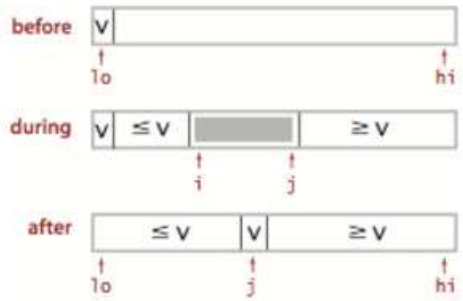
3-way Quicksort

by
Chingwei Lin
Taeko Owaki

Background

Quicksort is an efficient sorting algorithm by putting the pivot value at the correct position for each run, dividing the group as 'less than or equal' and 'larger than or equal' the pivot value, and then recursively doing the same for each partition until the whole array is sorted. Below is an overview of the quicksort.

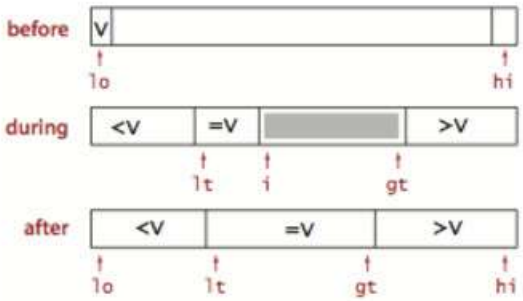


Quicksort partitioning overview

However, in the reality, not all the values are unique in unsorted data. For example, in {4, 6, 4, 6, 4, 9, 4, 3, 4, 1}, if we pick '4' as the pivot in a basic quicksort, we only place the first '4' at the correct position in the first run and recursively process remaining occurrences.

3-way Quicksort

3-way quicksort is designed to optimize the basic quicksort when there are many repeated values in the unsorted data. Each run, all the same pivot values are placed at their correct position. Below is an overview of the 3-way quicksort.



3-way partitioning overview

Implementation

The biggest difference is the partitioning part. With a little change of the code and a few more pointers we can improve the quicksort with 3-way partitioning.

[definition]
lo: the first element of the input array
hi: the last element of the input array
lt: the value less than the pivot
gt: the value greater than the pivot
i: the index of the array

[algorithm]

1. Choose the first element as the pivot
2. While the current index does not pass the gt pointer
    a. If [index] is less than the pivot, swap with [lt], and increment both pointers
    b. If [index] is greater than the pivot, swap with [gt], and decrement the gt pointer
    c. If [index] is equal to the pivot, increment the index
3. Recursively sort the left subarray
4. Recursively sort the right subarray

[code]

```
private static void sort(int[] a, int lo, int hi) {
      if (hi <= lo)
         return;

      int lt = lo, gt = hi;
      int v = a[lo];
      int i = lo;

      while (i <= gt)
      {
         if(a[i] < v)
         {
            swap(a, lt++, i++);
         }
         else if (a[i] > v)
         {
            swap(a, i, gt--);
         }
         else
            i++;
      }

      // a[lo..lt-1] < v = a[lt..gt] < a[gt+1..hi].
      sort(a, lo, lt-1);
      sort(a, gt+1, hi);
   }
```

Example

Considering the unsorted input of {4, 6, 4, 6, 4, 9, 4, 3, 4, 1}, the basic quicksort needs 7 runs to finish the sorting, while the 3-way quicksort needs 5 runs.

[Quicksort]

4 6 4 6 4 9 4 3 4 1
4 1 4 3 4 9 4 6 4 6
3 1 4 4 4 9 4 6 4 6
1 3 4 4 4 9 4 6 4 6
1 3 4 4 4 6 4 6 4 9
1 3 4 4 4 4 4 6 6 9
1 3 4 4 4 4 4 6 6 9

[3-way Quicksort]

4 6 4 6 4 9 4 3 4 1
1 3 4 4 4 4 4 9 6 6
1 3 4 4 4 4 4 9 6 6
1 3 4 4 4 4 4 6 6 9
1 3 4 4 4 4 4 6 6 9

Performance Test

Input: 100000000 array items, value: 0~1000
Quicksort: 5.437s
3-Way Quicksort: 4.267s

Input: 100000000 array items, value: 0~10
Quicksort: 3.834s
3-Way Quicksort: 1.438s

Input: 100000000 array items, value: same
Quicksort: 2.669s
3-Way Quicksort: 0.066s

Input: 10000 array items, already sorted in increasing order
Quicksort: 0.025s
3-Way Quicksort: 0.0s

Input: 10000 array items, already sorted in decreasing order
Quicksort: 0.032s
3-Way Quicksort: 0.084s

Conclusion

3-way quicksort is a better version than the traditional quicksort in many cases. Even for the worst case it has only slightly degradation than the traditional one. Although the time complexity for both algorithms are the same, O(NlogN) for the average case and $O(N^2)$ for the

worst case, 3-way quicksort can have less swap operations. And there is no reason to not improve the performance with only a few changes of the code.