Sold to Jiuling Wang using Selz (#P7H78SSD)

Edition

1

LEETCODE

50 COMMON INTERVIEW QUESTIONS

Clean Code Handbook

LEETCODE

Clean Code Handbook

© 2014 LeetCode
admin@leetcode.com

Chapter 0: Foreword

Hi, fellow LeetCoders:

First, I would like to express thank you for buying this eBook: “Clean Code Handbook: 50 Common Interview Questions”. As the title suggested, this is the definitive guide to write clean and concise code for interview questions. You will learn how to write clean code. Then, you will ace the coding interviews.

This eBook is written to serve as the perfect companion for LeetCode Online Judge (OJ).

Each problem has a “Code it now” link that redirects to the OJ problem statement page. You can write the code and submit it to the OJ system, which you will get immediate feedback on whether your solution is correct. If the “Code it now” link says “Coming soon”, it means the problem will be added very soon in the future, so stay tuned.

On the top right side of each problem are the Difficulty and Frequency ratings. There are three levels of difficulty: Easy, Medium, and Hard. Easy problems are problems that are easy to come up with ideas and the implementation should be pretty straightforward. Most interview questions fall into this level of difficulty.

On the other end, there are hard problems. Hard problems are usually algorithmic in nature and require more thoughts beforehand. There could be some coding questions that fall into Hard, but that is rare.

There are three frequency rating: Low, Medium, and High. The frequency of a problem being asked in a real interview is based on data collected from the user survey: “*Have you met this question in a real interview?*” This should give you an idea of what kind of questions is currently being asked in real interviews.

Each question may contain a section called “Example Questions Candidate Might Ask”. These are examples of good questions to ask your interviewer. Asking clarifying questions is important and is a good chance to demonstrate your thought process.

Each question is accompanied with as many approaches as possible. Each approach begins with a runtime and space complexity summary so you can quickly compare between different approaches. The analysis of the runtime and space complexity is usually provided along with the solution. Analyzing complexity is frequently being asked in a technical interview, so you should definitely prepare for it.

You learn best when you solve a problem by yourself. If you get stuck, there are usually hints in the book to help you. If you are still stuck, read the analysis and try to write the code yourself in the Online Judge

Even though you might think a problem is *easy*, writing code that is concise and clean is ***not*** as easy as most people think. For example, if you are writing more than 30 lines of code during a coding interview, your code is probably not concise enough. Most code in this eBook fall between 20 — 30 lines of code.

1337c0d3r

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. $A_i + A_j >$ target. Increasing i isn't going to help us, as it makes the sum even bigger. Therefore we should decrement j.

ii. $A_i + A_j <$ target. Decreasing j isn't going to help us, as it makes the sum even smaller. Therefore we should increment i.

iii. $A_i + A_j ==$ target. We have found the answer.

```java
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.

```java
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*** = "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:

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 “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 $2n$ 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.

```java
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.

```java
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: Coming soon! Difficulty: Hard, Frequency: N/A

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: Coming soon!

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:

没能处理溢出问题

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 $O(n^3)$.

$O(n^2)$ runtime, $O(n)$ space – Dynamic programming:

To improve over the brute force solution from a DP approach, first think how we can avoid unnecessary re-computation in validating palindromes. Consider the case "ababa". If we already knew that "bab" is a palindrome, it is obvious that "ababa" must be a palindrome since the two left and right end letters are the same.

Stated more formally below:

Define P[i, j] ← true iff the substring $S_i \ldots S_j$ is a palindrome, otherwise false.

Therefore,

P[i, j] ← (P[i+1, j-1] and Si = Sj)

The base cases are:

P[i, i] ← true

P[i, i+1] ← (Si = Si+1)

This yields a straight forward DP solution, which we first initialize the one and two letters palindromes, and work our way up finding all three letters palindromes, and so on…

This gives us a runtime complexity of $O(n^2)$ and uses $O(n^2)$ space to store the table.

