Fall 2021: CSOR 4231 - Analysis of Algorithms

Aayush Kumar Verma, UNI: av2955

Due Date: October 15, 2021

Problem 1:

(i). Let X be a random variable that denotes the number of empty bins.
Let X_i be an indicator random variable denoting an empty bin after all the balls are tossed.

$E[X] = \sum_{i=1}^{n} E[X_i]$
$E[X] = (\frac{n-1}{n})^n$

$E[X] = n.(1 - \frac{1}{n})^n$ $\leftarrow$answer

(ii). Now, let X_i be an indicator random variable denoting the i^{th} bin containing only 1 ball and let X denote the total number of bins that contain only 1 ball.

$E[X] = \sum_{i=1}^{n} E[X_i]$

$E[X] = \frac{1}{n}.\binom{n}{1}.\sum_{i=1}^{n}(\frac{n-1}{n})^{n-1}$

$= n.(\frac{n-1}{n})^{n-1} = n(1 - \frac{1}{n})^{n-1}$ $\leftarrow$answer

Problem 2:

(i). Let A be an array in which we have to search for the majority element. We divide A in to 2 parts $A_1 = A[1 \dots \lfloor n/2 \rfloor]$ and $A_2 = A[\lceil n/2 \rceil \dots n]$. Let m denote the majority element in A.

If m is $\leqslant$ half in both the halves, then it is $\leqslant$ half in the whole array, hence, a reverse positive statement for the required proof. If m is $\geqslant$ half in both the parts, then it is by default $\geqslant$ half in the whole array.

Consider the trivial case when $n = 1$ and there is just 1 element in the array so this element is the majority element. When we merge 2 arrays of size 1 and if both the elements of the 2 arrays are equal, then it is the majority element of the merged array, else there is no majority element. We repeat this process for larger values of n.

In case when m_1 is the majority element in A_1 and m_2 is the majority element in A_2 and $m_1 \neq m_2$, we count the occurrences of m_1 in A_2 and if it $\geqslant n/2$, then it is the majority element. We check the same thing for m_2 in A_1. Hence by this, we can conversely state that if m is the majority element in an array A, it is also the majority element in either or both the 2 parts of A, i.e., A_1 and A_2.

Based on this approach, the algorithm using **Divide and Conquer** can be defined as follows:

We define a function *isMajorityElement (arr[], left, right)* which takes 3 parameters as inputs - array, left index, and right index and returns the number which is the candidate majority element in the array. We also define a function *count (arr[], m)* which takes as input - array, number (m), and counts the number of occurrences of m in *arr*[] and returns the count. The main function called *checkMajority(arr[], n)* receives the candidate majority element from *isMajorityElement* and verifies if it actually is the majority element by calculating it's count and if the count is $\geqslant$ half the size of the array, then it is the majority element.

Base Case: $n = 1$, a single element in the array, i.e., $left == right$, so the majority element is $arr[left]$.

Divide Step: Split the array in two halves from the mid point by using the formula $(left + right)/2$ or to handle large values of $left$ and $right$, $[left + (right - left)/2]$

Conquer Step: Recursively calculate the majority element in each half of the array. Store the results in 2 variables m_1 and m_2 respectively.

Combine Step: If $(m_1 == m_2)$, i.e., both majority elements are equal, then it is the majority element of the merged array, so return either of them. If $(m_1 \neq m_2)$, then calculate *count (A, m_1)* and *count (A, m_2)* and return the one with greater count.

Time Complexity Analysis and Proof of Correctness:

1. Divide Step = $O(1)$
2. Conquer Step = $T(n/2) + T(n/2) = 2T(n/2)$
3. Combine Step = Time required to obtain the number of occurrences for 2 candidate majority elements = $O(n) + O(n) = O(n)$

Therefore, overall time = $2T(n/2) + O(n) + O(1) = 2T(n/2) + O(n)$

Using Master's Theorem, we get the overall complexity as $\boldsymbol{O(nlog(n))}$.

This algorithm works correctly because we have shown above that if there is a candidate element for majority element in the entire array, then it is also the majority element in the either or both of the 2 halves of the array.

