

Chapter 1: Array/String

1. Two Sum

Code it now: <https://oj.leetcode.com/problems/two-sum/>

Difficulty: Easy, Frequency: High

Question:

Given an array of integers, find two numbers such that they add up to a specific target number.

The function *twoSum* should return indices of the two numbers such that they add up to the target, where *index1* must be less than *index2*. Please note that your returned answers (both *index1* and *index2*) are not zero-based.

You may assume that each input would have *exactly* one solution.

Solution:

$O(n^2)$ runtime, $O(1)$ space – Brute force:

The brute force approach is simple. Loop through each element x and find if there is another value that equals to $target - x$. As finding another value requires looping through the rest of array, its runtime complexity is $O(n^2)$.

$O(n)$ runtime, $O(n)$ space – Hash table:

We could reduce the runtime complexity of looking up a value to $O(1)$ using a hash map that maps a value to its index.

```
public int[] twoSum(int[] numbers, int target) {
    Map<Integer, Integer> map = new HashMap<>();
    for (int i = 0; i < numbers.length; i++) {
        int x = numbers[i];
        if (map.containsKey(target - x)) {
            return new int[] { map.get(target - x) + 1, i + 1 };
        }
        map.put(x, i);
    }
    throw new IllegalArgumentException("No two sum solution");
}
```

Follow up:

What if the given input is already sorted in ascending order? See Question [2. Two Sum II – Input array is sorted].

2. Two Sum II – Input array is sorted

Code it now: Coming soon!

Difficulty: Medium, Frequency: N/A

Question:

Similar to Question [1. Two Sum], except that the input array is already sorted in ascending order.

Solution:

Of course we could still apply the [Hash table] approach, but it costs us $O(n)$ extra space, plus it does not make use of the fact that the input is already sorted.

$O(n \log n)$ runtime, $O(1)$ space – Binary search:

For each element x , we could look up if $target - x$ exists in $O(\log n)$ time by applying binary search over the sorted array. Total runtime complexity is $O(n \log n)$.

```
public int[] twoSum(int[] numbers, int target) {
    // Assume input is already sorted.
    for (int i = 0; i < numbers.length; i++) {
        int j = bsearch(numbers, target - numbers[i], i + 1);
        if (j != -1) {
            return new int[] { i + 1, j + 1 };
        }
    }
    throw new IllegalArgumentException("No two sum solution");
}

private int bsearch(int[] A, int key, int start) {
    int L = start, R = A.length - 1;
    while (L < R) {
        int M = (L + R) / 2;
        if (A[M] < key) {
            L = M + 1;
        } else {
            R = M;
        }
    }
    return (L == R && A[L] == key) ? L : -1;
}
```

$O(n)$ runtime, $O(1)$ space – Two pointers:

Let's assume we have two indices pointing to the i^{th} and j^{th} elements, A_i and A_j respectively. The sum of A_i and A_j could only fall into one of these three possibilities:

- i. $A_i + A_j > \text{target}$. Increasing i isn't going to help us, as it makes the sum even bigger. Therefore we should decrement j .
- ii. $A_i + A_j < \text{target}$. Decreasing j isn't going to help us, as it makes the sum even smaller. Therefore we should increment i .
- iii. $A_i + A_j == \text{target}$. We have found the answer.

```

public int[] twoSum(int[] numbers, int target) {
    // Assume input is already sorted.
    int i = 0, j = numbers.length - 1;
    while (i < j) {
        int sum = numbers[i] + numbers[j];
        if (sum < target) {
            i++;
        } else if (sum > target) {
            j--;
        } else {
            return new int[] { i + 1, j + 1 };
        }
    }
    throw new IllegalArgumentException("No two sum solution");
}

```

3. Two Sum III – Data structure design

Code it now: Coming soon!

Difficulty: Easy, Frequency: N/A

Question:

Design and implement a *TwoSum* class. It should support the following operations: *add* and *find*.

add(input) – Add the number *input* to an internal data structure.

find(value) – Find if there exists any pair of numbers which sum is equal to the *value*.

For example,

add(1); add(3); add(5); find(4) → true; find(7) → false

Solution:

***add* – $O(n)$ runtime, *find* – $O(1)$ runtime, $O(n^2)$ space – Store pair sums in hash table:**

We could store all possible pair sums into a hash table. The extra space needed is in the order of $O(n^2)$. You would also need an extra $O(n)$ space to store the list of added numbers. Each *add* operation essentially go through the list and form new pair sums that go into the hash table. The *find* operation involves a single hash table lookup in $O(1)$ runtime.

This method is useful if the number of *find* operations far exceeds the number of *add* operations.

