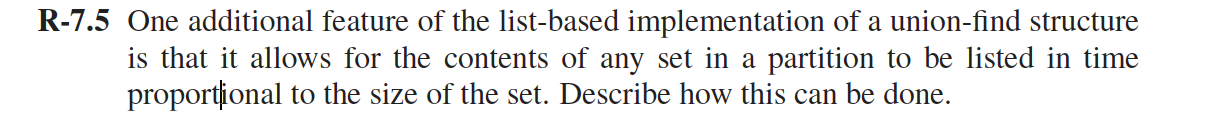
R-7.5



Answer：

we use a collection of linked lists, one for each set. Each linked list contains a head node which contains:

a) size of the set

b) name of the set

c) pointer to the first node (and maybe last node too) of the linked list. Each node of the linked list contains the pointer to one element of the set and a pointer to head node of the list.

**Algorithm To find all elements of the set in which x belongs:**

Algorithm Find-Set(x):

temp= x.head ;

temp = temp->next; //reach to first node of linked list

while(temp != NULL)

{

print temp->data;

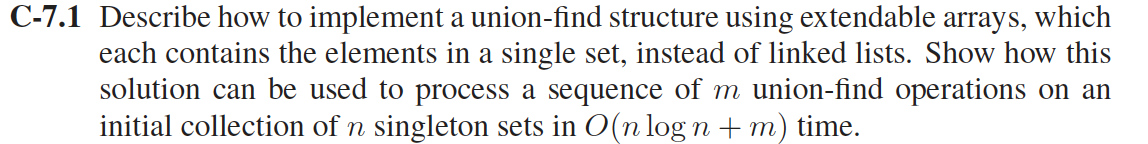
temp=temp->next;

}

end

**Time complexity:** O(n) where n is the number of elements in the set.

C-7.1



Using two arrays:

Each set is using an extendable array A and another array B with the same position to store the first element in the array, the set’s representative.

B[0] store the last position of element of A.

* Make-Set(*x*): create a singleton set containing *x*
* Find-Set(*x*): traverse the array A to find the list object, and then get to array B to find the representative
* Union(*x*, *y*): append *y*'s array onto the end of *x*'s array.
  + Use *x*'s array B to find the end.
  + Need to update the array B of y back to the set object for every element on *y’s array*.

If there are *n* Make-Sets and *n* Unions, the amortized time per operation is O(*n*)!. Each Make-Set and Find-Set still takes O(1).

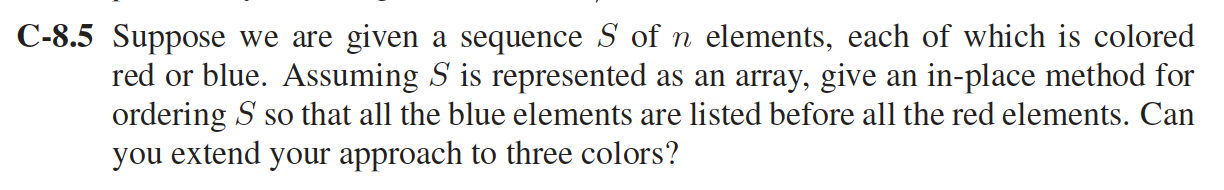
Each representative set for a given element is updated ≤ log *n* times, and there are *n* elements plus *m* operations. a sequence of *m* operations on *n* elements takes O(*m* + *n* log *n*) time.

A-7.5

Use the Union find method.

1. around the edge of the board, place in each cell either a red piece or a blue piece. This initialization makes it easy to check for a winning move. This uses at most O(n) MakeSet operations
2. Then, we check if the representative of the set containing any cell on the left boundary is the same as the representative of the set containing any cell on the right boundary. If they are, this is a winning move and quit the game. Otherwise, continue. This uses 2 FindSet operations.

C-8.5

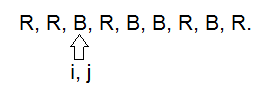


Assuming the given sequence is R, R, B, R, B, B, R, B, R.

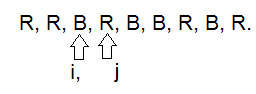
1. Assume to two pointers i and j pointing to the elements of the sequence.

2．Search for first occurrence of the B element (Blue element). And point i and j to that particular element.

Ex: For sequence R, R, B, R, B, B, R, B, R , it is:



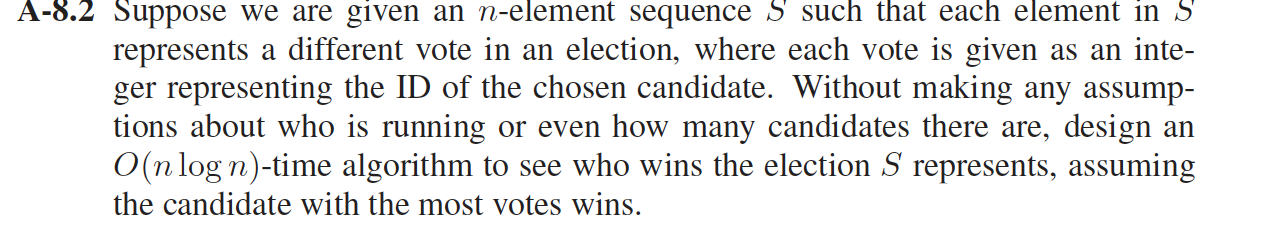
1. Then, from place where 'j' is pointing, search for the first occurrence of R and point 'j' to R.



1. Swap the elements i and j are pointing. And increment i pointer.
2. Repeat from step 3 until j reaches end of the sequence.

**Time complexity:** The S will be scanned for one time, and the time complexity is O(n).

A-8.2



1) Sort the sequence using Quick sort algorithm which has O(nlogn) complexity.