Algorithm 1 Majority Element - Divide and Conquer:

```
isMajorityElement(A, left, right):

  if left == right then
      return A[left]
  end if

  mid = left + (right - left) / 2
  m1 = isMajorityElement(A, left, mid)
  m2 = isMajorityElement(A, mid + 1, right)

  if m1 == m2 then
      return m1
  end if

  m1Count = count(A, left, right, m1)
  m2Count = count(A, left, right, m2)

  if m1Count > m2Count then
      return m1
  else
      return m2
  end if

count(A, left, right, m):
  cnt = 0

  for i = left to right do:
      if A[i] == m then
          cnt = cnt + 1
      end if
      i = i + 1
  end for

  return cnt

checkMajority(A, n):
  m = isMajorityElement(A, 1, n)
  cnt = count(A, 1, n, m)
  if cnt $\geqslant \frac{n+1}{2}$ then
      return m
  end if
```

(ii). Let x be the majority element in array A of length n. Hence, we know that frequency $(x) \geqslant n/2$.

(a). We know that x is the majority element and n is even. Now, after splitting the array in 2 halves, we will get 2 types of pairs. One in which both elements are same and the other in which both elements are different. In the case when both elements are same, the values may or may not be equal to x. Let p be the count of such pairs and q be the count of the pairs that are different. For x to be the majority, at least half of p must be that of good pairs. Let the count of pairs having the same value and value equal to x be t.

Now, assuming that $t < \frac{p}{2}$ and among the pairs which had different values, there can at max $q/2$ values of x, i.e. $t' \leqslant \frac{q}{2}$. Therefore, by this, in total we get:

$t + t' < \frac{p}{2} + \frac{q}{2}$

Now, $\frac{p}{2} + \frac{q}{2} = \frac{n}{2}$ and $t + t' =$ frequency(x). So, we get the inequality as $count(x) < \frac{n}{2}$ which is a contradiction. Hence, proved that when n is even then in the pairs that remain, x would still be the majority element.

(b). Given that n is odd. The rule we will follow is that initially, we will keep the last element aside initially and process the remaining elements (even in number) as shown in the above question. The, if the number of elements left are even, then we add the extra element as there may be a case when 2 elements can be a possible majority element and we would need a "tiebreaker" element in this case. If the number of elements remaining is odd, then there would be a clear majority so we will not add the extra element in this case.

For example, if we have an array as $[a, b, a, b, a]$, with a as the majority element, then the possible pairs can be:

$[a, a] \longrightarrow [a]$
$[b, b] \longrightarrow [b]$
$[a] \longrightarrow [a]$

So, in this case, both a and b can be the majority element so we will need the tiebreaker which ensures that the majority element in the remaining elements is the same majority element of the original array.

Another case can be:

$[a, b]$
$[a, a] \longrightarrow [a]$
$[b]$

Here, there is no tie. Hence we will not consider the tiebreaker in this case. This ensures that by using this rule, we take care of odd lengths also.

(c). We will use the concepts from (a) and (b) above to come up with a $O(n)$ solution to find the majority element in an array A of integers. We will first pair up the elements and only keep those elements in another temporary array which are equal and discard those pairs which have different numbers. This will give us a "candidate" majority element which we will verify with another helper function that returns the number of occurrences of this "candidate" majority element. If it is $\geqslant n/2$, then this is the majority element, otherwise, no majority element exists in the array (We keep this condition because we are going to design the algorithm assuming that we do not have any prior knowledge of whether the array has a majority element or not).

Time Complexity Analysis and Proof of Correctness:

1. Generating pairs = $T(n/2)$
2. Finding count = $O(n)$

Therefore, overall time $T(n) = T(n/2) + O(n)$

By Master's Theorem, we get $\boldsymbol{T(n) = O(n)}$

This algorithm works correctly as shown above that our modified Divide and Conquer gives the same majority element in the remaining pairs as well.

