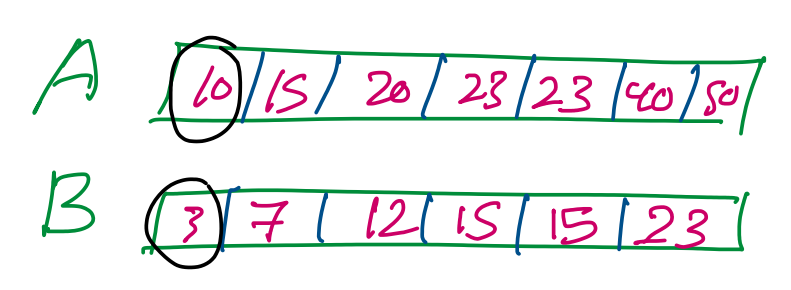
assignment- 5

Friday, 1 November 2024 11:14 PM

R-4.5 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). Give an O(*n*)-time pseudo-code algorithm for computing a sequence representing the set A ∪ B (with no duplicates).

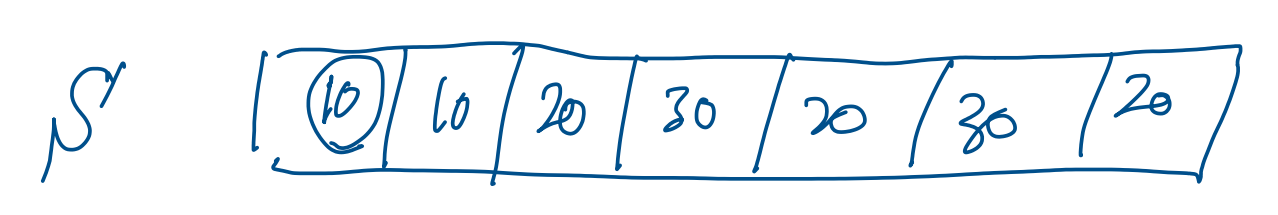
```
Algorithm unionAB( A, B)
    n = length of A
    i = 0      // Index of A
    j = 0      // Index of B
    C = empty sequence   // output sequence

    While i < n and j < n
        a = A[i]
        b = B[j]
        if a == b
            if C is empty or C[-1] ≠ a        // C[-1] means the last element in C
                C.append(a)
            j++
            i++

        else if a < b
            if C is empty or C[-1] ≠ a
                C.append(a)
            i++

        else            // a > b
            if C is empty or C[-1] ≠ b
                C.append(b)
            j++

    While i < n                  // Adding any Remaing elements of A
        a = A[i]
        if C is empty or C[-1] ≠ a
            C.append(a)
        i++

    While j < n                  // Adding any remaing elements of B
        b = B[j]
        if C is empty or C[-1] ≠ b
            C.append(b)
        j++

    return C
```



R-4.9 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 rank (index) $\lfloor n/2 \rfloor$, 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?

The running time of the modified Quicksort algorithm on an already Sorted sequence is $O(n \log n)$.

C-4.10 Suppose we are given an *n*-element sequence S such that each element in S represents a different vote in an election, where each vote is given as an integer representing the ID of the chosen candidate. Without making any assumptions about who is running or even how many candidates there are, design an *O(n* log *n)*-time algorithm to see who wins the election S represents, assuming the candidate with the most votes wins.



- We here use mergeSort to sort the S sequence
- then we traverse the sorted sequence & count the consecutive identical IDs to find the candidate with maximum votes.

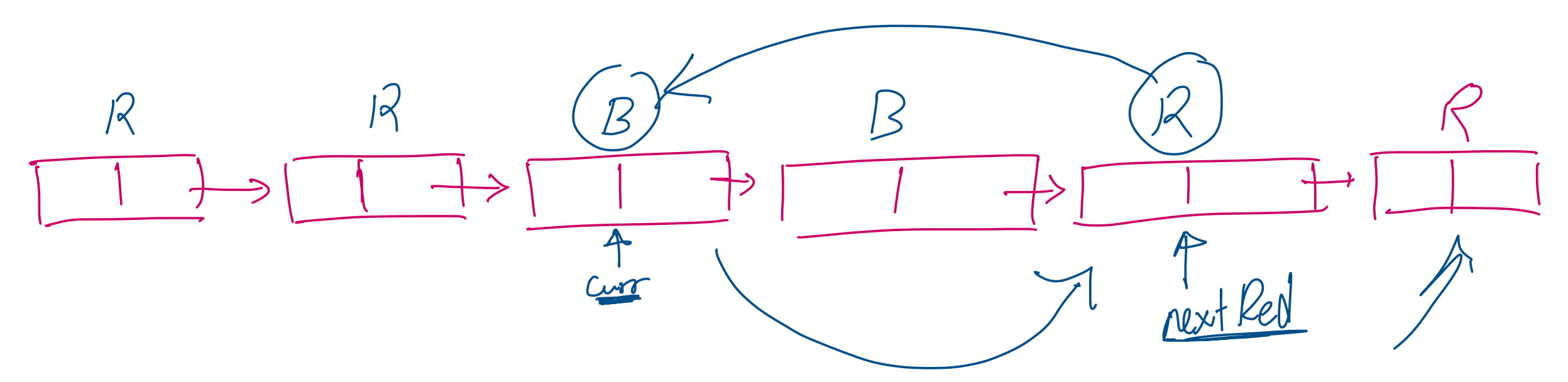
```
Algorithm FindWinner ( S)
    n = length of S
    Sort ( S)   // MergeSort   O(n log n) time
    max_candidate = S[0]
    max_Count = 1
    Current_Count = 1

    for i from 1 to n-1
        if S[i] == S[i-1]
            Current_Count ++
        else
            if Current_Count > max_Count
                max_Candidate = S[i-1]
                max_Count = Current_Count
            Current_Count = 1

    if current_Count > max_Count
        max_Candidate = S[n-1]

    return max_candidate
```

