Quick Sort with Three Partitions

Ashish Sharma
Rengakrishnan Subramanian
November 8, 2001

This paper is concerned with finding a linear-time algorithm for implementing a function *pivotbis,* which is used in Quick Sort algorithm. The function partitions an array into three parts, elements less than, equal to and greater than an element pivot, which is the first element of the array. The algorithm scans the input array only once and uses no auxiliary arrays.

Algorithm

Prelude

The algorithm takes an input array T [i…j], where i=0, j >0. Pointers *k* and *l* are initialized to *i* +1 and *j+1* respectively and pointer *q* is initialized to 0. Pointer *k* is incremented until T [k] >p, where p=T [i] is the pivot element. At every increment of pointer *k*, we check whether *q* has been initialized, the initialization condition of *q* being T [k]=p. While incrementing k, we make sure that every element to the left of T [q] is always less than the pivot p. This is achieved by swapping the element T [k] with T [q] and incrementing q, if T [k]. Pointer l is decremented until T [l] ≤ p. Now T[k] and T[l] are interchanged. This process continues as long as $k < l$. Finally, T [q-1] and T [l] are interchanged to put the pivot in the correct position and to ensure that all elements to the left of q are strictly less than pivot and all the elements to the right of k are greater than pivot. The elements between q and k (q inclusive) are equal to the pivot.

Algorithm

procedure pivotbis (T [i…j])

{
Permutes the elements in the array T [i…j] and returns the value q and k such that, at the end, $i \leq q \leq k \leq j$, where $T[x] < p$ for all $i \leq x < q$, $T[x] = p$ for all $q \leq x < k$ and $T[x] > p$, for all $k \leq x \leq j$.
}

q ← 0
p ← T[i]
k ← i + 1
l ← j +1

```
while true do
        repeat
                if ( T [k] = p and q = 0 )
                        q ← k
                if ( T [k] < p and q > 0 )

                        swap (T [k], T [q])
                        q ← q + 1
                k ← k + 1

        until ( T [k] >p or k≥ j )

        repeat l ← l - 1 until ( T [l] ≤ p )

        if ( k < l )
                swap ( T [k], T [l] )
        else
                if ( q = 0 ) q ← l + 1
                swap ( T [q-1], T [i] )
                return(q-1, k)
        end if
end while
```

Example

An example working of the above algorithm is depicted here for the following set of input.

3	1	2	3	7	3	4	8	3	2	6	9	3	5	7

Initially, q = 0, p = pivot = 3, k = 2, l = 16 (k is stroked, l is underlined)

During the course of the algorithm, the above array changes as following:

(k is stroked, l is underlined, q is double-stroked)

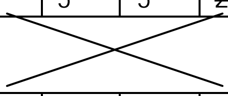
3	1	2	3	7	3	4	8	3	2	6	9	3	5	7

3	1	2	3	7	3	4	8	3	2	6	9	3	5	7

3	1	2	3	3	3	4	8	3	2	6	9	7	5	7

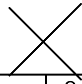
3	1	2	~~3~~	3	3	4	8	3	2	6	9	7	5	7

3	1	2	~~3~~	3	3	~~2~~	8	3	4	6	9	7	5	7



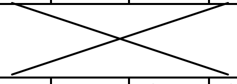
3	1	2	2	~~3~~	3	~~3~~	8	3	4	6	9	7	5	7

3	1	2	2	~~3~~	3	3	~~8~~	3	4	6	9	7	5	7



3	1	2	2	~~3~~	3	3	~~3~~	8	4	6	9	7	5	7

3	1	2	2	~~3~~	3	3	3	~~8~~	4	6	9	7	5	7



2	1	2	3	~~3~~	3	3	3	~~8~~	4	6	9	7	5	7

Finally, after the loop,

2	1	2	~~3~~	3	3	3	3	~~8~~	4	6	9	7	5	7

The procedure returns the following values

k = 8, q = 3, l = 7

Proof of Correctness

Proof by Loop Invariant

The loop invariant conditions for the above algorithm:

1. if $(q > 0) \wedge (i < x < q)$ then T [x] < pivot
2. if $(q > 0) \wedge (q \leq x < k)$ then T [x] = pivot
3. if $(l < x \leq j)$ then T [x] > pivot

