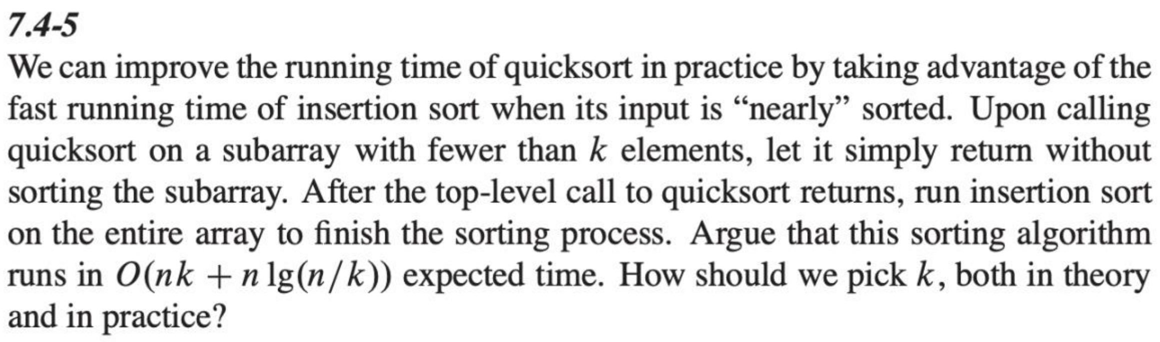


7.4-5

We can improve the running time of quicksort in practice by taking advantage of the fast running time of insertion sort when its input is "nearly" sorted. Upon calling quicksort on a subarray with fewer than k elements, let it simply return without sorting the subarray. After the top-level call to quicksort returns, run insertion sort on the entire array to finish the sorting process. Argue that this sorting algorithm runs in $O(nk + n \lg(n/k))$ expected time. How should we pick k, both in theory and in practice?

Based on the insertion sort algorithm, the run time complexity of sorting the arrays would be O(nk),
If we use quick sort untilt he problem size becomes smaller than k, we need lg(n/k) step,

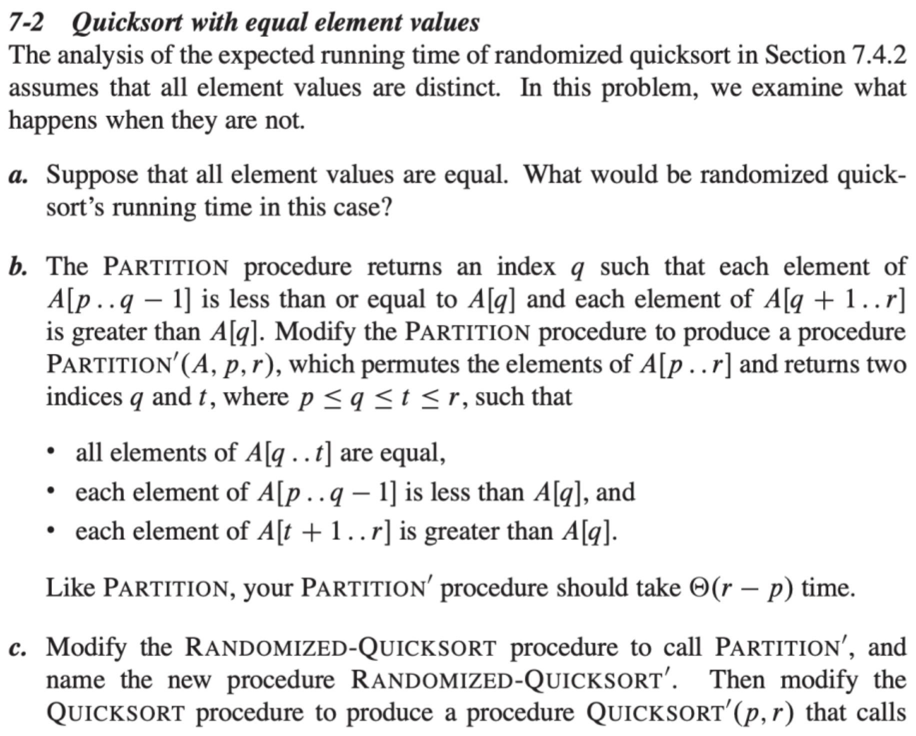
In theory, we should pick k to minimize the complexiy, where k could satisfy the equation:

C as constant

c_1 * nlgb >= c_2 n k + c_1 n lg(n/k)

So that we have lgk >= c2/c1 * k

In practice, we need a huge data set to try to find the best various values of K.



