6. Reverse Words in a String

Code it now: https://oj.leetcode.com/problems/reverse-words-in-a-string/ Difficulty: Medium, Frequency: High

Question:

Given an input string ***s***, reverse the string word by word.

For example, given ***s*** = "the sky is blue", return "blue is sky the".

Example Questions Candidate Might Ask:

Q: What constitutes a word?
A: A sequence of non-space characters constitutes a word.

Q: Does tab or newline character count as space characters?
A: Assume the input does not contain any tabs or newline characters.

Q: Could the input string contain leading or trailing spaces?
A: Yes. However, your reversed string should not contain leading or trailing spaces.

Q: How about multiple spaces between two words?
A: Reduce them to a single space in the reversed string.

Solution:

O(*n*) runtime, *O*(*n*) space:

One simple approach is a two-pass solution: First pass to split the string by spaces into an array of words, then second pass to extract the words in reversed order.

We can do better in one-pass. While iterating the string in reverse order, we keep track of a word's begin and end position. When we are at the beginning of a word, we append it.

```
public String reverseWords(String s) {
   StringBuilder reversed = new StringBuilder();
   int j = s.length();
   for (int i = s.length() - 1; i >= 0; i--) {
      if (s.charAt(i) == ' ') {
         j = i;
      } else if (i == 0 || s.charAt(i - 1) == ' ') {
         if (reversed.length() != 0) {
            reversed.append(' ');
         }
         reversed.append(s.substring(i, j));
      }
   }
   return reversed.toString();
}
```

Follow up:

If the input string does not contain leading or trailing spaces and the words are separated by a single space, could you do it *in-place* without allocating extra space? See Question [7. Reverse Words in a String II].

7. Reverse Words in a String II

Code it now: Coming soon! Difficulty: Medium, Frequency: N/A

Question:

Similar to Question [6. Reverse Words in a String], but with the following constraints:

"The input string does not contain leading or trailing spaces and the words are always separated by a single space."

Could you do it *in-place* without allocating extra space?

Solution:

$O(n)$ runtime, $O(1)$ space – In-place reverse:

Let us indicate the i^{th} word by w_i and its reversal as w_i'. Notice that when you reverse a word twice, you get back the original word. That is, $(w_i')' = w_i$.

The input string is $w_1\ w_2\ \ldots\ w_n$. If we reverse the entire string, it becomes $w_n'\ \ldots\ w_2'\ w_1'$. Finally, we reverse each individual word and it becomes $w_n\ \ldots\ w_2\ w_1$. Similarly, the same result could be reached by reversing each individual word first, and then reverse the entire string.

```
public void reverseWords(char[] s) {
   reverse(s, 0, s.length);
   for (int i = 0, j = 0; j <= s.length; j++) {
      if (j == s.length || s[j] == ' ') {
         reverse(s, i, j);
         i = j + 1;
      }
   }
}

private void reverse(char[] s, int begin, int end) {
   for (int i = 0; i < (end - begin) / 2; i++) {
      char temp = s[begin + i];
      s[begin + i] = s[end - i - 1];
      s[end - i - 1] = temp;
   }
}
```

Challenge 1:

Implement the two-pass solution without using the library's split function.

Challenge 2:

Rotate an array to the right by k steps in-place without allocating extra space. For instance, with $k = 3$, the array [0, 1, 2, 3, 4, 5, 6] is rotated to [4, 5, 6, 0, 1, 2, 3].

8. String to Integer (atoi)

Code it now: https://oj.leetcode.com/problems/string-to-integer-atoi/ Difficulty: Easy, Frequency: High

Question:

Implement *atoi* to convert a string to an integer.

The *atoi* function first discards as many whitespace characters as necessary until the first non-whitespace character is found. Then, starting from this character, takes an optional initial plus or minus sign followed by as many numerical digits as possible, and interprets them as a numerical value.

The string can contain additional characters after those that form the integral number, which are ignored and have no effect on the behavior of this function.

If the first sequence of non-whitespace characters in *str* is not a valid integral number, or if no such sequence exists because either *str* is empty or it contains only whitespace characters, no conversion is performed.

If no valid conversion could be performed, a zero value is returned. If the correct value is out of the range of representable values, the maximum integer value (2147483647) or the minimum integer value (–2147483648) is returned.

Solution:

The heart of this problem is dealing with overflow. A direct approach is to store the number as a string so we can evaluate at each step if the number had indeed overflowed. There are some other ways to detect overflow that requires knowledge about how a specific programming language or operating system works.

