

# Coding Interview in Java

Program Creek

# Contents

<b>1</b>	<b>Remove Duplicates from Sorted Array</b>	<b>14</b>
<b>2</b>	<b>Remove Duplicates from Sorted Array II</b>	<b>16</b>
<b>3</b>	<b>Remove Element</b>	<b>18</b>
<b>4</b>	<b>Move Zeroes</b>	<b>19</b>
<b>5</b>	<b>Candy</b>	<b>20</b>
<b>6</b>	<b>Trapping Rain Water</b>	<b>22</b>
<b>7</b>	<b>Product of Array Except Self</b>	<b>24</b>
<b>8</b>	<b>Minimum Size Subarray Sum</b>	<b>26</b>
<b>9</b>	<b>Summary Ranges</b>	<b>28</b>
<b>10</b>	<b>Missing Ranges</b>	<b>29</b>
<b>11</b>	<b>Merge Intervals</b>	<b>30</b>
<b>12</b>	<b>Insert Interval</b>	<b>31</b>
<b>13</b>	<b>Partition Labels</b>	<b>34</b>
<b>14</b>	<b>Find And Replace in String</b>	<b>36</b>
<b>15</b>	<b>One Edit Distance</b>	<b>37</b>
<b>16</b>	<b>Merge Sorted Array</b>	<b>38</b>
<b>17</b>	<b>Is Subsequence</b>	<b>40</b>
<b>18</b>	<b>Backspace String Compare</b>	<b>41</b>
<b>19</b>	<b>Repeated String Match</b>	<b>43</b>
<b>20</b>	<b>Container With Most Water</b>	<b>44</b>
<b>21</b>	<b>Reverse Vowels of a String</b>	<b>45</b>
<b>22</b>	<b>Valid Palindrome</b>	<b>46</b>
<b>23</b>	<b>Shortest Word Distance</b>	<b>47</b>
<b>24</b>	<b>Shortest Word Distance II</b>	<b>48</b>

<b>25</b>	<b>Shortest Word Distance III</b>	<b>50</b>
<b>26</b>	<b>Intersection of Two Arrays</b>	<b>52</b>
<b>27</b>	<b>Intersection of Two Arrays II</b>	<b>54</b>
<b>28</b>	<b>Two Sum II Input array is sorted</b>	<b>56</b>
<b>29</b>	<b>Two Sum III Data structure design</b>	<b>57</b>
<b>30</b>	<b>3Sum</b>	<b>58</b>
<b>31</b>	<b>4Sum</b>	<b>60</b>
<b>32</b>	<b>3Sum Closest</b>	<b>62</b>
<b>33</b>	<b>Wiggle Sort</b>	<b>63</b>
<b>34</b>	<b>Wiggle Subsequence</b>	<b>64</b>
<b>35</b>	<b>Longest Common Prefix</b>	<b>65</b>
<b>36</b>	<b>Next Permutation</b>	<b>66</b>
<b>37</b>	<b>Search Insert Position</b>	<b>68</b>
<b>38</b>	<b>Median of Two Sorted Arrays</b>	<b>70</b>
<b>39</b>	<b>Find Minimum in Rotated Sorted Array</b>	<b>72</b>
<b>40</b>	<b>Find Minimum in Rotated Sorted Array II</b>	<b>74</b>
<b>41</b>	<b>Find First and Last Position of Element in Sorted Array</b>	<b>76</b>
<b>42</b>	<b>Guess Number Higher or Lower</b>	<b>78</b>
<b>43</b>	<b>First Bad Version</b>	<b>80</b>
<b>44</b>	<b>Search in Rotated Sorted Array</b>	<b>82</b>
<b>45</b>	<b>Search in Rotated Sorted Array II</b>	<b>84</b>
<b>46</b>	<b>Longest Increasing Subsequence</b>	<b>85</b>
<b>47</b>	<b>Count of Smaller Numbers After Self</b>	<b>88</b>
<b>48</b>	<b>Russian Doll Envelopes</b>	<b>91</b>
<b>49</b>	<b>HIndex</b>	<b>93</b>
<b>50</b>	<b>HIndex II</b>	<b>95</b>
<b>51</b>	<b>Valid Anagram</b>	<b>96</b>
<b>52</b>	<b>Group Shifted Strings</b>	<b>98</b>
<b>53</b>	<b>Palindrome Pairs</b>	<b>100</b>
<b>54</b>	<b>Line Reflection</b>	<b>102</b>

<b>55 Isomorphic Strings</b>	<b>103</b>
<b>56 Two Sum</b>	<b>104</b>
<b>57 Maximum Size Subarray Sum Equals k</b>	<b>105</b>
<b>58 Subarray Sum Equals K</b>	<b>107</b>
<b>59 Maximum Subarray</b>	<b>108</b>
<b>60 Maximum Product Subarray</b>	<b>111</b>
<b>61 Longest Substring Without Repeating Characters</b>	<b>112</b>
<b>62 Longest Substring with At Most K Distinct Characters</b>	<b>114</b>
<b>63 Substring with Concatenation of All Words</b>	<b>116</b>
<b>64 Minimum Window Substring</b>	<b>118</b>
<b>65 Longest Substring with At Least K Repeating Characters</b>	<b>120</b>
<b>66 Permutation in String</b>	<b>122</b>
<b>67 Longest Consecutive Sequence</b>	<b>124</b>
<b>68 Majority Element</b>	<b>126</b>
<b>69 Majority Element II</b>	<b>128</b>
<b>70 Increasing Triplet Subsequence</b>	<b>129</b>
<b>71 Find the Second Largest Number in an Array</b>	<b>131</b>
<b>72 Word Ladder</b>	<b>132</b>
<b>73 Word Ladder II</b>	<b>134</b>
<b>74 Top K Frequent Elements</b>	<b>136</b>
<b>75 Meeting Rooms II</b>	<b>139</b>
<b>76 Meeting Rooms</b>	<b>141</b>
<b>77 Range Addition</b>	<b>142</b>
<b>78 Merge K Sorted Arrays in Java</b>	<b>144</b>
<b>79 Merge k Sorted Lists</b>	<b>146</b>
<b>80 Rearrange String k Distance Apart</b>	<b>147</b>
<b>81 Minimum Cost to Hire K Workers</b>	<b>149</b>
<b>82 Contains Duplicate</b>	<b>151</b>
<b>83 Contains Duplicate II</b>	<b>152</b>
<b>84 Contains Duplicate III</b>	<b>153</b>

<b>85</b>	<b>Max Sum of Rectangle No Larger Than K</b>	<b>155</b>
<b>86</b>	<b>Maximum Sum of Subarray Close to K</b>	<b>158</b>
<b>87</b>	<b>Sliding Window Maximum</b>	<b>160</b>
<b>88</b>	<b>Moving Average from Data Stream</b>	<b>162</b>
<b>89</b>	<b>Find Median from Data Stream</b>	<b>163</b>
<b>90</b>	<b>Data Stream as Disjoint Intervals</b>	<b>165</b>
<b>91</b>	<b>Linked List Random Node</b>	<b>167</b>
<b>92</b>	<b>Shuffle an Array</b>	<b>168</b>
<b>93</b>	<b>Sort List</b>	<b>170</b>
<b>94</b>	<b>Quicksort Array in Java</b>	<b>172</b>
<b>95</b>	<b>Kth Largest Element in an Array</b>	<b>175</b>
<b>96</b>	<b>Sort Colors</b>	<b>177</b>
<b>97</b>	<b>Maximum Gap</b>	<b>178</b>
<b>98</b>	<b>Group Anagrams</b>	<b>180</b>
<b>99</b>	<b>Ugly Number</b>	<b>181</b>
<b>100</b>	<b>Ugly Number II</b>	<b>182</b>
<b>101</b>	<b>Super Ugly Number</b>	<b>183</b>
<b>102</b>	<b>Find K Pairs with Smallest Sums</b>	<b>184</b>
<b>103</b>	<b>Rotate Array in Java</b>	<b>185</b>
<b>104</b>	<b>Reverse Words in a String II</b>	<b>188</b>
<b>105</b>	<b>Missing Number</b>	<b>189</b>
<b>106</b>	<b>Find the Duplicate Number</b>	<b>190</b>
<b>107</b>	<b>First Missing Positive</b>	<b>192</b>
<b>108</b>	<b>Queue Reconstruction by Height</b>	<b>194</b>
<b>109</b>	<b>Binary Watch</b>	<b>195</b>
<b>110</b>	<b>Search a 2D Matrix</b>	<b>198</b>
<b>111</b>	<b>Search a 2D Matrix II</b>	<b>199</b>
<b>112</b>	<b>Kth Smallest Element in a Sorted Matrix</b>	<b>201</b>
<b>113</b>	<b>Design Snake Game</b>	<b>203</b>
<b>114</b>	<b>Number of Islands II</b>	<b>205</b>

