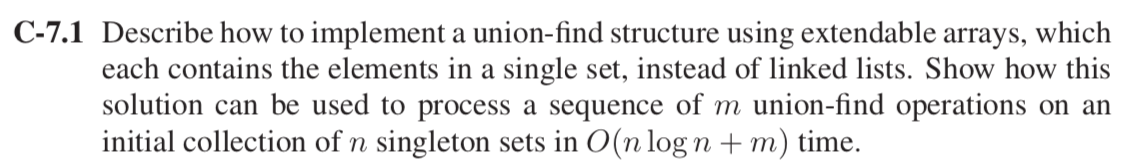


**Solution:**

Representing the list-based implementation using **linked list** will allow the contents of any set in a partition to be listed in time proportional to the size of the set.

Consider a list for set A contains the head node which stores the pointer to the first and last nodes of linked list containing pointers to all the elements of A. Each node of a linked list for A stores a pointer to an element belonging to that set and pointer to the head node for A.

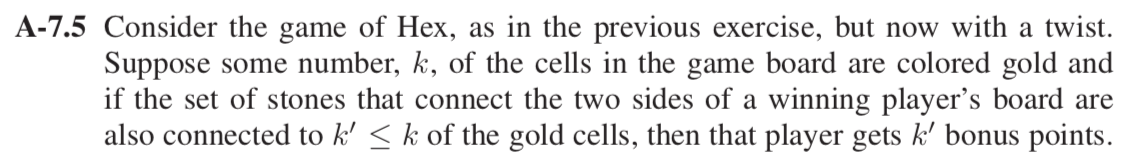
Constructing a set using linked list will be equal to the insertion operation for n element in linked list that is O(n)



**Solution:**

Given the elements in a single set, say {1}, {2}, {3},….., {n}. Perform UNION operation on the set by merging smaller set into larger set which helps in reducing the amortized cost. As the walking for all the elements occur from 1 to *n* in an effective way.

Consider an element *x* in the sequence of *m* union-find operation. Each time *x’*s element pointer changes the size of the set containing it gets double(extendable array implementation). The size of the set containing *x* after n union will be 2n. Each node can have its updatepointer changed at most log n times. Thus for all the n nodes of union-find data structure it will take O(n log n). Amortized time per union is O(log n). In union-find structure **Find** and **Make-Set** operation will take O(1) time in worst case and sequence of *m* union operation will take O(m + n log n).



**Solution:**

Using Union-Find data structure, we can detect when a player wins and how many bonus points it will earn.

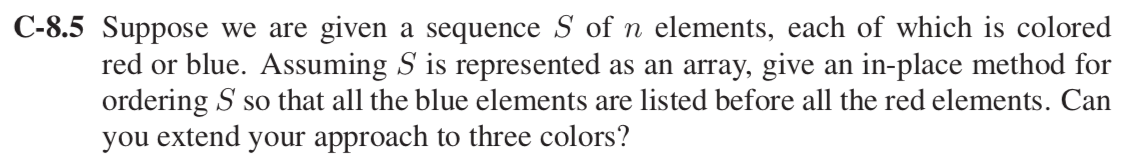
1. Firstly, named all the nodes in the n\*n grid from 0 to n2 -1. And make the singleton set for the n nodes. Example: {1}, {2}, {3}, ………{n}. Placing black color and white color stone from top to bottom starting from both corners and left to right respectively.
2. Now using **union** operation to connect each new cell with the adjacent cell if they have same color (black or white). Starting from two end points (left-right or top- bottom)
3. Use **find** operation to check whether the added cell belongs to same color cell. Also check any set containing any cell on the left boundary is same as the set containing any cell on the right boundary (similarly for top-bottom). If this condition holds this is the winning move. Else continue with the union-find.

Similarly, conditions can be checked from top to bottom.

Completing the chain from one end to other (left to right or top to bottom)

After completing the chain count the number of gold cells (*k’)* in the final set and that will be the bonus points for winning candidate.

