Interview questions: Quicksort

Question 1

Nuts and bolts. A disorganized carpenter has a mixed pile of n nuts and n bolts. The goal is to find the corresponding pairs of nuts and bolts. Each nut fits exactly one bolt and each bolt fits exactly one nut. By fitting a nut and a bolt together, the carpenter can see which one is bigger (but the carpenter cannot compare two nuts or two bolts directly). Design an algorithm for the problem that uses at most proportional to nlogn compares (probabilistically).

Hint:

modify the quicksort partitioning part of quicksort. Remark: This research paper gives an algorithm that runs in $n\log^4 n$ time in the worst case.

Algorithm design

Data structure: two arrays of n Comparable items of type Nut and Bolt.

Both types have common property - a size, which in metric system expressed in millimeters and is denoted as 'M<N>' where N expresses a diameter (in mm) of part of the nut or bolt with a thread, i.e. M2, M3, M4 etc. Any bolt and nut can be fastened with each other ('fits') if they have same size, 'M8' for example (simplified case, neglecting other characteristics, like 'pitch/lead of thread', which must be also equal for proper compatibility).

- Fitting nuts and bolts requires resolving a matching problem.

1) Partition, after shuffling(!), one of the array -Nuts , with a pivot Bolt of the particular size, i.e. 'M8' as example, so that:

- Item/Nut a[j] is in place
- No larger item/nut to the left of j (only 'M5', 'M4', 'M3' and so on)
- No smaller item/nut to the right of j (only 'M8', 'M10',

'M12' and larger).

- Takes time O(n)

2) Next, using nut a[j] as a pivot - partition another array (Bolts),- same as Nuts, also takes time O(n) .

First and second operations also make nuts and bolts array partitioned. Now we apply this partitioning recursively on the left and right sub-array of nuts and bolts.

Note: "This recursive algorithm strongly resembles quick-sort and it's well-known that the ‘average-case’ running time of quicksort is Θ(n log n). Consequently, the average number of nut-bolt comparisons is also Θ(n log n). The proposed algorithm not only matches the Nuts and Bolts, but also sorts them by size. In fact, the problems of sorting and matching nuts and bolts are equivalent, in the following sense. If the bolts were sorted, we could match the nuts and bolts in O(n log n) time by performing a binary search with each nut. Thus, if we had an algorithm to sort the bolts in O(n log n) time, we would immediately have an algorithm to match the nuts and bolts, starting from scratch, in O(n log n) time." (Randomized Algorithms, Jeff Erickson)

Question 2

Selection in two sorted arrays. Given two sorted arrays $a[]$ and $b[]$, of lengths $n1$ and $n2$ and an integer $0 \le k < n1+n2$, design an algorithm to find a key of rank k. The order of growth of the worst case running time of your algorithm should be logn, where n=n1+n2.

Hint: there are two basic approaches.

- Approach A: Compute the median in a[] and the median in b[]. Recur in a subproblem of roughly half the size.
- Approach B: Design a constant-time algorithm to determine whether a[i] is a key of rank k. Use this subroutine and binary search.

Dealing with corner cases can be tricky.

- Version 1: n1=n2 (equal length arrays) and k=n/2 (median).

As two arrays a[] and b[] are sorted and have equal length (n1=n2), the key of rank median k/2 is:

- Option I (corner case), b[0] >= a[n1] or b[n2] <=a[0], i.e. arrays are not intersected (except, possibly, "border" elements) , then k/2 = (a[n1] + b[0])/2 or (b[n2] + a[0])/2 .

Order of growth of the worst case running time is constant: upper bound O(n) = lower bound W (n) = 1

- Option II, arrays are intersected (source https://www.geeksforgeeks.org/median-of-two-sorted-arrays/)

1) Take the medians m1 and m2 of the input arrays a[] and b[] respectively.
2) If m1 and m2 both are equal then we are done.
   return m1 (or m2)
3) If m1 is greater than m2, then median is present in one of the below two subarrays.
   a) From first element of a to m1 (a[0...l_n1/2_l])
   b) From m2 to last element of b (b[l_n2/2_l...n2-1])
4) If m2 is greater than m1, then median is present in one of the below two subarrays.
   a) From m1 to last element of a (a[l_n1/2_l...n-1])
   b) From first element of b to m2 (b[0...l_n2/2_l])
5) Repeat the above process until size of both the subarrays becomes 2.
6) If size of the two arrays is 2 then use below formula to get the median.
   Median = (max(a[0], b[0]) + min(a[1], b[1]))/2

Order of growth of the worst case running time is logn (n=n1+n2), as each recursive step divides each subarray by half.