<b>115 Number of Connected Components in an Undirected Graph</b>	<b>207</b>
<b>116 Most Stones Removed with Same Row or Column</b>	<b>210</b>
<b>117 Longest Increasing Path in a Matrix</b>	<b>213</b>
<b>118 Word Search</b>	<b>216</b>
<b>119 Word Search II</b>	<b>219</b>
<b>120 Number of Islands</b>	<b>223</b>
<b>121 Find a Path in a Matrix</b>	<b>226</b>
<b>122 Sudoku Solver</b>	<b>228</b>
<b>123 Valid Sudoku</b>	<b>231</b>
<b>124 Walls and Gates</b>	<b>233</b>
<b>125 Surrounded Regions</b>	<b>236</b>
<b>126 Set Matrix Zeroes</b>	<b>240</b>
<b>127 Spiral Matrix</b>	<b>243</b>
<b>128 Spiral Matrix II</b>	<b>247</b>
<b>129 Rotate Image</b>	<b>249</b>
<b>130 Range Sum Query 2D Immutable</b>	<b>250</b>
<b>131 Shortest Distance from All Buildings</b>	<b>253</b>
<b>132 Best Meeting Point</b>	<b>255</b>
<b>133 Game of Life</b>	<b>256</b>
<b>134 TicTacToe</b>	<b>258</b>
<b>135 Sparse Matrix Multiplication</b>	<b>261</b>
<b>136 Add Two Numbers</b>	<b>263</b>
<b>137 Reorder List</b>	<b>265</b>
<b>138 Linked List Cycle</b>	<b>269</b>
<b>139 Copy List with Random Pointer</b>	<b>270</b>
<b>140 Merge Two Sorted Lists</b>	<b>272</b>
<b>141 Odd Even Linked List</b>	<b>274</b>
<b>142 Remove Duplicates from Sorted List</b>	<b>276</b>
<b>143 Remove Duplicates from Sorted List II</b>	<b>278</b>
<b>144 Partition List</b>	<b>279</b>

<b>145 Intersection of Two Linked Lists</b>	<b>280</b>
<b>146 Remove Linked List Elements</b>	<b>282</b>
<b>147 Swap Nodes in Pairs</b>	<b>283</b>
<b>148 Reverse Linked List</b>	<b>285</b>
<b>149 Reverse Linked List II</b>	<b>287</b>
<b>150 Reverse Double Linked List</b>	<b>289</b>
<b>151 Remove Nth Node From End of List</b>	<b>291</b>
<b>152 Palindrome Linked List</b>	<b>293</b>
<b>153 Delete Node in a Linked List</b>	<b>296</b>
<b>154 Reverse Nodes in kGroup</b>	<b>297</b>
<b>155 Plus One Linked List</b>	<b>300</b>
<b>156 Binary Tree Preorder Traversal</b>	<b>302</b>
<b>157 Binary Tree Inorder Traversal</b>	<b>303</b>
<b>158 Binary Tree Postorder Traversal</b>	<b>305</b>
<b>159 Binary Tree Level Order Traversal</b>	<b>308</b>
<b>160 Binary Tree Level Order Traversal II</b>	<b>310</b>
<b>161 Binary Tree Vertical Order Traversal</b>	<b>312</b>
<b>162 Invert Binary Tree</b>	<b>314</b>
<b>163 Kth Smallest Element in a BST</b>	<b>315</b>
<b>164 Binary Tree Longest Consecutive Sequence</b>	<b>317</b>
<b>165 Validate Binary Search Tree</b>	<b>319</b>
<b>166 Flatten Binary Tree to Linked List</b>	<b>321</b>
<b>167 Path Sum</b>	<b>323</b>
<b>168 Path Sum II</b>	<b>325</b>
<b>169 Construct Binary Tree from Inorder and Postorder Traversal</b>	<b>327</b>
<b>170 Construct Binary Tree from Preorder and Inorder Traversal</b>	<b>329</b>
<b>171 Convert Sorted Array to Binary Search Tree</b>	<b>331</b>
<b>172 Convert Sorted List to Binary Search Tree</b>	<b>332</b>
<b>173 Minimum Depth of Binary Tree</b>	<b>334</b>
<b>174 Binary Tree Maximum Path Sum</b>	<b>336</b>

<b>175 Balanced Binary Tree</b>	<b>337</b>
<b>176 Symmetric Tree</b>	<b>339</b>
<b>177 Binary Search Tree Iterator</b>	<b>340</b>
<b>178 Binary Tree Right Side View</b>	<b>342</b>
<b>179 Lowest Common Ancestor of a Binary Search Tree</b>	<b>344</b>
<b>180 Lowest Common Ancestor of a Binary Tree</b>	<b>345</b>
<b>181 Most Frequent Subtree Sum</b>	<b>347</b>
<b>182 Verify Preorder Serialization of a Binary Tree</b>	<b>349</b>
<b>183 Populating Next Right Pointers in Each Node</b>	<b>351</b>
<b>184 Populating Next Right Pointers in Each Node II</b>	<b>354</b>
<b>185 Unique Binary Search Trees</b>	<b>356</b>
<b>186 Unique Binary Search Trees II</b>	<b>358</b>
<b>187 Sum Root to Leaf Numbers</b>	<b>360</b>
<b>188 Count Complete Tree Nodes</b>	<b>362</b>
<b>189 Closest Binary Search Tree Value</b>	<b>365</b>
<b>190 Binary Tree Paths</b>	<b>367</b>
<b>191 Maximum Depth of Binary Tree</b>	<b>369</b>
<b>192 Recover Binary Search Tree</b>	<b>370</b>
<b>193 Same Tree</b>	<b>371</b>
<b>194 Serialize and Deserialize Binary Tree</b>	<b>372</b>
<b>195 Inorder Successor in BST</b>	<b>375</b>
<b>196 Inorder Successor in BST II</b>	<b>376</b>
<b>197 Find Leaves of Binary Tree</b>	<b>378</b>
<b>198 Largest BST Subtree</b>	<b>380</b>
<b>199 Implement Trie (Prefix Tree)</b>	<b>382</b>
<b>200 Add and Search Word Data structure design</b>	<b>386</b>
<b>201 Range Sum Query Mutable</b>	<b>390</b>
<b>202 The Skyline Problem</b>	<b>394</b>
<b>203 Implement Stack using Queues</b>	<b>396</b>
<b>204 Implement Queue using Stacks</b>	<b>398</b>