A desirable solution does not require any assumption on how the language works. In each step we are appending a digit to the number by doing a multiplication and addition. If the current number is greater than 214748364, we know it is going to overflow. On the other hand, if the current number is equal to 214748364, we know that it will overflow only when the current digit is greater than or equal to 8. Remember to also consider edge case for the smallest number, –2147483648 (-2^{31}).

```java
private static final int maxDiv10 = Integer.MAX_VALUE / 10;

public int atoi(String str) {
   int i = 0, n = str.length();
   while (i < n && Character.isWhitespace(str.charAt(i))) i++;
   int sign = 1;
   if (i < n && str.charAt(i) == '+') {
      i++;
   } else if (i < n && str.charAt(i) == '-') {
      sign = -1;
      i++;
   }
   int num = 0;
   while (i < n && Character.isDigit(str.charAt(i))) {
      int digit = Character.getNumericValue(str.charAt(i));
      if (num > maxDiv10 || num == maxDiv10 && digit >= 8) {
         return sign == 1 ? Integer.MAX_VALUE : Integer.MIN_VALUE;
      }
      num = num * 10 + digit;
      i++;
   }
   return sign * num;
}
```

9. Valid Number

Code it now: https://oj.leetcode.com/problems/valid-number/ Difficulty: Easy, Frequency: Low

Question:

Validate if a given string is numeric.

Some examples:

"0" → true
"0.1" → true
"abc" → false

Example Questions Candidate Might Ask:

Q: How to account for whitespaces in the string?
A: When deciding if a string is numeric, ignore both leading and trailing whitespaces.

Q: Should I ignore spaces in between numbers – such as "1 1"?
A: No, only ignore leading and trailing whitespaces. "1 1" is not numeric.

Q: If the string contains additional characters after a number, is it considered valid?
A: No. If the string contains any non-numeric characters (excluding whitespaces and decimal point), it is not numeric.

Q: Is it valid if a plus or minus sign appear before the number?
A: Yes. "+1" and "-1" are both numeric.

Q: Should I consider only numbers in decimal? How about numbers in other bases such as hexadecimal (0xFF)?
A: Only consider decimal numbers. "0xFF" is not numeric.

Q: Should I consider exponent such as "1e10" as numeric?
A: No. But feel free to work on the challenge that takes exponent into consideration. (The Online Judge problem does take exponent into account.)

Solution:

This problem is very similar to Question [8. String to Integer (atoi)]. Due to many corner cases, it is helpful to break the problem down to several components that can be solved individually.

A string could be divided into these four substrings in the order from left to right:

1. *s1.* Leading whitespaces (optional).
2. *s2.* Plus (+) or minus (–) sign (optional).
3. *s3.* Number.
4. *s4.* Optional trailing whitespaces (optional).

We ignore *s1, s2, s4* and evaluate whether *s3* is a valid number. We realize that a number could either be a whole number or a decimal number. For a whole number, it is easy: We evaluate whether *s3* contains only digits and we are done.

On the other hand, a decimal number could be further divided into three parts:

a. Integer part
b. Decimal point
c. Fractional part

The integer and fractional parts contain only digits. For example, the number “3.64” has integer part (3) and fractional part (64). Both of them are optional, but *at least* one of them must present. For example, a single dot ‘.’ is not a valid number, but “1.”, “.1”, and “1.0” are all valid. Please note that “1.” is valid because it implies “1.0”.

By now, it is pretty straightforward to translate the requirements into code, where the main logic to determine if *s3* is numeric from line 6 to line 17.

```
public boolean isNumber(String s) {
   int i = 0, n = s.length();
   while (i < n && Character.isWhitespace(s.charAt(i))) i++;
   if (i < n && (s.charAt(i) == '+' || s.charAt(i) == '-')) i++;
   boolean isNumeric = false;
   while (i < n && Character.isDigit(s.charAt(i))) {
      i++;
      isNumeric = true;
   }
   if (i < n && s.charAt(i) == '.') {
      i++;
      while (i < n && Character.isDigit(s.charAt(i))) {
         i++;
         isNumeric = true;
      }
   }
   while (i < n && Character.isWhitespace(s.charAt(i))) i++;
   return isNumeric && i == n;
}
```

Further Thoughts:

A number could contain an optional exponent part, which is marked by a character ‘e’ followed by a whole number (exponent). For example, “1e10” is numeric. Modify the above code to adapt to this new requirement.