Complexity

1. Sorting $O(n \log n)$
2. Travese $O(n)$

$T(n) = O(n \log n)$

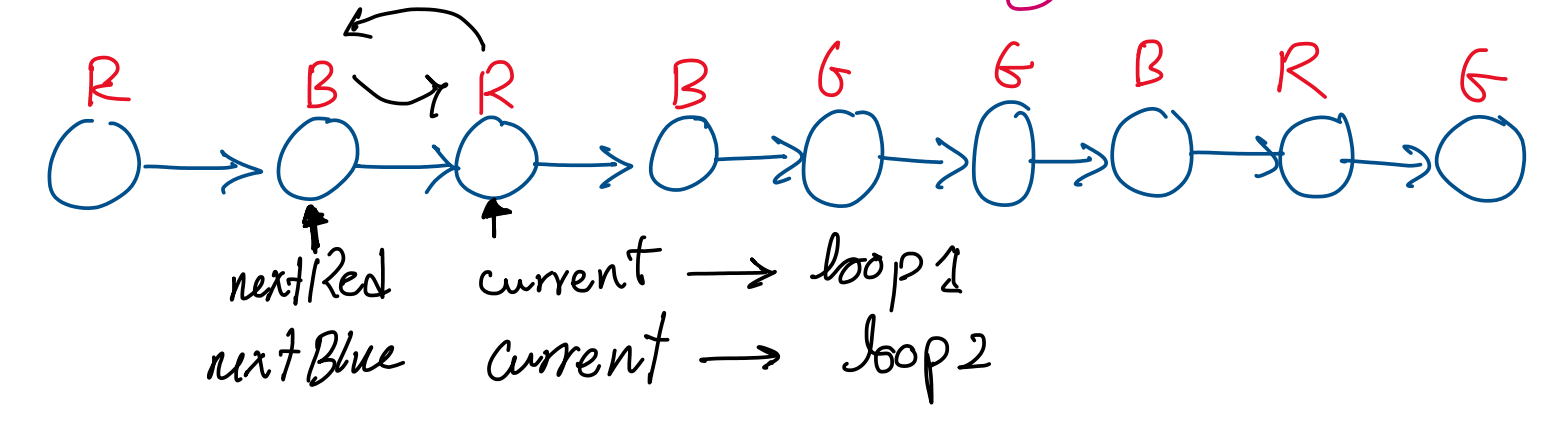
A. Let L be a **List** of objects colored either red or blue. Design an **in-place** algorithm **sortRB(L)** that places all red objects in list L before the blue colored objects. Thus the resulting List will have all the red objects followed by the blue objects. **Hint:** use the method swapElements to move the elements around in the List. **To receive full credit**, you must use Positions for traversal, e.g., first, last, after, before, swapElements, etc. which is necessary to make it in-place and efficient.



```
Algorithm Sort RB( L)
    if L.isEmpty() then
        return
    nextRed = L.first()
    while ! L.isLast(nextRed) ∧ nextRed.Color == 'Red' do
            nextRed = L.after(nextRed)
    Cur := nextRed
    while ! L.isLast( curr)   do
        Color = Color( Cur)
        if Color != 'Red'  then
            L.swapElement(cur, nextRed)
            nextRed := L. after(nextRed)      *?
        curr := L.after(Cur)                  *?
```

B. Let L be a **List** of objects colored either red, green, or blue. Design an **in-place** algorithm **sortRBG(L)** that places all red objects in list L before the blue colored objects, and all the blue objects before the green objects. Thus the resulting List will have all the red objects followed by the blue objects, followed by the green objects. **Hint:** use the method swapElements to move the elements around in the List. **To receive full credit**, you must use Positions for traversal, e.g., first, last, after, before, swapElements, etc. which is necessary to make it in-place.