We need to show that the loop invariants hold for each iteration of the loop. We show it by ensuring the invariants hold true:

a. At initialization – before the loop starts,
b. During maintenance – during the execution of the loop and
c. At termination – when the loop is completed

Initialization

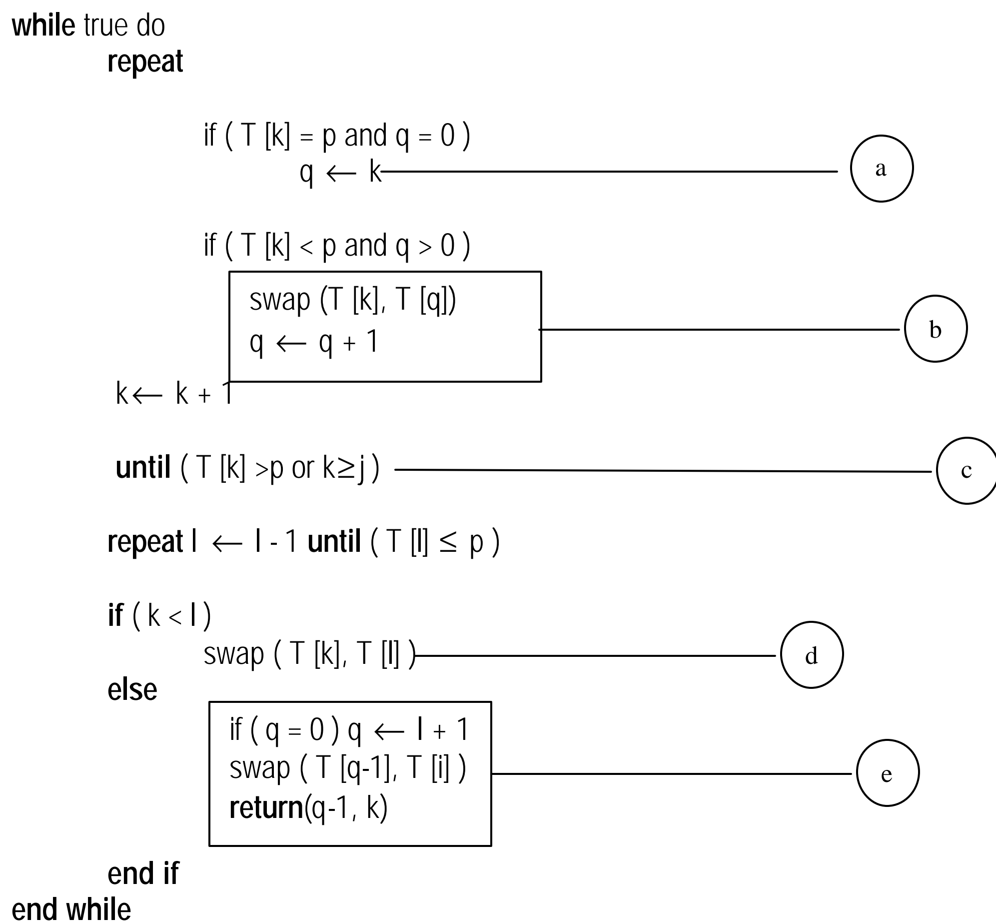
1. Before the first iteration,
   q = 0
   k = i + 1
   l = j + 1
   Since q = 0, the conditions 1 and 2 are met at initialization.

2. The 3rd condition states that x lies between *l* and *j*. However, at initialization, no such x exists, since, *l = j + 1*. So, condition 3 is also met at initialization.

Maintenance

To prove that the invariants hold during the iterations of the while loop, we will divide the algorithm into different set of statements (as shown below) and then prove that invariants hold during execution of each set of statement.

The algorithm is divided as follows:



Case a)

```
if ( T [k] = p and q = 0 )
        q ← k
```

If T [x] = pivot and q = 0, indicates that we have encountered the first element which is equal to the pivot and the element is at index k. If the condition is met, the assignment statement *q ← k* makes pointer q point to the first element in the array, which is equal to the pivot, thus, satisfying condition 2.
Note that if T [x] = pivot and also q > 0, it still satisfies condition 2, because all elements that lie between q and k (q inclusive) are equal to the pivot. Also, this indicates that all elements that lie between i and q (both i and q exclusive), are less than the pivot, thus, satisfying condition 1. Condition 3 is met because this statement does not affect the value of l.