Additional exercise:

Could you improve the above space complexity further and how?

$O(n^2)$ runtime, $O(1)$ space – Simpler solution:

In fact, we could solve it in $O(n^2)$ time using only constant space.

We observe that a palindrome mirrors around its center. Therefore, a palindrome can be expanded from its center, and there are only $2n - 1$ such centers.

You might be asking why there are $2n - 1$ but not n centers? The reason is the center of a palindrome can be in between two letters. Such palindromes have even number of letters (such as "abba") and its center are between the two 'b's.

Since expanding a palindrome around its center could take $O(n)$ time, the overall complexity is $O(n^2)$.

```
public String longestPalindrome(String s) {
   int start = 0, end = 0;
   for (int i = 0; i < s.length(); i++) {
      int len1 = expandAroundCenter(s, i, i);
      int len2 = expandAroundCenter(s, i, i + 1);
      int len = Math.max(len1, len2);
      if (len > end - start) {
         start = i - (len - 1) / 2;
         end = i + len / 2;
      }
   }
   return s.substring(start, end + 1);
}

private int expandAroundCenter(String s, int left, int right) {
   int L = left, R = right;
   while (L >= 0 && R < s.length() && s.charAt(L) == s.charAt(R)) {
      L--;
      R++;
   }
   return R - L - 1;
}
```

$O(n)$ runtime, $O(n)$ space – Manacher's algorithm:

There is even an $O(n)$ algorithm called Manacher's algorithm, explained here in detail. However, it is a non-trivial algorithm, and no one expects you to come up with this algorithm in a 30 minutes coding session. But, please go ahead and understand it, I promise it will be a lot of fun.

14. One Edit Distance

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

Question:

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

Hint:

1. If | n – m | is greater than 1, we know immediately both are not one-edit distance apart.
2. It might help if you consider these cases separately, m == n and m ≠ n.
3. Assume that *m* is always ≤ *n*, which greatly simplifies the conditional statements. If m > n, we could just simply swap *S* and *T*.
4. If *m* == *n*, it becomes finding if there is *exactly* one modified operation. If m ≠ n, you do not have to consider the delete operation. Just consider the insert operation in *T*.

Solution:

Let us assume *m* = length of *S*, *n* = length of *T*.

Although this problem is solvable by directly applying the famous Edit distance dynamic programming algorithm with runtime complexity of $O(mn)$ and space complexity of $O(mn)$ (could be optimized to $O(\min(m,n))$, it is far from desirable as there exists a simpler and more efficient one-pass algorithm.

O(*n*) runtime, *O*(1) space – Simple one-pass:

For the case where *m* is equal to *n*, it becomes finding if there is exactly one modified character. Now let's assume $m \leq n$. (If $m > n$ we could just swap them).

Assume *X* represents the one-edit character. There are *three* one-edit distance operations that could be applied to *S*.

i. Modify operation – Modify a character to *X* in *S*.

```
S = "abcde"
T = "abXde"
```

ii. Insert operation – *X* was inserted before a character in *S*.

```
S = "abcde"
T = "abcXde"
```

iii. Append operation – *X* was appended at the end of *S*.

```
S = "abcde"
T = "abcdeX"
```

We make a first pass over S and T concurrently and stop at the first non-matching character between S and T.

1. If S matches all characters in T, then check if there is an extra character at the end of T. (*Modify* operation)
2. If $| n - m | == 1$, that means we must skip this non-matching character only in T and make sure the remaining characters between S and T are exactly matching. (*Insert* operation)
3. If $| n - m | == 0$, then we skip both non-matching characters i*n* S an*d* T and make sure the remaining characters betwee*n* S an*d* T are exactly matching. (*Append* operation)

```
public boolean isOneEditDistance(String s, String t) {
   int m = s.length(), n = t.length();
   if (m > n) return isOneEditDistance(t, s);
   if (n - m > 1) return false;
   int i = 0, shift = n - m;
   while (i < m && s.charAt(i) == t.charAt(i)) i++;
   if (i == m) return shift > 0;
   if (shift == 0) i++;
   while (i < m && s.charAt(i) == t.charAt(i + shift)) i++;
   return i == m;
}
```

15. Read N Characters Given Read4

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

Question:

The API: *int read4(char *buf)* reads 4 characters at a time from a file.

