EL-GY 9343 Homework 5

Yihao Wang
yw7486@nyu.edu

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 $O(nk + n\log(\frac{n}{k}))$ expected time. How should we pick k, both in theory and in practice?

   1. Assume that in the m-th level, the whole array is partioned into subarrays with k elements in each, i.e., $\frac{n}{2^m} = k$. In each level, the time complexity is $O(n)$, thus the total time complexity of recursive partition is $m \cdot O(n) = O(n\log(\frac{n}{k}))$.
   2. Moreover, because the m levels of quick sort has split the original array into $\frac{n}{k}$ subarray, and $\forall i = 0, 1, \ldots, \frac{n}{k} - 1$, all elements within the i-th subarray are smaller than those in the $(i+1)$-th subarray. Therefore, when running insertion sort on the returned entire array, it can be considered within each of the $\frac{n}{k}$ subarray, each contributing to the time complexity by $O(k^2)$. In total, the final insertion sort gives a time complexity of $\frac{n}{k}\dot{O}(k^2) = O(nk)$.
   3. To sum up, the total time complexity is $O(nk) + O(n\log(\frac{n}{k})) = \underline{O(nk + n\log(\frac{n}{k}))}$.

In theory, we expect to see that $O(nk + n\log(\frac{n}{k})) = O(n\log n + nk - n\log k)$ grows slower than $O(n\log n)$. If $k = n$, the worst-case time complexity degenerates to $O(n^2)$. So we can choose relatively small k to avoid the degradation.
In practice, we can choose some small k, such as 5 or 10, depending on the concrete situation.

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?
   It's $\underline{O(n^2)}$ because every partitions will be extremely unbalanced.

   (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 $p \leq q \leq t \leq r$, 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.

Algorithm 1 PARTITION'(A, p, r)

```
Input: $A$ as input array to be partitioned within the index range of $[p, r]$
Output: Two indices $q$ and $t$, where $p \le q \le t \le r$
 1: $x \leftarrow A[p]$
 2: $i \leftarrow p + 1$
 3: for $j = p+1, p+2, \ldots, r$ do      ▷ move all elements equals to $x$ to the front of the array
 4:     if A[p] == x then
 5:         EXCHANGE $A[i] \leftrightarrow A[j]$
 6:         $i \leftarrow i + 1$
 7:     end if
 8: end for
 9: $c \leftarrow i - p$                  ▷ $c$ is the number of elements equals to $x$
10: $i \leftarrow i - 1$
11: $j \leftarrow r + 1$
12: while True do                    ▷ do a normal PARTITION, but with strictly less and greater
13:     repeat
14:         $j \leftarrow j - 1$
15:     until $A[j] < x$
16:     repeat
17:         $i \leftarrow i + 1$
18:     until $A[i] > x$
19:     if $i < j$ then
20:         EXCHANGE $A[i] \leftrightarrow A[j]$
21:     else
22:         $t \leftarrow i - 1$
23:     end if
24: end while
25: $i \leftarrow p$
26: for $i = 0, 1, \ldots, c$ do      ▷ exchange the elements equal to $x$ from the front to the middle
27:     EXCHANGE $A[p+i] \leftrightarrow A[j]$
28:     $j \leftarrow j - 1$
29: end for
30: $q \leftarrow j + 1$
31: return $q$, $t$
```

(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.

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

```
Input: $A$ as input array to be partitioned within the index range of $[p, r]$
Output: Two indices $q$ and $t$, where $p \leq q \leq t \leq r$
1: $i \leftarrow RANDOM(p, r)$
2: EXCHANGE $A[i] \leftrightarrow A[p]$
3: return PARTITION’$(A, p, r)$
```

Algorithm 3 QUICKSORT’(A, p, r)

```
Input: $A$ as input array to be partitioned within the index range of $[p, r]$
Output: The sorted version of $A$
1: if $p < r$ then
2:     $q, t \leftarrow$ RANDOMIZED-PARTITION’$(A, p, r)$
3:     RANDOMIZED-PARTITION’$(A, p, q)$
4:     RANDOMIZED-PARTITION’$(A, t, r)$
5: end if
```

(d) Using QUICKSORT’, how would you adjust the analysis of randomized quicksort to avoid the assumption that all elements are distinct?

Now we exclude the elements that equal to the pivot from the recurrence, so we will not recursively compare all those equal numbers, thus the total number of comparison will not increase due to duplication. The total time complexity is still $O(n \log n)$ using QUICKSORT’.

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

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

Since the maximum of array A is 9, we first create a frequency array C' of length 9 and count the frequency of each number showing up in the array A:

$C' =$	1	2	1	0	2	1	1	0	2

Then we calculate the cumulative sums:

$C =$	1	3	4	4	6	7	8	8	10

Now we put every element of A into a new array B in a reversed order. At the same time, we decrease the corresponding position of cumulative C.