7-2 ***Quicksort with equal element values***

The analysis of the expected running time of randomized quicksort in Section 7.4.2 assumes that all element values are distinct. In this problem, we examine what happens when they are not.

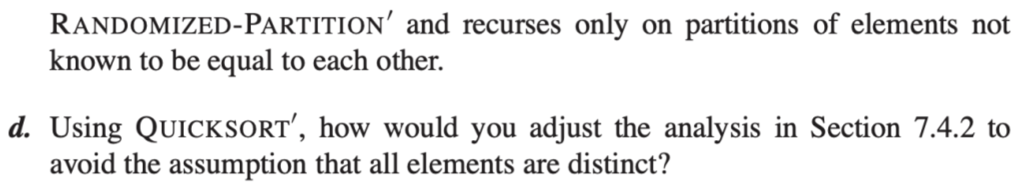
a. Suppose that all element values are equal. What would be randomized quicksort's running time in this case?

b. The PARTITION procedure returns an index q such that each element of $A[p \mathinner{..} q-1]$ is less than or equal to $A[q]$ and each element of $A[q+1 \mathinner{..} r]$ is greater than $A[q]$. Modify the PARTITION procedure to produce a procedure PARTITION$'(A, p, r)$, which permutes the elements of $A[p \mathinner{..} r]$ and returns two indices q and t, where $p \le q \le t \le r$, such that

- all elements of $A[q \mathinner{..} t]$ are equal,
- each element of $A[p \mathinner{..} q-1]$ is less than $A[q]$, and
- each element of $A[t+1 \mathinner{..} r]$ is greater than $A[q]$.

Like PARTITION, your PARTITION$'$ procedure should take $\Theta(r-p)$ time.

c. Modify the RANDOMIZED-QUICKSORT procedure to call PARTITION$'$, and name the new procedure RANDOMIZED-QUICKSORT$'$. Then modify the QUICKSORT procedure to produce a procedure QUICKSORT$'(p, r)$ that calls



RANDOMIZED-PARTITION$'$ and recurses only on partitions of elements not known to be equal to each other.

d. Using QUICKSORT$'$, how would you adjust the analysis in Section 7.4.2 to avoid the assumption that all elements are distinct?

A,

Since all elements are equal, randomized quicksort, the run time would always q = r so that the time would be T(n) = T(n- 1) + O(n) = O(n ^ 2)

B,

```
PARTITION'(A, p, r)
    x = A[p]
    low = p
    high = p
    for j = p + 1 to r
        if A[j] < x:
            y = A[j]
            A[j] = A[high + 1]
            A[high + 1] = A[low]
            A[low] = y
            low = low + 1
            high = high + 1
        else if A[j] == x
            exchange A[high + 1] with A[j]
            high = high + 1
    return (low, high)
```

C,

```
QUICKSORT'(A, p, r)
    if p < r
        (low, high) = RANDOMIZED-PARTITION'(A, p, r)
        QUICKSORT'(A, p, low - 1)
        QUICKSORT'(A, high + 1, r)
```

D,

3 similar to figure 8.2, illustrate the operation of counting-sort on
A = [5,7,3,1,3,6,2,1,3,5]

Answers:

First we need to create a count array to store the count of each unique object,

Then we modigy the count array such that each element at each index stores the sum of previous counts,

The modified count array indicateds the position of each object in the outpout sequences,

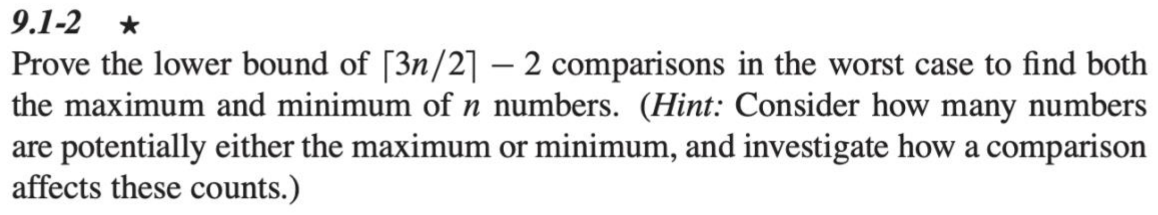
Then we rotates the array clockwise for one time,