The return value is the actual number of characters read. For example, it returns 3 if there is only 3 characters left in the file.

By using the *read4* API, implement the function *int read(char *buf, int n)* that reads *n* characters from the file.

Note: The *read* function will only be called once for each test case.

Solution:

This seemingly easy coding question has some tricky edge cases. When *read4* returns less than 4, we know it must reached the end of file. However, take note that *read4* returning 4 could mean the last 4 bytes of the file.

To make sure that the buffer is not copied more than *n* bytes, copy the remaining bytes (*n* – readBytes) or the number of bytes read, whichever is smaller.

```
/* The read4 API is defined in the parent class Reader4.
      int read4(char[] buf); */

public class Solution extends Reader4 {
   /**
    * @param buf Destination buffer
    * @param n   Maximum number of characters to read
    * @return    The number of characters read
    */
   public int read(char[] buf, int n) {
      char[] buffer = new char[4];
      int readBytes = 0;
      boolean eof = false;
      while (!eof && readBytes < n) {
         int sz = read4(buffer);
         if (sz < 4) eof = true;
         int bytes = Math.min(n - readBytes, sz);
         System.arraycopy(buffer /* src */, 0 /* srcPos */,
               buf /* dest */, readBytes /* destPos */, bytes /* length */);
         readBytes += bytes;
      }
      return readBytes;
   }
}
```

Follow up:

What if read could be called multiple times? See Question [16. Read N Characters Given Read4 – Call multiple times].

16. Read N Characters Given Read4 – Call multiple times

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

Question:

Similar to Question [15. Read N Characters Given Read4], but the *read* function may be called multiple times.

Solution:

This makes the problem a lot more complicated, because it can be called multiple times and involves storing states.

Therefore, we design the following class member variables to store the states:

i. *buffer* – An array of size 4 use to store data returned by *read4* temporarily. If the characters were read into the buffer and were not used partially, they will be used in the next call.

ii. *offset* – Use to keep track of the offset index where the data begins in the next *read* call. The buffer could be read partially (due to constraints of reading up to *n* bytes) and therefore leaving some data behind.

iii. *bufsize* – The real buffer size that stores the actual data. If *bufsize* > 0, that means there is partial data left in buffer from the last *read* call and we should consume it before calling *read4* again. On the other hand, if *bufsize* == 0, it means there is no data left in buffer.

This problem is a very good coding exercise. Coding it correctly is extremely tricky due to the amount of edge cases to consider.

```java
/* The read4 API is defined in the parent class Reader4.
      int read4(char[] buf); */

public class Solution extends Reader4 {
   private char[] buffer = new char[4];
   int offset = 0, bufsize = 0;
   /**
    * @param buf Destination buffer
    * @param n   Maximum number of characters to read
    * @return    The number of characters read
    */
   public int read(char[] buf, int n) {
      int readBytes = 0;
      boolean eof = false;
      while (!eof && readBytes < n) {
         int sz = (bufsize > 0) ? bufsize : read4(buffer);
         if (bufsize == 0 && sz < 4) eof = true;
         int bytes = Math.min(n - readBytes, sz);
         System.arraycopy(buffer /* src */, offset /* srcPos */,
               buf /* dest */, readBytes /* destPos */, bytes /* length */);
         offset = (offset + bytes) % 4;
         bufsize = sz - bytes;
         readBytes += bytes;
      }
      return readBytes;
   }
}
```

Chapter 2: Math

17. Reverse Integer

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

Question:

Reverse digits of an integer. For example: $x = 123$, return 321.

