EL9343 Homework 5

Due: Oct. 13th 11:00 a.m.

1. 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 $O(nk + n\lg(\frac{n}{k}))$ expected time. How should we pick k, both in theory and in practice?
   Solution:
   As we are doing quicksort until the problem size becomes less than or equal to k, we expect there to be $\log(\frac{n}{k})$ levels in the recursion tree, making the $n\lg(\frac{n}{k})$ term. After the quicksort returns, each element is within k of its final position. This means that insertion sort will take the shifting of at most k elements for every element that needed to change position, leading to the nk term. Putting these together, we have $O(nk + n\lg(\frac{n}{k}))$ expected time.
   In theory, we should pick k to minimize the expression. Let the real running time be $c_1nk + c_2n\lg(\frac{n}{k})$, take a derivative on k and let that equal zero, then $c_1n - c_2n\frac{1}{k} = 0$ and $k = \frac{c_2}{c_1}$. Therefore, the value of k depends on the relative size of the scalar in the nk term and the $n\lg(\frac{n}{k})$ term.
   In practice, we try it with large numbers of input size n for various values of k, because there are gritty properties of the machine not considered here such as cache line size.

2. **Quicksort with equal element values**. The analysis of the expected running time of the randomized quicksort 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?
   Solution:
   Since all elements are the same, the initial random choice of index and swap change nothing. Thus, randomized quicksort's running time is the same as that of quicksort. Since all elements are equal, PARTITION(A, p, r) always returns $r-1$, making it worst-case partitioning, so the runtime is $\Theta(n^2)$

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

   Like PARTITION, your PARTITION' procedure should take $\Theta(r-p)$ time.
   Solution:
   The PARTITION' is shown in algorithm 1.

   (c) Modify the RANDOMIZED-PARTITION procedure to call PARTITION', and name the new procedure RANDOMIZED-PARTITION'. Then modify the QUICKSORT procedure to produce a procedure QUICKSORT'(A, p, r) that calls RANDOMIZED-PARTITION' and recurses only on partitions of elements not known to be equal to each other.
   Solution:
   The RANDOMIZED-PARTITION' is shown in algorithm 2. The QUICKSORT' is shown in algorithm 3.

   (d) Using QUICKSORT', how would you adjust the analysis of randomized quicksort to avoid the assumption that all elements are distinct?
   Solution:
   Let d be the number of distinct elements in A. The running time is dominated by the time spent in the partition procedure, and there can be at most d calls to PARTITION'. If X is the number of comparisons performed in partition (line 4 as in lecture slides) over the entire execution of QUICKSORT', then the running time is $O(d + X)$.
   It remains true that each pair of elements is compared at most once. If z_i is the i^{th} smallest element, we need to compute the probability that z_i is compared to z_j. This time, once a pivot x is chosen

Algorithm 1 PARTITION'(A, p, r)

```
1: x = A[p]
2: i = p − 1
3: k = p
4: for j = p + 1, . . . , r do
5:    if A[j] < x then
6:       i = i + 1
7:       k = k + 1
8:       Exchange A[i], A[j]
9:       Exchange A[k], A[j]
10:   end if
11:   if A[j] = x then
12:      k = k + 1
13:      Exchange A[k], A[j]
14:   end if
15: end for
16: return i + 1, k
```

Algorithm 2 RANDOMIZED-PARTITION'(A, p, r)

```
1: i = RANDOM(p, r)
2: Exchange A[p], A[i]
3: return PARTITION'(A, p, r)
```

Algorithm 3 QUICKSORT'(A, p, r)

```
1: if p < r then
2:    (q, t) = RANDOMIZED-PARTITION'(A, p, r)
3:    QUICKSORT'(A, p, q − 1)
4:    QUICKSORT'(A, t + 1, r)
5: end if
```

with $z_i \leq x \leq z_j$, we know that z_i and z_j cannot be compared at any subsequent time. This is where the analysis differs, because there could be many elements of the array equal to z_i or z_j, so the probability that z_i and z_j are compared decreases. However, the expected percentage of the distinct elements in a random array tends to be $1 - \frac{1}{e}$ (check the link[1] for this value), so asymptotically the expected number of comparisons is the same.

3. Similar to the example shown in slides, illustrate the operation of COUNTING-SORT on

$$A = [5, 9, 7, 2, 3, 6, 9, 5, 2, 1]$$

Solution:

		1	2	3	4	5	6	7	8	9	10
	A	5	9	7	2	3	6	9	5	2	1
		0	1	2	3	4	5	6	7	8	9
	C	0	1	2	1	0	2	1	1	0	2
(a)		0	1	2	3	4	5	6	7	8	9
	C	0	1	3	4	4	6	7	8	8	10
(b)		1	2	3	4	5	6	7	8	9	10
	B	1									
		0	1	2	3	4	5	6	7	8	9
	C	0	0	3	4	4	6	7	8	8	10
(c)		1	2	3	4	5	6	7	8	9	10
	B	1		2							
		0	1	2	3	4	5	6	7	8	9
	C	0	0	2	4	4	6	7	8	8	10
(d)		1	2	3	4	5	6	7	8	9	10
	B	1		2			5				
		0	1	2	3	4	5	6	7	8	9
	C	0	0	2	4	4	5	7	8	8	10
					...						
(Final)		1	2	3	4	5	6	7	8	9	10
	B	1	2	2	3	5	5	6	7	9	9
		0	1	2	3	4	5	6	7	8	9
	C	0	0	1	3	4	4	6	7	8	8

4. Prove the lower bound of $\lceil \frac{3n}{2} \rceil - 2$ 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.)

Solution:

If n is odd, the number of comparisons are calculated as,

$$1 + \frac{3(n-3)}{2} + 2 = \frac{3n}{2} - \frac{3}{2} = \lceil \frac{3n}{2} \rceil - 2$$

[1] https://math.stackexchange.com/questions/41519/expected-number-of-unique-items-when-drawing-with-replacement

Note: In the worst case, the last left-out element has to be compared twice, with both the MAX and MIN.
If n is even, the number of comparisons are calculated as,

$$1 + \frac{3(n-2)}{2} = \frac{3n}{2} - 2 = \lceil \frac{3n}{2} \rceil - 2$$

5. **Largest i numebrs 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.
   Solution:
   The running time of sorting the numbers is $O(n \lg n)$, and the running time of listing the i largest is $O(i)$, Therefore, the total running time is $O(n \lg n + i)$.
   Note: We expect both i and n occur in the results. If you get $O(n \lg n + i)$, then simplify it to $O(n \lg n)$ with $i \leq n$, you will get full points. If you directly write $O(n \lg n)$, there will be a small deduction in points.

   (b) Build a max-priority queue from the numbers, and call EXTRACT-MAX i times.
   Solution:
   The running time of building a max-priority queue (using a heap) is $O(n)$, and the running time of each call EXTRACT-MAX is $O(\lg n)$, Therefore, the total running time is $O(n + i \lg n)$.

   (c) Use an order-statistic algorithm to find the ith largest number, partition around that number, and sort the i largest numbers.
   Solution:
   The running time of finding and partitioning around the i^{th} largest number is $O(n)$, and the running time of sorting the i largest numbers is $O(i \lg i)$, Therefore, the total running time is $O(n + i \lg i)$.