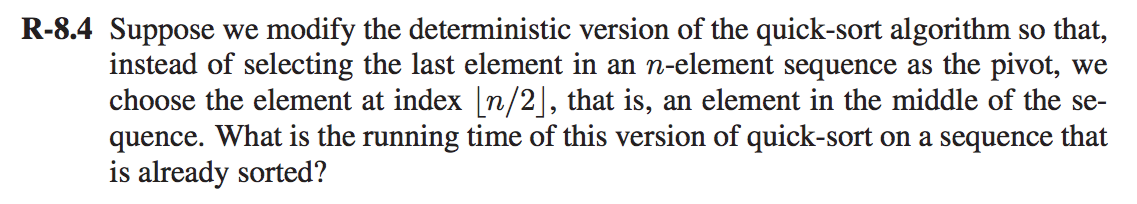
CS 600 Homework 5 | CWID 10430147 | Divyendra Patil | Username: dpatil3  
Date: 10/06/2017

Chapter 8:



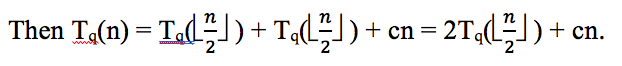
**Solution R-8.4:**

In the deterministic version, we assume to choose the pivot to be first or the last element unlike in the non-deterministic version where we choose the pivot by random or by the median of three random selections.

In the case of selecting the middle element (n/2) as the pivot, each iteration will divide the partition in half recursively.

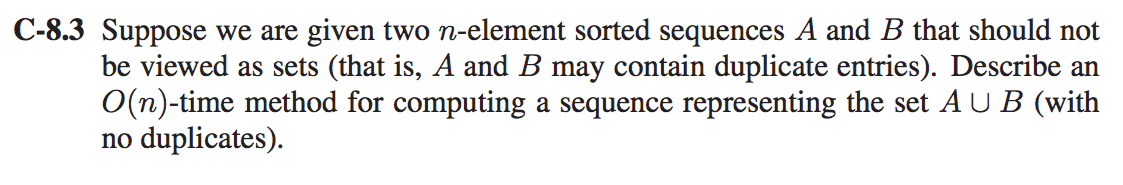
Moreover, if the data is already sorted, there would be no swaps and this will result in log n recursive calls amounting to a cost of O(n).

This version of quicksort would always split the sequence into two almost equal sorted pieces since the pivot is always smack in the middle of it. Let T(n) be the worst-case running time of this quicksort algorithm on a sequence of size n.



The above equation a somewhat similar to merge sort. Hence Tq(n) = O(Tm(n)) given the same constant c and the same running time given n ≤ 1.

Since Tm(n) = O(n log n), Tq(n) = O(O(n log n)) = O(n log n).



**Solution C-8.3:**

**Input**: Two sorted n-element sequences A and B (they might contain duplicate values)

**Output**: An (n-z) element array which contains both the elements of A and B void of duplicates. Where we denote z = no. of duplicates

Merge sequences A and B into a new sequence C by using ordinary merge sort.

**MergeSort(arr[], l, t)**

If t > l

a. Find the middle point to divide the array into two halves:

middle m = (l+t)/2

b. Call mergeSort for first half:

Call mergeSort(arr, l, m)

c. Call mergeSort for second half:

Call mergeSort(arr, m+1, t)

d. Merge the two halves sorted in step 2 and 3:

Call merge(arr, l, m, t)

Once, all the elements from A and B including duplicates are merged into a new array C, we can perform a linear scan through the this new array to find the duplicate elements that needs to be removed.

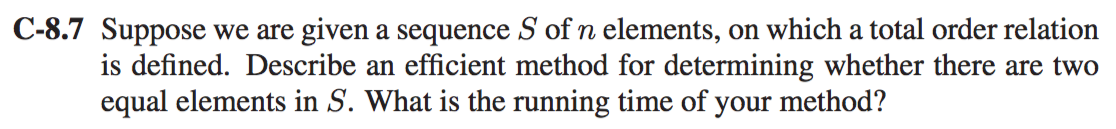
To remove the duplicate elements :

a. Scan through the array C.

b. if (nextElement == currentElement)

remove(element).