**Time Complexity**: Make set operation will take O(n)time. Union operation for n elements will take O(n log n) time. So total time complexity is O(n log n).



**Solution:**

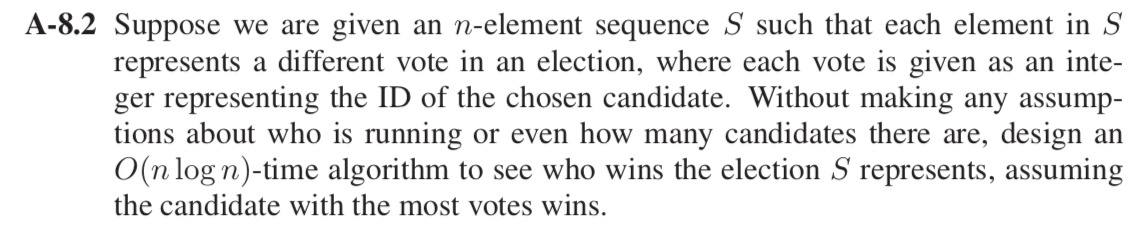
By using an in-place (Quick-sort) method for ordering S so that all the blue elements are listed before all the red elements.

Take an integer ‘***l’*** which scan elements rightward and ‘***r’*** which scans elements leftward. If ***l*** is at the blue element than increment the value for ***l*** until it reached to red element.

If ***r*** is at the red element than decrement the value for ***r*** until it reached to blue element.

When ***l*** reached the red element and ***r*** reached the blue element then swap the elements S[***l***] and S[***r***]. Recursively call this procedure until ***l*** and ***r*** reached to same position.

Yes, we can extend this approach to three colors. By implementing the above approach twice, we can solve it for three colors. Firstly, we can solve it for one color by moving one of the colors to the left of the array and swapping other two colors at the right of the array. Now again applying the same approach to the other two colors which were moved to the right. Starting point for this will be the ending point of the sorted color ***l*** and ***r*** be at the right most index of the array and then apply the similar approach as mentioned above.

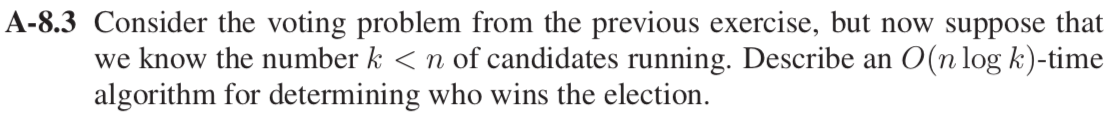


**Solution:**

Given an *n*-element sequence S such that each element in S represent a different vote in election, where each vote is given as an integer.

1. As the sequence is given, we have to sort the elements in S using quicksort or mergesort.
2. Traverse the list and keep track of the number of votes for individual candidates. The person who receives maximum votes say ID*max* and also keep track of the count and check with other candidates in the list.
3. If the number of votes for any other individual is greater than ID*max* then change the maximum votes with the current votes.
4. After that traverse the list, the maximum number of votes given to a particular person will wins the election.

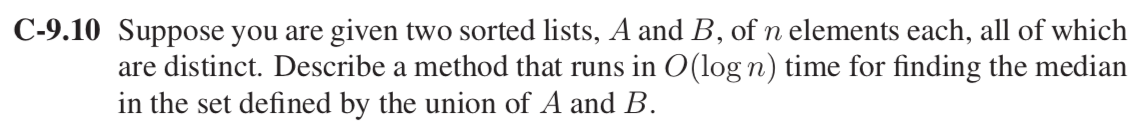
**Time Complexity**: Sorting the elements will take O(n log n) and traversing all the elements of S i.e. *n*  will take O(n) time. So the above algorithm will run in O(n log n).



