Homework 4

Name: Atiq Patel CWID: 10432883

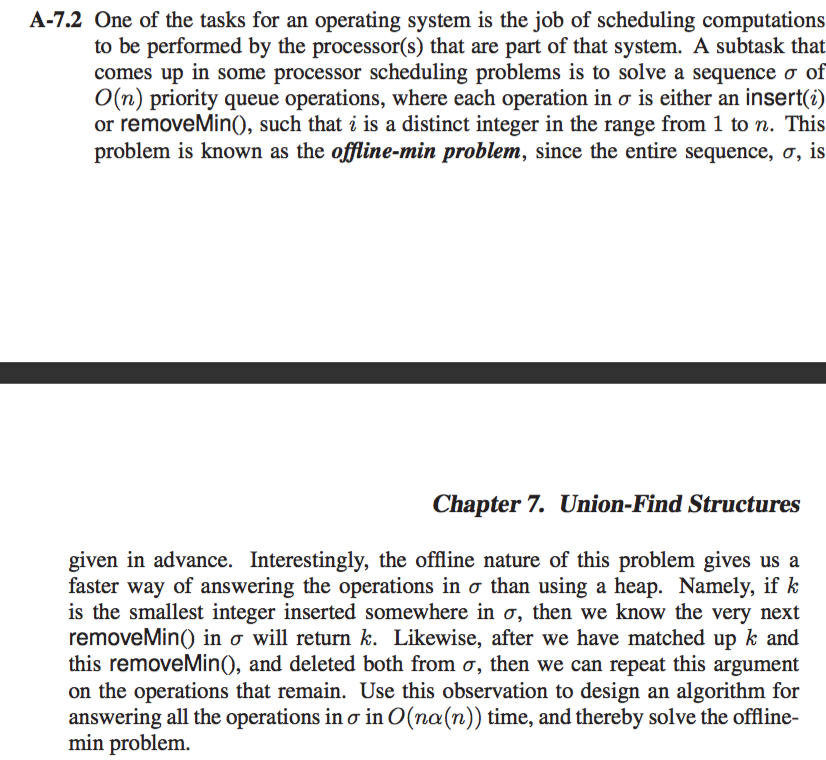
Screen%20Shot%202017-09-28%20at%202.38.37%20PM.png

**Solution:**

We can remove the duplicates from by sorting them and then removing duplications. We can use Divide and Conquer technique to sort them out as its worst case is **O (nlog n)**.

Now, we have to delete the elements, we are going to be using Linked List as the data structure. Deletion time in a linked list would differ from where the object is getting deleted if it is a known place then, O (1), but finding that position would be O (n). We consider the worst case **O (n)**.

So the total time taken would be O (n log n) + O (n) which is O(n(1+ log n)) and that is **O (nlog n)**.



**Solution:**

**Algorithm OffMin(Sequence σ):**

**Input:** A sequence σ of priority queue operations.

**Output:** The desired output of the sequence σ of operations that are done in the array R.

**for** i🡨0 to num(σ) **do** //num (σ)= total number of sequence operations

**if** σ[i] == removeMin() **then** //perform removal of minimum number

RemoveCounter🡨RemoveCounter + 1 & index 🡨i

**else do** //perform insertion

R🡨insert(σ(i))

**If** index <num(σ) **then**

FinalResult=R[index – RemoveCounter]

**else do** //storing the result

FinalResult=R

MergeSort(FinalResult)

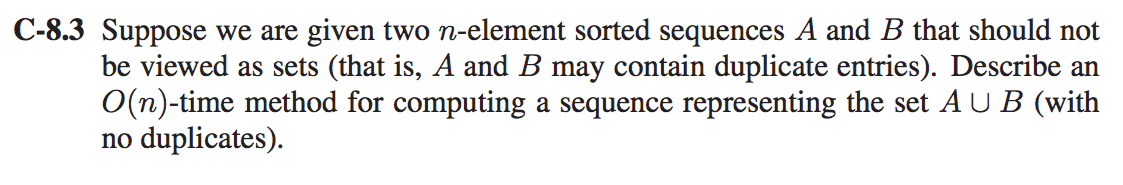
Consider a sequence be [insert(8),insert(10),insert(4),RemoveMin(),1,RemoveMin()]

The output in FinalResult should be 8,10.

Consider α be the operations performed which include RemoveMin() and insert(i) where i is the element. If there are n number of α operations. Then the time taken would be **nα**. This particular algorithm iterates till n numbers of sequence giving us O (n) time. Hence, the overall run time would be

T(n)= O(n)\*O(nα)=**O(nα(n))**

Thus, the run time for the whole algorithm would be **O(nα(n))**.

**Solution:**

We are given two n element sorted sequence but we haven’t been told sorted in what way i.e ascending or descending so considering that we will use Merge Sort as its worst case is O (n log n).

**Algorithm MergeSort(A, Low,High):**

**Input:** The two n sorted element array sequences A and B.

**Output:** An n-z array which contains its duplicates as well.

**if** high>low **do**

Middle=(Low+High)/2

MergeSort(A,Low,Middle)

Merge(A,low,Middle,High)

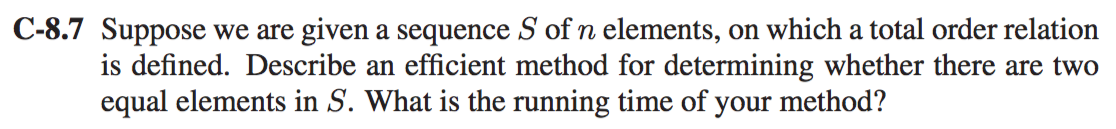
//After MergeSort is done it is stored in a result array R will contain the duplicates. Duplicates can be //removed by parsing linearly through R and comparing the previous and the current element.