Thus, the sequence representing the sets A U B with no duplicates takes O(n) time.



**Solution C-8.7:**

**Input**: An array having n elements

**Output**: Result whether the array contains equal elements or not

We need to sort the array using any one of the sorting algo’s. Once the array is sorted we can search for equal elements using the following algorithm,

Consider a counter i < n where n is the length of the input array

i = 0

while (i<n)

{

if(arr[i] = = arr[i+1])

{

return “Equal elements found in array”

}

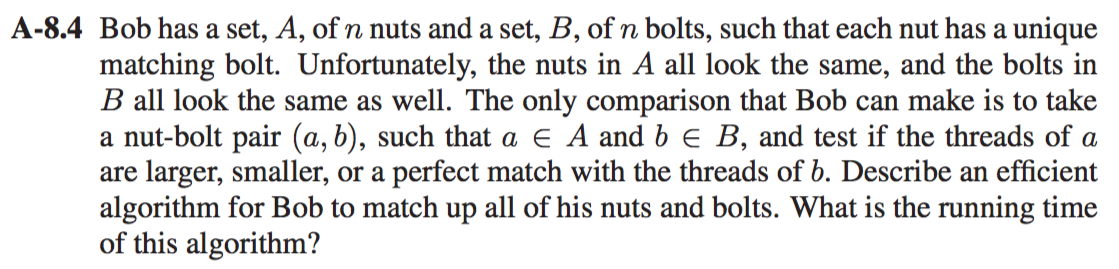
i++

}

The **TOTAL** time complexity of the above method would be the complexity of the sorting algorithm & (Addition) to the complexity of the search array traversal.

Therefore time complexity = O(n log n) + O(n). Considering the sorting algorithm to be a quick sort.

Therefore the overall time complexity is O(n log n)



**Solution A-8.4:**

1] We apply the randomized QuickSort algorithm in the following manner:

2] Pick a nut (at random), use this to sort the bolts into two sets, those having diameters smaller than the nut and larger than the nut, finding the matching bolt in the process.

3] Now that you have found one matching nut/bolt pair, use bolt to sort nuts into two piles, those with threads larger or smaller than the bolt.

4] Now you have two pile of nuts and two piles of bolts. note that since each bolt has a matching nut, and vice-versa (and the matches are unique), then the pile of bolts with the “smaller” threads is equal in size to the pile of nuts with the “smaller” threads, and the same is true of the other pile of nuts and bolts. Then you repeat this process on each of the matching piles of nuts and bolts.

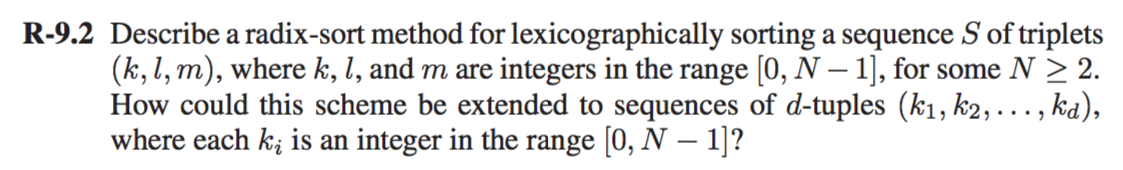
5] If we use the randomized procedure to choose our “pivots” to ensure the piles aren’t too big, then, in expectation since it’s a randomized procedure, we’ll be doing at most O(n log n) comparisons. The first round takes n + n, which comes out to be O(n).

6] The second round we’ll do at most O(n) comparisons again.

7] The number of rounds until we’re done will be at most log n, as in the case of the usual randomized QuickSort procedure.

8] So we have at most O(n log n) comparisons

Chapter 9:



Radix sort algorithm sorts given integer sequence by clustering integer keys by the individual digits which have same significant position and value. The radix sort performs stable bucket on each individual digits. Here we use a radix-sort on (𝑘, 𝑙, 𝑚) where 𝑘,𝑙 𝑎𝑛𝑑 𝑚∈[0,𝑁−1] 𝑎𝑛𝑑 𝑁≥2

And for lexicographical sorting (𝑘1, 𝑙1, 𝑚1)<(𝑘2,𝑙2,𝑚2)<⋯<(𝑘𝑑,𝑙𝑑,𝑚𝑑)

The radix sort algorithm performs three times stable bucket sort, starting from least significant digit to most significant digit.

