Quick Sort and Order Statistics

Arnab Ganguly, Assistant Professor
Department of Computer Science, University of Wisconsin – Whitewater
Theory of Algorithms (CS 433)

1 Quick Sort

Quick-sort is yet another **divide-and-conquer** algorithm for sorting numbers. However, as opposed to merge-sort discussed previously, this has an advantage. Quick-sort is an **in-place** algorithm, i.e., we do not require any extra space other than the input array and some variables. Recall that merge-sort on the other hand requires an extra array. This makes quick-sort more space-efficient (with almost consuming roughly half the space that of merge sort).

1.1 Recursive Algorithm for Quick Sort

- First select a number in the array as **pivot** (based on some rules or arbitrarily).
- Then call the **partition method**, which reorders the array in the following manner:
  - LEFT-PART: numbers to the left of the pivot are $\leq$ pivot, and
  - RIGHT-PART: numbers to the right of the pivot are $>$ pivot.
- Finally, recursively quick-sort the LEFT-PART and the RIGHT-PART
- Note that once the array has one element, sorting is not necessary; hence, we stop.

1.2 The Partition Method

Step 1: First, we locate the index in the array where the pivot will lie after partition; this is exactly the x^{th} index starting from $left$ of the array, where x is the number of values in the $left$ through $right$ portion of the array that are less than or equal to the pivot. Call this index, the $partitionIndex$. Also, find an index in the array that contains the pivot; call this the $pivotIndex$.

Step 2: Now bring the pivot to the correct spot by swapping the array contents at $partitionIndex$ and $pivotIndex$.

Step 3: Sweep a variable i from $left$ to $partitionIndex$ until we see a number higher than the pivot. Sweep a variable j from $right$ to $(partitionIndex + 1)$ until we see a number less than or equal to the pivot. If $i < j$, then $A[i]$ and $A[j]$ are out of order; hence, swap them; move i and j. Repeat this until i crosses j.

See next for an illustration and the algorithm.[1]

[1] You can also check CLRS for a slightly more (practically) efficient implementation.

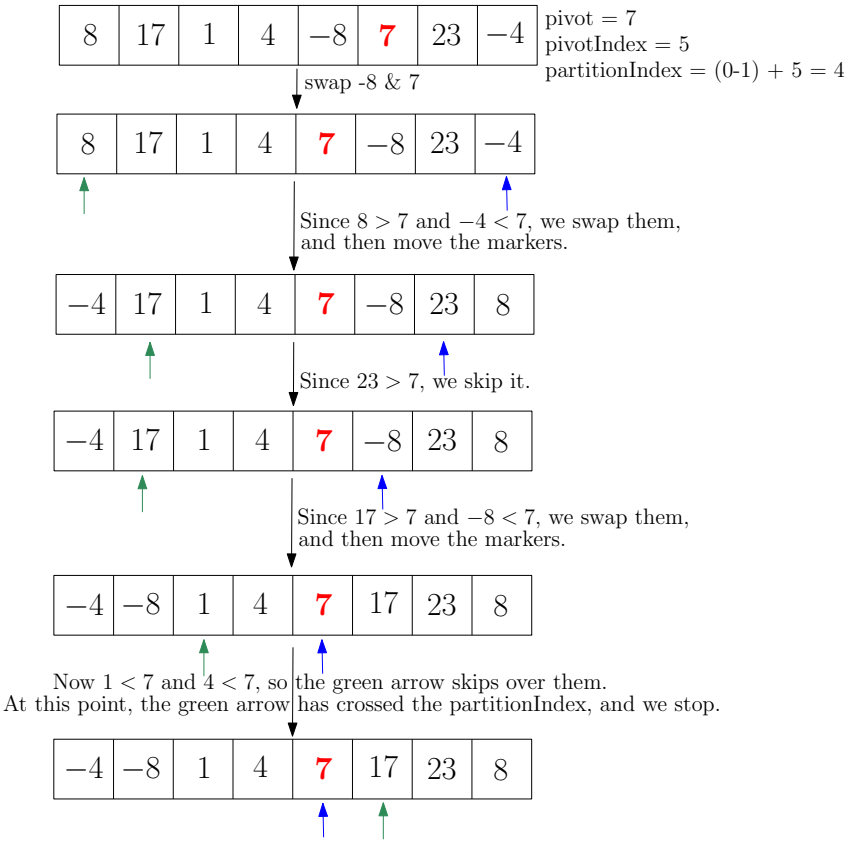

Figure 1: Illustration of the Partition method. Bold red text indicates the pivot.

