Quicksort

Manuel Serna-Aguilera

Introduction

Quicksort is a very popular sorting algorithm, despite its worst-case running time being $\Theta(n^2)$, it is nonetheless very effecient on average with an expected running time of $\Theta(n \cdot \log_2(n))$. The constant factors hidden by the asymptotic notation are small. Assuming distinct input elements, its expected running time is $O(n \cdot \log_2(n))$.

Like merge sort, quicksort utilizes the divide-and-conquer paradigm, taking in an array $A[p\,..\,r]$, it performs the following steps.

> **Divide**: Partition (rearrange) the array $A[p\,..\,r]$ into two (possibly empty) subarrays $A[p\,..\,q-1]$ and $A[q+1\,..\,r]$ such that each element of $A[p\,..\,q-1]$ is less than or equal to $A[q]$, which is, in turn, less than or equal to each element of $A[q+1\,..\,r]$. Compute the index q as part of this partitioning procedure (this can be a simple or complex process).
>
> **Conquer**: Sort the two subarrays $A[p\,..\,q-1]$ and $A[q+1\,..\,r]$ by recursive calls to quicksort.
>
> **Combine**: The subarrays are already sorted, and no work is needed to combine them (as quicksort sorts in place).

Here are the procedures for quicksort.

```
QUICKSORT(A, p, r)
1  if p < r
2       q = PARTITION(A, p, r)
3       QUICKSORT(A, p, q − 1)
4       QUICKSORT(A, q + 1, r)
```