red → blue → green

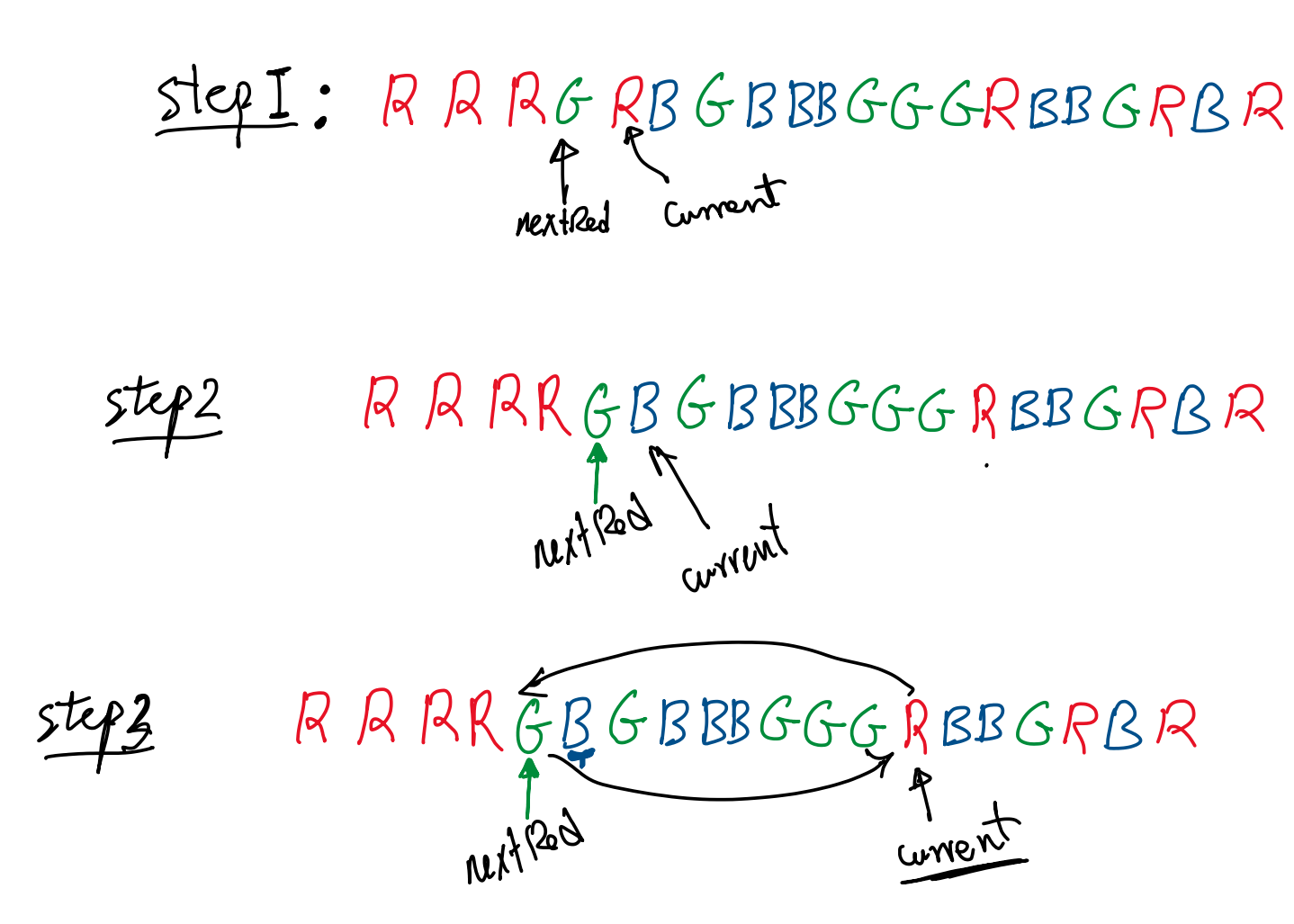


```
Algorithm sortRBG(L)

        nextRed := L.First( )
        while !L.isLast(nextRed) ∧ nextRed == "Red" then
            nextRed := L.after(nextRed)
        current := nextRed
        while ! L.isLast(Current)   do
            Color := Color(current)
            if( Color == 'Red')
                swapElements( current, nextRed)
                nextRed := L.after(nextRed)
        Current := L.after(current)
        nextBlue := nextRed          // initialize with last Red
        Current := nextBlue
        while !L.islast(Current)   do
            Color := Color(Current)
            if ( Color == 'Blue'
                swap Element( Current, nextBlue)
                nextBlue := L.after(nextBlue)
        current := L.after(Current)
```

move Red in the front

More the Blue after the Red

By then → All the Green will be at the end.


step I: R R R G R B G B B B G G G R B B G R B R
nextRed
Current
step2 R R R R G B G B B B G G G R B B G R B R
nextRed
current
step3 R R R R G B G B B B G G G R B B G R B R
nextRed
current