- Version 2: k=n/2 (median).

Length of arrays a[] and b[] - not equal (the first array a[] is smaller than the second array b[]); a[] and b[] are 0-based indexed.

The approach is similar to Version 1,- get the median of

both arrays a[] and b[], discard half of each array depends on medians comparison.

First of all, consider the following corner cases (source https://www.geeksforgeeks.org/median-of-two-sorted-arrays-of-different-sizes/):

If the size of smaller array a[n1] is 0. Return the median of a larger array b[n2/2] (odd) or (b[n2/2] +b[n2/2-1])/2 (even elements).

1. If the size of smaller array a[] is 1,- a[0]:
1. The size of the larger array b[] also 1 - return the median between two numbers: (a[0]+b[0])/2;
2. If the size of the large b[] array is odd, then merged output array will be even and the median will be an average of two mid elements . The element from the smaller array a[0] will affect the median if and only if it lies in the range from (n2/2 – 1)th to (n2/2 + 1)th element of the larger array (i.e. a[0] will be one of the two mid elements). So, find the median in between the four elements,- the element of the smaller array a[0], b[n2/2], b[n2/2 – 1] and b[n2/2 + 1] element of a larger array;
3. Similarly, if the size of the large b[] array is even, then merged output array will odd and the median will be the mid element. So, check for the median in between of the three elements,- the element of the smaller array a[0], b[n2/2]th and b[n2/2 – 1]th element of a larger array (i.e., a[0] will be the median if b[n2/2] <= a[0] >= b[n2/2 – 1], otherwise it will be one of the b[n2/2]th or b[n2/2 – 1]th).
2. If the size of the smaller array a[] is 2:
1. In case the large array b[] has also size 2,- calculate the median by the formula:

Median = (max(a[0], b[0]) + min(a[1], b[1]))/2

1. If the larger array b[] has an odd number of elements (beginning from 3) , then the median will be one of the following three elements (simply, compare/sort that three to find the middle one which is the median of the merged output of two initial arrays):

- Middle element of larger array b[]

- Max of the first element of smaller array and element just before the middle, i.e b[n2/2-1]th element, in a larger array;
- Min of the second element of smaller array and element just after the middle in the bigger array, i.e b[n2/2+1]th element in the bigger array.

1. If the larger array has **even** number of elements (beginning from 4), then the median will be one of the following 3 elements
   - The middle between two elements of the larger array;
   - Max of the first element of smaller array and element just before the first middle element in the bigger array, i.e b[n2/2-2]nd ('0'-based indexing) element;
   - Min of the second element of smaller array and element just after the second middle element in the bigger array, b[n2/2+1]th element;

- Assuming that size of both arrays is greater than 2 and the first array a[] is smaller than the second array b[]. If the median of the first (smaller) array a[n1/2] is less than the median of second (larger) b[n2/2] , then all elements of first half of a[] will be in the first half of the output (merged) array.
- Version 3: no restrictions.

Length of arrays a[] and b[] - not equal; also, let's assume that the first array a[] is smaller than the second array b[]); rank of k - any integer from $0 \le k < n1+n2$; also, $k>0$; a[] and b[] have 0-based index.

- Corner cases:

1. If length of one of the arrays is 0, the answer is kth element of the second array;
2. If k== 1, take min element out of a[0], b[0];
3. Arrays are not intersected (except, possibly, "border" elements) ,
   a. If (a[n1]<=b[0]), then the answer is:
      i. If k <= n1,- a[k - 1];
      ii. else,- b[k - n1 - 1].
   a. If b[n2] <=a[0], the answer is:
      i. If k <= n2,- b[k - 1];

ii. else,- a[k - n2 - 1].