<b>205 Implement a Stack Using an Array in Java</b>	<b>399</b>
<b>206 Implement a Queue using an Array in Java</b>	<b>401</b>
<b>207 Evaluate Reverse Polish Notation</b>	<b>403</b>
<b>208 Valid Parentheses</b>	<b>406</b>
<b>209 Longest Valid Parentheses</b>	<b>407</b>
<b>210 Min Stack</b>	<b>408</b>
<b>211 Max Chunks To Make Sorted</b>	<b>410</b>
<b>212 Maximal Rectangle</b>	<b>411</b>
<b>213 Mini Parser</b>	<b>413</b>
<b>214 Flatten Nested List Iterator</b>	<b>415</b>
<b>215 Nested List Weight Sum</b>	<b>417</b>
<b>216 Nested List Weight Sum II</b>	<b>419</b>
<b>217 Decode String</b>	<b>421</b>
<b>218 Evaluate math expression with plus, minus and parentheses</b>	<b>423</b>
<b>219 Partition to K Equal Sum Subsets</b>	<b>425</b>
<b>220 Permutations</b>	<b>427</b>
<b>221 Permutations II</b>	<b>430</b>
<b>222 Permutation Sequence</b>	<b>432</b>
<b>223 Number of Squareful Arrays</b>	<b>434</b>
<b>224 Generate Parentheses</b>	<b>437</b>
<b>225 Combination Sum</b>	<b>439</b>
<b>226 Combination Sum II</b>	<b>441</b>
<b>227 Combination Sum III</b>	<b>442</b>
<b>228 Combination Sum IV</b>	<b>443</b>
<b>229 Wildcard Matching</b>	<b>444</b>
<b>230 Regular Expression Matching</b>	<b>445</b>
<b>231 Expressive Words</b>	<b>448</b>
<b>232 Get Target Number Using Number List and Arithmetic Operations</b>	<b>450</b>
<b>233 Flip Game</b>	<b>452</b>
<b>234 Flip Game II</b>	<b>453</b>

<b>235 Word Pattern</b>	<b>454</b>
<b>236 Word Pattern II</b>	<b>455</b>
<b>237 Scramble String</b>	<b>457</b>
<b>238 Remove Invalid Parentheses</b>	<b>458</b>
<b>239 Shortest Palindrome</b>	<b>460</b>
<b>240 Lexicographical Numbers</b>	<b>462</b>
<b>241 Combinations</b>	<b>464</b>
<b>242 Letter Combinations of a Phone Number</b>	<b>465</b>
<b>243 Restore IP Addresses</b>	<b>467</b>
<b>244 Factor Combinations</b>	<b>469</b>
<b>245 Subsets</b>	<b>470</b>
<b>246 Subsets II</b>	<b>472</b>
<b>247 Coin Change</b>	<b>474</b>
<b>248 Palindrome Partitioning</b>	<b>477</b>
<b>249 Palindrome Partitioning II</b>	<b>479</b>
<b>250 House Robber</b>	<b>480</b>
<b>251 House Robber II</b>	<b>482</b>
<b>252 House Robber III</b>	<b>483</b>
<b>253 Jump Game</b>	<b>484</b>
<b>254 Jump Game II</b>	<b>486</b>
<b>255 Best Time to Buy and Sell Stock</b>	<b>487</b>
<b>256 Best Time to Buy and Sell Stock II</b>	<b>488</b>
<b>257 Best Time to Buy and Sell Stock III</b>	<b>489</b>
<b>258 Best Time to Buy and Sell Stock IV</b>	<b>491</b>
<b>259 Dungeon Game</b>	<b>493</b>
<b>260 Decode Ways</b>	<b>494</b>
<b>261 Perfect Squares</b>	<b>495</b>
<b>262 Word Break</b>	<b>496</b>
<b>263 Word Break II</b>	<b>499</b>
<b>264 Minimum Window Subsequence</b>	<b>502</b>

<b>265 Maximal Square</b>	<b>503</b>
<b>266 Minimum Path Sum</b>	<b>505</b>
<b>267 Unique Paths</b>	<b>507</b>
<b>268 Unique Paths II</b>	<b>509</b>
<b>269 Paint House</b>	<b>511</b>
<b>270 Paint House II</b>	<b>513</b>
<b>271 Edit Distance in Java</b>	<b>515</b>
<b>272 Distinct Subsequences Total</b>	<b>518</b>
<b>273 Longest Palindromic Substring</b>	<b>520</b>
<b>274 Longest Common Subsequence</b>	<b>522</b>
<b>275 Longest Common Substring</b>	<b>524</b>
<b>276 LRU Cache</b>	<b>526</b>
<b>277 Insert Delete GetRandom O(1)</b>	<b>529</b>
<b>278 Insert Delete GetRandom O(1) Duplicates allowed</b>	<b>531</b>
<b>279 Design a Data Structure with Insert, Delete and GetMostFrequent of O(1)</b>	<b>533</b>
<b>280 Design Phone Directory</b>	<b>536</b>
<b>281 Design Twitter</b>	<b>537</b>
<b>282 Single Number</b>	<b>540</b>
<b>283 Single Number II</b>	<b>541</b>
<b>284 Twitter Codility Problem Max Binary Gap</b>	<b>542</b>
<b>285 Number of 1 Bits</b>	<b>544</b>
<b>286 Reverse Bits</b>	<b>545</b>
<b>287 Repeated DNA Sequences</b>	<b>546</b>
<b>288 Bitwise AND of Numbers Range</b>	<b>548</b>
<b>289 Sum of Two Integers</b>	<b>549</b>
<b>290 Counting Bits</b>	<b>550</b>
<b>291 Maximum Product of Word Lengths</b>	<b>552</b>
<b>292 Gray Code</b>	<b>553</b>
<b>293 UTF8 Validation</b>	<b>554</b>
<b>294 Course Schedule</b>	<b>555</b>

<b>295 Course Schedule II</b>	558
<b>296 Minimum Height Trees</b>	560
<b>297 Graph Valid Tree</b>	562
<b>298 Clone Graph</b>	564
<b>299 Reconstruct Itinerary</b>	567
<b>300 Pow(x, n)</b>	568

Every title in the PDF is linked back to the original blog. When it is clicked, it opens the original post in your browser. If you want to discuss any problem, please go to the post and leave your comment there.

I'm not an expert and some solutions may not be optimal. So please leave your comment if you see any problem or have a better solution. I will reply your comment as soon as I can.

This collection is updated from time to time. Please check out this link for the latest version: <http://www.programcreek.com/2014/05/10-algorithms-for-coding-interview/>

# 1 Remove Duplicates from Sorted Array

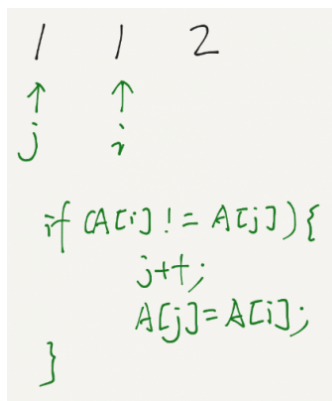
Given a sorted array, remove the duplicates in place such that each element appear only once and return the new length. Do not allocate extra space for another array, you must do this in place with constant memory.

For example, given input array A = [1,1,2], your function should return length = 2, and A is now [1,2].

## 1.1 Analysis

The problem is pretty straightforward. It returns the length of the array with unique elements, but the original array need to be changed also. This problem is similar to [Remove Duplicates from Sorted Array II](#).

## 1.2 Java Solution



The diagram shows an array with elements 1, 1, 2. Below the first two '1's are green arrows pointing to indices 'j' and 'i' respectively. Below this, the following code snippet is written in green:

```
if (A[i] != A[j]) {  
    j++;  
    A[j] = A[i];  
}
```

---

```
public static int removeDuplicates(int[] A) {  
    if (A.length < 2)  
        return A.length;  
  
    int j = 0;  
    int i = 1;  
  
    while (i < A.length) {  
        if (A[i] != A[j]) {  
            j++;  
            A[j] = A[i];  
        }  
  
        i++;  
    }  
  
    return j + 1;  
}
```

---

Note that we only care about the first unique part of the original array. So it is ok if input array is 1, 2, 2, 3, 3, the array is changed to 1, 2, 3, 3, 3.