Algorithm 1 Partition Method

```
function PARTITION(int A[ ], int left, int right, int pivot)
    int pivotIndex = left, partitionIndex = (left – 1);
    for (k = left to right) do
        if (A[k] equals pivot) then
            pivotIndex = k;
        if (A[k] ≤ pivot) then
            partitionIndex++;
    swap A[pivotIndex] and A[partitionIndex]
    int i = left, j = right;
    while (i < j) do
        increment i as long as (i ≤ partitionIndex && A[i] ≤ pivot);
        decrement j as long as (j > partitionIndex && A[j] > pivot);
        if (i < j) then
            swap A[i] and A[j];
            i++;
            j--;
    return partitionIndex;
```

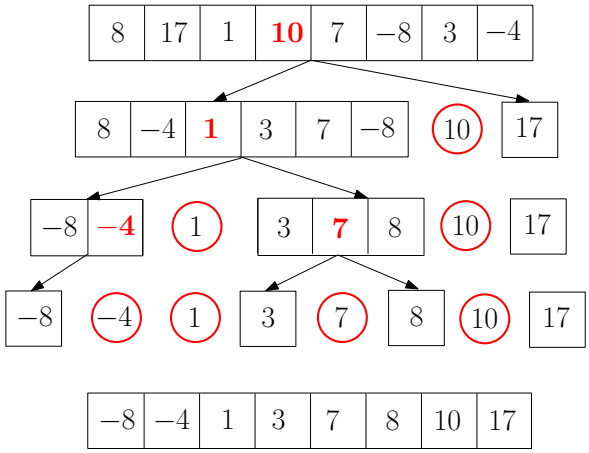

Figure 2: Illustration of the Quick-sort algorithm. Bold red text indicates a pivot. The array is partitioned around the pivot – left part is smaller or equal, whereas the right side is larger.

Algorithm 2 Quick-sort Algorithm

```
function QUICKSORT(int A[ ], int left, int right)
    if (left < right) then
        int pivot = GENERATEPIVOT(A, left, right);
        int partitionIndex = PARTITION(A, left, right, pivot);
        QUICKSORT(A, left, partitionIndex – 1);
        QUICKSORT(A, partitionIndex + 1, right);
```

1.3 Pivot Generation and Complexity

Let $T(n)$ be the time take to quick-sort the array, where n is the length of the array. Note that the partition procedure takes time $\Theta(n)$ in the worst case. The complexity of quick-sort is determined based on how the array is partitioned, which in turn depends on the choice of the pivot.

Selecting the first or last number in the array as Pivot. Easy to implement, but can easily become very slow. For example, say the array is sorted (either ascending or descending). Then, this pivot will always partition the array into one part, whereas the other part will be empty; hence, in the worst-case, $T(n) = T(n-1) + \Theta(n)$. Hence, $T(n) = \Theta(n^2)$.

Median of Three as Pivot. Select the median of the first, the last, and the middle numbers in the array. Note that this overcomes the problem of the previous case (when the array is sorted and we select the first or last number as pivot); in fact, in this case, the median of three will always divide the array into two equal parts. Hence, in the best case $T(n) = 2T(n/2) + \Theta(n)$, i.e., $T(n) = \Theta(n \log n)$. In the worst case, however, it can be shown that this still has complexity $\Theta(n^2)$.

Median of the Array as Pivot. In this scenario, the partition always divides the array into two roughly equal parts. Moreover, there exists deterministic algorithms that can find the median in $O(n)$ time in the worst case[2]; hence, $T(n) = 2T(n/2) + \Theta(n)$. Thus, $T(n) = \Theta(n \log n)$.

[2] We shall discuss a simpler (randomized) version of this; for the deterministic method, you can check CLRS.

Random Array Entry as Pivot. Choose a pivot randomly. The worst case is $\Theta(n^2)$, but the best case is $\Theta(n \log n)$.[3] In fact, this method is quite practical (partly due to its simplicity), and the average case complexity is $\Theta(n \log n)$.

2 Order Statistics

Problem Statement: Given an array A of n numbers and a positive integer $k \leq n$, find the k^{th} smallest number in the array A.

One simple way to solve the problem is to sort the array in $\Theta(n \log n)$ time (using say Merge Sort) and then simply return $A[k-1]$.[4] In what follows, we see how one can employ the PARTITION method to solve this problem.

2.1 Selection/Quick-Select Algorithm

Suppose, we partition the array $A[left, right]$ and obtain $partitionIndex$. Recall that there are exactly $(partitionIndex - left + 1)$ numbers in $A[left, right]$ which are less than or equal to the pivot. If k equals $(partitionIndex - left + 1)$, then our desired answer is the pivot. If $k < (partitionIndex - left + 1)$, then the answer is the k^{th} smallest number to the left of the pivot in the partitioned array. Otherwise, there are $(partitionIndex - left + 1)$ many numbers smaller than the answer in the partitioned array; hence, we recursively look for the $(k-(partitionIndex-left+1))^{th}$ smallest number to the right of the pivot in the partitioned array.