**Solution:**

1. For implementing the algorithm, we can use balanced search tree for storing the vote counts with respect to the ID of all candidates. Where each node represents the ID and store the value of votes received by particular ID.
2. Initially the count for votes is 0, as we begin traversing the elements in S sequence, increment the count of the votes for a particular ID.
3. Go through all the nodes in the tree and find the ID which contains maximum number of votes.

**Time Complexity**: The search and update operation in the tree for *k* elements will take O(log k) time. And traversing the sequence of *n* elements in S will take O(n) time. So, in total the running time of the algorithm is O(n log k).



**Solution:**

Given: Two sorted list A and B of *n* elements each, all of which are distinct.

1. First calculate the medians medA and medB of sorted list A and B respectively.
2. If medA and medB are equal, then return either medA or medB.
3. Else if medA is greater than medB then median is present in one of the two subarrays defined below:
   * From first element of A to medA (A[0] …..A[n/2]) or
   * From medB to last element of B (B[n/2] to B[n-1])
4. Else if medA is smaller than medB them median is present in one of the two subarrays defined below:
   * From medA to last element of list A (A[n/2] …...A[n-1]) or
   * From first element of list B to medB (B[0] …..B[n/2)
5. Recursively call above process until the size of both the subarrays becomes 2.
6. Using formula to calculate the median when size of the lists become 2 is:

Median = (max(A[0], B[0]) + min (A[1], B[1])) / 2

**Time Complexity**: Above algorithm runs in O(log n) time.

Example: A = {2, 11, 17, 26, 35}

B = {3, 12, 20, 30, 40}

medA = 17

medB = 20

medA < medB (apply step d)

[17, 26, 35] and [3, 12, 20]

In above two medA and medB is 26 and 12 respectively

Now medA > medB (apply step c)

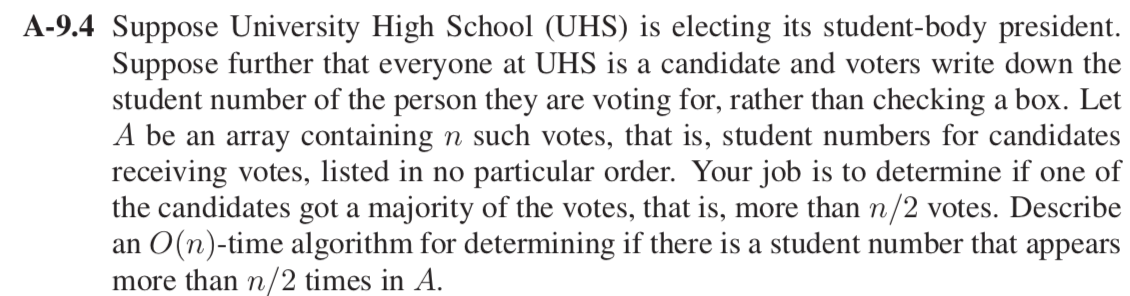
Subarray becomes [17, 26] and [12, 20]

Now, checking step f from above method, there are 2 elements left

Median = (max(17, 12) + min(26, 20)) / 2

= (17 + 20)/2

= 18.5



**Solution:**

For determining if there is a student number that appears more than n/2 times in A we have to use Randomized Quick select algorithm based on prune and search paradigm.

Student number will work as the index and the value of vote stored in a particular index. First count the votes for a particular candidate and then apply prune and search.

**Prune:** In an array A containing *n* votes, select a random element *x* and partition *A* into three sequences.

L: elements less than *x*

E: elements equals *x*

G: elements greater than *x*

In this algorithm, depending on the count of votes candidates getting less votes will be in the left of *x* and getting majority of votes will be on right side of *x*.

**Search**: So, recursively calling the quick select on G will give the result for one of the candidates who got majority of votes.

**Time Complexity**: The above algorithm will run in O(n). First step(prune) will take O(n) to partition all the elements into three L, E, G. And second step will also take O(n) as recursively call the quick select.