## 2 Remove Duplicates from Sorted Array II

Follow up for "Remove Duplicates": What if duplicates are allowed at most twice?

For example, given sorted array A = [1,1,1,2,2,3], your function should return length = 5, and A is now [1,1,2,2,3]. So this problem also requires in-place array manipulation.

### 2.1 Java Solution 1

We can not change the given array's size, so we only change the first k elements of the array which has duplicates removed.

---

```
public int removeDuplicates(int[] nums) {
    if(nums==null){
        return 0;
    }
    if(nums.length<3){
        return nums.length;
    }

    int i=0;
    int j=1;
    /*

        i, j  1 1 1 2 2 3
    step1 0 1   i j
    step2 1 2   i j
    step3 1 3   i j
    step4 2 4   i j

    */
    while(j<nums.length){
        if(nums[j]==nums[i]){
            if(i==0){
                i++;
                j++;
            }else if(nums[i]==nums[i-1]){
                j++;
            }else{
                i++;
                nums[i]=nums[j];
                j++;
            }
        }else{
            i++;
            nums[i]=nums[j];
            j++;
        }
    }

    return i+1;
}
```

---



The problem with this solution is that there are 4 cases to handle. If we shift our two points to right by 1 element, the solution can be simplified as the Solution 2.

## 2.2 Java Solution 2

---

```
public int removeDuplicates(int[] nums) {
    if(nums==null){
        return 0;
    }

    if (nums.length <= 2){
        return nums.length;
    }
    /*
    1,1,1,2,2,3
    i j
    */
    int i = 1; // point to previous
    int j = 2; // point to current

    while (j < nums.length) {
        if (nums[j] == nums[i] && nums[j] == nums[i - 1]) {
            j++;
        } else {
            i++;
            nums[i] = nums[j];
            j++;
        }
    }

    return i + 1;
}
```

---

## 3 Remove Element

Given an array and a value, remove all instances of that value in place and return the new length. (Note: The order of elements can be changed. It doesn't matter what you leave beyond the new length.)

### 3.1 Java Solution

This problem can be solve by using two indices.

---

```
public int removeElement(int[] A, int elem) {
    int i=0;
    int j=0;

    while(j < A.length){
        if(A[j] != elem){
            A[i] = A[j];
            i++;
        }

        j++;
    }

    return i;
}
```

---

## 4 Move Zeroes

Given an array `nums`, write a function to move all 0's to the end of it while maintaining the relative order of the non-zero elements.

For example, given `nums = [0, 1, 0, 3, 12]`, after calling your function, `nums` should be `[1, 3, 12, 0, 0]`.

### 4.1 Java Solution 2

We can use the similar code that is used to solve [Remove Duplicates from Sorted Array I, II, Remove Element](#).

---

```
public void moveZeroes(int[] nums) {
    int i=0;
    int j=0;

    while(j<nums.length){
        if(nums[j]==0){
            j++;
        }else{
            nums[i]=nums[j];
            i++;
            j++;
        }
    }

    while(i<nums.length){
        nums[i]=0;
        i++;
    }
}
```

---

## 5 Candy

There are N children standing in a line. Each child is assigned a rating value. You are giving candies to these children subjected to the following requirements:

1. Each child must have at least one candy.
2. Children with a higher rating get more candies than their neighbors.

What is the minimum candies you must give?

### 5.1 Analysis

This problem can be solved in  $O(n)$  time.

We can always assign a neighbor with 1 more if the neighbor has higher a rating value. However, to get the minimum total number, we should always start adding 1s in the ascending order. We can solve this problem by scanning the array from both sides. First, scan the array from left to right, and assign values for all the ascending pairs. Then scan from right to left and assign values to descending pairs.

This problem is similar to [Trapping Rain Water](#).

### 5.2 Java Solution

---

```
public int candy(int[] ratings) {
    if (ratings == null || ratings.length == 0) {
        return 0;
    }

    int[] candies = new int[ratings.length];
    candies[0] = 1;

    //from let to right
    for (int i = 1; i < ratings.length; i++) {
        if (ratings[i] > ratings[i - 1]) {
            candies[i] = candies[i - 1] + 1;
        } else {
            // if not ascending, assign 1
            candies[i] = 1;
        }
    }

    int result = candies[ratings.length - 1];

    //from right to left
    for (int i = ratings.length - 2; i >= 0; i--) {
        int cur = 1;
        if (ratings[i] > ratings[i + 1]) {
            cur = candies[i + 1] + 1;
        }

        result += Math.max(cur, candies[i]);
        candies[i] = cur;
    }
}
```

```
    return result;  
}
```

---

## 6 Trapping Rain Water

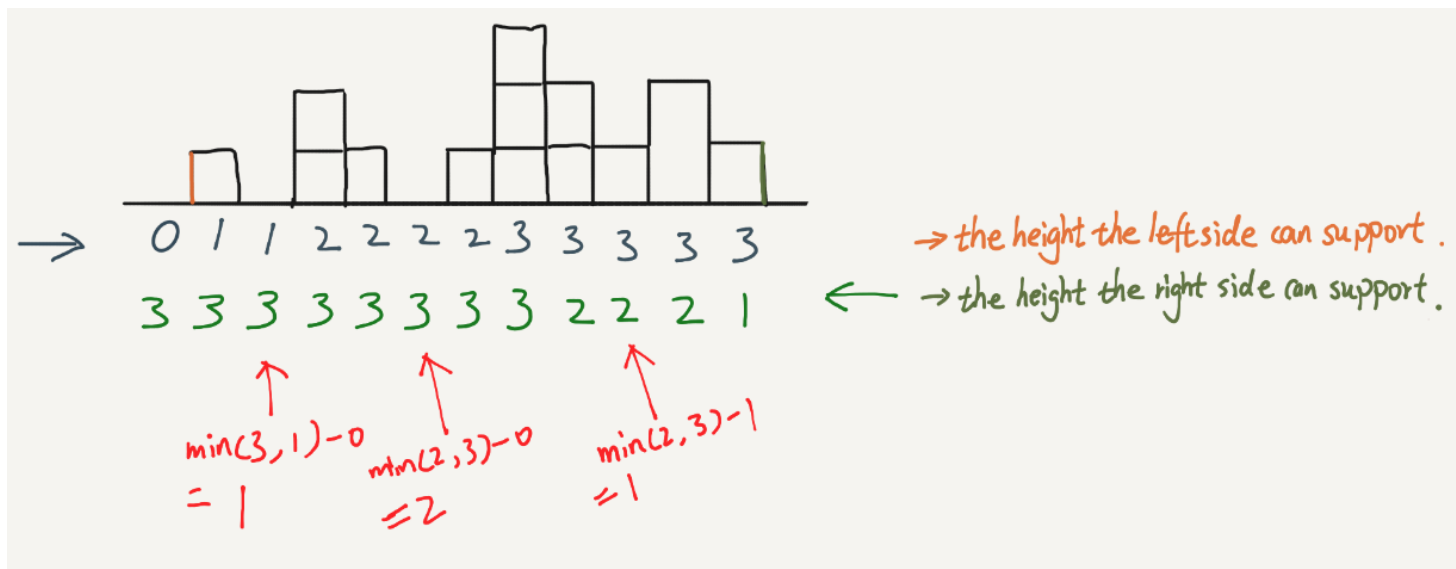
Given  $n$  non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining.

For example, given  $[0,1,0,2,1,0,1,3,2,1,2,1]$ , return 6.

### 6.1 Analysis

This problem is similar to [Candy](#). It can be solve by scanning from both sides and then get the total.