This is pretty straightforward to extend from the previous solution. The added block of code is highlighted as below.

```
public boolean isNumber(String s) {
   int i = 0, n = s.length();
   while (i < n && Character.isWhitespace(s.charAt(i))) i++;
   if (i < n && (s.charAt(i) == '+' || s.charAt(i) == '-')) i++;
   boolean isNumeric = false;
   while (i < n && Character.isDigit(s.charAt(i))) {
      i++;
      isNumeric = true;
   }
   if (i < n && s.charAt(i) == '.') {
      i++;
      while (i < n && Character.isDigit(s.charAt(i))) {
         i++;
         isNumeric = true;
      }
   }
   if (isNumeric && i < n && s.charAt(i) == 'e') {
      i++;
      isNumeric = false;
      if (i < n && (s.charAt(i) == '+' || s.charAt(i) == '-')) i++;
      while (i < n && Character.isDigit(s.charAt(i))) {
         i++;
         isNumeric = true;
      }
   }
   while (i < n && Character.isWhitespace(s.charAt(i))) i++;
   return isNumeric && i == n;
}
```

10. Longest Substring Without Repeating Characters

Code it now: https://oj.leetcode.com/problems/longest-substring-without-repeating-characters/

Difficulty: Medium, Frequency: Medium

Question:

Given a string, find the length of the longest substring without repeating characters. For example, the longest substring without repeating letters for “abcabcbb” is “abc”, which the length is 3. For “bbbbb” the longest substring is “b”, with the length of 1.

Solution:

***O*(*n*) runtime, *O*(1) space – Two iterations:**

How can we look up if a character exists in a substring instantaneously? The answer is to use a simple table to store the characters that have appeared. Make sure you communicate with your interviewer if the string can have characters other than ‘a’–‘z’. (ie, Digits? Upper case letter? Does it contain ASCII characters only? Or even unicode character sets?)

The next question is to ask yourself what happens when you found a repeated character? For example, if the string is “abcdcedf”, what happens when you reach the second appearance of ‘c’?

When you have found a repeated character (let’s say at index *j*), it means that the current substring (excluding the repeated character of course) is a potential maximum, so update the maximum if necessary. It also means that the repeated character must have appeared before at an index *i*, where *i* is less than *j*.

Since you know that all substrings that start before or at index *i* would be less than your current maximum, you can safely start to look for the next substring with head which starts exactly at index *i* + 1.

Therefore, you would need two indices to record the head and the tail of the current substring. Since *i* and *j* both traverse at most *n* steps, the worst case would be 2*n* steps, which the runtime complexity must be *O*(*n*).

Note that the space complexity is constant *O*(1), even though we are allocating an array. This is because no matter how long the string is, the size of the array stays the same at 256.

```
public int lengthOfLongestSubstring(String s) {
   boolean[] exist = new boolean[256];
   int i = 0, maxLen = 0;
   for (int j = 0; j < s.length(); j++) {
      while (exist[s.charAt(j)]) {
         exist[s.charAt(i)] = false;
         i++;
      }
      exist[s.charAt(j)] = true;
      maxLen = Math.max(j - i + 1, maxLen);
   }
   return maxLen;
}
```

What if the character set could contain unicode characters that is out of ascii's range? We could modify the above solution to use a Set instead of a simple boolean array of size 256.

$O(n)$ runtime, $O(1)$ space – Single iteration:

The above solution requires at most 2n steps. In fact, it could be optimized to require only n steps. Instead of using a table to tell if a character exists or not, we could define a mapping of the characters to its index. Then we can skip the characters immediately when we found a repeated character.

```
public int lengthOfLongestSubstring(String s) {
   int[] charMap = new int[256];
   Arrays.fill(charMap, -1);
   int i = 0, maxLen = 0;
   for (int j = 0; j < s.length(); j++) {
      if (charMap[s.charAt(j)] >= i) {
         i = charMap[s.charAt(j)] + 1;
      }
      charMap[s.charAt(j)] = j;
      maxLen = Math.max(j - i + 1, maxLen);
   }
   return maxLen;
}
```

11. Longest Substring with At Most Two Distinct Characters

Code it now: Difficulty: Hard, Frequency: N/A
https://oj.leetcode.com/problems/longest-substring-with-at-most-two-distinct-characters/

Question:

Given a string *S*, find the length of the longest substring *T* that contains at most two distinct characters.

For example,

Given *S* = “eceba”,

T is "ece" which its length is 3.

Solution:

First, we could simplify the problem by assuming that *S* contains two or more distinct characters, which means *T* must contain exactly two distinct characters.

The brute force approach is $O(n^3)$ where *n* is the length of *S*. We can form every possible substring, and for each substring insert all characters into a Set which the Set’s size indicating the number of distinct characters. This could be easily improved to $O(n^2)$ by reusing the same Set when adding a character to form a new substring.

The trick is to maintain a sliding window that always satisfies the invariant where there are always at most two distinct characters in it. When we add a new character that breaks this invariant, how can we move the begin pointer to satisfy the invariant? Using the above example, our first window is the substring “abba”. When we add the character ‘c’ into the sliding window, it breaks the invariant. Therefore, we have to readjust the window to satisfy the invariant again. The question is which starting point to choose so the invariant is satisfied.

Let’s look at another example where *S* = “abaac”. We found our first window “abaa”. When we add ‘c’ to the window, the next sliding window should be “aac”.

