4	Median of Two Sorted Arrays	23.4%	Hard	
10	Regular Expression Matching	24.3%	Hard	
17	Letter Combinations of a Phone Number	37.1%	Medium	
20	Valid Parentheses	34.2%	Easy	
22	Generate Parentheses	48.8%	Medium	
23	Merge k Sorted Lists	28.9%	Hard	
31	Next Permutation	29.1%	Medium	
42	Trapping Rain Water	38.1%	Hard	
50	Pow(x, n)	26.2%	Medium	

54	Spiral Matrix	27.6%	Medium	
56	Merge Intervals	32.4%	Medium	
66	Plus One	39.9%	Easy	
128	Longest Consecutive Sequence	38.6%	Hard	
139	Word Break	31.9%	Medium	
140	Word Break II	24.8%	Hard	
146	LRU Cache	20.4%	Hard	
155	Min Stack	32.0%	Easy	
162	Find Peak Element	39.3%	Medium	

166	Fraction to Recurring Decimal	18.3%	Medium	
173	Binary Search Tree Iterator	44.0%	Medium	
200	Number of Islands	37.2%	Medium	
208	Implement Trie (Prefix Tree)	31.7%	Medium	
218	The Skyline Problem	29.4%	Hard	
224	Basic Calculator	29.1%	Hard	
228	Summary Ranges	32.7%	Medium	
231	Power of Two	40.9%	Easy	
240	Search a 2D Matrix II	39.1%	Medium	

253	Meeting Rooms II	39.7%	Medium	
279	Perfect Squares	38.0%	Medium	
280	Wiggle Sort	58.9%	Medium	
289	Game of Life	37.6%	Medium	
297	Serialize and Deserialize Binary Tree	35.5%	Hard	
312	Burst Balloons	43.9%	Hard	
315	Count of Smaller Numbers After Self	35.1%	Hard	
316	Remove Duplicate Letters	30.6%	Hard	
326	Power of Three	40.9%	Easy	

336	Palindrome Pairs	27.3%	Hard	
341	Flatten Nested List Iterator	43.6%	Medium	
345	Reverse Vowels of a String	39.5%	Easy	
374	Guess Number Higher or Lower	37.1%	Easy	
387	First Unique Character in a String	47.3%	Easy	
388	Longest Absolute File Path	37.6%	Medium	
406	Queue Reconstruction by Height	56.6%	Medium	
421	Maximum XOR of Two Numbers in an Array	48.2%	Medium	
448	Find All Numbers Disappeared in an Array	51.2%	Easy	

463	Island Perimeter	58.3%	Easy	
535	Encode and Decode TinyURL	74.0%	Medium	
657	Judge Route Circle	68.6%	Easy	
739	Daily Temperatures	53.3%	Medium	
760	Find Anagram Mappings	76.3%	Easy	

253. Meeting Rooms II

Given an array of meeting time intervals consisting of start and end times `[[s1,e1],[s2,e2],...]` ($s_i < e_i$), find the minimum number of conference rooms required.

Example 1:

```
Input: [[0, 30],[5, 10],[15, 20]]
Output: 2
```

Example 2:

```
Input: [[7,10],[2,4]]
Output: 1
```

```
/**
 * Definition for an interval.
 * public class Interval {
 *     int start;
 *     int end;
 *     Interval() { start = 0; end = 0; }
 *     Interval(int s, int e) { start = s; end = e; }
 * }
 */
class Solution {
    public int minMeetingRooms(Interval[] intervals) {

    }
}
```

C++ solution using a map. total 11 lines

Last Edit: April 9, 2018 1:15 AM

hsuyuanyuan
51

class Solution {
public:
int minMeetingRooms(vector& intervals) {
map<int, int> mp; // key: time; val: +1 if start, -1 if end

```
    for(int i=0; i< intervals.size(); i++) {
        mp[intervals[i].start] ++;
        mp[intervals[i].end] --;
```

```
    }

    int cnt = 0, maxCnt = 0;
    for(auto it = mp.begin(); it != mp.end(); it++) {
        cnt += it->second;
        maxCnt = max( cnt, maxCnt);
    }

    return maxCnt;
}
```

};

280. Wiggle Sort

Given an unsorted array `nums`, reorder it **in-place** such that `nums[0] <= nums[1] >= nums[2] <= nums[3]...`.