### 6.2 Java Solution



```
public int trap(int[] height) {  
    int result = 0;  
  
    if(height==null || height.length<=2)  
        return result;  
  
    int left[] = new int[height.length];  
    int right[] = new int[height.length];  
  
    //scan from left to right  
    int max = height[0];  
    left[0] = height[0];  
    for(int i=1; i<height.length; i++){  
        if(height[i]<max){  
            left[i]=max;  
        }  
    }  
}
```

```
    }else{
        left[i]=height[i];
        max = height[i];
    }
}

//scan from right to left
max = height[height.length-1];
right[height.length-1]=height[height.length-1];
for(int i=height.length-2; i>=0; i--){
    if(height[i]<max){
        right[i]=max;
    }else{
        right[i]=height[i];
        max = height[i];
    }
}

//calculate totoal
for(int i=0; i<height.length; i++){
    result+= Math.min(left[i],right[i])-height[i];
}

return result;
}
```

---

## 7 Product of Array Except Self

Given an array of  $n$  integers where  $n > 1$ , `nums`, return an array `output` such that `output[i]` is equal to the product of all the elements of `nums` except `nums[i]`.

Solve it without division and in  $O(n)$ .

For example, given `[1,2,3,4]`, return `[24,12,8,6]`.

### 7.1 Java Solution 1

---

```
public int[] productExceptSelf(int[] nums) {
    int[] result = new int[nums.length];

    int[] t1 = new int[nums.length];
    int[] t2 = new int[nums.length];

    t1[0]=1;
    t2[nums.length-1]=1;

    //scan from left to right
    for(int i=0; i<nums.length-1; i++){
        t1[i+1] = nums[i] * t1[i];
    }

    //scan from right to left
    for(int i=nums.length-1; i>0; i--){
        t2[i-1] = t2[i] * nums[i];
    }

    //multiply
    for(int i=0; i<nums.length; i++){
        result[i] = t1[i] * t2[i];
    }

    return result;
}
```

---

### 7.2 Java Solution 2

We can directly put the product values into the final result array. This saves the extra space to store the 2 intermediate arrays in Solution 1.

---

```
public int[] productExceptSelf(int[] nums) {
    int[] result = new int[nums.length];

    result[nums.length-1]=1;
    for(int i=nums.length-2; i>=0; i--){
        result[i]=result[i+1]*nums[i+1];
    }
}
```

---



```
int left=1;
for(int i=0; i<nums.length; i++){
    result[i]=result[i]*left;
    left = left*nums[i];
}

return result;
}
```

---

## 8 Minimum Size Subarray Sum

Given an array of  $n$  positive integers and a positive integer  $s$ , find the minimal length of a subarray of which the sum  $\geq s$ . If there isn't one, return 0 instead.

For example, given the array  $[2,3,1,2,4,3]$  and  $s = 7$ , the subarray  $[4,3]$  has the minimal length of 2 under the problem constraint.

### 8.1 Analysis

We can use 2 points to mark the left and right boundaries of the sliding window. When the sum is greater than the target, shift the left pointer; when the sum is less than the target, shift the right pointer.

### 8.2 Java Solution - two pointers

A simple sliding window solution.

---

```
public int minSubArrayLen(int s, int[] nums) {
    if(nums==null || nums.length==1)
        return 0;

    int result = nums.length;

    int start=0;
    int sum=0;
    int i=0;
    boolean exists = false;

    while(i<=nums.length){
        if(sum>=s){
            exists=true; //mark if there exists such a subarray
            if(start==i-1){
                return 1;
            }

            result = Math.min(result, i-start);
            sum=sum-nums[start];
            start++;

        }else{
            if(i==nums.length)
                break;
            sum = sum+nums[i];
            i++;
        }
    }

    if(exists)
        return result;
    else
        return 0;
}
```

---

```
}
```

---

Similarly, we can also write it in a more readable way.

---

```
public int minSubArrayLen(int s, int[] nums) {
    if(nums==null||nums.length==0)
        return 0;

    int i=0;
    int j=0;
    int sum=0;

    int minLen = Integer.MAX_VALUE;

    while(j<nums.length){
        if(sum<s){
            sum += nums[j];
            j++;
        }else{
            minLen = Math.min(minLen, j-i);
            if(i==j-1)
                return 1;

            sum -=nums[i];
            i++;
        }
    }

    while(sum>=s){
        minLen = Math.min(minLen, j-i);

        sum -=nums[i++];
    }

    return minLen==Integer.MAX_VALUE? 0: minLen;
}
```

---

## 9 Summary Ranges

Given a sorted integer array without duplicates, return the summary of its ranges for consecutive numbers.  
For example, given [0,1,2,4,5,7], return ["0->2","4->5","7"].

### 9.1 Analysis

When iterating over the array, two values need to be tracked: 1) the first value of a new range and 2) the previous value in the range.

### 9.2 Java Solution

---

```
public List<String> summaryRanges(int[] nums) {
    List<String> result = new ArrayList<String>();

    if(nums == null || nums.length==0)
        return result;

    if(nums.length==1){
        result.add(nums[0]+"");
    }

    int pre = nums[0]; // previous element
    int first = pre; // first element of each range

    for(int i=1; i<nums.length; i++){
        if(nums[i]==pre+1){
            if(i==nums.length-1){
                result.add(first+"->"+nums[i]);
            }
        }else{
            if(first == pre){
                result.add(first+"");
            }else{
                result.add(first + "->"+pre);
            }

            if(i==nums.length-1){
                result.add(nums[i]+"");
            }

            first = nums[i];
        }

        pre = nums[i];
    }

    return result;
}
```

---

## 10 Missing Ranges

Given a sorted integer array `nums`, where the range of elements are in the inclusive range `[lower, upper]`, return its missing ranges.

Example:

Input: `nums = [0, 1, 3, 50, 75]`, `lower = 0` and `upper = 99`, Output: `["2", "4->49", "51->74", "76->99"]`

### 10.1 Java Solution

---

```
public List<String> findMissingRanges(int[] nums, int lower, int upper) {
    List<String> result = new ArrayList<>();
    int start = lower;

    if(lower==Integer.MAX_VALUE){
        return result;
    }

    for(int i=0; i<nums.length; i++){
        //handle duplicates, e.g., [1,1,1] lower=1 upper=1
        if(i<nums.length-1 && nums[i]==nums[i+1]){
            continue;
        }

        if(nums[i] == start){
            start++;
        }else{
            result.add(getRange(start, nums[i]-1));
            if(nums[i]==Integer.MAX_VALUE){
                return result;
            }
            start = nums[i]+1;
        }
    }

    if(start<=upper){
        result.add(getRange(start, upper));
    }

    return result;
}

private String getRange(int n1, int n2) {
    return n1 == n2 ? String.valueOf(n1) : String.format("%d->%d" , n1, n2);
}
```

---

## 11 Merge Intervals

Given a collection of intervals, merge all overlapping intervals.

For example, Given [1,3],[2,6],[8,10],[15,18], return [1,6],[8,10],[15,18].

### 11.1 Analysis

The key to solve this problem is defining a Comparator first to sort the arraylist of Intervals.

### 11.2 Java Solution

---

```
public List<Interval> merge(List<Interval> intervals) {
    if(intervals == null || intervals.size()<=1){
        return intervals;
    }

    Collections.sort(intervals, Comparator.comparing((Interval itl)->itl.start));

    List<Interval> result = new ArrayList<>();
    Interval t = intervals.get(0);

    for(int i=1; i<intervals.size(); i++){
        Interval c = intervals.get(i);
        if(c.start <= t.end){
            t.end = Math.max(t.end, c.end);
        }else{
            result.add(t);
            t = c;
        }
    }

    result.add(t);

    return result;
}
```

---

## 12 Insert Interval

Problem:

*Given a set of non-overlapping & sorted intervals, insert a new interval into the intervals (merge if necessary).*

---

Example 1:

Given intervals [1,3],[6,9], insert and merge [2,5] in as [1,5],[6,9].

