Assignement 5

R-4.5 Merging Sorted Sequences with Duplicates

```
MERGE-SORTED-SEQUENCES(A, B)
  i ← 1
  j ← 1
  k ← 1
  while i ≤ A.length and j ≤ B.length
    if A[i] < B[j]
      C[k] ← A[i]
      i ← i + 1
    else if A[i] > B[j]
      C[k] ← B[j]
      j ← j + 1
    else // A[i] = B[j]
      C[k] ← A[i]
      i ← i + 1
      j ← j + 1
    k ← k + 1

  // Copy remaining elements from A, if any
  while i ≤ A.length
    C[k] ← A[i]
    i ← i + 1
    k ← k + 1

  // Copy remaining elements from B, if any
  while j ≤ B.length
    C[k] ← B[j]
```

```
    j ← j + 1
    k ← k + 1

  return C
```

R-4.9 Modified Quick-Sort on Sorted Sequence

If we choose the middle element as the pivot in quick-sort, the algorithm will always split the sequence into two halves of equal size. In the worst case, when the sequence is already sorted, this leads to a quadratic time complexity of O(n^2).

C-4.10 Finding the Election Winner

```
FIND-ELECTION-WINNER(S)
  for i ← 1 to S.length
    if votes[S[i]] == 0
      votes[S[i]] ← 1
    else
      votes[S[i]] ← votes[S[i]] + 1

  max_votes ← 0
  winner ← 0
  for i ← 1 to S.length
    if votes[i] > max_votes
      max_votes ← votes[i]
      winner ← i

  return winner
```

A. In-Place Sorting of Red and Blue Objects

```
SORT-RB(A)
  red ← 1
  blue ← A.length
  while red < blue
```

```
    if A[red] == RED
      red ← red + 1
    else if A[blue] == BLUE
      blue ← blue - 1
    else
      SWAP(A[red], A[blue])
```

B. In-Place Sorting of Red, Green, and Blue Objects

```
SORT-RGB(A)
  red ← 1
  mid ← 1
  blue ← A.length

  while mid ≤ blue
    if A[mid] == RED
      SWAP(A[red], A[mid])
      red ← red + 1
      mid ← mid + 1
    else if A[mid] == BLUE
      SWAP(A[mid], A[blue])
      blue ← blue - 1
    else
      mid ← mid + 1
```

C.

```
inPlacePartition(S, lo, hi)
```

```
pivot_index ← RANDOM(lo, hi)
pivot ← S[pivot_index]

// Move pivot to the end
SWAP(S[pivot_index], S[hi])

// Two pointers: smaller and equal
smaller ← lo - 1
equal ← lo - 1

for i ← lo to hi - 1
  if S[i] < pivot
    smaller ← smaller + 1
    equal ← equal + 1
    SWAP(S[i], S[smaller])
    SWAP(S[smaller], S[equal])
  else if S[i] == pivot
    equal ← equal + 1
    SWAP(S[i], S[equal])

// Move pivot to its final position
SWAP(S[equal + 1], S[hi])

return (smaller + 1, equal + 1)
```

1. Pivot Selection: A random pivot is chosen and moved to the end of the segment.
2. Partitioning:

- smaller and equal pointers are initialized to lo - 1.
- The loop iterates from lo to hi - 1.
- If the current element is less than the pivot, it's swapped with the element at smaller and equal positions, effectively placing it in the first segment.
- If the current element is equal to the pivot, it's swapped with the element at the equal position, placing it in the second segment.

3. Pivot Placement: The pivot is moved to its final position, which is just after the equal pointer.
4. Return Indices: The function returns the indices p1 and p2, which mark the beginning and end of the second segment (elements equal to the pivot).

This modification to inPlaceQuickSort handles duplicate keys effectively and maintains the overall O(n log n) time complexity.