**Problem Set 2**
**Erdun E**
**September 16, 2024**
**Worked with: Fengkai Liu, Qingyuan Wan**

**CS 5800: Algorithm**
**Problem Set 2**

# Exercises

**(Goddard) A3, problem 3. Suppose we have a list of $n$ numbers. The list is guaranteed to have a number which appears more than $n/2$ times on it. Devise a good algorithm to find the Majority element.**

**Answer:** Given a number that appears more than half of the list. Sorting is a good algorithm to find out the majority element.

**Sorting:** Sort the list, because this number is more than half of the list, and return the element which at the middle index is the majority element.

**Algorithm Steps:**
1. Calculate the length of the list
2. Determining special cases (list.length = 0 or 1)
3. Sorting the list in ascending order
4. Return the element at the middle index which is the majority element, because it appears more than n/2.

**Time Complexity:** O(nlogn), sorting takes O(nlogn) time.

**Algorithm: 1 Sorting Psesudocode**

```
length = list.length()
if if length ≤ 1 then
    return list
end if
list.sort()
return list[length / 2]
```

**(Dasgupta) 2.14. You are given an array of $n$ elements, and you notice that some of the elements are duplicates; that is, they appear more than once in the array. Show how to remove all duplicates from the array in time $O(n \log n)$.**

**Answer:** Given remove all duplicates from the array in time O(n log n). Divide and Conquer could approach.

**Divide and Conquer:** First, the array is divided into two smaller subarrays to reduce the size of the problem; then, duplicates are removed from each subarray by recursively removing them until they are small enough to be processed directly; and finally, the final array is ensured to contain no duplicates by merging the processed subarrays.

**Algorithm Steps:**
1. Calculate the length of the array
2. Determining special cases (array.length = 0 or 1)
3. Splits the array into two subarrays and recursively processes each half.
4. Each subarray is processed recursively. When the size of the subarray is 1 or less than 1, there is no duplication.
5. Merges two sorted arrays while removing duplicates.

**Time Complexity:** O(nlogn). Because sorting takes O(nlogn) and merging takes O(n) across logn recursive levels, making sorting is the most time-consuming.

Psesudocode on the next page.

**Algorithm: 2 Sorting Psesudocode**

```
1: function removeDuplicates(array)
2:     return function divideAndConquer(array)
3: end function
4: function divideAndConquer(array)
5:     if if array.length() ≤ 1 then
6:         return array
7:     end if
8:     mid = array.length() / 2
9:     left = copyOfRange(array, 0, mid)
10:    right = copyOfRange(array, mid, array.length())
11:    left = function divideAndConquer(left)
12:    right = function divideAndConquer(right)
13:    left.sort()
14:    right.sort()
15:    return function mergeAndRemoveDuplicates(left, right)
16: end function
17: function mergeAndRemoveDuplicates(left, right)
18:    merged = new empty array
19:    i = 0
20:    j = 0
21:    while i < left.length() and j < right.length() do
22:        if left[i] < right[j] then
23:            if merged.length() == 0 or merged[merged.length() - 1] != left[i] then
24:                merged.append(left[i])
25:            end if
26:            i = i + 1
27:        else if left[i] > right[j] then
28:            if merged.length() == 0 or merged[merged.length() - 1] != right[i] then
29:                merged.append(right[i])
30:            end if
31:            j = j + 1
32:        else
33:            if merged.length() == 0 or merged[merged.length() - 1] != left[i] then
34:                merged.append(left[i])
35:            end if
36:            i = i + 1
37:            j = j + 1
38:        end if
39:    end while
40:    while i < left.length() do
41:        if merged.length() == 0 or merged[merged.length() - 1] != left[i] then
42:            merged.append(left[i])
43:        end if
44:        i = i + 1
45:    end while
46:    while j < right.length() do
47:        if merged.length() == 0 or merged[merged.length() - 1] != right[i] then
48:            merged.append(right[i])
49:        end if
50:        j = i + 1
51:    end while
52:    return mergerd
53: end function
```

**(Dasgupta) 2.15. In our median-finding algorithm (Section 2.4), a basic primitive is the** split **operation, which takes as input an array $S$ and a value $v$ and then divides $S$ into three sets: the elements less than $v$, the elements equal to $v$, and the elements greater than $v$. Show how to implement this** split **operation in place, that is, without allocating new memory.**

**Answer:** Given implement this split operation in place without allocating new memory. Split in-place algorithm is a good algorithm to implement this split operation.

**Split In-Place:** Set up three pointers. Point to the beginning of the array, the element being examined, and the end of the array. Iterate through the array, swapping values less than v to the front of v and values greater than v to the back of v.

**Algorithm Steps:**
1. Initial 3 pointers as low, mid, and high
2. Iterate the array until mid greater than high
3. If array[mid] less than v, swap array[mid] with array[low], pointers move
4. If array[mid] greater than v, swap array[mid] with array[high], pointer high moves, but mid doesn't because need to check the swapped value again.
5. For others, pointer mid moves due to equal to v

**Time Complexity:** O(n), each element being checked at most once, and n is the length of the array.

**Algorithm: 3 Split In-Place Psesudocode**

```
1: low = 0, mid = 0
2: high = S.length() - 1
3: while mid ≤ high do
4:     if S[mid] < v then
5:         swap low and mid
6:         low = low + 1
7:         mid = mid + 1
8:     else if S[mid] > v then
9:         swap mid and high
10:        high = high - 1
11:    else
12:        mid = mid + 1
13:    end if
14: end while
15: return S
```