Assignment 3 Answers

COT 6405 - Introduction to Theory of Algorithms

1) **(10 points) For QUICKSORT, the PARTITION function is called n times to sort an array of n elements. Prove that when the array is already sorted in ascending or descending order, every call to the PARTITION (A, p ,r) function generates an empty array and an array of size p − r .**

Answer: Assume an ascending array is [a_1 , a_2 , . . . , a_n ,], where a_1 , $< a_2$, . . . , $< a_n$. When the partition function is called for the first time (PARTITION(A, 1 ,n)), the pivot $r = n$. According to the partition function, the For loop compares a_j with a_r for $1 \leq$ j $\leq$ n-1. Since a_r is the largest element of this array, when the For loop finishes, no element is moved and $i = n$-1 and $j = n$-1. Then, the partition function returns $q = i$+1 = n as the pivot element. This means that PARTITION (A, 1, n) generates an empty array and an array of size n – 1. This array is exactly [a_1 , a_2 , . . . , a_{n-1}], for which the pivot $r = n$-1. Again, a_r is the largest element of this array. When the For loop finishes, $i = n$-2 and $j = n$-2, and the partition function returns $q = i$+1 = n-1 as the pivot element. QUICKSORT continues calling QUICKSORT (A, 1, n-2) and QUICKSORT (A, n, n-1). The first recursive call to QUICKSORT will process the array of [a_1 , a_2 , . . . , a_{n-2}] , and the second recursive call to QUICKSORT will process an empty array. The same analysis applies for all calls to partition, and every call to the PARTITION function generates an empty array and an array of size p - r. By using an similar analysis, you can prove this for an array in descending order.

2) **(10 points) Is it correct to say that the worst-case running time of RANDOMIZED-QUICKSORT on textbook 7.3 (page 179) is O(nlgn)? If YES, please give your reason. If NO, please explain under which situation the worst-case running time can be triggered.**

Answer: Not correct. The worst-case running time of RANDOMIZED-QUICKSORT is O(n^2). This happens under the situation that the random function of RANDOMIZD-PARTITION always returns the current largest/smallest element as the pivot element. In this case, PARTITION (A, p ,r) always generates a bad split, i.e., an empty array and an array of size $p - r$.

3) **(10 points) See below for Triple-QUICKSORT(A, p, r), which partitions an array into three subarrays A1, A2, and A3, such that all elements in A1 are less than those in A2, and all elements in A2 are less than those in A3.**

```
Triple-QUICKSORT(A, p, r)
{
  if (p < r)
  {
    [q1, q2] = Triple-PARTITION(A, p, r);
     Triple-QUICKSORT (A, p, q1-1);
     Triple-QUICKSORT (A, q1+1, q2-1);
     Triple-QUICKSORT (A, q2+1, r);
  }
}
```

Answer the following questions:

a) Triple-PARTITION (A, p, r) returns two pivots q1 and q2. When this function returns, all elements in A[p... q1-1] are less than or equal to the pivot A[q1], all elements in A[q1+1... q2-1] are larger than A[q1] and less than or equal to the pivot A[q2], and all elements in A[q2+1... r] are larger than the pivot A[q2]. Complete the code of Triple-PARTITION (A, p, r).

```
Triple-PARTITION(A, p, r)
{              x = A[r]
i = p – 1
for j = p to r – 1
if A[j] ≤ x
i = i + 1
exchange A[i] ↔ A[j]
exchange A[i + 1] ↔ A[r]
q1 = _________
             x = A[r]
i = q1
for j = _________ to _________
if A[j] ≤ _________
i = i + 1
exchange _________
exchange _________
q2 = _________
      return [q1, q2]
        }
```

Answer:

```
Triple-PARTITION(A, p, r)
{       x = A[r]
i = p – 1
```

for $j = p$ to $r - 1$
if A[j] $\leq x$
$i = i + 1$
exchange A[i] $\leftrightarrow$ A[j]
exchange A[$i + 1$] $\leftrightarrow$ A[r]
q1 $= i + 1$
$x =$ A[r]
$i = q1$
for $j = q1 + 1$ to $r - 1$
if A[j] $\leq x$
$i = i + 1$
exchange A[i] $\leftrightarrow$ A[j]
exchange A[$i + 1$] $\leftrightarrow$ A[r]
q2 $= i + 1$
return [q1, q2]
}

b) Is it possible for Line 1 or line 9 of Triple-PARTITION choose the same elements as the pivots (In other words, q1 = q2)? Please explain your reason.