Example Questions Candidate Might Ask:

Q: What about negative integers?
A: For input $x = -123$, you should return -321.
Q: What if the integer's last digit is 0? For example, $x = 10, 100, \ldots$
A: Ignore the leading 0 digits of the reversed integer. 10 and 100 are both reversed as 1.
Q: What if the reversed integer overflows? For example, input $x = 1000000003$.
A: In this case, your function should return 0.

Solution:

Let's start with a simple implementation. We do not need to handle negative integers separately, because the modulus operator works for negative integers as well (e.g., $-43 \% 10 = -3$).

```
public int reverse(int x) {
    int ret = 0;
    while (x != 0) {
        ret = ret * 10 + x % 10;
        x /= 10;
    }
    return ret;
}
```

There is a flaw in the above code – the reversed integer could overflow/underflow. Take $x = 1000000003$ for example. To check for overflow/underflow, we could check if $ret > 214748364$ or $ret < -214748364$ before multiplying by 10. On the other hand, if $ret == 214748364$, it must not overflow because the last reversed digit is guaranteed to be 1 due to constraint of the input x.

```java
public int reverse(int x) {
    int ret = 0;
    while (x != 0) {
        // handle overflow/underflow
        if (Math.abs(ret) > 214748364) {
            return 0;
        }
        ret = ret * 10 + x % 10;
        x /= 10;
    }
    return ret;
}
```

18. Plus One

Code it now: https://oj.leetcode.com/problems/plus-one/ Difficulty: Easy, Frequency: High

Question:

Given a number represented as an array of digits, plus one to the number.

Example Questions Candidate Might Ask:

Q: Could the number be negative?

A: No. Assume it is a non-negative number.

Q: How are the digits ordered in the list? For example, is the number 12 represented by [1,2] or [2,1]?

A: The digits are stored such that the most significant digit is at the head of the list.

Q: Could the number contain leading zeros, such as [0,0,1]?

A: No.

Solution:

Iterate from the least significant digit, and simulate by adding one to it. Adding one to a digit less than nine is straightforward – Add one to it and we are done.

On the other hand, adding one to a digit of 9 brings it to 10, so we set the digit to 0 and continues with a carry digit of one to its left digit. Notice this recursive behavior? Yes, we are adding one again to its left digit and this behavior continues until the most significant digit.

Finally, be sure that you handle the edge case where each digit of the number is 9.

```
public void plusOne(List<Integer> digits) {
   for (int i = digits.size() - 1; i >= 0; i--) {
       int digit = digits.get(i);
       if (digit < 9) {
         digits.set(i, digit + 1);
         return;
       } else {
         digits.set(i, 0);
       }
   }
   digits.add(0);
   digits.set(0, 1);
}
```

When all digits are 9, we did something slightly strange (See line 11). We append the digit 0 and modify the most significant digit to 1. Some of you might ask why not insert 1 to the front of list? Assume that the list is implemented as an ArrayList, appending an element is far more efficient than inserting to the front, because all elements have to be shifted one place to the right otherwise.

19. Palindrome Number

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

Question:

Determine whether an integer is a palindrome. Do this without extra space.

Example Questions Candidate Might Ask:

Q: Does negative integer such as –1 qualify as a palindrome?
A: For the purpose of discussion here, we define negative integers as non-palindrome.

Solution:

The most intuitive approach is to first represent the integer as a string, since it is more convenient to manipulate. Although this certainly does work, it violates the restriction of not using extra space. (ie, you have to allocate n characters to store the reversed integer as string, where n is the maximum number of digits). I know, this sound like an unreasonable requirement (since it uses so little space), but don't most interview problems have such requirements?