See next for the algorithm and an illustration of it.

Algorithm 3 Selection Algorithm

```
function QUICKSELECT(int A[ ], int left, int right, int k)
    if (left == right) then
        return A[left];
    int pivot = GENERATEPIVOT(A, left, right);
    int partitionIndex = PARTITION(A, left, right, pivot);
    if (k equals partitionIndex − left + 1) then
        return pivot;
    else if (k < partitionIndex − left + 1) then
        return QUICKSELECT(A, left, partitionIndex – 1, k);
    else
        return QUICKSELECT(A, partitionIndex + 1, right, k – (partitionIndex – left + 1));
```

2.2 Complexity

Let $T(n)$ be the time taken by the selection algorithm, where n is the length of the array. The complexity, once again, is determined by the choice of the pivot.

[3] In the worst case, you may be extremely unlucky and always end up selecting the minimum (or maximum). In the best case, you may be extremely lucky and always end up selecting the median of the array.

[4] We can improve this to $O(n + k \log n)$ using heaps. Note that $k \leq n$; hence, this is asymptotically $O(n \log n)$ and much better for small values of k.

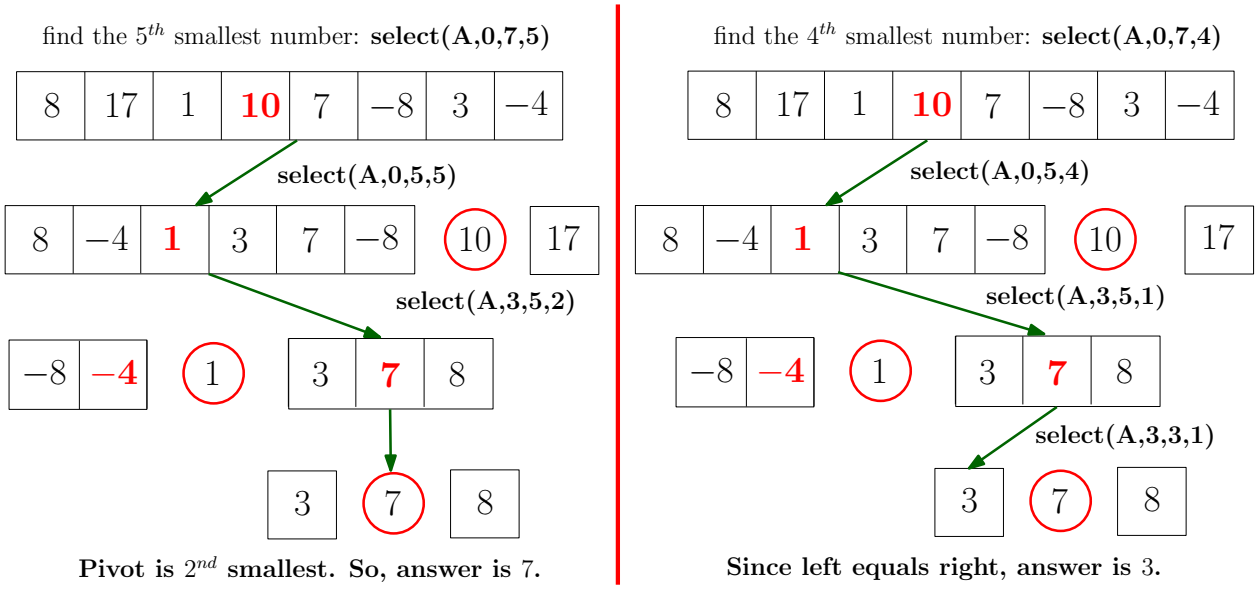

Figure 3: Illustration of the Selection algorithm. Bold red text indicates a pivot. Green arrows shows the direction in which the search progresses.

Let us start with the **median of the array as pivot**. Note that as oppose to quick sort, we will only recurse on one half of the array; hence, $T(n) = T(n/2) + \Theta(n)$. Solving, we get $T(n) = \Theta(n)$.

If we use **the first or last number in the array as pivot** or if we use the **median of three as pivot**, then $T(n) = \Theta(n^2)$ in the worst case. In the best case (by sheer luck in the first/last number method or when the array is sorted in the median of three method), we end up selecting the median, resulting in $T(n) = \Theta(n)$.

If **choose a pivot randomly**, the worst case, as in quick sort, is $\Theta(n^2)$; in the best case, complexity is $\Theta(n)$. However, this method is pretty practical (partly due to its simplicity); in fact, the average case complexity is $\Theta(n)$.