Report on Assignment -1 (Divide *& Conquer*)

***S*ubmitted by**

Suresh Kumar Amalapuram

Roll No. CS19RESCH11001

AdvancedDataStructures & Algorithms

Problem 1- Solution Overview

- This section focuses on details of the Solution
- Sort the Input array using Randomized Quick Sort, guaranteed to sort in $O(n \log n)$(proved in the class), where n is number of elements
- We compare the element at the index **i** with the element at index **i+mid**,where mid =n/2 and end this comparison at the index n/2
- Returns true, if any Majority element exist
- Implementation is a **in-place** algorithm, with time complexity$O(n \log n)$ and space complexity $O(n)$

Problem 1-pseudo code of the Solution

0.1 Find the Majority of the given array of elements

Algorithm 1 Algorithm for finding Majority of the given array of elements

Input: **n** =Size for the array, Array **a[]** of n integers

Output: **true** or **false** based on presence of Majority element

```
1: $a[1...n] \longleftarrow$ apply RandomizedQuickSort(a[1...n])
2: $Mid \longleftarrow n/2$
3: for each $i$ in 1..to..n/2 do
4:     if a[$i$] equals a[$i$+mid] then
5:         return true
6:     end if
7: end for
8: return false
```

0.2 Randomized Quick Sort on given array of elements

Algorithm 2 Randomized QuickSort

```
Input: n =Size for the array, Array a[1..n] of n integers
Output: Array a[1..n] of sorted elements
 1: Randomized QuickSort(a[1..n], start,end )
 2: Randomly choose an index i , swap the Last element with the element at index i
 3: Pivot Index (Pi) ⟵ apply Partition Procedure (a[1..n],start,end);
 4: Apply Randomized QuickSort(a[1..n] start,pi-1)
 5: Apply Randomized QuickSort(a[1..n],pi+1,end )
```

0.3 Partition Procedure

Algorithm 3 Partition Procedure

```
Input: n =Size for the array, Array a[1..n] of n integers ,start ,end index
Output: Position of pivot element in the sorted array
 1: Partition(a[1..n], start,end )
 2: pivot = a[end]
 3: leftend=start
 4: for i=start+1 to end do
 5:     if a[i] < pivot then
 6:         leftend++;
 7:         swap(a[i],a[leftend]
 8:     end if
 9: end for
10: swap(a[i],a[leftend]
11: return leftend
```

Correctness of the Algorithm

1. Randomized Quick sort will be resulted in Sorted array (proven in the class)
2. In the Sorted Array if any Majority element exists, it should occur as an sub array of similar elements sequentially with sub array length greater than n/2
   - n = size of the input array

Efficiency of the Algorithm

0.4 Time Complexity

1. Expected time for Randomized Quick Sort(proved in the class) = $O(n\log n)$
   - n = size of the input array
2. Worst case time for Majority element check = $O(n/2)$
3. Total time Complexity $= O(n\log n + n) = O(n\log n)$

0.5 Space Complexity

1. Space for input Array = $O(n)$
   - n = size of the input array
2. Depth of the stack space = $O(n)$
   - **Stack Space will be characterized by number of activation records** it has.
   - For **Each Recursive call there will be an one activation record** maintained in the stack space

- Stack space will contains **n activation records, when the input is already is sorted and largest element is selected as pivot** in every subsequent Quick sort calls.

3. Different integer variables used= $O(c)$

   - c is some non-negative constant

4. Total Space Complexity $= O(n+n+c) = O(n)$

Description of each Function in Code

0.5.1 void swap(int* a,int* b)

- Swaps two elements using pointer Assignment

0.5.2 bool isSorted(int a, int b)

- Returns true if two elements were sorted

0.5.3 int getRandomNumber(int size)

- Returns a random number generated in the range [0, size-1]

0.5.4 int partition(int* p_array,int p_beginIndex,int p_endIndex)

- Returns the pivot index and Rearranges the array using a selecting a randomly selected element known as Pivot

0.5.5 void convertchararraytoInt(int* p_array,char* p_chararray)

- Converts the character array representation of elements to Integer array Representation

0.5.6 void applyRandomizedQuickSort(int* p_array,int p_start,int p_end)

- This is beginning point for applying Randomized quicksort on given array of elements

0.5.7 bool findMajority(int* p_arrray,int p_size)

- Return true if majority element found in the sorted sequence

Problem 2- Solution Overview

- This section focuses on details of the Solution
- Pivot index is obtained after applying **Partition Procedure** on given array of elements.Input array split into two parts left sub array and right array
- Left sub array contains all the elements $\leq$ pivot and right sub array contains elements $>$ pivot
- Now we scan the left sub array from right to left to move all the duplicate pivot elements to the right end of the left sub array. This **guarantees that all duplicate pivot elements moved to the right end of the left sub array**
- Now we can apply Randomized quick sort on the Right sub array and non-pivot elements of the left sub array
- Implementation is a **in-place** algorithm, with time complexity$O(n\log n)$ and space complexity $O(n)$