Algorithm 2 Majority Element in Linear Time:

```
getMajorityElement (A, n):
  m = findCandidate(A, n, None)
  f = findCount(A, n, m)
  if f > $\lfloor n/2 \rfloor + 1$ then
      return m
  else
      return -1
  end if

findCandidate(A, n, tiebreaker):
  if n == 0 then
      return tiebreaker
  end if
  if n == 2 then
      if A[1] == A[2] then
          return A[1]
      else
          return -1
      end if
  end if

  if n % 2 == 1 then
      tiebreaker = A[:-1]
  end if
  pairs = []
  for i = 1 to n-1, (skip = 2) do
      if A[i] == A[i+1] then
          append A[i] to pairs
      end if
      i = i + 1
  end for
  return findCandidate (pairs, length (pairs), tiebreaker)

findCount(A, n, m):
  cnt = 0
  for i = 1 to n do
      if A[i] == m then
          cnt = cnt + 1
      end if
      i = i + 1
  end for
  return cnt
```

Problem 3:

We are given an unsorted array A of size n and another integer $k < n$. We will use another array which stores the absolute distance of all the numbers of the array from the first element. The distances can have duplicate values, hence, we will then use the Deterministic Selection Algorithm which uses the modified randomized-partition routine (described in Problem 5.ii - Algorithm 5). In this way, we get the k^{th} nearest distance in $O(n)$. After getting the k^{th} nearest distance, we traverse the original array A and pick the first k elements which have the absolute distance $\leqslant k^{th}$ nearest distance. In the Deterministic Selection Algorithm, we will not use a random pivot, but select the pivot as the median of medians of groups of 5 of the numbers in the array.

Algorithm 3 k^{th} Closest Elements:

```
SELECTION(A, p, r, i):
  if p == r then
      return A[p]
  end if
  q1, q2 = ModifiedPartition(A, p, r)
  x1 = q1 - p + 1
  x2 = q2 - p + 1
  if x1 ⩽ i ⩽ x2 then
      return A[q2]
  else
      if i < x1 then
          return SELECTION(A, p, q1 - 1, i)
      else
          return SELECTION(A, q2 + 1, r, i - x2)
      end if
  end if

findElements(A, n, k, x):
  res = []
  for (i = 2 to n) and (k > 0) do
      if abs (A[i] - A[0]) ⩽ x then
          res.append (A[i])
          k = k - 1
      end if
      i = i + 1
  end for
  return res
```

Time Complexity Analysis and Proof of Correctness:

From the class, we know that the Selection Algorithm runs in $O(n)$ time. The *findElements(A, n, k, x)* method also runs in $O(n)$ time. Hence, the overall time complexity for the algorithm is $O(n)$. This algorithm is correct since, as proved in the class, the algorithm is correct and the only modification to the discussed algorithm is the use of a modified partition routine which returns 2 indices instead of 1 index.

Problem 4:

Finding (k - 1) elements of rank $\lceil \frac{n}{k} \rceil, \lceil \frac{2n}{k} \rceil, \ldots, \lceil \frac{(k-1)n}{k} \rceil$ is equivalent to finding the k^{th} quantiles of array A and is given by the below mentioned algorithm which runs in $O(nlogk)$ time.

Algorithm 4 Ranking with Quantiles:

```
RANKING(A, k, Q, n):
  if k ≠ 1 then
      d = $\lfloor \frac{k}{2} \rfloor$
      p = SELECT(A, $\lfloor i.n/k \rfloor$)
      PARTITION(A, p)
      Q.append (RANKING(A[1... $\lfloor d.n/k \rfloor$], $\lfloor k/2 \rfloor$, Q, n))
      Q.append (RANKING(A[$\lfloor d.n/k+1 \rfloor \ldots n$], $\lceil k/2 \rceil$, Q, n))
      return p
  end if
  return None
```

Time Complexity Analysis and Proof of Correctness:

If we draw a recursion tree for this, we see that the root gets split into $\lceil \frac{k-1}{2} \rceil$ and $\lfloor \frac{k-1}{2} \rfloor$ order of k and cost to sum up this level is $O(n)$. At the root level, we have order-k, which also requires $O(n)$ cost. At any depth d, we have the order as 2^d and $0 \leqslant d \leqslant log_2(k-1)$ and the cost is $O(n)$ because the number of elements at any level is n. The depth is $log_2(k-1)$. Hence, combining the two, the overall complexity comes to $O(nlogk)$.

