QuickSort

Manoj Gupta

September 11, 2016

Given an unsorted array A and its first element $A[1]$ (lets call it a **pivot**), can we put the pivot in its correct position in the final sorted array? Note that we don't actually aim to sort the array at this point but just want to find the final position of the pivot. We now give a simple algorithm that use two extra array $S_<$ and $S_>$ of size n. $S_<$ contains all the elements of A with value less than pivot and $S_>$ contains all the elements in A with value greater than pivot.

```
1 $S_< \leftarrow$elements in $A$ with value less than pivot;
2 $S_> \leftarrow$ elements in $A$ with value greater than pivot;
3 Put the elements in $S_<$ at the start of $A$;
4 Then put the pivot;
5 Then put all the elements in $S_>$ at the end of $A$;
6 return the index of pivot;
```

Figure 1: PARTITION(A): The partition function finds the correct place of pivot in A.

One can prove that the above algorithm correctly finds the final place of pivot. Note only that the array to the left of pivot contains all the elements less than pivot and the array to the right of pivot contains all the elements greater than pivot. We will use this property of PARTITION crucially in our sorting algorithm called QUICKSORT. One can also check that the running time of PARTITION(A) is $O(n)$ if A is of size n.

We will now describe our sorting algorithm:

```
1 if $n = 1$ then
2     return;
3 end
4 else
5     $i \leftarrow$ PARTITION$(A)$;
6     QUICKSORT$(1 \dots i-1)$;
7     QUICKSORT$(i+1 \dots n)$;
8 end
```

Figure 2: QUICKSORT$(A[1 \dots n])$

Lets us ask some simple question regarding the above algorithm. Assume that $A = [1, 2, 3, \dots, n]$. So, the first element of A, pivot=1. Note that the pivot is already at it correct position, so PARTITION(A), even after taking $O(n)$, will leave A untouched. Since, there is no sub-array to the left of the pivot, we will then call QUICKSORT$(A[2 \dots n])$. And the same pattern follows in the next call when pivot=2. We claim that the running time of the above algorithm on this input can be calculated as follows: $T(n) = T(n-1) + cn$, where we know that after the first iteration we have found the final place of (pivot=)1 in $O(n)$ time and then we call QUICKSORT of size $n-1$. One can calculate that $T(n) = c\binom{n}{2} = O(n^2)$.

Thus, we see that in the worst case QUICKSORT takes $O(n^2)$ time. What would then be the best case for quick sort? One feels that if the pivot's final place is in the middle of the array, then we would have paritioned A into two sub-array of same size. Let us assume that in each invocation of QUICKSORT, pivot's final position is in the middle of the concerned array. Thus, the running time of QUICKSORT on such a *nice* input can be *roughly* calculated as: $T(n) = 2T(n/2) + cn$, where $T(n/2)$ is roughly the time taken by QUICKSORT to sort the left and right array to the pivot and cn is the time taken by PARTITION. We have seen above recurrence relation while analyzing MERGESORT and we know $T(n) = O(n \log n)$.

Thus, the running time of QUICKSORT varies from worst case $O(n^2)$ to best case $O(n \log n)$. However, in practice QUICKSORT turns out to the best sorting algorithm. This leads us to believe that it might be very fast *on average*. Thus, we now find the average running time of QUICKSORT.

The average running time of QUICKSORT, $T(n) = \frac{\sum_{S' \subset S} \text{Time take by QUICKSORT on } S'}{|S|}$

where $|S| = n!$. $T(n)$ can be simplified as follows:

$$T(n) = \frac{\sum_{i=1}^{n} \text{Time taken by QUICKSORT on all sequences with first element } i}{n!}$$

$$= \frac{\sum_{i=1}^{n} (n-1)!(T(i-1) \ + T(n-i) + \ cn)}{n!}$$

Since there are $(n-1)!$ sequences with first element i

$$= \frac{1}{n} \sum_{i=1}^{n} T(i-1) \ + T(n-i) + \ cn$$

$$= \Big(\frac{1}{n} \sum_{i=1}^{n} (T(i-1) \ + T(n-i)\Big) + \ cn$$

Thus,

$$nT(n) = \Big(\sum_{i=1}^{n} T(i-1) + T(n-i)\Big) + \ cn^2 \tag{0.1}$$

Substituting n by $n-1$, we get:

$$(n-1)T(n-1) = \Big(\sum_{i=1}^{n-1} T(i-1) + T(n-1-i)\Big) + \ c(n-1)^2 \tag{0.2}$$