***add* – $O(\log n)$ runtime, *find* – $O(n)$ runtime, $O(n)$ space – Binary search + Two pointers:**

Maintain a sorted array of numbers. Each *add* operation would need $O(\log n)$ time to insert it at the correct position using a modified binary search (See Question [48. Search Insert Position]). For *find* operation we could then apply the [Two pointers] approach in $O(n)$ runtime.

***add* – $O(1)$ runtime, *find* – $O(n)$ runtime, $O(n)$ space – Store input in hash table:**

A simpler approach is to store each input into a hash table. To find if a pair sum exists, just iterate through the hash table in $O(n)$ runtime. Make sure you are able to handle duplicates correctly.

```

public class TwoSum {
    private Map<Integer, Integer> table = new HashMap<>();

    public void add(int input) {
        int count = table.containsKey(input) ? table.get(input) : 0;
        table.put(input, count + 1);
    }

    public boolean find(int val) {
        for (Map.Entry<Integer, Integer> entry : table.entrySet()) {
            int num = entry.getKey();
            int y = val - num;
            if (y == num) {
                // For duplicates, ensure there are at least two individual numbers.
                if (entry.getValue() >= 2) return true;
            } else if (table.containsKey(y)) {
                return true;
            }
        }
        return false;
    }
}

```

4. Valid Palindrome

Code it now: <https://oj.leetcode.com/problems/valid-palindrome/>

Difficulty: Easy, Frequency: Medium

Question:

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

For example,

"A man, a plan, a canal: Panama" is a palindrome.

"race a car" is *not* a palindrome.

Example Questions Candidate Might Ask:

Q: What about an empty string? Is it a valid palindrome?

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

Solution:

$O(n)$ runtime, $O(1)$ space:

The idea is simple, have two pointers – one at the head while the other one at the tail. Move them towards each other until they meet while skipping non-alphanumeric characters.

Consider the case where given string contains only non-alphanumeric characters. This is a valid palindrome because the empty string is also a valid palindrome.

```
public boolean isPalindrome(String s) {
    int i = 0, j = s.length() - 1;
    while (i < j) {
        while (i < j && !Character.isLetterOrDigit(s.charAt(i))) i++;
        while (i < j && !Character.isLetterOrDigit(s.charAt(j))) j--;
        if (Character.toLowerCase(s.charAt(i))
            != Character.toLowerCase(s.charAt(j))) {
            return false;
        }
        i++; j--;
    }
    return true;
}
```

5. Implement strstr()

Code it now: <https://oj.leetcode.com/problems/implement-strstr/>

Difficulty: Easy, Frequency: High

Question:

Implement *strstr()*. Returns the index of the first occurrence of *needle* in *haystack*, or -1 if *needle* is not part of *haystack*.

Solution:

$O(nm)$ runtime, $O(1)$ space – Brute force:

There are known efficient algorithms such as [Rabin-Karp algorithm](#), [KMP algorithm](#), or the [Boyer-Moore algorithm](#). Since these algorithms are usually studied in an advanced algorithms class, it is sufficient to solve it using the most direct method in an interview – The *brute force method*.

The brute force method is straightforward to implement. We scan the *needle* with the *haystack* from its first position and start matching all subsequent letters one by one. If one of the letters does not match, we start over again with the next position in the *haystack*.

Assume that n = length of *haystack* and m = length of *needle*, then the runtime complexity is $O(nm)$.

Have you considered these scenarios?

- i. *needle* or *haystack* is empty. If *needle* is empty, always return 0. If *haystack* is empty, then there will always be no match (return -1) unless *needle* is also empty which 0 is returned.
- ii. *needle*'s length is greater than *haystack*'s length. Should always return -1 .
- iii. *needle* is located at the end of *haystack*. For example, "aaaba" and "ba". Catch possible off-by-one errors.
- iv. *needle* occur multiple times in *haystack*. For example, "mississippi" and "issi". It should return index 2 as the *first* match of "issi".
- v. Imagine two very long strings of equal lengths = n , *haystack* = "aaa...aa" and *needle* = "aaa...ab". You should not do more than n character comparisons, or else your code will get Time Limit Exceeded in OJ.

Below is a clean implementation – no special if statements for all the above scenarios.

```
public int strStr(String haystack, String needle) {
    for (int i = 0; ; i++) {
        for (int j = 0; ; j++) {
            if (j == needle.length()) return i;
            if (i + j == haystack.length()) return -1;
            if (needle.charAt(j) != haystack.charAt(i + j)) break;
        }
    }
}
```

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 $s = \text{"the sky is blue"}$, 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 $w_1 w_2 \dots w_n$. If we reverse the entire string, it becomes $w_n' \dots w_2' w_1'$. Finally, we reverse each individual word and it becomes $w_n \dots w_2 w_1$. 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}).

```

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:

- s1*. Leading whitespaces (optional).
- s2*. Plus (+) or minus (–) sign (optional).
- s3*. Number.
- 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 “bbbb” 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 $2n$ 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;
}

```