Step 1:

$A =$	5	9	7	2	3	6	9	5	2	1
$B =$	1									
$C =$	0	3	4	4	6	7	8	8	10	

Step 2:

$A =$	5	9	7	2	3	6	9	5	2	1
$B =$	1		2							
$C =$	0	2	4	4	6	7	8	8	10	

Step 3:

$A =$	5	9	7	2	3	6	9	5	2	1
$B =$	1		2			5				
$C =$	0	2	4	4	5	7	8	8	10	

Step 4:

$A =$	5	9	7	2	3	6	9	5	2	1
$B =$	1		2			5				9
$C =$	0	2	4	4	5	7	8	8	9	

Step 5:

$A =$	5	9	7	2	3	6	9	5	2	1
$B =$	1		2			5	6			9
$C =$	0	2	4	4	5	6	8	8	9	

Step 6:

$A =$	5	9	7	2	3	6	9	5	2	1
$B =$	1		2	3		5	6			9
$C =$	0	2	3	4	5	6	8	8	9	

Step 7:

$$A = \begin{array}{|c|c|c|c|c|c|c|c|c|c|}\hline 5 & 9 & 7 & 2 & 3 & 6 & 9 & 5 & 2 & 1 \\ \hline\end{array}$$

$$B = \begin{array}{|c|c|c|c|c|c|c|c|c|c|}\hline 1 & 2 & 2 & 3 & & 5 & 6 & & & 9 \\ \hline\end{array}$$

$$C = \begin{array}{|c|c|c|c|c|c|c|c|c|}\hline 0 & 1 & 3 & 4 & 5 & 6 & 8 & 8 & 9 \\ \hline\end{array}$$

Step 8:

$$A = \begin{array}{|c|c|c|c|c|c|c|c|c|c|}\hline 5 & 9 & 7 & 2 & 3 & 6 & 9 & 5 & 2 & 1 \\ \hline\end{array}$$

$$B = \begin{array}{|c|c|c|c|c|c|c|c|c|c|}\hline 1 & 2 & 2 & 3 & & 5 & 6 & 7 & & 9 \\ \hline\end{array}$$

$$C = \begin{array}{|c|c|c|c|c|c|c|c|c|}\hline 0 & 1 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \hline\end{array}$$

Step 8:

$$A = \begin{array}{|c|c|c|c|c|c|c|c|c|c|}\hline 5 & 9 & 7 & 2 & 3 & 6 & 9 & 5 & 2 & 1 \\ \hline\end{array}$$

$$B = \begin{array}{|c|c|c|c|c|c|c|c|c|c|}\hline 1 & 2 & 2 & 3 & & 5 & 6 & 7 & 9 & 9 \\ \hline\end{array}$$

$$C = \begin{array}{|c|c|c|c|c|c|c|c|c|}\hline 0 & 1 & 3 & 4 & 5 & 6 & 7 & 8 & 8 \\ \hline\end{array}$$

Step 8:

$$A = \begin{array}{|c|c|c|c|c|c|c|c|c|c|}\hline 5 & 9 & 7 & 2 & 3 & 6 & 9 & 5 & 2 & 1 \\ \hline\end{array}$$

$$B = \begin{array}{|c|c|c|c|c|c|c|c|c|c|}\hline 1 & 2 & 2 & 3 & 5 & 5 & 6 & 7 & 9 & 9 \\ \hline\end{array}$$

$$C = \begin{array}{|c|c|c|c|c|c|c|c|c|}\hline 0 & 1 & 3 & 4 & 4 & 6 & 7 & 8 & 8 \\ \hline\end{array}$$

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.)

Algorithm 4 **FIND-MIN-MAX**(A)

```
Input: $A$ as input array to find the maximum and minimum elements, indexed from 1 to $n$
Output: Maximum element $m_{\max}$ and minimum element $m_{\min}$
 1: if $A[1] < A[2]$ then          ▷ Compare the first two elements and initialize $m_{\max}$ and $m_{\min}$
 2:     $m_{\min} \leftarrow A[1]$, $m_{\max} \leftarrow A[2]$          ▷ with the larger and smaller ones respectively
 3: else
 4:     $m_{\min} \leftarrow A[2]$, $m_{\max} \leftarrow A[1]$
 5: end if
 6: for $i = 1, \ldots, \lceil \frac{n-2}{2} \rceil$ do          ▷ traverse the remaining array $A$ in pairs
 7:     if $A[2i+1] > A[2i+2]$ then          ▷ compare within the pair
 8:         EXCHANGE $A[2i+1] \leftrightarrow A[2i+2]$
 9:     end if
10:     if $A[2i+1] < m_{\min}$ then          ▷ compare the smaller one with the minimum so far
11:         $m_{\min} \leftarrow A[2i]$
12:     end if
13:     if $A[2i+2] > m_{\max}$ then          ▷ compare the larger one with the maximum so far
14:         $m_{\max} \leftarrow A[2i+1]$
15:     end if
16: end for
```

From the Algo.4 above, we can clearly see that in total we need at least $1 + \lceil \frac{n-2}{2} \rceil \times 3 = \lceil \frac{3n}{2} \rceil - 2$ times of comparison to find the maximum and minimum element in an array of size n.

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.

It takes $O(n \log n)$ to sort n numbers with a comparison-based algorithm, thus the total time complexity is $\underline{O(n \log n)}$.

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

It takes $O(n)$ to build a max heap from n numbers. And it takes $O(\log n)$ when calling EXTRACT-MAX on a heap of size n, which in total takes

$$\begin{aligned} & O(\log n) + O(\log(n-1)) + \cdots + O(\log(n-i+1)) \\ =& \sum_{j=1}^{n} O(\log j) - \sum_{k=1}^{n-i} O(\log k) \\ =& O(\log n!) - O(\log(n-i)!) \\ \simeq& \underline{O(n \log n) - O(\log(n-i)!)} \end{aligned}$$

Note we use the Stirling's approximation at the last equal sign.

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

It takes O(n) to find the largest ith number, and it takes another $O(n)$ to partition around that number. Finally, it takes $O(i \log i)$ to sort the larger part. In total, the time complexity is $\underline{O(n + i \log i)}$