Subtracting Equation 0.2 from Equation 0.1, we get,

$$\begin{aligned}
nT(n) - (n-1)T(n-1) &= T(n-1) + T(n-1) + c(n^2 - (n-1)^2) \\
nT(n) - (n-1)T(n-1) &= 2T(n-1) + c(2n-1) \\
nT(n) &\leq (n+1)T(n-1) + 2cn
\end{aligned}$$

Thus,

$$\frac{T(n)}{n+1} \leq \frac{T(n-1)}{n} + \frac{2c}{n+1} \tag{0.3}$$

Replacing n by $(n-1)$, we get

$$\frac{T(n-1)}{n} \leq \frac{T(n-2)}{n-1} + \frac{2c}{n} \tag{0.4}$$

At this point, we will use the substitution method and substitute Equation 0.4 in Equation 0.3. We get:

$$\begin{aligned} \frac{T(n)}{n+1} &\leq \frac{T(n-2)}{n-1} + \frac{2c}{n} + \frac{2c}{n+1} \\ &\leq \frac{T(n-3)}{n-2} + \frac{2c}{n-1} + \frac{2c}{n} + \frac{2c}{n+1} \end{aligned}$$

Thus, we can do repeated substitution to get the following:

$$\frac{T(n)}{n+1} \leq \frac{T(2)}{1} + \frac{2c}{2} + \frac{2c}{2} + \cdots + \frac{2c}{n-1} + \frac{2c}{n} + \frac{2c}{n+1}$$

Assuming that $T(1) = 2c$, we get:

$$T(n) \leq 2c(n+1)\Big(\frac{1}{1} + \frac{1}{2} + \frac{1}{2} + \cdots + \frac{1}{n-1} + \frac{1}{n} + \frac{1}{n+1}\Big)$$

The last term in the above equation is a harmonic series whose rough approximate value is $O(\log(n+1))$. Thus, $T(n) \approx 2c(n+1)\log(n+1)$. Thus $T(n) = O(n \log n)$.

Similar to MERGESORT, we can prove by induction that QUICKSORT correctly sorts an array. The reader is encouraged to prove it himself. We will be move on the an important aspect of QUICKSORT. In our current version of QUICKSORT, we use two extra array $S_<$ and $S_>$. Can we execute the PARTITION procedure without using extra, that is, using $O(1)$ extra memory only.

1 PARTITION using $O(1)$ extra memory

Since we have constant memory at our disposal, in one pass, we can find the final position of pivot. To this end, we will maintain two counters $c_<$ and $c_>$ and count the number of elements less than and greater than pivot.

```
$c_< \leftarrow 0$;
$c_> \leftarrow 0$;
$pivot \leftarrow A[1]$;
foreach $i \leftarrow 2$ to $n$ do
    if $A[i] < pivot$ then
        $c_< \leftarrow c_< + 1$ ;
    end
    else
        $c_> \leftarrow c_> + 1$ ;
    end
end
$swap(A[1], A[c_<])$;
```

Figure 3: The first step in our new PARTITION function

At the end of the above step, we claim that *pivot* is now at it correct place. However, we have still not *partitioned* the array. To do it, we have to go over the left array of the pivot and check if it contains any element greater than pivot. If yes, we have to move it to the array to the right of pivot. However, it means that we need to *displace* some element at the right. Since, we have only $O(1)$ extra memory with us, we cannot even store this out of place elements. So, it seems natural to *displace* that element on the right of pivot whose value is less than pivot. Our algorithm will mimic the above idea: after executing the the procedure in Figure 3, assume that we have place pivot at index j in the array A. Now, we maintain two counters c_1 which runs from 1 to $j-1$ and counter c_2 which runs from $j+1$ to n. We will use the following idea:

We will increment c_1 only if $A[c_1] < pivot$. Similarly, we will increment c_2 only if $A[c_2] > pivot$

Thus, we have the following algorithm:

```
$c_1 \leftarrow 1$;
$c_2 \leftarrow j+1$;
**while** *true* **do**
    **while** $A[c_1] < pivot$ *and* $c_1 \leq j-1$ **do**
        $c_1 \leftarrow c_1 + 1$;
    **end**
    **while** $A[c_2] > pivot$ *and* $c_2 \leq n$ **do**
        $c_2 \leftarrow c_2 + 1$;
    **end**
    **if** $c_1 = j$ *or* $c_2 = n+1$ **then**
        break;
    **end**
    **else**
        $swap(A[c_1], A[c_2])$;
        $c_1 \leftarrow c_1 + 1$;
        $c_2 \leftarrow c_2 + 1$;
    **end**
**end**
```

Figure 4: The last few steps in our new PARTITION function

We now prove the following important observation of the above *second step* of PARTITION.