4. If k <= n1 and a[k - 1] < b[0],- the answer is a[k - 1];

5. If k <= n2 and b[k - 1] < a[0],- the answer is b[k - 1].

Order of growth of the worst case running time is constant: upper bound O(n) = lower bound W (n) = 1

- General approach is: comparing the middle index elements from two subarrays based on their length condition (source https://stackoverflow.com/questions/40189065/find-kth-smallest-element-from-two-sorted-arrays?noredirect=1&lq=1).
- Simplifying trick: during recursion on each iteration, exchange parameters order to maintain that the first array (with it's current index) is always smaller to avoid writing additional if-else conditions (i.e. before comparing of middle elements, ensure that a[].length <= b[].length, otherwise - swap them by recalling a 'findKth' function with smaller arrays as the first parameter;
- Also, a[middle] to be calculated as min out of k/2 or a[].length - index of the current element of a[] (remember, that a[] is a smaller array on the current iteration);
- On each iteration, recalculating the middle index elements from two subarrays, ensure invariant of a[middle] + b[middle] == k;
- 'index of the current element', - 'i'- for a[], 'j'- for b[], starts from 0, on each recursive iteration increases by middle index element of corresponding array depending on comparison of middle elements,-
  - if a[i + middle] <= b[j + middle] make next iteration with i= i + a'th middle index, k= k - a'th middle index;
  - Else, with j= j + b'th middle index,k= k - b'th middle index;

Example of 'findKth' on Java:

```
findKth(int[]A,inti,int[]B,intj,intk) {
if((A.length-i)>(B.length-j))
{
```

```
return findKth(B,j,A,i,k);
}

if(i>=A.length)
{
returnB[j+k-1];
}
if(k==1)
{
return Math.min(A[i],B[j]);
}

Int aMid=Math.min(k/2,A.length-i);
Int bMid=k-aMid;

if(A[i+aMid-1]<=B[j+bMid-1])
{
return findKth(A,i+aMid,B,j,k-aMid);
}
returnfindKth(A,i,B,j+bMid,k-bMid);
}
```

Order of growth of the worst case running time is: log(k) = log(n1+n2)

Question 3

Decimal dominants. Given an array with n keys, design an algorithm to find all values that occur more than n/10 times. The expected running time of your algorithm should be linear.

Note: A value in the array is called a decimal dominant if it occurs more than n/10 times in the array.

Hint: determine the n/10-th largest value using 'Quick-select' and check if it occurs more than n/10 times.

Alternative hint: use 9 counters

I. Using Quick-select (QS) algorithm

General approach: If a value appears k times and the array is divided into intervals of length k, then the value must appear at the end of one of the intervals.

Algorithm design: finding all values that appear n/k times: compute the $i \cdot n$/k-largest elements (k 'quantile' points) for $1 \leq i \leq$ k, and for each of them, check whether the value appears at least n/k times. Each iteration takes $O(n)$ time using Quick-select, for a total of $O(kn)$ time.

Possible optimisation (source: https://stackoverflow.com/questions/50558730/how-to-find-all-values-that-occur-more-than-n-k-times-in-an-array, rici)

- since the k quantile points are sorted, you could reduce the k to log k by doing one scan and binary searching each element in the list of quantiles. But since k is fixed, it's O(n) regardless;
- find the same element which is presented inside k quantile points twice or more, which in turn guarantees that the value appears at least n/k times. This save from having to do a scan for it. But one still have to scan for the other ones (if there are any).

II. Boyer-Moore Voting algorithm (https://www.geeksforgeeks.org/majority-element/)

General approach: finds all the elements that occurs more than n/k in a input which runs in O(nk). This is a two-step process.

- The first step gives the element that maybe the majority element in the array. If there is a majority element in an array, then this step will definitely return majority element, otherwise, it will return candidate for majority element.
- Check if the element obtained from the above step is majority element. This step is necessary as there might be no majority element.

Step 1: Finding a Candidate

The algorithm for the first phase that works in O(n) is known as

Moore's Voting Algorithm. Basic idea of the algorithm is that if each occurrence of an element *e* can be cancelled with all the other elements that are different from *e* then *e* will exist till end if it is a majority element.

Step 2: Check if the element obtained in step 1 is majority element or not.

Traverse through the array and check if the count of the element found is greater than half the size of the array, then print the answer else print "No majority element".

Algorithm design:

1. Loop through each element and maintains a count of majority element, and a majority index, *maj_index*
2. If the next element is same then increment the count if the next element is not same then decrement the count.
3. if the count reaches 0 then changes the maj_index to the current element and set the count again to 1.
4. Now again traverse through the array and find the count of majority element found.
5. If the count is greater than half the size of the array, print the element
6. Else print that there is no majority element

———————————————————————————————