Two Pointers, Sliding Window, Merge Intervals

Ben Johnson

Two Pointers

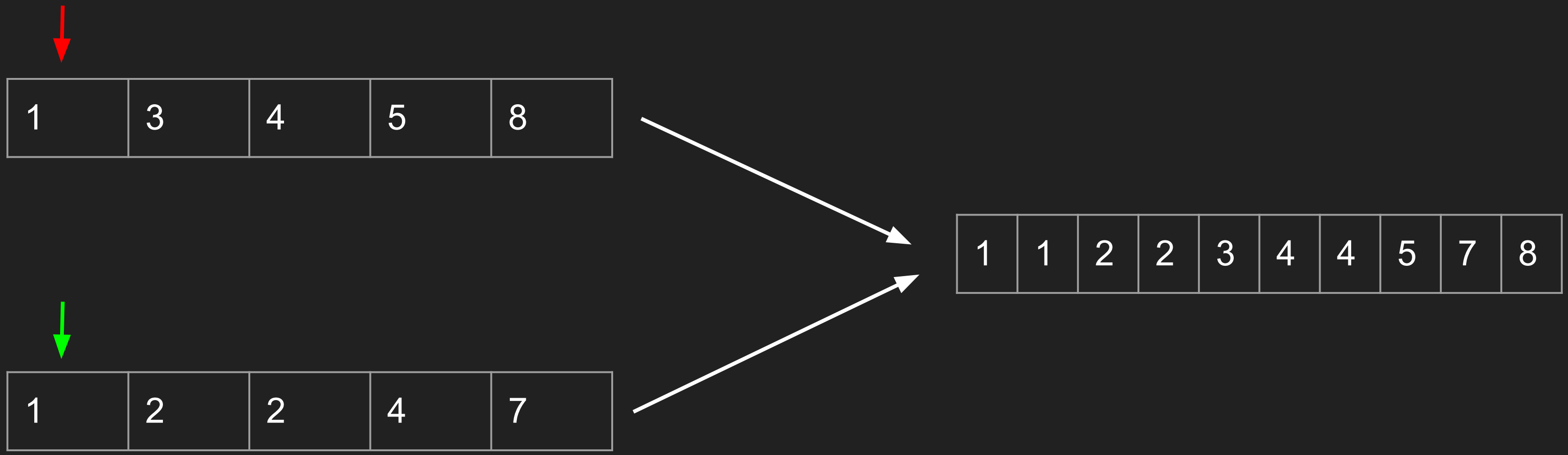
- Mostly useful for array/list data structures
- Maintain two pointers/indices, continuously update them to solve problem
- For example, slide pointers toward each other
- Usually, advance and/or use 1 of 2 pointers depending on some condition (e.g. array elements)

Two Pointers - Example 1 (Merge Sort)

- Merge two sorted arrays into one sorted array
- Maintain two pointers, one for each input array, initially at the beginning
- For each step, compare values of elements at each pointer, add smaller to output array, and increment that pointer
- Keep going until you reach end of both arrays
- Leetcode: https://leetcode.com/problems/merge-sorted-array/description/?envType=problem-list-v2&envId=two-pointers

Two Pointers - Example 1 (Merge Sort)

(Solve on whiteboard)



Two Pointers - Example 1 (Merge Sort)

```cpp
vector<int> merge_sort(const vector<int> &v1, const vector<int> &v2) {
   vector<int> result;
   int v1_pointer = 0, v2_pointer = 0;
   while (v1_pointer < v1.size() || v2_pointer < v2.size()) {
       if (v2_pointer >= v2.size() || (v1_pointer < v1.size() && v1[v1_pointer] < v2[v2_pointer])) {
           result.push_back(v1[v1_pointer++]);
       } else {
           result.push_back(v2[v2_pointer++]);
       }
   }

   return result;
}
```

Two Pointers - Example 2 (Find K Closest Elements)

- Given a sorted array and two integers k and x, return the k closest integers to x in the array (in sorted order)
- “Closest” meaning minimum absolute value of difference
- Break ties by choosing the smaller element
- Leetcode: https://leetcode.com/problems/find-k-closest-elements/description/?envType=problem-list-v2&envId=two-pointers