Example:

```
Input: nums = [3,5,2,1,6,4]
Output: One possible answer is [3,5,1,6,2,4]
class Solution {
    public void wiggleSort(int[] nums) {

    }
}
```

Solution

Approach #1 (Sorting) [Accepted]

The obvious solution is to just sort the array first, then swap elements pair-wise starting from the second element. For example:

```
   [1, 2, 3, 4, 5, 6]
       ↑   ↑   ↑   ↑
       swap   swap

=> [1, 3, 2, 5, 4, 6]
public void wiggleSort(int[] nums) {
    Arrays.sort(nums);
    for (int i = 1; i < nums.length - 1; i += 2) {
        swap(nums, i, i + 1);
    }
}

private void swap(int[] nums, int i, int j) {
    int temp = nums[i];
    nums[i] = nums[j];
    nums[j] = temp;
}
```

Complexity analysis

- Time complexity : O(n \log n)

  $O(n \log n)$. The entire algorithm is dominated by the sorting step, which costs O(n \log n)

  $O(n \log n)$ time to sort n

  n elements.

- Space complexity : O(1)

  $O(1)$. Space depends on the sorting implementation which, usually, costs O(1)

$O(1)$auxiliary space if `heapsort` is used.

Approach #2 (One-pass Swap) [Accepted]

Intuitively, we should be able to reorder it in one-pass. As we iterate through the array, we compare the current element to its next element and if the order is incorrect, we swap them.

```
public void wiggleSort(int[] nums) {
    boolean less = true;
    for (int i = 0; i < nums.length - 1; i++) {
        if (less) {
            if (nums[i] > nums[i + 1]) {
                swap(nums, i, i + 1);
            }
        } else {
            if (nums[i] < nums[i + 1]) {
                swap(nums, i, i + 1);
            }
        }
        less = !less;
    }
}
```

We could shorten the code further by compacting the condition to a single line. Also observe the boolean value of `less` actually depends on whether the index is even or odd.

```
public void wiggleSort(int[] nums) {
    for (int i = 0; i < nums.length - 1; i++) {
        if (((i % 2 == 0) && nums[i] > nums[i + 1])
                || ((i % 2 == 1) && nums[i] < nums[i +
1])) {
            swap(nums, i, i + 1);
        }
    }
}
```

Here is another amazing solution by @StefanPochmann who came up with originally here.

```
public void wiggleSort(int[] nums) {
    for (int i = 0; i < nums.length - 1; i++) {
        if ((i % 2 == 0) == (nums[i] > nums[i + 1])) {
            swap(nums, i, i + 1);
        }
    }
}
```

Complexity analysis

- Time complexity : $O(n)$

  $O(n)$. In the worst case we swap at most n \over 2

  2

  n

   times. An example input is `[2,1,3,1,4,1]`.

- Space complexity : $O(1)$

  $O(1)$.

760. Find Anagram Mappings

Given two lists `A`and `B`, and `B` is an anagram of `A`. `B` is an anagram of `A` means `B` is made by randomizing the order of the elements in `A`.

We want to find an *index mapping* `P`, from `A` to `B`. A mapping `P[i] = j` means the `i`th element in `A` appears in `B` at index `j`.

These lists `A` and `B` may contain duplicates. If there are multiple answers, output any of them.

For example, given

```
A = [12, 28, 46, 32, 50]
B = [50, 12, 32, 46, 28]
```

We should return

```
[1, 4, 3, 2, 0]
```

as `P[0] = 1` because the `0`th element of `A` appears at `B[1]`, and `P[1] = 4` because the `1`st element of `A` appears at `B[4]`, and so on.

Note:

1. `A, B` have equal lengths in range `[1, 100]`.
2. `A[i], B[i]` are integers in range `[0, 10^5]`.

```
class Solution {
    public int[] anagramMappings(int[] A, int[] B) {


    }
}
```

Approach #1: Hash Table [Accepted]

Intuition

Take the example `A = [12, 28, 46]`, `B = [46, 12, 28]`. We want to know where the `12` occurs in `B`, say at position `1`; then where the `28` occurs in `B`, which is position `2`; then where the `46` occurs in `B`, which is position `0`.

If we had a dictionary (hash table) `D = {46: 0, 12: 1, 28: 2}`, then this question could be handled easily.

Algorithm

Create the hash table `D` as described above. Then, the answer is a list of `D[A[i]]` for `i = 0, 1, ...`.

```
class Solution {
    public int[] anagramMappings(int[] A, int[] B) {
        Map<Integer, Integer> D = new HashMap();
        for (int i = 0; i < B.length; ++i)
            D.put(B[i], i);

        int[] ans = new int[A.length];
        int t = 0;
        for (int x: A)
            ans[t++] = D.get(x);
        return ans;
    }
}
```

```
class Solution(object):
    def anagramMappings(self, A, B):
        D = {x: i for i, x in enumerate(B)}
```

```
return [D[x] for x in A]
```

Complexity Analysis

- Time Complexity: O(N)

  $O(N)$, where N

  N is the length of A

  A.

- Space Complexity: O(N)

  $O(N)$.

Analysis written by: @awice.