Answer: Not possible. In line 10, i has an initial value of q1. When the second for loop finishes, i should be larger than or equal to q1, i.e., i $\geq q1$. In line 16, q2 $= i + 1 \geq q1 + 1$. Thus, q1 $\neq$ q2.

In a special case, when the array is already sorted in ascending order, q1 = n, and the second For loop is not executed, because i = q1 = n and j starts from n+1. According to the code, q2 = n+1 and thus q1 is not equal to q2.

c) Analyze the worst-case time complexity of Triple-QUICKSORT (A, p, r).

Answer: when the array is already sorted, Triple-PARTITION generates a split of 0 : 0 : n-1 and T(n) = T(n-1) + T(0) + T(0) + $\Theta(n) = \Theta(n^2)$. Therefore, the worst-case time complexity happens.

4) **Use induction to prove that LSD Radix Sort works. Where does your proof need the assumption that the intermediate sort is stable?**

To prove by induction, we start with a base case. The base case should be an array of size one that is a single digit. Radix sort will sort on this single digit and produce a sorted array. So we can get a sorted result. We can say LSD Radix sort works on the base case.

Now the base case is proven. We then continue our inductive prove hypothesis. We assume that lower-order digits {j:j<i} are sorted. We want to prove that Radix sort works with i digits. By hypothesis, we know that all digits from 1 to j are correctly ordered. Now Radix sort will order the numbers based on i digit.

If two digits at position i are different, ordering numbers by that digit is correct (lower-order digits are irrelevant)

If they are the same, numbers are already sorted on the lower-order digits. Since we use a stable sort, the numbers stay in the right order

Thus, if the intermediate sort was stable, the sorting will remain correct.

5) **Given n = 80,000,000 numbers and each number of 64 bits. We first divide each number into d digits and then use LSD Radix Sort to sort these numbers. What's the running time if each digit is of 32 bits, 16 bits, 8 bits, $\lceil \lg n \rceil$, and $\lfloor \lg n \rfloor$ bits? Please explain your answer.**

Answer:

a) 32bits: Each number has 64bits/32bits = 2 digits. Assume the stable sort is counting sort.

For each counting sort, the sorting time is O(n+k) n is the number, k is range of 32 bits, 2^{32}

In total we have 2 counting sorts for 2 digits.

Total running time = 2 * O(1) * (80,000,000 + 2^{32})

b) 16 bits: Each number has 64bits/16bits = 4 digits.

As above the total running time = 4 * O(1) * (n+k) = 4 * O(1) * (80,000,000 + 2^{16})

c) 8 bits: Each number has 64bits/16bits = 8 digits.

As above the total running time = 8 * O(1) * (n+k) = 8 * O(1) * (80,000,000 + 2^{8})

d) $\lceil \lg n \rceil$: $\lceil \lg n \rceil = \lceil \lg 80{,}000{,}000 \rceil = 23$. Each number has 64/23 = 2.78, **3** digits.

Total running time= 3 * O(1) * (n+k) = 3 * O(1) * (80,000,000 + 2^{23})

e) $\lfloor \lg n \rfloor$: $\lfloor \lg n \rfloor = 22$. In this case each number has 64/22 = 3 digits.

Total running time = 3 * O(1) * (n+k) = 3 * O(1) * (80,000,000 + 2^{22})

6) **Each element of an array A of n elements falls in the range of [0... $k * n^{100} - 1$], where k is a constant that is less than n. Can we sort these numbers in O(n) time? Why?**

Answer: Yes we can sort these numbers in O(n) time by using Radix sort. Each word is of b bits and has a maximum value of $k * n^{100} - 1$ and therefore $b = \lg(\ k * n^{100}\ )$. Substituting b into the running time equation on page 37 of lecture 10 (how to choose r), the result is O(n).