Another approach is to first reverse the number. If the number is the same as its reversed, then it must be a palindrome. You could reverse a number by doing the following:

```
public int reverse(int num) {
  assert num >= 0;  // for non-negative integers only.
  int rev = 0;
  while (num != 0) {
    rev = rev * 10 + num % 10;
    num /= 10;
  }
  return rev;
}
```

This seemed to work too, but did you consider the possibility that the reversed number might overflow? If it overflows, the behavior is language specific (For Java the number wraps around on overflow, but in C/C++ its behavior is undefined). Yuck.

Of course, we could avoid overflow by storing and returning a type that has larger size than int (ie, long long). However, do note that this is language specific, and the larger type might not always be available on all languages.

We could construct a better and more generic solution. One pointer is that, we must start comparing the digits somewhere. And you know there could only be two ways, either expand from the middle or compare from the two ends.

It turns out that comparing from the two ends is easier. First, compare the first and last digit. If they are not the same, it must not be a palindrome. If they are the same, chop off one digit from both ends and continue until you have no digits left, which you conclude that it must be a palindrome.

Now, getting and chopping the last digit is easy. However, getting and chopping the first digit in a generic way requires some thought. I will leave this to you as an exercise. Please think your solution out before you peek on the solution below.

```
public boolean isPalindrome(int x) {
  if (x < 0) return false;
  int div = 1;
  while (x / div >= 10) {
    div *= 10;
  }
  while (x != 0) {
    int l = x / div;
    int r = x % 10;
    if (l != r) return false;
    x = (x % div) / 10;
    div /= 100;
  }
  return true;
}
```

Chapter 3: Linked List

20. Merge Two Sorted Lists

Code it now: https://oj.leetcode.com/problems/merge-two-sorted-lists/ Difficulty: Easy, Frequency: Medium

Question:

Merge two sorted linked lists and return it as a new list. The new list should be made by splicing together the nodes of the first two lists.

Solution:

We insert a dummy head before the new list so we don't have to deal with special cases such as initializing the new list's head. Then the new list's head could just easily be returned as dummy head's next node.

Using dummy head allows you to write simpler code and adds as a powerful tool to your interview arsenal. To see more examples of dummy head usage, please see these questions: [21. Add Two Numbers], [22. Swap Nodes in Pairs], and [23. Merge K Sorted Linked Lists].

```
public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
   ListNode dummyHead = new ListNode(0);
   ListNode p = dummyHead;
   while (l1 != null && l2 != null) {
      if (l1.val < l2.val) {
         p.next = l1;
         l1 = l1.next;
      } else {
         p.next = l2;
         l2 = l2.next;
      }
      p = p.next;
   }
   if (l1 != null) p.next = l1;
   if (l2 != null) p.next = l2;
   return dummyHead.next;
}
```

21. Add Two Numbers

Code it now: https://oj.leetcode.com/problems/add-two-numbers/ Difficulty: Medium, Frequency: High

Question:

You are given two linked lists representing two non-negative numbers. The digits are stored in reverse order and each of their nodes contains a single digit. Add the two numbers and return it as a linked list.

Input: (2 → 4 → 3) + (5 → 6 → 4)

Output: 7 → 0 → 8

Solution:

Keep track of the carry using a variable and simulate digits-by-digits sum from the head of list, which contains the least-significant digit.

Take extra caution of the following cases:

- When one list is longer than the other.
- The sum could have an extra carry of one at the end, which is easy to forget. (e.g., (9 → 9) + (1) = (0 → 0 → 1))

```
public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
   ListNode dummyHead = new ListNode(0);
   ListNode p = l1, q = l2, curr = dummyHead;
   int carry = 0;
   while (p != null || q != null) {
      int x = (p != null) ? p.val : 0;
      int y = (q != null) ? q.val : 0;
      int digit = carry + x + y;
      carry = digit / 10;
      curr.next = new ListNode(digit % 10);
      curr = curr.next;
      if (p != null) p = p.next;
      if (q != null) q = q.next;
   }
   if (carry > 0) {
      curr.next = new ListNode(carry);
   }
   return dummyHead.next;
}
```

22. Swap Nodes in Pairs

Code it now: https://oj.leetcode.com/problems/swap-nodes-in-pairs/ Difficulty: Medium, Frequency: Medium

Question:

Given a linked list, swap every two adjacent nodes and return its head.