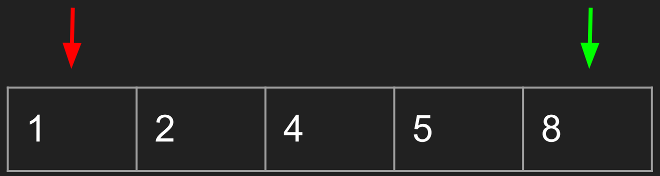
Two Pointers - Example 2 (Find K Closest Elements)

- **Solution:** Maintain two pointers (left and right of what will eventually be the k closest elements), starting at the start and end of the array
- While there are more than k elements between left and right pointer:
    - “Bring in” (i.e. increase left/decrease right) pointer with greater distance from x (left wins ties)
- Return the elements between left and right pointers

Two Pointers - Example 2 (Find K Closest Elements)

k = 2, x = 3

(Solve on whiteboard)



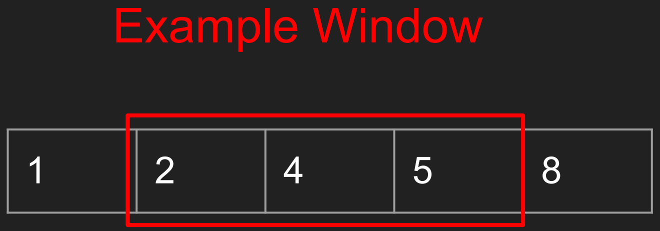
Two Pointers - Example 2 (Find K Closest Elements)

```
vector<int> findClosestElements(vector<int>& arr, int k, int x) {
   int left_pointer = 0, right_pointer = arr.size() - 1;
   while ((right_pointer - left_pointer) >= k) {
       int left_difference = abs(arr[left_pointer] - x);
       int right_difference = abs(arr[right_pointer] - x);
       if (left_difference <= right_difference) {
           right_pointer--;
       } else {
           left_pointer++;
       }
   }

   // Add all elements between left_pointer and right_pointer to the result
   vector<int> result(k);
   for (int i = 0; i < k; i++) {
       result[i] = arr[left_pointer + i];
   }
   return result;
}
```

Sliding Window

- Maintain a window (contiguous subarray) of an array
- Continuously update window (e.g. grow/shrink) to maintain an invariant or optimize a quantity
- Can be thought of as a specific type of two pointers algorithm where the pointers are start & end indices of a window



Sliding Window - Example 1 (Longest Repeating Character Replacement)

- Given a string s and an integer k, return the length of the longest substring you can get after perform the following operation at most k times:
  - Choose any character of the string and change it to any other character
- Input string consists only of uppercase letters
- 1 <= s.length <= 10^5
- Leetcode: https://leetcode.com/problems/longest-repeating-character-replacement/description/?envType=problem-list-v2&envId=sliding-window

Sliding Window - Example 1 (Longest Repeating Character Replacement)

- **Solution:** maintain a window of the longest possible substring satisfying these constraints
- Continuously slide the end index right by 1, then slide the start index right by 1 until we satisfy the constraint, then record window length (take max)
- Maintain an array of counts of each of 26 uppercase letters in window, update by incrementing/decrementing when sliding window
- Constraint translates to: window size <= highest count + k

Sliding Window - Example 1 (Longest Repeating Character Replacement)

(Solve on whiteboard)

Example input:

s = "AABABBA", k = 1

Sliding Window - Example 1 (Longest Repeating Character Replacement)

```cpp
int characterReplacement(string s, int k) {
    int n = s.size();
    int window_start = 0, window_end = 0;
    int result = 0;
    vector<int> char_counts(26, 0);
    char_counts[s[0] - 'A'] = 1;
    while (window_end < n) {
        int window_size = window_end - window_start + 1;

        int most_common_char_count = 0;
        for (int count : char_counts) {
            most_common_char_count = max(most_common_char_count, count);
        }
        if (most_common_char_count + k < window_size) {
            char_counts[s[window_start] - 'A']--;
            window_start++;
        } else {
            result = max(result, window_size);
            window_end++;
            if (window_end < n) {
                char_counts[s[window_end] - 'A']++;
            }
        }
    }
    return result;
}
```