7) **Describe an algorithm that, given n integers in the range 0 to k, preprocess its input and then answers any query about how many of the n integers fall into a range [a... b] in O(1) time. Your algorithm should use $\Theta(n + k)$ preprocessing time.**

Answer: Create two arrays C1 and C2. C1[i] counts the number of elements that are less than i and C2[i] counts the number of elements that are larger than i. So the number of integers fall into [$a \ldots b$] is n- C1[a]- C2[b].

Algorithm 1 Range(a,b, A)

[a, b] is the range that we want to select from A.
A is the array with n integers in the range 0 to k

1: **If** NOT preprocessed **then**
2: Create arrays C1 and C2 of length k+1 (index from 0 to k), initialize every element in C1, C2 as 0
3: **For** every number x in A:
4: Add 1 to $C1[x+1], C1[x+2]$ until $C1[k]$
5: Add 1 to $C2[0], C2[1]$ until $C2[x-1]$
6: **End For**
7: preprocessed = True
8: **End If**
9: **Return** $n - C1[a] - C2[b]$

8) **Suppose we use RANDOMIZED-SELECT to select the minimum element of the array A = { 3,2,9,0,7,5,4,8,6,1} . Describe a sequence of partitions that results in a worst-case performance of RANDOMIZED-SELECT.**

Answer: To get a worst case performance, every time the Randomized select will select the largest element of the array as pivot to cause the most unbalanced partitions.

So first partition, choose 9 as pivot, {3, 2, 0, 7, 5, 4, 8, 6, 1}
2nd: choose 8 {3, 2, 0, 7, 5, 4, 6, 1}
3rd: choose 7, {3, 2, 0, 5, 4, 6, 1}
4th: choose 6, {3, 2, 0, 5, 4, 1}
5th: choose 5,{3, 2, 0, 4, 1}
6th: choose 4,{3, 2, 0, 1}
7th: choose 3,{2, 0, 1}
8th: choose 2,{0, 1}
9th: choose 1,{0}
10th: choose 0,

9) **Analyze SELECT to show that if $n \geq$ 140, then at least $\lceil \frac{n}{4} \rceil$ elements are greater than the median-of-medians x, and at least $\lceil \frac{n}{4} \rceil$ elements are less than x**

Answer: from textbook page 221, we know that at least $3n/10 - 6$ elements are less than x. When $n \geq$ 140, $\lceil \frac{n}{4} \rceil \leq 3n/10 - 6$ and therefore we can say that at least $\lceil \frac{n}{4} \rceil$ elements are less than x. Due to the same reason, at least $\lceil \frac{n}{4} \rceil$ elements are greater than x

10) **Describe an O(n) algorithm (other than linear sorting algorithm) that, given a set S of n distinct numbers and a positive integer $k \leq n$, determines the k numbers in S that are closest to the median of S.**

In a set S of n distinct numbers with $i = n/2$th order statistic being the median, we want to find the k elements that are nearest the median. Thus, we wish to find all i-th order statistics such that

$$n/2 - \lfloor k/2 \rfloor < i < n/2 + \lfloor k/2 \rfloor$$

Answer: The obvious solution is to run Select k times for a runtime of **O(nk)**. However, this does not give our desired runtime of O(n), because k can range between 1 and n.

To begin, we need to know the absolute values of the differences between each element and the median of S. First, we run Select (n/2) **(O(n))** to find the median. Then, we create a new array of length n called T, and iterate through S, filling the matching indices of T with the absolute values of the differences between the element at that index in S and the median of S (**O(n)**).

In T, the 0th order statistic will reside at the index of the median, the 1st and 2nd order statistics are guaranteed to reside at the indexes of the closest elements to the median of S. The k-th order statistic will reside at the index of the farthest element which is still within the k closest elements. So, we run Select(k) on T to get the index of our furthest element from the median of S (**O(n)**).

When we know this element and the absolute value of its difference with the median (let's call this absolute value a), we know that all elements with a smaller difference must fall within the range of the k elements surrounding the median. Thus, we simply iterate through T; when we see an absolute value less than or equal to a, we note its index, take the element at that index in S, and add it to an array to be returned. When we finish iterating through T (**O(n)**), our array to be returned will be of size k and contain the numbers from S closest to its median.

The total runtime of this algorithm is **O(n) + O(n) + O(n) + O(n) = O(4n) = O(n)**.