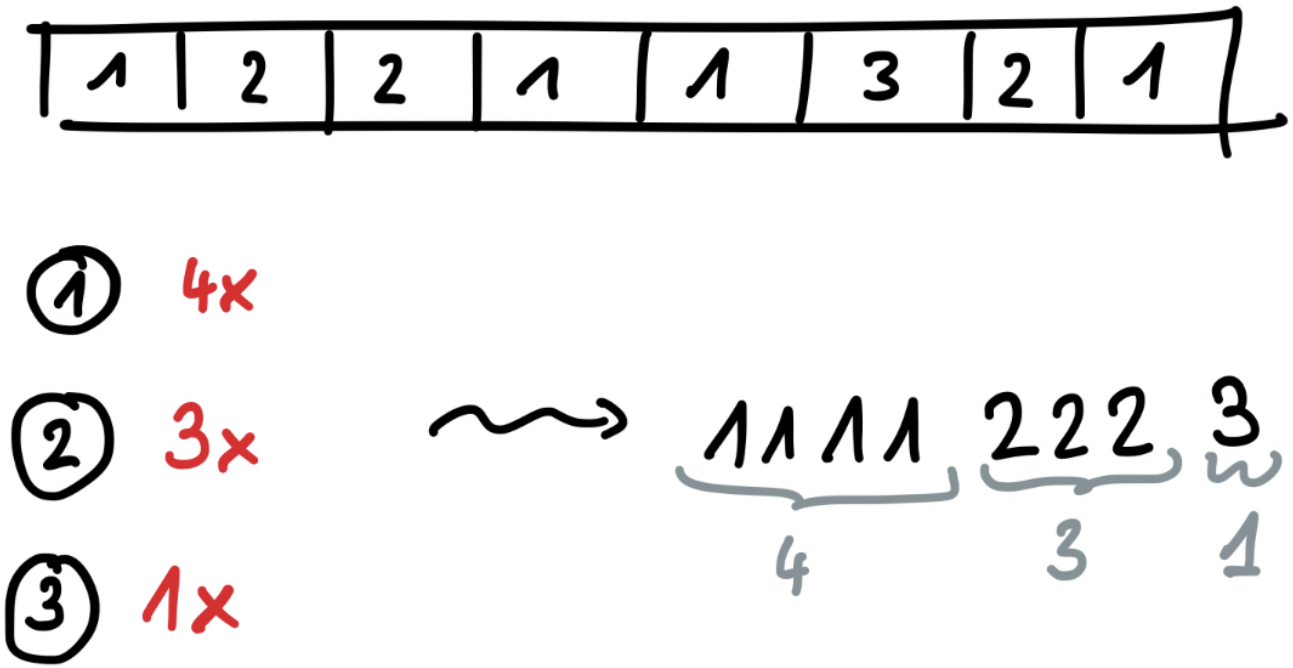
Idea: Given n integers, create a heap by sequentially inserting each element into it. This takes time $\log(1) + \log(2) + \cdots + \log(n) = O(n \log n)$. Then repeatedly retrieve and remove the largest element from the heap to create a sorted list. This takes times $O(n)$. Thus heapsort runs in time $O(n \log n)$.

Counting sort

Comparison based algorithms work for any kind of ordered data. If we know more about our data we can design algorithms better suited for it. Suppose we want to sort n positive integers that take m different values. We can use counting sort in time $O(n+m)$. This can be faster if m is small, because we **do not use comparisons**.

Idea: Go through the array once, and count how many times a given number occurs, save this in an array $c[0 \dots m]$. Then go through the array c once from the smallest to the largest integer and create a list with $c[i]$ entries equal to i.

Example



Data centers: task statement

Task: We are given a list $a_1, a_2, \ldots, a_n$ of positive integers. We are then given s queries of the following form: Sort the list from largest to smallest, then reduce the c_i biggest ones by m_i each. The output should be the resulting list, sorted from largest to smallest.
Constraints: $n \leq 100000, s \leq 5000, a_i \leq 10^9$, all operations will be possible.
Subtasks:

- 1. $s = 0$.
- 2. $s \leq 100$.
- 3. $a_i \leq 1000$.
- 4. $c_i = 1$.
- 5. No further constraints.

Data centers: solution

- 1: Sort the list once using any sorting algorithm.
- 2: Simulate everything using C++ sorting routine.
- 3: Counting sort, because $a_i \leq 1000$.
- 4: Heap sort (or a priority queue in C++). Build the queue in $O(n \log n)$, remove the largest elements in $O(1)$, substract m_i and put them back in in $O(\log n)$.
- 5: Use the merge step in merge sort to combine the two parts of the list, the part that was changed, and the part that was left intact.

Homework

Look at the following problems:

- IOI '21 Registers
- BOI '21 Swaps