This method iterates *n* times and therefore its runtime complexity is $O(n)$. We use three pointers: *i*, *j*, and *k*.

```
public int lengthOfLongestSubstringTwoDistinct(String s) {
   int i = 0, j = -1, maxLen = 0;
   for (int k = 1; k < s.length(); k++) {
      if (s.charAt(k) == s.charAt(k - 1)) continue;
      if (j >= 0 && s.charAt(j) != s.charAt(k)) {
         maxLen = Math.max(k - i, maxLen);
         i = j + 1;
      }
      j = k - 1;
   }
   return Math.max(s.length() - i, maxLen);
}
```

Further Thoughts:

Although the above method works fine, it could not be easily generalized to the case where *T* contains at most *k* distinct characters.

The key is when we adjust the sliding window to satisfy the invariant, we need a counter of the number of times each character appears in the substring.

```
public int lengthOfLongestSubstringTwoDistinct(String s) {
   int[] count = new int[256];
   int i = 0, numDistinct = 0, maxLen = 0;
   for (int j = 0; j < s.length(); j++) {
      if (count[s.charAt(j)] == 0) numDistinct++;
      count[s.charAt(j)]++;
      while (numDistinct > 2) {
         count[s.charAt(i)]--;
         if (count[s.charAt(i)] == 0) numDistinct--;
         i++;
      }
      maxLen = Math.max(j - i + 1, maxLen);
   }
   return maxLen;
}
```

12. Missing Ranges

Code it now: https://oj.leetcode.com/problems/missing-ranges/ Difficulty: Medium, Frequency: N/A

Question:

Given a sorted integer array where the range of elements are [0, 99] inclusive, return its missing ranges.

For example, given [0, 1, 3, 50, 75], return ["2", "4->49", "51->74", "76->99"]

Example Questions Candidate Might Ask:

Q: What if the given array is empty?
A: Then you should return ["0->99"] as those ranges are missing.

Q: What if the given array contains all elements from the ranges?
A: Return an empty list, which means no range is missing.

Solution:

Compare the gap between two neighbor elements and output its range, simple as that right? This seems deceptively easy, except there are multiple edge cases to consider, such as the first and last element, which does not have previous and next element respectively. Also, what happens when the given array is empty? We should output the range "0->99".

As it turns out, if we could add two "artificial" elements, –1 before the first element and 100 after the last element, we could avoid all the above pesky cases.

Further Thoughts:

i. List out test cases.

ii. You should be able to extend the above cases not only for the range [0,99], but any arbitrary range [start, end].

```
public List<String> findMissingRanges(int[] vals, int start, int end) {
   List<String> ranges = new ArrayList<>();
   int prev = start - 1;
   for (int i = 0; i <= vals.length; i++) {
      int curr = (i == vals.length) ? end + 1 : vals[i];
      if (curr - prev >= 2) {
         ranges.add(getRange(prev + 1, curr - 1));
      }
      prev = curr;
   }
   return ranges;
}

private String getRange(int from, int to) {
   return (from == to) ? String.valueOf(from) : from + "->" + to;
}
```

13. Longest Palindromic Substring

Code it now: https://oj.leetcode.com/problems/longest-palindromic-substring/ Difficulty: Medium, Frequency: Medium

Question:

Given a string *S*, find the longest palindromic substring in *S*. You may assume that the maximum length of *S* is 1000, and there exists one unique longest palindromic substring.

Hint:

First, make sure you understand what a palindrome means. A palindrome is a string which reads the same in both directions. For example, "aba" is a palindome, "abc" is not.

A common mistake:

Some people will be tempted to come up with a quick solution, which is unfortunately flawed (however can be corrected easily):

> Reverse S and become S'. Find the longest common substring between S and S', which must also be the longest palindromic substring.

This seemed to work, let's see some examples below.

For example, S = "caba", S' = "abac".

The longest common substring between S and S' is "aba", which is the answer.

Let's try another example: S = "abacdfgdcaba", S' = "abacdgfdcaba".

The longest common substring between S and S' is "abacd". Clearly, this is not a valid palindrome.

We could see that the longest common substring method fails when there exists a reversed copy of a non-palindromic substring in some other part of S. To rectify this, each time we find a longest common substring candidate, we check if the substring's indices are the same as the reversed substring's original indices. If it is, then we attempt to update the longest palindrome found so far; if not, we skip this and find the next candidate.

This gives us an $O(n^2)$ DP solution which uses $O(n^2)$ space (could be improved to use $O(n)$ space). Please read more about Longest Common Substring here.

$O(n^3)$ runtime, $O(1)$ space – Brute force:

The obvious brute force solution is to pick all possible starting and ending positions for a substring, and verify if it is a palindrome. There are a total of $\binom{n}{2}$ such substrings (excluding the trivial solution where a character itself is a palindrome).

Since verifying each substring takes $O(n)$ time, the run time complexity is $O(n^3)$.