**For** i🡨0 to len(R) **do**

if(presentelement==previouselement) **then**

Delete(presentelement)

The run time for this particular algorithm is **O (n)**.



**Solution:**

The most efficient way would be sorting the particular sequence S of n elements and further parsing it linearly and then comparing the current element to the previous one.

We will use merge Sort to sort out the sequence which has a worst case time complexity of O (n log n).

**Algorithm GetThisDone(Array A):**

**Input:** An Array A having a sequence of n elements.

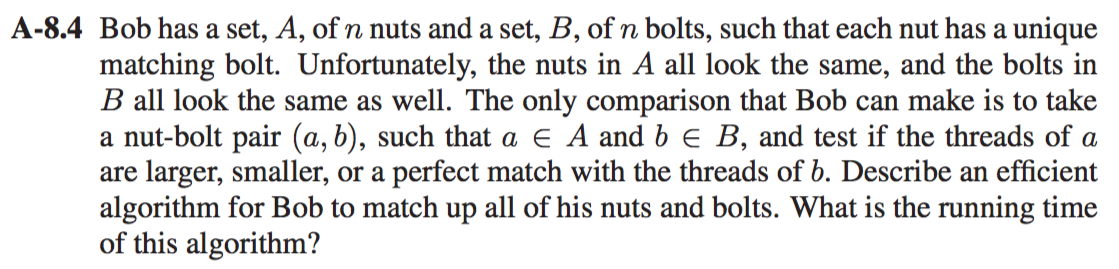
**Ouput:** Letting the user know if there is a presence of two equal elements in S.

**for** i🡨0 to n-1 **do**

**if** (A[i]==A[i+1]) **do**

**return** Equal element found

The sorting takes O (n log n) time and this algorithm parses linearly so it takes O (n) time. Therefore the overall complexity would be O (n(1+log n)) which is **O (nlog n)**.

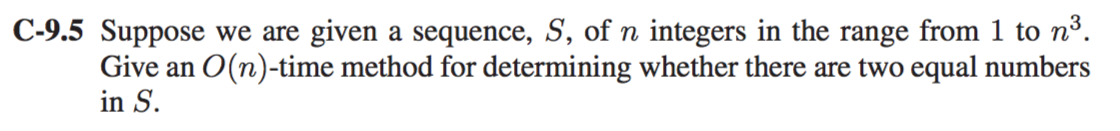


**Solution:**

We can solve this problem by taking the following steps: -

1. Apply QuickSort algorithm with slight modifications: -
   * Pick a random pivot point (nut) and use it to sort the bolts into two sets, one being smaller and other being larger, which would lead to finding the right bolts for the particular nut.
   * Once we have found a matching nut and bolt, use the bolts to arrange nuts into two piles, one being smaller and other being larger than the bolt.
2. We now have two piles of bolts. Each nut has ended up with a matching bolt and each bolt has ended up with matching nut. We keep on repeating the process on each of the matching piles of bolts and nuts.
3. Quick sort has a good average case and typically it works out well for choosing randomized pivots to know that the piles are not to big. Either way the worst case would O(n log n).
4. The first iteration takes 2n which turns to be O (n).
5. The second set of iterations take O (n) as well.
6. The number of rounds until we have all the matching nuts and bolts will keep on decreasing leading to O(log n).

Thus, the overall run time for this algorithm would be **O (n log n)**.



**Solution:**

The best approach for the solution of this problem is by using the Radix Sort.

Radix Sort will sort the first component and then second component and give out a lexicographically ordered sequence. The only work needed to be done is to compare each component and if it matches out then we have two equal numbers.

**Algorithm Match(Sequence S):**

**Input:** An Array A contains sequence S of n integers.

**Output:** Check whether there are two equal numbers.

i🡨0

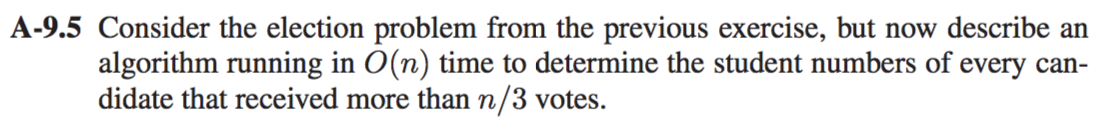
**while** i<Array.length(A)-1 **do**

**if** (firstcomponent==secondcomponent) **do**

**return** “Two Equal numbers found”

i🡨i+1

The run time for the above method is **O (n)** as it requires the traversal of the whole array A.



**Solution:**

One way we can solve this problem is by using Bucket Sort to sort the Array A containing n votes to find the candidate having more than n/3 votes.

We create a result array R to find out which student numbers are repeated over and over again and store it in a Result Array R. We sort the student numbers in the and store them in the array.

We use bucket sort to sort the whole array R based on the student elements and place them within ranges say 0-10,11-20... and so on.

For this to happen we need to parse the whole list once which takes **O (n)** runtime.

Two for loops have to be used in this case, one for parsing the whole array R and then parsing inside each bucket. It would take maximum of 2n iterations when worst case is considered. First it has to check every Bucket that’s n and if all the elements are present in the same bucket which is another n. That makes a total of 2n.

The overall time required to sort the array is the summation of time required to count O (n) and the looping to determine the student numbers of every candidate greater than n/3 also accounts for O (n) which takes a total of O(n+n) = O (2n) which is equal to **O (n)** run time.

Thus, the run time to determine to find if the student id have greater votes than n/3 is **O(n).**