Example 2:

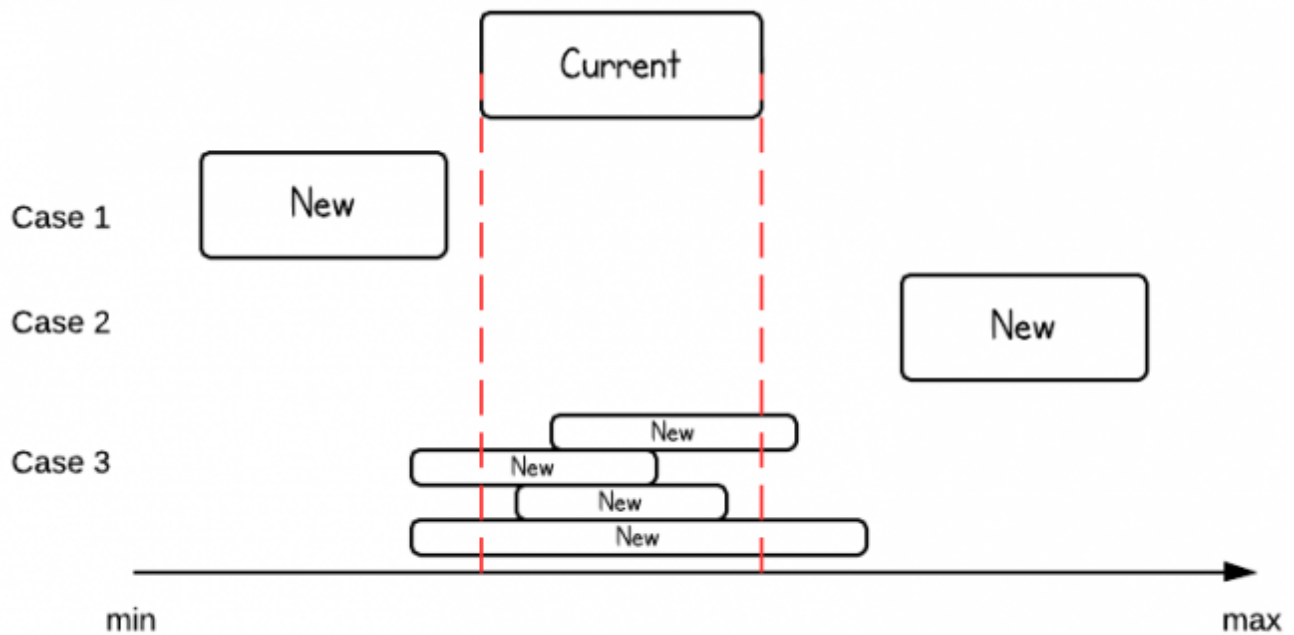
Given [1,2],[3,5],[6,7],[8,10],[12,16], insert and merge [4,9] in as [1,2],[3,10],[12,16].

This is because the **new** interval [4,9] overlaps with [3,5],[6,7],[8,10].

---

### 12.1 Java Solution 1

When iterating over the list, there are three cases for the current range.



---

```
/**
 * 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; }
```

```

* }
*/
public class Solution {
    public ArrayList<Interval> insert(ArrayList<Interval> intervals, Interval newInterval) {

        ArrayList<Interval> result = new ArrayList<Interval>();

        for(Interval interval: intervals){
            if(interval.end < newInterval.start){
                result.add(interval);
            }else if(interval.start > newInterval.end){
                result.add(newInterval);
                newInterval = interval;
            }else if(interval.end >= newInterval.start || interval.start <= newInterval.end){
                newInterval = new Interval(Math.min(interval.start, newInterval.start),
                    Math.max(newInterval.end, interval.end));
            }
        }

        result.add(newInterval);

        return result;
    }
}

```

---

## 12.2 Java Solution 2 - Binary Search

If the intervals list is an ArrayList, we can use binary search to make the best search time complexity  $O(\log(n))$ . However, the worst time is bounded by shifting the array list if a new range needs to be inserted. So time complexity is still  $O(n)$ .

```

public List<Interval> insert(List<Interval> intervals, Interval newInterval) {
    List<Interval> result = new ArrayList<>();

    if (intervals.size() == 0) {
        result.add(newInterval);
        return result;
    }

    int p = helper(intervals, newInterval);
    result.addAll(intervals.subList(0, p));

    for (int i = p; i < intervals.size(); i++) {
        Interval interval = intervals.get(i);
        if (interval.end < newInterval.start) {
            result.add(interval);
        } else if (interval.start > newInterval.end) {
            result.add(newInterval);
            newInterval = interval;
        } else if (interval.end >= newInterval.start || interval.start <= newInterval.end) {
            newInterval = new Interval(Math.min(interval.start, newInterval.start),
                Math.max(newInterval.end, interval.end));
        }
    }

    result.add(newInterval);
}

```



```
        return result;
    }

    public int helper(List<Interval> intervals, Interval newInterval) {
        int low = 0;
        int high = intervals.size() - 1;

        while (low < high) {
            int mid = low + (high - low) / 2;

            if (newInterval.start <= intervals.get(mid).start) {
                high = mid;
            } else {
                low = mid + 1;
            }
        }

        return high == 0 ? 0 : high - 1;
    }
}
```

---

The best time is  $O(\log(n))$  and worst case time is  $O(n)$ .

## 13 Partition Labels

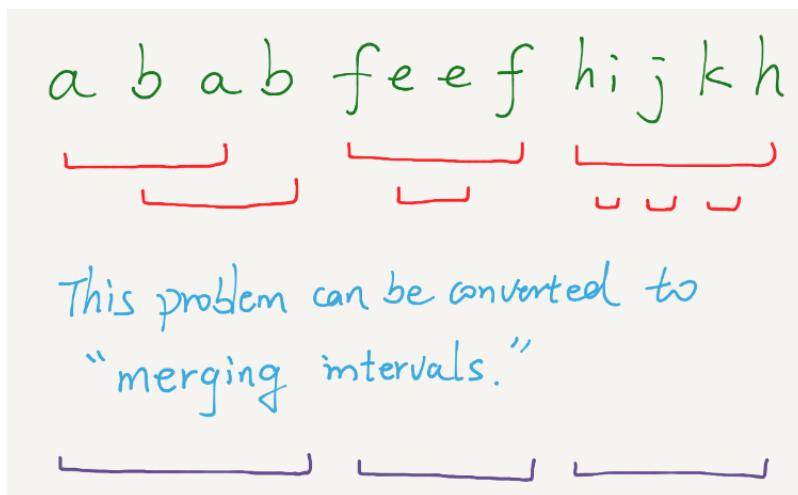
A string *S* of lowercase letters is given. We want to partition this string into as many parts as possible so that each letter appears in at most one part, and return a list of integers representing the size of these parts.

For example:

Input: *S* = "ababfeefhijkh" Output: [4,4,5]

Explanation: The partition is "abab", "feef", "hijkh". This is a partition so that each letter appears in at most one part.

### 13.1 Java Solution



---

```
public List<Integer> partitionLabels(String S) {
    ArrayList<Integer> result = new ArrayList<>();

    HashMap<Character, int[]> map = new HashMap<>();
    for(int i=0; i<S.length(); i++){
        char c = S.charAt(i);
        int[] arr = map.get(c);

        if(arr == null){
            arr = new int[]{i, i};
            map.put(c, arr);
        }else{
            arr[1]=i;
        }
    }

    ArrayList<int[]> list = new ArrayList<>();
    list.addAll(map.values());

    Collections.sort(list, Comparator.comparing((int[] arr) -> arr[0]));
```

```
int[] t = list.get(0);
for(int i=1; i<list.size(); i++){
    int[] range = list.get(i);

    if(range[1]<=t[1]){
        continue;
    }else if(range[0]>t[1]){ //impossible be equal
        result.add(t[1]-t[0]+1);
        t = range;
    }else{
        t[1] = range[1];
    }
}

result.add(t[1]-t[0]+1);

return result;
}
```

---

## 14 Find And Replace in String

To some string *S*, we will perform some replacement operations that replace groups of letters with new ones (not necessarily the same size).