Problem 2-pseudo code of the Solution

0.6 3-way Randomized Quick Sort on given array of elements

Algorithm 4 Randomized 3-way QuickSort

```
Input: n =Size for the array, Array a[1..n] of n integers
Output: Array a[1..n] of sorted elements
1: 3-way Randomized QuickSort(a[1..n], start,end )
2: Randomly choose an index i , swap the Last element with the element at index i
3: Pivot Index (Pi) ⟵ apply Partition Procedure (a[1..n],start,end);
4: Move all duplicate pivot elements to the adjacent positions of the pi in the left
   sub-array
5: DupeCount ⟵ Counting the number of duplicate Pivot elements in the left sub-
   array
6: Apply 3-way Randomized QuickSort(a[1..n], start,pi-DupeCount-1)
7: Apply 3-way Randomized QuickSort(a[1..n],pi+1,end )
```

0.7 Partition Procedure

Algorithm 5 Partition Procedure

Input: **n** =Size for the array, Array **a[1..n]** of n integers ,start ,end index

Output: Array **Position** of pivot element in the sorted array

```
1: Partition(a[1..n], start,end )
2: pivot = a[end]
3: leftend=start
4: for i=start+1 to end do
5:     if a[i] < pivot then
6:         leftend++;
7:         swap(a[i],a[leftend]
8:     end if
9: end for
10: swap(a[i],a[leftend]
11: return leftend
```

Correctness of the Algorithm

1. Randomized Quick sort is guaranteed to sort the array (proven in the class)

2. In addition, we are scanning the left sub array (array of elements to the left of pivot index obtained after applying partition procedure) from right to left to count duplicate pivot elements and disqualify them to participate in the subsequent Quick sort calls.

3. This Reduces the cost of the Quick sort by finding the duplicate pivot elements leads to following Recurrence Relation

   T**(n)=T(q-DupesCount-1)+T(n-q)+O(n)+O(q)**

   T(n)-Time Complexity of 3-way Randomized Quick Sort

   T(q-DupesCount-1) - Time Complexity of left sub array after Removing Duplicate Pivot element from it

   T(n-q)-Time Complexity for right sub array

   O(n)-Time for Partition procedure (where n= end-start +1)

   O(q) - Time for Linear scan of left sub array (obtained after Partition) to find Duplicate pivot elements

   - n = size of the input array

Efficiency of the Algorithm

0.8 Time Complexity

1. Time complexity for 3-way Quick sort depends on time for Partition, time for quick sort on left sub array contains (q-DupesCount) elements, time for right sub array containing n-q elements, time for linear scan of the at most q elements, where q is the index of pivot element in the sorted array

2. Recurrence Relation for 3- way Randomized Quick Sort

   T**(n)=T(q-DupesCount-1)+T(n-q)+O(n)+O(q)**

$q \leq n$ **, if input array is already sorted and largest element is selected as pivot**
= T(q-DupesCount-1)+T(n-q)+O(n)+O(n)
=T(q-DupesCount-1)+T(n-q)+O(n)
$\leq$ **T(q)+T(n-q)+O(n)** (eq for Randomized Quick Sort)

- n = size of the input array

=$O(n\log n)$(Proved in class)

3. Running time for 3-way Randomized Quick Sort = $O(n\log n)$

0.9 Space Complexity

1. Space for input Array = $O(n)$
   - n= size of the input array
2. Depth of the stack space = $O(n)$
   - **Stack Space will be characterized by number of activation records** it has.
   - For **Each Recursive call there will be an one activation record** maintained in the stack space
   - Stack space will contains **n activation records, when the input is already is sorted and largest element is selected as pivot** in every subsequent Quick sort calls.
3. Different integer variables used= $O(c)$
   - c is some non-negative constant
4. Total Space Complexity $= O(n+n+c) = O(n)$

Description of each Function in Code

0.9.1 void swap(int* a,int* b)

- Swaps two elements using pointer Assignment

0.9.2 bool isSorted(int a, int b)

- Returns true if two elements were sorted

0.9.3 int getRandomNumber(int size)

- Returns a random number generated in the range [0, size-1]

0.9.4 int partition(int* p_array,int p_beginIndex,int p_endIndex)

- Returns the pivot index and Rearranges the array using a selecting a randomly selected element known as Pivot

0.9.5 void convertchararraytoInt(int* p_array,char* p_chararray)

- Converts the character array representation of elements to Integer array Representation

0.9.6 void apply3wayRandomizedQuickSort(int* p_array,int p_start,int p_end)

- This is beginning point for applying 3-way Randomized quicksort on given array of elements