2) Now we have the sorted array S[1...n], winner=S[1], count=1, maxVote=1

3) for i= 2 to n

{

while(S[i-1] == S[i])

{

count++;

i++;

}

if(count>maxVote)

{

maxVote=count;

winner=S[i-1];

}

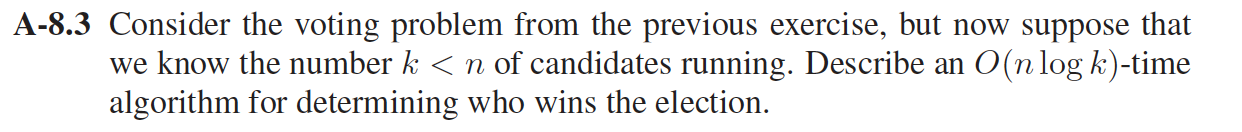
count=1;

i++;

}

This will take O(n) time because the inner(while) loop is affecting the outer(for) loop's deriving variable i.

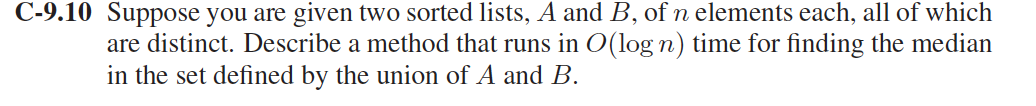
A-8.3



In this case, the candidates can be stored in a balanced binary tree (for example, an AVL Tree).

1. Each node should store a student ID and the number of votes they have received so far. As each vote in the sequence is processed, search the tree for the student ID of the chosen candidate (which takes *O*(log *k*) time).
2. If the ID is found, add one to its number of votes. Otherwise, create a new node with this ID and with one vote. At the end, go through all the nodes in the tree to find the one with the most votes.
3. The process could be sped up even further in the average case (though not in the worst case) by replacing the AVL Tree with a Hash Table.

C-9.10



The median of a sorted list of numbers, can be found, by arranging all the numbers from lowest value number to highest value number and then picking the middle number.

**Comparing the medians of two sorted lists**

In this method we will first obtain the medians of the two sorted lists and then we will compare them.

Suppose we are given two sorted lists A and B.

**Algorithm:**

1) Calculate the medians m1 and m2 from given list A and B respectively.

2) If m1 and m2 both are equal then return either m1 or m2.

3) If m1 > m2, then median is present in one of the below two sublists.

a) From first element of A1 to m1 (A1[0...|n/2|])

b) From m2 to last element of B2 (B2[|n/2|...n-1])

4) If m2 >m1, then median is present in one of the below two sublists.

a) From m1 to last element of A1 (A1[|n/2|...n-1])

b) From first element of B2 to m2 (B2[0...|n/2|])

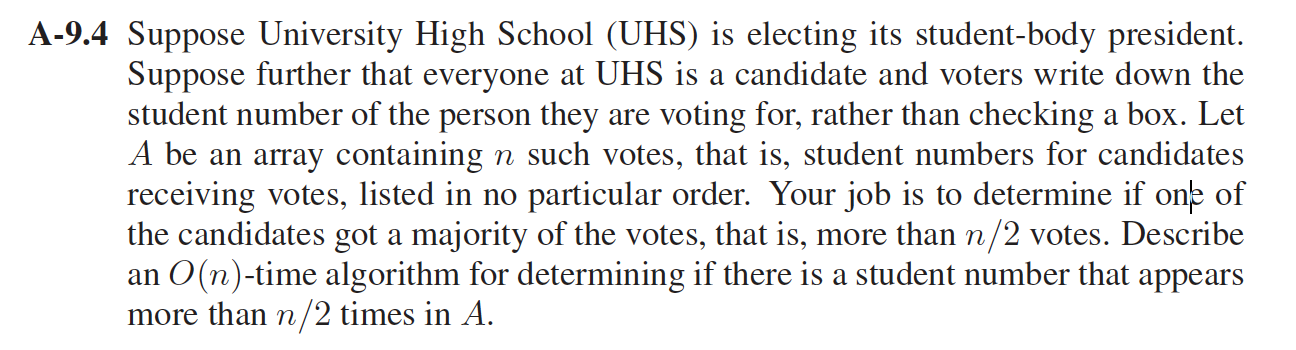
5) Repeat the above steps until size of both the sublists becomes 2.

6) If size of the two lists is 2 then use this formula to get the median.

Median = (max(A1[0], B2[0]) + min(A1[1], B2[1]))/2

Time complexity for above algorithm is O(log n)

A-9.4



1. The first step gives the element that may be majority element in the array. If there is a majority element in an array, then this step will definitely return majority element, otherwise it will return candidate for majority element.
2. Check if the element obtained from above step is majority element.This step is necessary as we are not always sure that element return by first step is majority element.

**1. Finding a Candidate :**

We use the Moore’s Voting Algorithm which works in O(n).

Basic idea of the algorithm is that if we cancel out each occurrence of an element *e* with all the other elements that are different from *e* then *e* will exist till end if it is a majority element.

findCandidate(a[], size)

Initialize index and count of majority element

maj\_index = 0, count = 1

Loop for i = 1 to size – 1

If a[maj\_index] == a[i]

count++

Else

count--;

If count == 0

maj\_index = i;

count = 1

Return a[maj\_index]

**2.Check if the element obtained in step 1 is majority element or not :**

printMajority (a[], size)

Find the candidate for majority

If candidate is majority. i.e., appears more than n/2 times.

Print the candidate

Else Print "No Majority Element"

In this second phase we need to check if the candidate is really a majority element. Second phase is simple and can be easily done in O(n). We just need to check if count of the candidate element is greater than n/2.