Case b)

```
if ( T [k] < p and q > 0 )
        swap (T [k], T [q])
        q ← q + 1
```

If T [x] < pivot and q > 0, it indicates that we have encountered an element which is less than pivot and it also indicates that we have already encountered an element equal to pivot, whose position is pointed by q. By executing {Swap (T [q], T [k]), q ←q +1} statement, we ensure that all the elements which are less are less than pivot are positioned to the left of *q*, and q points to left most element which is equal to pivot, there by satisfying the invariant 1. Since k was the first element less than pivot occurring after element q, we can also say that by swapping it we can say that all the elements between q and k are equal to pivot thus satisfying invariant 2. Condition 3 is met because this statement does not affect the value of l.

Case c)

repeat l ← l - 1 **until** (T [l] ≤ p)

The statement *l ← l – 1* is repeated until an element that is greater than or equal to the pivot is met. Since l←j + 1 during the initialization of the loop, it is ensured that at the end of the above loop, all elements that lie between l and j are strictly greater than the pivot element, thus, satisfying the invariant 3.

Case d)

swap (T [k], T [l])

Before the execution of the statement, it is true that the element at *l* is less than or equal to the pivot element and the element at *k* is greater than the pivot element. After the swap, it is ensured that, the element at k is strictly less than or equal to the pivot and the element at l is strictly greater than the pivot element. We can thus say that all the elements between *l* and *j* are now greater than pivot satisfying invariant 3. Invariants 1 and 2 still hold because this statement does not affect values of *q* or *k.*

Case e)

if (q = 0) q ← l +1

swap (T [q-1], T [i])
return(q-1, k)

This set of statements are executed iff $k > l$. That is, this happens only once and as the last statement(s) of the while loop. If during the course of the algorithm, we never encountered an element that was equal to the pivot, we assign q value of $l + 1$ ($q \leftarrow l + 1$). If q > 0 at this point, we know that q points to the leftmost element which is equal to pivot. So, we can safely swap the element T [q-1] and T [i] (Remember that T [i] = pivot). At the end of the swap statement, it is ensured that all elements to the left of q are strictly less than pivot and all elements to the right of l are strictly greater than pivot, thus invariant 3 is satisfied. As we know that element at q points to second left most element equal to q to and by returning q-1 we are effectively returning leftmost element equal to pivot, thus we satisfy invariant 2. Also since pointer q points to the leftmost element in T, all the elements to left to q are less than pivot satisfying invariant 1.

Termination

At termination, k > l and the elements are divided into three sets:

a. A set with all elements less than pivot. These are the elements to the left of q-1.
b. A set with all elements equal to pivot. These are the elements that lie between q-1 and k (q-1 inclusive).
c. A set with all elements greater than pivot. These are the elements that lie between k and j (both k and j inclusive).

Thus, conditions 1, 2 and 3 are met at the termination of the algorithm.

Time Complexity Analysis

We can observe that in order for the algorithm to terminate k and l must cross each other. The statement which determines the number of iterations of the loop are $k = k + 1$ and $l = l - 1$. Hence the runtime complexity of the algorithm can be stated as Ω (n).
Assume that every swap takes O (1). In the worst case, the *swap (T [k], T [q])* happens for every $k = k + 1$, except first i.e. (the first element of k is equal to pivot and the rest of elements in the array are less than pivot), hence $k = j$ at the end of the first repeat loop. In this case, there are *n-1* swaps, taking a time of O (n-1). Also, we can observe that, in this worst case, the *swap (T [k], T [l])* will not be executed, because k=l.
If the *swap (T [k], T [l])* happens for every iteration of the while loop, it can happen for at most n/2 times, where each swap take O (1) time, so the total time is still O (n).
In conclusion, we can say that *pivotbis* runs at Θ (n).

References

[1] Gilles Brassard and Paul Bratley. *Fundamentals of Algorithmics.* Prentice Hall of India, 1998.