This algorithm works correctly as we are using the similar selection and partition algorithms taught in the class, so the selection algorithm ensures that we get the correct pivot for the k^{th} rank.

Problem 5:

(i). Even if all the elements are equal, then the partition (finding random index) and swapping would not change and the randomized Quicksort algorithm will have the same run-time as the normal Quicksort algorithm. Since all elements are equal, the partition function PARTITION(A, p, r) will return (r - 1) always thus giving a worst case complexity of $O(n^2)$.

(ii). Yes, if we are expecting many equal elements, then it is better to modify the PARTITION function such that it partitions in the array A like $A[1 \dots p], A[p+1 \dots q_1]$, and $A[q_1+1 \dots q_2]$. The new partition function will now return two values for q_1 and q_2.

Algorithm 5 Modified-PARTITION:

```
ModifiedPartition(A, p, r):
  x = A[r]
  i = p-1
  Swap (A[r], A[p])
  k = p
  for j = p+1 to r-1 do
      if A[j] < x then
          i = i+1
          k = i+2
          Swap (A[i], A[j])
          Swap (A[j], A[k])
      end if
      if A[j] == x then
          k = k+1
          Swap (A[j], A[k])
      end if
      j = j + 1
  end for
  Swap (A[i+1], A[r])
  return (i+1) and (k+1)
```

Time Complexity Analysis and Proof of Correctness:

The only loop that runs in the modified Partition algorithm is a *for loop* which makes $(r-1)-(p+1)$ iterations, which is $O(r-p)$. For the entire array, $r = n$ and $p = 0$, so the complexity is $O(n)$.
As, we are modifying the partition algorithm only by changing the return values from 1 to 2, thereby signifying that all values lesser than the pivot element lie to the left (between p and $q1$), all values equal to the pivot element lie in the middle (between $q1$ and $q2$), and all the values greater than the pivot element lie on its right (between $q2$ and r). Hence, the algorithm works correctly.

(iii). If we use the modified partition function define above with our randomized Quicksort algorithm, then the modified partition function will be called $log(n)$ times and each time it will be bounded by a $O(n)$ runtime. Hence, the overall time complexity of the randomized Quicksort algorithm will become $O(nlogn)$.

Problem 6:

(i). There are a total of $\binom{2n}{n}$ ways of dividing $2n$ numbers in to 2 sorted lists having n numbers each.

(ii). A decision tree to merge 2 sorted lists has $\binom{2n}{n}$ nodes, thus making it's height = number of operations $= log(\frac{(2n)!}{n!n!})$

$= log((2n)!) - 2log(n!)$

$= \sum_{i=1}^{2n} log(i) - 2\sum_{i=1}^{n} log(i)$

$= 2n + 2nlog(n) - 2nlog(n+1) - 2log(n+1)$

$= 2n - o(n)$ (by using Stirling's Formula)

(iii). We need to compare them. This is because as they are from different lists, the order of 2 consecutive elements in the sorted order cannot be determined when they are compared to other elements. If we do not compare them, then we cannot differentiate between the original lists and the lists obtained by swapping these 2 elements between the lists because we need to do different things to merge the elements correctly when the 2 elements are different, hence, we must compare them.

(iv). Consider 2 lists X and Y, where:

$$X = 1, 3, 5, \ldots, 2n-1$$
$$Y = 2, 4, 6, \ldots, 2n$$

By the property proven in part (iii), we will need to compare 1 with 2, 2 with 3, and so on until $2n-1$ with $2n$, i.e., the comparisons we would make take the following form (assuming 1-based indexing in the 2 lists):

$$X[1] \leqslant Y[1] \leqslant X[2] \leqslant Y[2] \leqslant X[3] \ldots \leqslant X[2n-1] \leqslant Y[2n]$$

Hence, we need to make at least the following comparisons:

$$\forall i \in [1, n], \text{ compare } (X[i], Y[i])$$

and,

$$\forall i \in [1, n-1], \text{ compare } (Y[i], X[i+1])$$

to merge the 2 lists, hence making a total of at least **$(2n-1)$ comparisons**.