Merge Intervals

- Given an array of intervals (each with a start and end index), merge all overlapping intervals return an array of merged, now non-overlapping intervals
- 1 <= intervals.length <= 10^4
- Leetcode: https://leetcode.com/problems/merge-intervals/description/

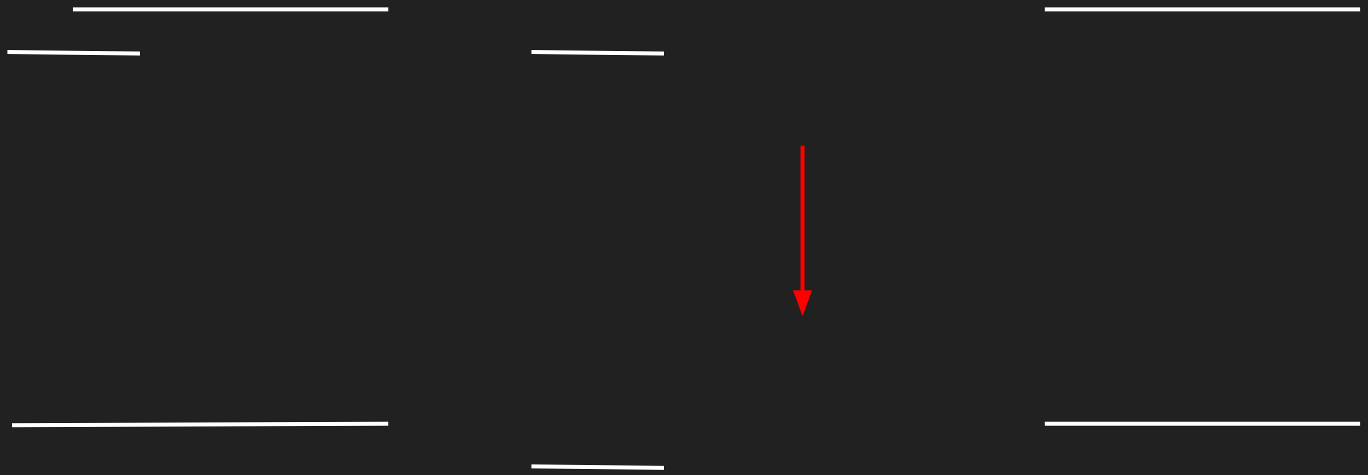
Merge Intervals

- **Solution**: Sort intervals by increasing end index. Add the first interval to the result, then iterate over the remaining intervals. For each interval:
  - Pop all overlapping intervals from the end of the result
  - Add the interval to the result; if we've popped any previous intervals, adjust start of interval to earliest start of itself & popped intervals
- **Runtime analysis**: O(n), where n is number of intervals, since each interval is pushed at most once and popped at most once (no need for binary search on earliest interval to remove)

Merge Intervals

Intervals = [[1,3],[2,6],[8,10],[15,18]]

(Solve on whiteboard)



Merge Intervals

```
vector<vector<int>> merge_intervals(vector<vector<int>>& intervals) {
   sort(intervals.begin(), intervals.end(), [&](auto &interval1, auto &interval2) {
       return interval1[1] < interval2[1] || (interval1[1] == interval2[1] && interval1[0] < interval2[0]);
   });

   vector<vector<int>> result = {intervals[0]};
   for (int i = 1; i < intervals.size(); i++) {
       auto &interval = intervals[i];

       int newIntervalStart = interval[0];
       while (result.size() > 0 && result[result.size() - 1][1] >= interval[0]) {
           newIntervalStart = min(newIntervalStart, result[result.size() - 1][0]);
           result.erase(result.begin() + result.size() - 1);
       }
       result.push_back({newIntervalStart, interval[1]});
   }
   return result;
}
```

Practice Contest

- https://vjudge.net/contest/667415