Once can check that the running time of the *second* step of PARTITION also takes $O(n)$ time. This is due to the fact that counter c_1 can only be incremented from 1 to $j-1$ and c_2 can be incremented from $j+1$ to n.

We now prove the correctness of the above algorithm. To this end, we make the following important observation on our new algorithm:

Observation 1.1. *Whenever c_1 is incremented, $A[1 \dots c_1]$ contains all the elements less than pivot. When c_2 is incremented, $A[j+1 \dots c_2]$ contain all the elements greater than pivot.*

If both the counters c_1 and c_2 move to j and $n+1$ respectively, then by above observation, we would have partitioned the array. The only problem is to show that c_1 and c_2 will *always* be incremented to their respective ends. To this end, we prove the following important lemma:

Lemma 1.2. *Before each iteration of the outer while loop in Figure 4, total number of elements greater than pivot in $A[1 \dots j-1]$ = total number of elements less than pivot in $A[j+1 \dots n]$.*

Proof. We will prove the above statement when we first enter the while loop. The statement follows for all the other iteration of while loop, as in each iteration we swap an element greater than pivot from $A[1 \dots j-1]$ with an element less than pivot in $A[j+1 \dots n]$, thus maintaining the equality.

Consider the array A when it first enters the while loop. Since the first step PARTITION is already done (in Figure 3), the pivot is at it correct final index j. Assume for contradiction that total number of elements greater than pivot in $A[1 \dots j-1]$, say $x >$ total number of elements less than pivot in $A[j+1 \dots n]$, say y. This implies that the total number of element less than pivot in $A[1 \dots j-1]$ = $j-1-x$. This implies that the total elements in A less than pivot = $j-1-x+y < j-1$ (since $x > y$). However, this contradicts our calculation of final position of pivot in Figure 3. This procedure calculated that the correct place for pivot is j because there are $j-1$ elements in A that are less than pivot. Thus, we arrive at a contradiction. Thus, our assumption that $x > y$ is not true.

Similarly, we can prove that x cannot be less than y too. □

Using Lemma 1.2 and Observation 1.1 and induction, we can show the following lemma:

Lemma 1.3. *With only $O(1)$ extra memeory,* PARTITION *partitions the array into three parts (1) all the elements that are less than pivot. (2) pivot and (3) all elements greater than pivot.*

In the above PARTITION algorithm, we make two pass over the array. First, to find the final place of the pivot and second to find the left and right partition of pivot.

However, QUICKSORT is the best sorting algorithm because we can execute PARTITION in one pass with $O(1)$ extra memory.

2 PARTITION in one pass with $O(1)$ memory

One has to observe that there is no need to waste the first pass to find the final place of pivot. So, we will straight away go to the second step of PARTITION which partitions the left and right sub-array of pivot. However, there is a problem with this approach – we don't know where to start the second counter c_2. So, we will do the following hack: we will move the counter c_2 from n decrementing it till a certain condition is satisfied.

We will use the following idea:

We will increment c_1 if $A[c_1] < pivot$. Similarly, we will decrement c_2 only if $A[c_2] > pivot$. We will stop when c_1 crosses c_2.

```
$pivot \leftarrow A[1]$;
$c_1 \leftarrow 2$;
$c_2 \leftarrow n$;
while true do
    while $A[c_1] < pivot$ and $c_1 \leq c_2$ do
        $c_1 \leftarrow c_1 + 1$;
    end
    while $A[c_2] > pivot$ and $c_1 \leq c_2$ do
        $c_2 \leftarrow c_2 - 1$;
    end
    if $c_1 > c_2$ then
        break;
    end
    else
        $swap(A[c_1], A[c_2])$;
        $c_1 \leftarrow c_1 + 1$;
        $c_2 \leftarrow c_2 - 1$;
    end
end
$swap(A[1], A[c_2])$;
```

Figure 5: PARTITION that takes only one pass and take $O(1)$ extra space.

We make the following observation about the above algorithm:

Observation 2.1. *In the while loop, after c_1 and c_2 crosses all elements in $A[2 \ldots c_2]$ are less than the pivot and all the elements in $A[c_1 \ldots n]$ are greater than pivot.*

As the last step, we swap pivot (that is $A[1]$) with $A[c_2]$. This will ensure that the pivot is at its correct place. Using the above observation, all the elements to the left of the pivot, that is $A[1 \ldots c_1 - 1]$ have value $< pivot$. Similarly, all the elements to the right of the pivot, that is $A[c_1 + 1 \ldots n]$ have value $> pivot$. Thus, we have partitioned the array into three parts.