I understand that the ‘optimal strategy’ part might be confusing if you’re seeing it for the first time, here’s how you can think about it.

Bob wants to maximize the score, and moves after Alice. He knows what she’s picked, of course he’s going to pick whichever option works best for him, i.e, max⁡(∣��−��−1∣,∣��−��+1∣)max(∣*Ai*​−*Ai*−1​∣,∣*Ai*​−*Ai*+1​∣).  
This is Bob’s optimal play.

Alice wants to minimize the score, and knows that Bob will move after her.  
This means she knows, if she picks ��*Ai*​, which of ��−1*Ai*−1​ and ��+1*Ai*+1​ Bob will choose.  
*Bob’s choice is not something she can influence after she has chosen ��Ai​.*

Her only optimal play is to thus pick whichever ��*Ai*​ minimizes max⁡(∣��−��−1∣,∣��−��+1∣)max(∣*Ai*​−*Ai*−1​∣,∣*Ai*​−*Ai*+1​∣), because that’s the best she can do.

This is generally what ‘optimal play’ means in game theory. Each player will make a move to the best of their ability, and has perfect information about what their opponents will do in response to their move (which is the information they use to decide their move in the first place).

Also, if you’re confused about a statement during a contest, you can always ask a clarification! The setter or an admin will usually reply reasonably quickly.

public class Solution {

public String reverseVowels(String s) {

if(s == null || s.length()==0) return s;

String vowels = "aeiouAEIOU";

char[] chars = s.toCharArray();

int start = 0;

int end = s.length()-1;

while(start<end){

while(start<end && !vowels.contains(chars[start]+"")){

start++;

}

while(start<end && !vowels.contains(chars[end]+"")){

end--;

}

char temp = chars[start];

chars[start] = chars[end];

chars[end] = temp;

start++;

end--;

}

return new String(chars);

}

class Solution {

public void moveZeroes(int[] nums) {

int count =0;

for(int i=0;i<nums.length;i++){

if(nums[i]!=0){

nums[count++] = nums[i];

}

}

while(count<nums.length){

nums[count++]=0;

}

}

}

class Solution {

public boolean wordPattern(String pattern, String s) {

String[] words = s.split(" ");

if (words.length != pattern.length()) {

return false;

}

Map<Character, String> charToWord = new HashMap<>();

Map<String, Character> wordToChar = new HashMap<>();

for (int i = 0; i < pattern.length(); i++) {

char c = pattern.charAt(i);

String word = words[i];

if (!charToWord.containsKey(c)) {

charToWord.put(c, word);

}

if (!wordToChar.containsKey(word)) {

wordToChar.put(word, c);

}

if (!charToWord.get(c).equals(word) || !wordToChar.get(word).equals(c)) {

return false;

}

}

return true;

}

}