For example,

Given 1 → 2 → 3 → 4, you should return the list as 2 → 1 → 4 → 3.

Your algorithm should use only constant space. You may not modify the values in the list, only nodes itself can be changed.

Example Questions Candidate Might Ask:

Q: What if the number of nodes in the linked list has only odd number of nodes?
A: The last node should not be swapped.

Solution:

Let's assume p, q, r are the current, next, and next's next node.

We could swap the nodes pairwise by adjusting where it's pointing next:
q.next = p;
p.next = r;

The above operations transform the list from { p → q → r → s } to { q → p → r → s }.

If the next pair of nodes exists, we should remember to connect p's next to s. Therefore, we should record the current node before advancing to the next pair of nodes.

To determine the new list's head, you look at if the list contains two or more elements. Basically, these extra conditional statements could be avoided by inserting an extra node (also known as the dummy head) to the front of the list.

```
public ListNode swapPairs(ListNode head) {
   ListNode dummy = new ListNode(0);
   dummy.next = head;
   ListNode p = head;
   ListNode prev = dummy;
   while (p != null && p.next != null) {
      ListNode q = p.next, r = p.next.next;
      prev.next = q;
      q.next = p;
      p.next = r;
      prev = p;
      p = r;
   }
   return dummy.next;
}
```

23. Merge K Sorted Linked Lists

Code it now: https://oj.leetcode.com/problems/merge-k-sorted-lists/ Difficulty: Hard, Frequency: High

Question:

Merge k sorted linked lists and return it as one sorted list. Analyze and describe its complexity.

Solution:

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

The brute force approach is to merge a list one by one. For example, if the lists = [$l1$, $l2$, $l3$, $l4$], we first merge $l1$ and $l2$, then merge the result with $l3$, and finally $l4$.

To analyze its time complexity, we are going to assume there are a total of k lists, and each list is of size n. There will be a total of $k - 1$ merge operations. The first merge operation will be two lists of size n, therefore in the worst case there could be $n + n$ comparisons. The second merge operation will be two lists of size $2n$ and n. Notice that each merge increase the size of the merged lists by n. Therefore, the total number of comparisons required is $2n + 3n + 4n + \ldots + kn = n\left(\frac{k(k+1)}{2} - 1\right) = O(nk^2)$.

$O(nk \log k)$ runtime, $O(k)$ space – Heap:

We could use a min heap of size k. The heap is first initialized with the smallest element from each list. Then as we extract the nodes out from the heap, we must remember to insert its next node into the heap. As each insert operation into the heap costs $\log(k)$ and there are a total of nk elements, the total runtime complexity is $O(nk \log k)$.

Ignoring the extra space that is used to store the output list, we only use extra space of $O(k)$ due to the heap.

```java
private static final Comparator<ListNode> listComparator =
      new Comparator<ListNode>() {
   @Override
   public int compare(ListNode x, ListNode y) {
      return x.val - y.val;
   }
};

public ListNode mergeKLists(List<ListNode> lists) {
   if (lists.isEmpty()) return null;
   Queue<ListNode> queue = new PriorityQueue<>(lists.size(), listComparator);
   for (ListNode node : lists) {
      if (node != null) {
         queue.add(node);
      }
   }
   ListNode dummyHead = new ListNode(0);
   ListNode p = dummyHead;
   while (!queue.isEmpty()) {
      ListNode node = queue.poll();
      p.next = node;
      p = p.next;
      if (node.next != null) {
         queue.add(node.next);
      }
   }
   return dummyHead.next;
}
```

$O(nk \log k)$ runtime, $O(1)$ space – Divide and conquer using two way merge:

If you still remember how merge sort works, we can use a divide and conquer mechanism to solve this problem. Here, we apply the merge two lists algorithm from Question [20. Merge Two Sorted Lists].

