**Chapter 8**

***R-8.4*** Suppose we modify the deterministic version of the quick-sort algorithm so that, instead of selecting the last element in an *n*-element sequence as the pivot, we choose the element at index *n/*2*\_*, that is, an element in the middle of the sequence. What is the running time of this version of quick-sort on a sequence that is already sorted?

**Answer:**

In the most deterministic version the first or the last value is used as pivot and in other versions pivot is chosen by random or median of three random selection. The question is n/2 is supposed to be pivot, so the sequence is split is in half each time it is iterated. The data is already sorted so there will be no swaps the height of quick sort tree is log n recursive calls would amount to O (n).

Here, the data is always split in to two equal sorted pieces. Since the pivot is in the middle of the set let T(n) be the worst case running time of the algorithm on a sequence of size n.

Tq(n) = Tq (|n/2|) +Tq(|n/2|) + cn =2Tq(|n/2|)+cn

Tq(n)= O (Tm(n)) it is similar to mergesort here the constant c and same running time given n≤1

Tm Mergesort- Tm(n) = O (n log n), Tq(n)=O (O (n log n)) = O (n log n)

[A, B, C, **D**, E, F, G, H]

[A, **B**, C] [E, **F**, G, H]

[A] [C] [5] [**7**, 8]

[8]

***C-8.3*** Suppose we are given two *n*-element sorted sequences *A* and *B* that should not be viewed as sets (that is, *A* and *B* may contain duplicate entries). Describe an *O*(*n*)-time method for computing a sequence representing the set *A ∪ B* (with no duplicates).

**Answer:**

There are two n-element sorted sequences A and B so mergesort can be used for creating a new sequence. While merging the two sequences if the values are same then it is eliminated.

Mergesort(array [], l, t) if t >I

The middle point is searched and via that point it can be divided into two halves: middle m = (l+t)/2

Merge sort for first half: mergeSort(array, l, m) and Merge sort for second half: mergeSort(array, m+1, t)

Merge the two divided half parts merge (array, l, m, t)

As soon as all the elements including the duplicate values are merged into an array lets name it C. The next step is a Linear search through the array C to find the duplicate elements can be removed.

If(nextElement ==currentElement)

Remove(element).

Hence the sequence representing the A U B with no duplicated would have the time complexity of O(n) time.

***C-8.7*** Suppose we are given a sequence *S* of *n* elements, on which a total order relation is defined. Describe an efficient method for determining whether there are two equal elements in *S*. What is the running time of your method?

**Answer:**

The first step to determine whether there are two equal elements is to sort the array then compare Then equal elements in the algorithm is searched by comparing the current element with the next element.

Let’s consider a counter i<n where n is the length of the input array

i= 0

while (i < n)

{ if(array[i] == array[i+1])

{

return “Equal Elements found in array”

}

i++

}

The time complexity to sort and search the duplicated in the array would be O (n log n) +O (n) let’s consider the sorting algorithm the quick sort algorithm.

Therefor Time= O (n log n)

***A-8.4*** Bob has a set, *A*, of *n* nuts and a set, *B*, of *n* bolts, such that each nut has a unique matching bolt. Unfortunately, the nuts in *A* all look the same, and the bolts in *B* all look the same as well. The only comparison that Bob can make is to take a nut-bolt pair (*a, b*), such that *a ∈ A* and *b ∈ B*, and test if the threads of *a* are larger, smaller, or a perfect match with the threads of *b*. Describe an efficient algorithm for Bob to match up all of his nuts and bolts. What is the running time of this algorithm?

**Answer:**

The quick sort algorithm is used where the nuts are chosen randomly and bolts are sorted into two sets, ones that has smaller than the nut and larger than the nut diameter. Through this method the matching pair is found. Now nut and bolts both can be stocked into two piles, larger and smaller than the bolt and nut. Both nut and bold have one to one relation so we can repeat the step until the right matches are found. The picking of the nut and bolt can be randomized to choose the pivots to ensure that the piles are not big due to randomization the time would be O (n log n) and the second round will do most O (n) comparison. The maximum number of round while using the randomized quicksort would take log n. So, the total tiem complexity is O (n log n)

**Chapter 9**

***R-9.2*** Describe a radix-sort method for lexicographically sorting a sequence *S* of triplets (*k, l,m*), where *k*, *l*, and *m* are integers in the range [0*,N −*1], for some *N ≥* 2. How could this scheme be extended to sequences of *d*-tuples (*k*1*, k*2*, . . ., kd*), where each *ki* is an integer in the range [0*,N −* 1]?

**Answer:**

Radix sort algorithm is a non-comparative integer sorting algorithm that sorts data with an integer sequence by combining/grouping integer keys by the individual digits which have same significant position and value. The Radix sort performs the stable bucket sort on each individual digit. Here, radix sort is used on k, l and m where k, l and m ∈ [0, N-1] and N ≥ 2. The radix sort algorithm performs three times stable bucket sort, starting from least significant to most significant digit.

Algorithm: Iteration through the index I of the tuples, starting from the last index (d-1) and going down to the first index 0. At each iteration, bucket sort the tuples of S by their i-th entry using N buckets.

BucketSort(array[], n)

First n empty buckets are created. Then array[i] is inserted into bucket [n\*array[i]] for every element array[i]. Then insertions sort is implemented on individual buckets. All the sorted buckets are then concatenated.

The time complexity of this algorithm would be O (d(n+N)) where n is the length of the integer sequence.

***C-9.5*** Suppose we are given a sequence, *S*, of *n* integers in the range from 1 to *n*3. Give an *O*(*n*)-time method for determining whether there are two equal numbers in *S*.

**Answer:**

Radix sort is utilized in this case to determine two equal number in the range of 1 to n3. Radix sort can view the numbers in the data as triples of numbers in the range from 1 to n, which will take O (n) time. To find the 2-equal number in the sequence S following algorithm can be implemented:

Let i< length where length of input array.

i = 0

while (i < length)

{

If(array[i] == array[i+1]

{

return “True”

}

i++

}

The time complexity of this algorithm would be the complexity of traversing the array once which is O (n).

***A-9.5*** Consider the election problem from the previous exercise, but now describe an algorithm running in *O*(*n*) time to determine the student numbers of every candidate that received more than *n/*3 votes.

**Answer:**

The bucket sort is implemented to sort the array and find the student that has more than n/3 votes. Suppose all the elements are in the list in the range of the array(n). A counter array is created to find the duplicate elements and are stored in the counter array. The count number of repeated elements we traverse the entire list once, the time taken to traverse the list is linear so the time complexity is O (n).

Two loops are used add/remove element & increase/ decrease the count. It is not quadratic as the maximum number of iteration is 2n. If all the elements are stored in the same bucket then the inner loop iterated n times so, other buckets remain empty. The inner loop does not iterate for other buckets.

Time required to sort the array to count is O (n) and the iteration to check number of votes greater than n/3 takes O (n). Total (n) = O (n) + O (n) = O (n).