Each replacement operation has 3 parameters: a starting index *i*, a source word *x* and a target word *y*. The rule is that if *x* starts at position *i* in the original string *S*, then we will replace that occurrence of *x* with *y*. If not, we do nothing.

For example, if we have *S* = "abcd" and we have some replacement operation *i* = 2, *x* = "cd", *y* = "ffff", then because "cd" starts at position 2 in the original string *S*, we will replace it with "ffff".

### 14.1 Java Solution

---

```
public String findReplaceString(String S, int[] indexes, String[] sources, String[] targets) {
    StringBuilder sb = new StringBuilder();
    TreeMap<Integer, String[]> map = new TreeMap<>();
    for (int i = 0; i < indexes.length; i++) {
        map.put(indexes[i], new String[]{sources[i], targets[i]});
    }

    int prev = 0;

    for (Map.Entry<Integer, String[]> entry : map.entrySet()) {
        int startIndex = entry.getKey();
        int endIndex = startIndex + entry.getValue()[0].length();

        if (prev != startIndex) {
            sb.append(S.substring(prev, startIndex));
        }

        String org = S.substring(startIndex, endIndex);
        if (org.equals(entry.getValue()[0])) {
            sb.append(entry.getValue()[1]);
            prev = endIndex;
        } else {
            sb.append(org);
            prev = endIndex;
        }
    }

    if (prev < S.length()) {
        sb.append(S.substring(prev));
    }

    return sb.toString();
}
```

---

## 15 One Edit Distance

Given two strings S and T, determine if they are both one edit distance apart.

### 15.1 Java Solution

---

```
public boolean isOneEditDistance(String s, String t) {
    if(s==null || t==null)
        return false;

    int m = s.length();
    int n = t.length();

    if(Math.abs(m-n)>1){
        return false;
    }

    int i=0;
    int j=0;
    int count=0;

    while(i<m&&j<n){
        if(s.charAt(i)==t.charAt(j)){
            i++;
            j++;
        }else{
            count++;
            if(count>1)
                return false;

            if(m>n){
                i++;
            }else if(m<n){
                j++;
            }else{
                i++;
                j++;
            }
        }
    }

    if(i<m||j<n){
        count++;
    }

    if(count==1)
        return true;

    return false;
}
```

---

## 16 Merge Sorted Array

*Given two sorted integer arrays A and B, merge B into A as one sorted array.*

Note: You may assume that A has enough space to hold additional elements from B. The number of elements initialized in A and B are m and n respectively.

### 16.1 Analysis

The key to solve this problem is moving element of A and B backwards. If B has some elements left after A is done, also need to handle that case.

The takeaway message from this problem is that the loop condition. This kind of condition is also used for [merging two sorted linked list](#).

### 16.2 Java Solution 1

---

```
public class Solution {
    public void merge(int A[], int m, int B[], int n) {

        while(m > 0 && n > 0){
            if(A[m-1] > B[n-1]){
                A[m+n-1] = A[m-1];
                m--;
            }else{
                A[m+n-1] = B[n-1];
                n--;
            }
        }

        while(n > 0){
            A[m+n-1] = B[n-1];
            n--;
        }
    }
}
```

---

### 16.3 Java Solution 2

The loop condition also can use m+n like the following.

---

```
public void merge(int A[], int m, int B[], int n) {
    int i = m - 1;
    int j = n - 1;
    int k = m + n - 1;

    while (k >= 0) {
        if (j < 0 || (i >= 0 && A[i] > B[j]))
            A[k--] = A[i--];
    }
}
```

---

```
        else  
            A[k--] = B[j--];  
    }  
}
```

---

## 17 Is Subsequence

Given a string *s* and a string *t*, check if *s* is subsequence of *t*.

You may assume that there is only lower case English letters in both *s* and *t*. *t* is potentially a very long (length = 500,000) string, and *s* is a short string ( $\leq 100$ ).

A subsequence of a string is a new string which is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (ie, "ace" is a subsequence of "abcde" while "aec" is not).

### 17.1 Java Solution

---

```
public boolean isSubsequence(String s, String t) {
    if(s.length()==0)
        return true;

    int i=0;
    int j=0;
    while(i<s.length() && j<t.length()){
        if(s.charAt(i)==t.charAt(j)){
            i++;
        }

        j++;

        if(i==s.length())
            return true;
    }

    return false;
}
```

---



## 18 Backspace String Compare

Given two strings S and T, return if they are equal when both are typed into empty text editors. # means a backspace character.

Example 1:

---

Input: S = "ab#c", T = "ad#c"

Output: true

Explanation: Both S and T become "ac".

---

Example 2:

---

Input: S = "a##c", T = "#a#c"

Output: true

Explanation: Both S and T become "c".

---

### 18.1 Java Solution

This problem requires  $O(N)$  time and  $O(1)$  space.

---

```
public boolean backspaceCompare(String S, String T) {
    int i = S.length()-1;
    int j = T.length()-1;

    while(i>=0 || j>=0){
        int c1=0;
        while(i>=0 && (c1>0 || S.charAt(i)=='#')){
            if(S.charAt(i)=='#'){
                c1++;
            }else{
                c1--;
            }
            i--;
        }

        int c2=0;
        while(j>=0 && (c2>0 || T.charAt(j)=='#')){
            if(T.charAt(j)=='#'){
                c2++;
            }else{
                c2--;
            }
            j--;
        }

        if(i>=0 && j>=0){
            if(S.charAt(i)!=T.charAt(j)){
                return false;
            }
        }
    }
    return true;
}
```

```
        }else{
            i--;
            j--;
        }
    }else{
        if(i>=0 || j>=0){
            return false;
        }
    }
}

return i<0 && j<0;
}
```

---

## 19 Repeated String Match

Given two strings A and B, find the minimum number of times A has to be repeated such that B is a substring of it. If no such solution, return -1.

For example, with A = "abcd" and B = "cdabcdab".

Return 3, because by repeating A three times ("abcdabcdabcd"), B is a substring of it; and B is not a substring of A repeated two times ("abcdabcd").

Note: The length of A and B will be between 1 and 10000.

### 19.1 Java Solution

The optimal solution's time complexity is  $O(n)$  where n is the length of the longer string from A and B.

---

```
public int repeatedStringMatch(String A, String B) {
    int i = 0;
    int j = 0;

    int result = 0;
    int k = 0;

    while (j < B.length()) {
        if (A.charAt(i) == B.charAt(j)) {
            i++;
            j++;

            if (i == A.length()) {
                i = 0;
                result++;
            }
        } else {
            k++;
            if (k == A.length()) {
                return -1;
            }
            i = k;
            j = 0;
            result = 0;
        }
    }

    if (i > 0) {
        result++;
    }

    return result;
}
```

---

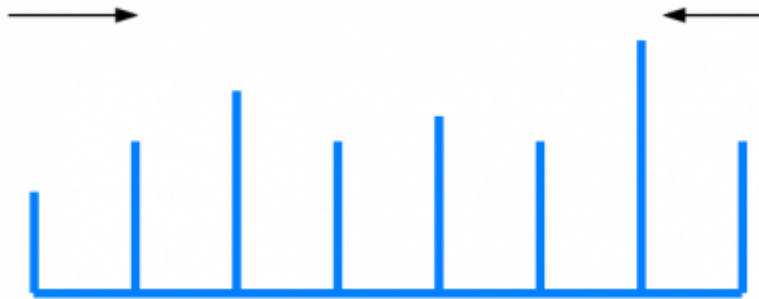
## 20 Container With Most Water

### 20.1 Problem

Given  $n$  non-negative integers  $a_1, a_2, \dots, a_n$ , where each represents a point at coordinate  $(i, a_i)$ .  $n$  vertical lines are drawn such that the two endpoints of line  $i$  is at  $(i, a_i)$  and  $(i, 0)$ . Find two lines, which together with  $x$ -axis forms a container, such that the container contains the most water.