Basically, the algorithm merges two lists at a time, so the number of lists reduces from:

$$k \to \frac{k}{2} \to \frac{k}{4} \to \cdots \to 2 \to 1$$

Similarly, the size of the lists increases from (Note that the lists could subdivide itself at most $\log(k)$ times):

$$n \to 2n \to 4n \to \cdots \to 2^{\log k} n$$

Therefore, the runtime complexity is:

$$\begin{aligned} & k \cdot n + \frac{k}{2} \cdot 2n + \frac{k}{4} \cdot 4n + \cdots + 2^{\log k} n \cdot 1 \\ &= nk + nk + nk + \cdots + nk \\ &= nk \log k \end{aligned}$$

Since we are implementing this divide and conquer algorithm iteratively, the space complexity is constant at $O(1)$, yay!

```java
public ListNode mergeKLists(List<ListNode> lists) {
   if (lists.isEmpty()) return null;
   int end = lists.size() - 1;
   while (end > 0) {
      int begin = 0;
      while (begin < end) {
         lists.set(begin, merge2Lists(lists.get(begin),
                                      lists.get(end)));
         begin++;
         end--;
      }
   }
   return lists.get(0);
}

private ListNode merge2Lists(ListNode l1, ListNode l2) {
   ListNode dummyHead = new ListNode(0);
   ListNode p = dummyHead;
   while (l1 != null && l2 != null) {
      if (l1.val < l2.val) {
         p.next = l1;
         l1 = l1.next;
      } else {
         p.next = l2;
         l2 = l2.next;
      }
      p = p.next;
   }
   if (l1 != null) p.next = l1;
   if (l2 != null) p.next = l2;
   return dummyHead.next;
}
```

24. Copy List with Random Pointer

Code it now: https://oj.leetcode.com/problems/copy-list-with-random-pointer/ Difficulty: Hard, Frequency: High

Question:

A linked list is given such that each node contains an additional random pointer that could point to any node in the list or null.

Return a deep copy of the list.

Solution:

Cloning a linked list without an additional random pointer is easy to solve. The trickier part however, is to clone the random list node structure.

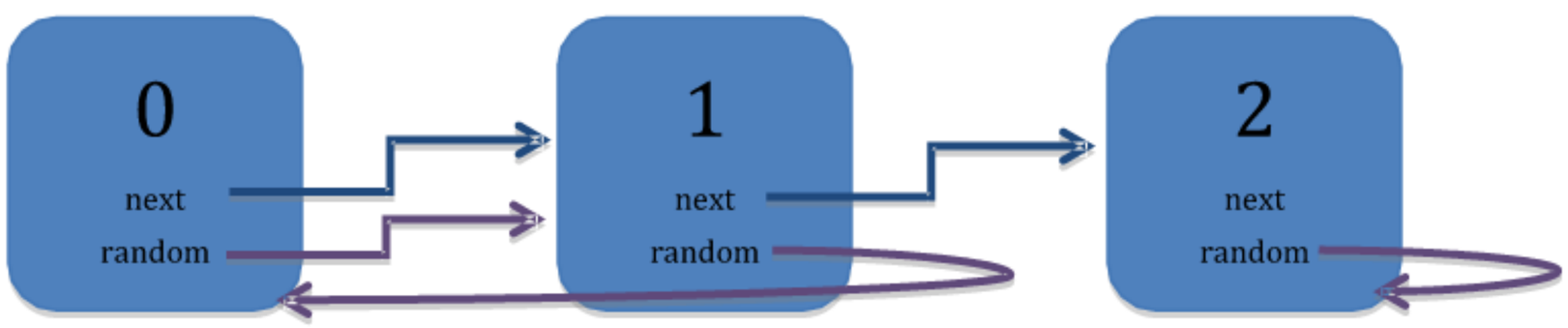


Figure 1: Linked list that has both next and random pointers.