The, Output each object from the input sequence followed by
increasing its count by 1

the process would be that :

count array:
[0, 0, 0, 0, 0, 0, 0]
[2, 1, 3, 0, 2, 1, 1]
[2, 3, 6, 6, 8, 9, 10]
[0, 2, 3, 6, 6, 8, 9]

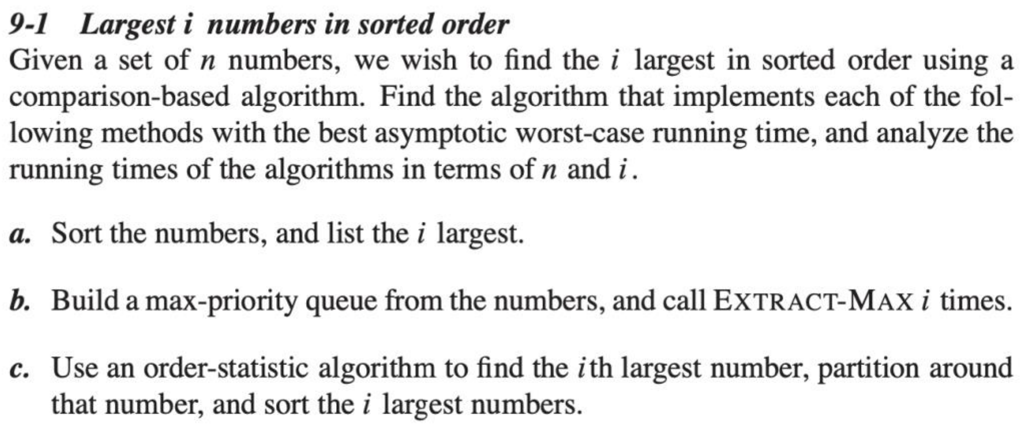
Outputarray =
[0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[1, 1, 2, 3, 3, 3, 5, 5, 6, 7]



9.1-2 ★
Prove the lower bound of $\lceil 3n/2 \rceil - 2$ comparisons in the worst case to find both the maximum and minimum of n numbers. (*Hint:* Consider how many numbers are potentially either the maximum or minimum, and investigate how a comparison affects these counts.)

Any number in the n numbers could be either the max one or the min one at the original status.
Set max array and min array,
We pick the items from the list of numerbers, a, and b, if a <= b, we remove a from max and remove b from min, reduce the counts of both sets by 1, there are n/2 comparisions until min and max are disjoint, and the sets will have size of n/2, we also could compare in the max array and in the min array and it will only reduce either one array by one.

In total there are three types of comparison, Therefore, we require a total of n/2 – 1 + n/2 – 1 = n – 2 comparison, adding the first one, n – 2 + n/2 = 3/2n – 2 in the worst case.



9-1 *Largest i numbers in sorted order*

Given a set of n numbers, we wish to find the i largest in sorted order using a comparison-based algorithm. Find the algorithm that implements each of the following methods with the best asymptotic worst-case running time, and analyze the running times of the algorithms in terms of n and i.

a. Sort the numbers, and list the i largest.

b. Build a max-priority queue from the numbers, and call EXTRACT-MAX i times.

c. Use an order-statistic algorithm to find the ith largest number, partition around that number, and sort the i largest numbers.

A,

The algorithm would first sort the numbers, the best asymptotic worst case, would be n*log(n), and the listing the I largest would take O(i) with a for loop, therefore the total running time would be. O (nlog(n) + i)

B, since we are building the max-priority queue, we are calling healpify process, this will take nlog(n) in the worst case in the best way, the call of extract-max with I times, would be I * log(n)

Therefore the total time would be O(nlog(n) + I * log(n))

C, the partitioning around the I th largest number would take O(n), sorting the subarray of lengthi would be ilog(i) so the total time would be O(n + ilog(i))