17. QUICKSORT

Mihai-Valentin DUMITRU
mihai.dumitru2201@upb.ro

December 2024

This lecture will serve as a "case-study" for the analysis of divide-and-conquer algorithms. We will introduce one of the most well-known and widely-used sorting algorithms, quicksort, by first presenting how it works, then proving its correctness and finally showing why it is *fast*.

1 The quicksort algorithm

Quicksort is a recursive algorithm, that works in the following way:

1. choose a *"pivot"*: an element in the array that will act as a point of reference for the future steps
2. move all elements lower than the pivot to the left of the pivot and all elements greater than the pivot to its right
3. quicksort the elements to the left, then quicksort the elements to the right

This description is not very exact. For example, one question that should instantly pop up is: "how do you choose the pivot?" We will address this (and many other questions) later, but first let's have a look at a possible pseudocode:

Algorithm 17.1 Quicksort pseudocode

```
1: function QUICKSORT(A, left, right)
2:     length ← right − left
3:     if length ≤ 1 then
4:         return
5:     pivot_index ← CHOOSE_PIVOT(left, right)
6:     new_pivot_index ← PARTITION(A, left, right, pivot_index)
7:     QUICKSORT(A, left, new_pivot_index)
8:     QUICKSORT(A, new_pivot_index + 1, right)
```

Things seem similar to merge sort: we have two recursive calls on two parts of the array and some extra processing.

The key difference is that, for merge sort, this extra processing is done *after* the recursive calls, in order to combine the solutions to the two subproblems into the final solution.

For quicksort, the extra work is done to split the initial problem into two subproblems in such a way that, after they are solved, there is no need to do anything to have the solution to the main problem.

We now address the two functions involved. First, let's convene that our pivot choice will always choose as a pivot **the first element** of the array; in our case this means returning `left`.

Then the pseudocode for partitioning will look like this:

Algorithm 17.2 Partitioning pseudocode

```
1: function PARTITION(A, left, right, pivot_index)
2:     SWAP(left, pivot_index)
3:     frontier ← left + 1
4:     for each i = left + 1, right do
5:         if A[i] ≤ A[left] then
6:             SWAP(frontier, i)
7:             frontier ← frontier + 1
8:     SWAP(left, frontier − 1)
9:     return frontier − 1
```

The goal is to place all elements lower than the pivot to its left and all elements greater than the pivot to its right. The simplest way to do this would be to allocate a new array, then start copying elements into its: lower elements go to the "front", greater elements go to the "back".

However, with some ingenuity, we can do this partitioning "in-place", without needing any extra space. The idea is to maintain a *"frontier"* of the elements smaller than the pivot. The frontier is an index: all elements with a smaller index ($<$) are lower than the pivot and all elements with a higher index ($\geq$) are greater than the pivot.

We then iterate the array with an index i and make sure that, at each iteration, the first i elements are properly partitioned around the frontier.

A good trick to do this is to first position the pivot as the first element of the array; we will need to compare all the elements with it and now we know exactly where it is at all steps.

We initialize the frontier to show that right now we have no elements to the left of the pivot, then we start iterating through all other elements:

- when we find an element greater than the pivot, there's nothing to do; this element is already to the right of the frontier, so it is properly partitioned.
- when we find an element less than the pivot, we need to place it before the frontier. The first idea that comes to mind is to create a "hole" over the current element, then shift all elements after the frontier one position to the right.

  This would maintain the current order of the already partitioned greater-than-the-pivot element, but we do not care about this order; all we want to achieve is partitioning.

  We can instead swap our current element with the one at the frontier, then advance the frontier by one element.

  This way, we maintain the property that all elements to the left of the frontier are lower than the pivot and all elements to the right are greater than the pivot.

After iterating through the elements, we have partitioned the array such that:

- the first element is the pivot
- there is an index, frontier, ≥ 1
- that all elements with a lower index than frontier are less than the pivot
- all elements with a greater index are greater than the pivot

In order to obtain the desired partitioning, with the pivot in the middle, we just swap the element right before the frontier with the first element.

2 Correctness

To prove quicksort correct, we must first prove the correctness of the partitioning subroutine.

In the previous section, we mentioned how, during partitioning, a certain property is always true: "all elements with an index lower than `frontier` are lower than the pivot and all elements with an index higher than `frontier` are greater than the pivot".

By "all elements", we actually mean "all elements examined by this step", i.e. with an index $\leq i$.

We have a property parameterized by an index i and, as this index grows, the property should always be true. This is called a *loop invariant*.

Invariants are an essential part of correctness proofs for iterative algorithms. However, due to time constraints, they are outside the scope of this course, so we will not provide a rigorous correctness proof of quicksort.