### 20.2 Analysis

Initially we can assume the result is 0. Then we scan from both sides. If  $\text{leftHeight} < \text{rightHeight}$ , move right and find a value that is greater than  $\text{leftHeight}$ . Similarly, if  $\text{leftHeight} > \text{rightHeight}$ , move left and find a value that is greater than  $\text{rightHeight}$ . Additionally, keep tracking the max value.



### 20.3 Java Solution

---

```
public int maxArea(int[] height) {
    if (height == null || height.length < 2) {
        return 0;
    }

    int max = 0;
    int left = 0;
    int right = height.length - 1;

    while (left < right) {
        max = Math.max(max, (right - left) * Math.min(height[left], height[right]));
        if (height[left] < height[right])
            left++;
        else
            right--;
    }

    return max;
}
```

---

## 21 Reverse Vowels of a String

Write a function that takes a string as input and reverse only the vowels of a string.

### 21.1 Java Solution

this is a simple problem which can be solved by using two pointers scanning from beginning and end of the array.

---

```
public String reverseVowels(String s) {
    ArrayList<Character> vowList = new ArrayList<Character>();
    vowList.add('a');
    vowList.add('e');
    vowList.add('i');
    vowList.add('o');
    vowList.add('u');
    vowList.add('A');
    vowList.add('E');
    vowList.add('I');
    vowList.add('O');
    vowList.add('U');

    char[] arr = s.toCharArray();

    int i=0;
    int j=s.length()-1;

    while(i<j){
        if(!vowList.contains(arr[i])){
            i++;
            continue;
        }

        if(!vowList.contains(arr[j])){
            j--;
            continue;
        }

        char t = arr[i];
        arr[i]=arr[j];
        arr[j]=t;

        i++;
        j--;
    }

    return new String(arr);
}
```

---

## 22 Valid Palindrome

Given a string, determine if it is a palindrome, considering only alphanumeric characters and ignoring cases.

For example, "Red rum, sir, is murder" is a palindrome, while "Programcreek is awesome" is not.

Note: Have you consider that the string might be empty? This is a good question to ask during an interview.

For the purpose of this problem, we define empty string as valid palindrome.

### 22.1 Java Solution

There are several different ways to solve this problem. The following is a solution with  $O(n)$  time complexity and  $O(1)$  space complexity.

---

```
public boolean isPalindrome(String s) {
    if(s==null){
        return false;
    }

    s = s.toLowerCase();

    int i=0;
    int j=s.length()-1;

    while(i<j){
        while(i<j && !((s.charAt(i)>='a' && s.charAt(i)<='z')
            || (s.charAt(i)>='0'&&s.charAt(i)<='9'))){
            i++;
        }

        while(i<j && !((s.charAt(j)>='a' && s.charAt(j)<='z')
            || (s.charAt(j)>='0'&&s.charAt(j)<='9'))){
            j--;
        }

        if(s.charAt(i) != s.charAt(j)){
            return false;
        }

        i++;
        j--;
    }

    return true;
}
```

---

## 23 Shortest Word Distance

Given a list of words and two words word1 and word2, return the shortest distance between these two words in the list.

For example, Assume that words = ["practice", "makes", "perfect", "coding", "makes"].

Given word1 = "coding", word2 = "practice", return 3. Given word1 = "makes", word2 = "coding", return 1.

### 23.1 Java Solution

---

```
public int shortestDistance(String[] words, String word1, String word2) {
    int m=-1;
    int n=-1;

    int min = Integer.MAX_VALUE;

    for(int i=0; i<words.length; i++){
        String s = words[i];
        if(word1.equals(s)){
            m = i;
            if(n!=-1)
                min = Math.min(min, m-n);
        }else if(word2.equals(s)){
            n = i;
            if(m!=-1)
                min = Math.min(min, n-m);
        }
    }

    return min;
}
```

---

## 24 Shortest Word Distance II

This is a follow up of Shortest Word Distance. The only difference is now you are given the list of words and your method will be called repeatedly many times with different parameters. How would you optimize it?

Design a class which receives a list of words in the constructor, and implements a method that takes two words word1 and word2 and return the shortest distance between these two words in the list.

For example, Assume that words = ["practice", "makes", "perfect", "coding", "makes"].

Given word1 = "coding", word2 = "practice", return 3. Given word1 = "makes", word2 = "coding", return 1.

### 24.1 Java Solution

---

```
public class WordDistance {
    HashMap<String, ArrayList<Integer>> map;
    public WordDistance(String[] words) {
        map = new HashMap<String, ArrayList<Integer>>();
        for(int i=0; i<words.length; i++){
            if(map.containsKey(words[i])){
                map.get(words[i]).add(i);
            }else{
                ArrayList<Integer> list = new ArrayList<Integer>();
                list.add(i);
                map.put(words[i], list);
            }
        }
    }

    public int shortest(String word1, String word2) {

        ArrayList<Integer> l1 = map.get(word1);
        ArrayList<Integer> l2 = map.get(word2);

        int result = Integer.MAX_VALUE;
        for(int i1: l1){
            for(int i2: l2){
                result = Math.min(result, Math.abs(i1-i2));
            }
        }
        return result;
    }
}
```

---

The time complexity for shortest method is  $O(M*N)$ , where M is frequency of word1 and N is the frequency of word2. This can be improved by the following:

---

```
public int shortest(String word1, String word2) {

    ArrayList<Integer> l1 = map.get(word1);
    ArrayList<Integer> l2 = map.get(word2);

    int result = Integer.MAX_VALUE;
```



```
int i=0;
int j=0;
while(i<l1.size() && j<l2.size()){
    result = Math.min(result, Math.abs(l1.get(i)-l2.get(j)));
    if(l1.get(i)<l2.get(j)){
        i++;
    }else{
        j++;
    }
}

return result;
}
```

---

The time complexity of the shortest method is now  $O(M+N)$ . Since  $M+N < \text{size of word list}$ , the time is  $O(K)$  where  $k$  is the list size.

## 25 Shortest Word Distance III

This is a follow-up problem of [Shortest Word Distance](#). The only difference is now word1 could be the same as word2.

Given a list of words and two words word1 and word2, return the shortest distance between these two words in the list.

word1 and word2 may be the same and they represent two individual words in the list.

For example, Assume that words = ["practice", "makes", "perfect", "coding", "makes"].

Given word1 = "makes", word2 = "coding", return 1. Given word1 = "makes", word2 = "makes", return 3.

### 25.1 Java Solution 1

In this problem, word1 and word2 can be the same. The two variables used to track indices should take turns to update.

---

```
public int shortestWordDistance(String[] words, String word1, String word2) {
    if(words==null || words.length<1 || word1==null || word2==null)
        return 0;

    int m=-1;
    int n=-1;

    int min = Integer.MAX_VALUE;
    int turn=0;
    if(word1.equals(word2))
        turn = 1;

    for(int i=0; i<words.length; i++){
        String s = words[i];
        if(word1.equals(s) && (turn ==1 || turn==0)){
            m = i;
            if(turn==1) turn=2;
            if(n!=-1)
                min = Math.min(min, m-n);
        }else if(word2.equals(s) && (turn==2 || turn==0)){
            n = i;
            if(turn==2) turn =1;
            if(m!=-1)
                min = Math.min(min, n-m);
        }
    }

    return min;
}
```

---

### 25.2 Java Solution 2

We can divide the cases to two: word1 and word2 are the same and not the same.