To sort an entire array A, the initial call is QUICKSORT$(A, 1, A.length)$.

```
PARTITION(A, p, r)
1  x = A[r] // this partition procedure always picks the right-most element
2  i = p − 1
3  for j = p to r − 1
4       if A[j] ≤ x
5            i = i + 1
6            exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

Figure 1 shows how the partition procedure works on an example 8-element array. This particular partition picks the right-most element in the subarray as its **pivot**, which is how we divide the subarrays; note that we can have any index be the partition, the book just does it this way for simplicity.

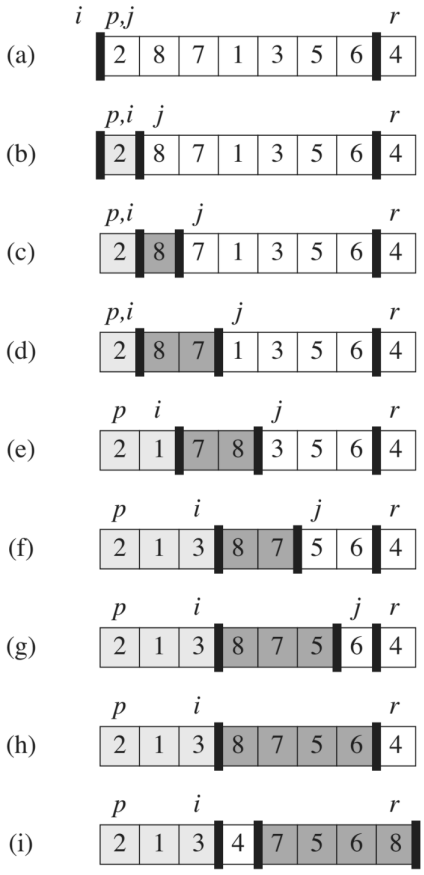

Figure 1: A visual example of the process of partition. Array entry $A[r]$ becomes the pivot element x. Lightly shaded array elements are all in the first partition with values no greater than x. Heavily shaded elements are in the second partition with values greater than x. The unshaded elements have not yet been put in one of the first two partitions, and the final white element is the pivot x. **(a)** The initial array and variable settings. None of the elements have been placed in either of the first two partitions. **(b)** The value 2 is "swapped with itself" and put in the partition of smaller values. **(c)**–**(d)** The values 8 and 7 are added to the partition of larger values. **(e)** The values 1 and 8 are swapped, and the smaller partition grows. **(f)**The values 3 and 7 are swapped, and the smaller partition grows. **(g)**–**(h)** The larger partition grows to include 5 and 6, and the loop terminates. **(i)** In lines 7–8, the pivot element is swapped so that it lies between the two partitions.

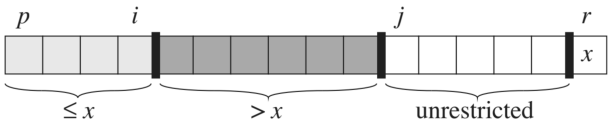

Figure 2: The four regions maintained by the procedure PARTITION on a subarray $A[p\,..\,r]$. The values in $A[p\,..\,i]$ are all less than or equal to x, the values in $A[i+1\,..\,j-1]$ are all greater than x, and $A[r] = x$. The subarray $A[p\,..\,r-1]$ can take on any values.

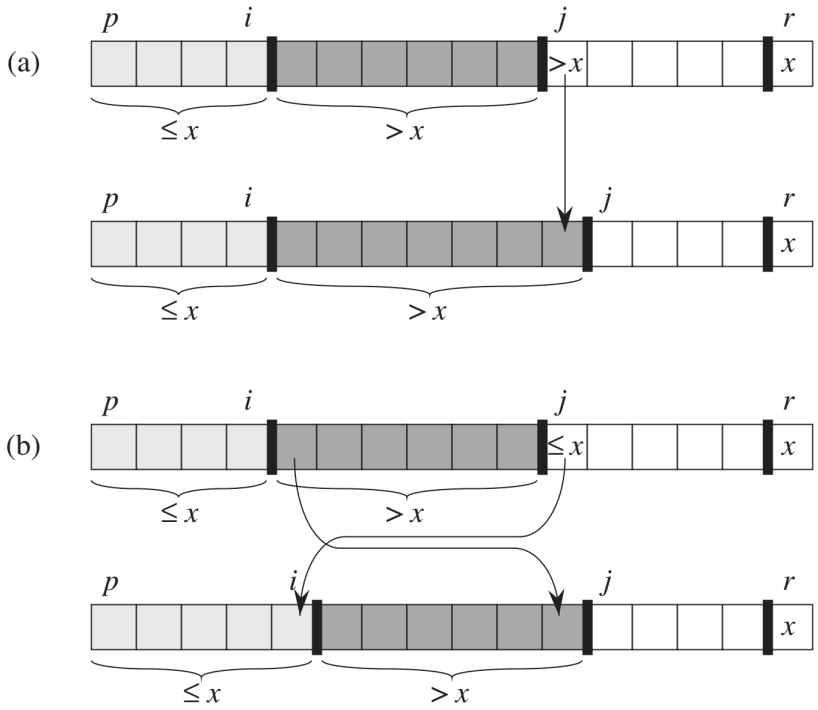

Figure 3: The two cases for one iteration of procedure PARTITION. **(a)** If $A[j] > x$, the only action is to increment j. **(b)** If $A[j] \leq x$, index i is incremented, $A[i]$ and $A[j]$ are swapped, and then j is incremented.

Performance

The **best case** of quicksort occurs when PARTITION evenly splits the array every time it is called. The actual sizes of the splits are $\lfloor n/2 \rfloor$ and $\lfloor n/2 \rfloor - 1$ since we do not include the partition value when we recurse. Even so, we can simplify the sizes to be the same since the time cost will pretty much be the same. This ideal situation gives the recurrence

$$T(n) = 2T(n/2) + cn + 1.$$

By the master method,

$$\begin{aligned} n^{\log_b a} &= n^{\log_2 2} = n^1 \\ f(n) &= cn = \theta(n). \qquad \text{(case 2)} \end{aligned}$$

Thus, the closed-form solution and the best-case running time for quicksort is

$$T(n) = \Theta(n \cdot \log_2(n)).$$

The **worst case** of quicksort occurs when the array is already sorted in increasing order, giving the following recurrence.

$$\begin{aligned} T(n) &= T(n-1) + cn \\ &= T(n-i) + i \cdot cn \\ &\qquad\qquad n - i = 1 \text{ (aside)} \\ &\qquad\qquad n - 1 + i \\ &\qquad\qquad T(1) = c \\ &= T(1) + cn(n-1) \\ &= 1 + cn^2 - cn \\ &= cn^2 - cn + 1 \end{aligned}$$

$T(n) = \Theta(n^2)$ is our closed-form solution and worst-case running time.