However, we describe intuitively the elements needed for such a proof:

- we need to prove that for $i = n$, our property means that the array is partitioned
- we need to prove that at the beginning of the `for` loop, the condition holds (i.e. for $i = \texttt{left} + 1$)
- we need to prove inductively that if the condition holds for i, it holds for $i + 1$
- we need to prove that the loop actually terminates and doesn't go on forever

For a good introduction to loop invariants, we recommend chapter 2.1 of "Introduction to Algorithms" [1], which introduces them in the context of proving the correctness of insertion sort.

Chapter 7.1 presents a complete treatment of quicksort, including the proof of correctness for partitioning.

3 Complexity analysis

Similarly to merge sort, we have a recursive sorting algorithm in which two calls are made at each step. However, the array is split in two parts based on some arbitrary *pivot*, it's not necessarily broken into two *halves*. Depending on the choice of pivot, the recurrence relation that models the running time varies.

3.1 Pivot choice – best case scenario

Let's assume that, at each step, we can magically select as a pivot the **median element** of the array; i.e. half of the elements are lower than it and half are greater.

Then the two partitions of the array are equally-sized halves and the recurrence relation on an array of size n becomes:

$$T_{best}(n) = 2T_{best}\left(\frac{n}{2}\right) + \Theta(n)$$

Identical to the one for merge sort! (Although note that, in this case, the linear term is not interpreted as "the amount of work needed to recombine the solutions to the subproblems", but rather "the amount of work needed to break the problem into subproblems").

3.2 Pivot choice – worst case scenario

Now let's assume that, at each step, we select as a pivot the **least element** of the array[†].

The partition with elements less than the pivot is empty, while the partition with elements greater then the pivot contains all other elements. For an array of n elements, the recurrence relation becomes:

$$T_{worst}(n) = T_{worst}(n-1) + \Theta(n)$$

The Master Method doesn't work on this recurrence, but we can easily see that the recursion tree is shaped as an n-node long "chain".

Guessing $T(n) \in O(n^2)$, we can apply the substitution method.

[†] A symmetrical argument works for the **greatest element**.

Proof. **Base case:** $1 \leq c$.

Induction step: Assume $T(n-1) \leq c(n-1)^2$.

Adding n, we get:

$$T(n) \leq c(n-1)^2 + n \tag{1}$$

$$= cn^2 - 2cn + c + n \tag{2}$$

$$= cn^2 + (n + c - 2cn) \qquad \leq cn^2 \tag{3}$$

The last inequality holds for $c \geq \frac{n}{2n-1}$. So $c \geq 1, n_0 = 1$ are ok.

□

With a similar argument for the lower bound we can provide a tight bound of $\Theta(n^2)$. This is asymptotically worse than the best-case scenario of $\Theta(n \log n)$.

But what is between these two extremes? Does this $\Theta(n^2)$ complexity even matter in practice? Unfortunately, the answer is yes; if our strategy is to always select the first element as a pivot, then any already-sorted, or already-reversed-sorted array will reach this worst-case complexity. In the real world, it is often the case that we find almost sorted arrays (or portions of the array) in our inputs.

There are also good news. For the best-case-complexity, we don't need to be able to select the *median* every time, just "something close to the median"; and it turns out that it is feasible to do so.

The key element of our pivot selection criterion will be to choose a *random element*. So let us first try to formalize the concept of *random*, then show why choosing the pivot in this way is advantageous.

3.3 Random algorithms

We can extend a Turing machine to allow for randomness by taking inspiration from the nondeterministic Turing machine: we have a machine with two transition functions, δ_0 and δ_1; at each step, the machine flips a coin: if it comes up heads, it uses δ_0, otherwise it uses δ_1.

This concept can be rigorously formalized and analyzed from both a computability and complexity perspective, but this is outside the scope of this course.

For a more high-level "RAM-machine"-like model, we can imagine a new instruction that generates a random number in one of the registers.

In either way, it is important to note that "flipping a coin" is actually the only random *primitive* we need; we can generate random objects of any kind starting from this. Just recall the first lectures on computability when we discussed how objects can be represented by finite strings over a finite alphabet and how the binary alphabet is enough to represent any object. For example, generating a n-bit random number amounts to flipping a coin n times etc.

3.4 Expected running time of quicksort

What if instead of always being able to magically select the median and get a 50-50 split of the array, we could instead magically select an element "close enough to the median" such that we get a 25-75 split (or better)?

For a split of exactly 25-75, we get the following recurrence relation:

$$T_{ok}(n) = T_{ok}\left(\frac{n}{4}\right) + T_{ok}\left(\frac{3n}{4}\right) + \Theta(n)$$

Let's guess an upper bound and use the substitution method to prove it (a symmetrical argument can prove a lower-bound; together we can get a tight bound of $\Theta(n \log n)$, but we will only prove the upper bound).