Let’s understand with example:

Integer sequence with three digit 𝑆=((8,5,7),(6,2,6),(3,5,7),(4,2,5),(4,3,5))

First we sort S using the least significant position 𝑚 𝑆𝑚=((3,2,**5**),(4,3,**5**),(6,2,**6**),(8,5,**7**),(3,5,**7**))

Second we sort S with the significant position on 𝑙 which gives us 𝑆𝑚,𝑙=((4,**2**,5),(6,**2**,6),(4,**3**,5),(8,**5**,7),(3,**5**,7))

After this we then sort S with the most significant position 𝑘 𝑆𝑚,𝑙,𝑘=((**3**,2,5),(**3**,5,7),(**4**,3,5),(**6**,2,6),(**8**,5,7))

Algorithm :

* Iterate 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(arr[], n)

1) Create n empty buckets (Or lists).

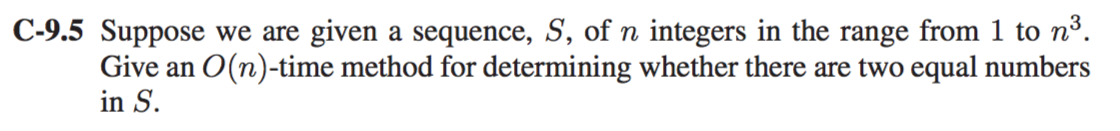
2) Do following for every array element arr[i].

a) Insert arr[i] into bucket[n\*array[i]]

3) Sort individual buckets using insertion sort.

4) Concatenate all sorted buckets.

=> The running-time of this algorithm is O(d(n + N)) where n is the length of S.



1] The way to determine two equal numbers in the range of 1 to n3 is utilizing Radix sort.

2] Radix sort will be a suitable algorithm for sorting in this case because we can view the elements as triples of numbers in the range from 1 to n,which will take O(n) time.

3] To find the two equal elements in sequence S, consider the following algorithm:

4] Consider i < len, where len is the length of input array.

i = 0

while (i<len)

{

if(arr[i] = = arr[i+1])

{

return “TRUE”

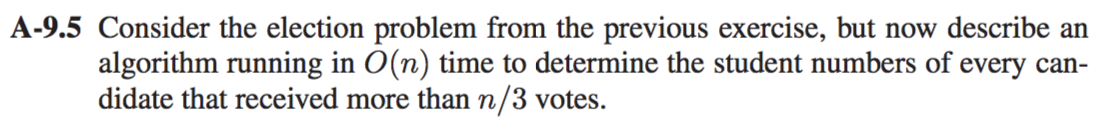
}

i++

}

The total time complexity of the above method would be the complexity of the traversing the array once i.e O(n).

Therefore the total time complexity is O(n).



We use the bucket sort to sort the array and find the student (student id) having more than n/3 votes.

**The Input:** Unsorted array A[n] with ‘n’ votes as ‘student id’

**The Output**: Student id with votes more than n/3

Let all the elements in the list lie in the range of the array(n). We then create a counter array to find the elements which are repeated and store it in the counter array. Arrange the elements in the array, depends on the counter array.

**Analysis of bucket sort**: To count the number of repeated elements, we traverse the entire list once. The time to traverse the list is linear. **Hence the traversing time is O(n).**

In order to copy back, we use two loops i.e add/remove an element or increase/decrease the count. But here we cannot judge that it is quadratic, because the maximum number of iteration is 2n. Cause if all the elements are stored in the same bucket then the inner loop iterates n times. Since all the other elements are stored in the same bucket other buckets remain empty. Hence the inner loop does not iterate for other buckets.

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

T(n) = O(n) + O(n) = O(n).

Therefore, the method to find the student id’s with greater votes than (n/3) using bucket sort is O(n).