Theorem 17.1.

$$T_{ok}(n) \in O(n \log n)$$

Proof. We need to show that $\exists c, n_0$ s.t. $\forall n \geq n_0, 0 \leq T_{ok}(n) \leq cn \log n$.

Note that this recurrence only makes sense if we have base cases for $T_{ok}(0), T_{ok}(1), T_{ok}(2), T_{ok}(3)$. Let's say that they are all 1.

As the base case for our induction, we must choose all values that depend on these, so: 4, 5, 6, 7.

$$T(4) = 6 \leq c \cdot 8 \tag{4}$$

$$T(5) = 7 \leq c \cdot 5 \log 5 \tag{5}$$

$$T(6) = 8 \leq c \cdot 6 \log 6 \tag{6}$$

$$T(7) = 9 \leq c \cdot 7 \log 7 \tag{7}$$

From this, $c \geq \frac{3}{4}$ (interpreting "log" to be $\log_2$).

We now assume that for all values lower than n (in particular $\frac{n}{4}$ and $\frac{3n}{4}$) the inequality holds:

$$T(\frac{n}{4}) \leq c\frac{n}{4} \log \frac{n}{4} \tag{8}$$

$$T(\frac{3n}{4}) \leq c\frac{3n}{4} \log \frac{3n}{4} \tag{9}$$

Summing these two terms and adding n we get:

$$T(n) \leq c\frac{n}{4} \log \frac{n}{4} + c\frac{3n}{4} \log \frac{3n}{4} + n \tag{10}$$

$$= c\frac{n}{4} \log \frac{n}{4} + c\frac{3n}{4} \log \frac{n}{4} + c\frac{3n}{4} \log 3 + n \tag{11}$$

$$= cn \log \frac{n}{4} + c\frac{3n}{4} \log 3 + n \tag{12}$$

$$= cn \log n - 2cn + c\frac{3n}{4} \log 3 + n \tag{13}$$

$$\leq cn \log n \tag{14}$$

The last inequality holds when $c\frac{3n}{4} \log 3 + n - 2cn \leq 0$, so when $c \geq \dfrac{1}{2 - \frac{3}{4} \log 3} \approx 1.23$.

So for $n_0 = 8, c = 1.5$, the relationship holds.

□

Thus if we get a 25-75 split or better, the recurrence relation that describes the running time of quicksort has a complexity of $\Theta(n \log n)$.

How many elements would offer a 25-75 split or better? Well, precisely half of them! We need, at each step to select a pivot, such that at least 25% of the elements and at most 75% of the elements are lower than it. This is true for 50% of the elements of a list[‡].

Note that this probability *does not refer to the input's structure*. We are still doing worst-case analysis, where any input is just as likely. We are **not** saying that for any input array n, quicksort is fast for half of the $n!$ possible permutations, but slow for the other half. The 50% probability refers to the internal randomness used in the algorithm to select the pivot.

[‡]There is a hidden assumption here that all elements are *distinct*; and that we are not dealing, for example, with an array of all 1s. Quicksort can be extended to also have $\Theta(n \log n)$ performance in this case, but we will not address this here.

A rigorous proof of the fact that $\Theta(n \log n)$ is the *expected complexity* of quicksort-with-random-pivot requires various notions of statistics. We do not present it here, but we recommend reading the material referenced in the next section.

4 References and further reading

Quicksort was first developed by Tony Hoare in 1959 and published in 1961 [2].

It enjoyed such popularity that the sorting function in the C standard library was named `qsort`[§] (but the C standard does not actually require that the implementation use quicksort).

The presentation in this lecture is mostly adapted from Tim Roughgarden's textbook "Algorithms Illuminated" [3] (chapter 5). We also recommend going through the relevant videos from his algorithms course at Stanford[††] (the videos starting with "5 " and "6 ").

Another good treatment of quicksort can be found in "Introduction to Algorithms" [1] (chapter 7).

Bibliography

[1] Thomas H Cormen et al. *Introduction to algorithms*. MIT press, 2009.

[2] C. A. R. Hoare. "Algorithm 64: Quicksort". In: *Commun. ACM* 4.7 (July 1961), p. 321.

[3] Tim Roughgarden. *Algorithms illuminated*. Soundlikeyourself publishing, 2022.

[§] `https://pubs.opengroup.org/onlinepubs/009696899/functions/qsort.html`

[††] `https://www.youtube.com/watch?v=yRM3sc57qOc&list=PLXFMmlkO3Dt7QOxr1PIAriY5623cKiH7V`