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

To get started, it is helpful to label each individual node with an index. According to the above list, node 0's random points to node 1, node 1's random point to node 0, and node 2's random points to itself, node 2.

We could rebuild the structure assume we have the above connections. As we know that node 0 random points to node 1, we need to connect the cloned version from node 0' to node 1'. We would have to iterate over the list each time to link the nodes and it takes $O(n)$, making the overall time complexity $O(n^2)$.

How do we represent the connection? We can build a map that maps the original node to its indices. Having this map will allow us to clone the structure. If we know that node 0's random points to node 1, we just have to connect them, right? The only issue is connecting them takes $O(n)$ complexity, because we have to traverse the cloned list to find the node to connect.

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

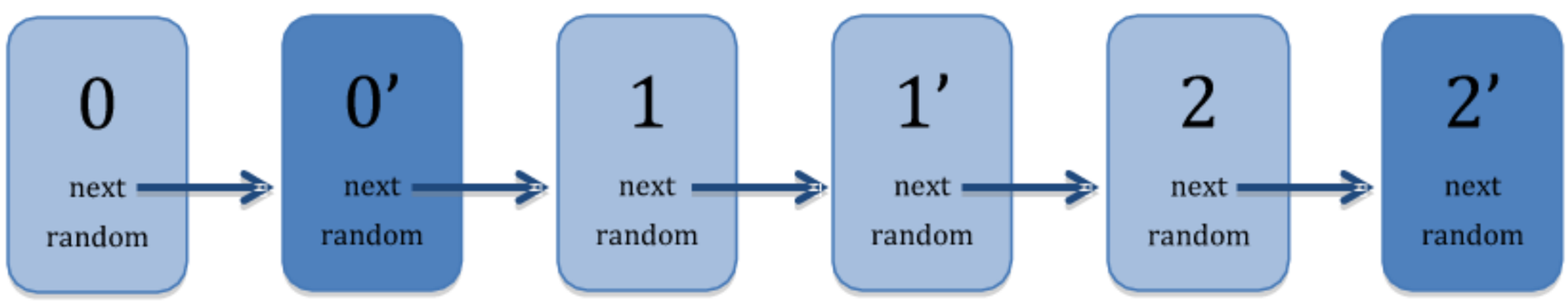
It is now natural to lead to a mapping so we can quickly lookup the node to connect. We can easily build the map of indices to cloned nodes. Therefore, we have reduced the complexity to $O(1)$ when connecting the random nodes.

This had got us started, although it requires two maps. On closer inspection, it turns out that the two maps could be shortened into one single map. We just need to map the original node to its random node directly.

```java
public RandomListNode copyRandomList(RandomListNode head) {
    Map<RandomListNode, RandomListNode> map = new HashMap<>();
    RandomListNode p = head;
    RandomListNode dummy = new RandomListNode(0);
    RandomListNode q = dummy;
    while (p != null) {
        q.next = new RandomListNode(p.label);
        map.put(p, q.next);
        p = p.next;
        q = q.next;
    }
    p = head;
    q = dummy;
    while (p != null) {
        q.next.random = map.get(p.random);
        p = p.next;
        q = q.next;
    }
    return dummy.next;
}
```

O(*n*) runtime, *O*(1) space – Modify original structure:

The above algorithm uses extra space *O*(*n*), can we not use extra space? What if we eliminate the map? The only way is to modify the original structure. Imagine if we modify the next node of the original node to point to its own copy.



Now, assume we have the above configuration we could assign the random node pointers of each copy easily with the following code:

```
node.next.random = node.random.next;
```

To summarize, we need three iterations over the list:

i. Create a copy of each of the original node and insert them in between two original nodes in an alternate fashion.

ii. Assign random pointer of each node copy.

iii. Restore the input to its original configuration.

We have achieved *O*(*n*) runtime complexity with using only constant extra space.

```java
public RandomListNode copyRandomList(RandomListNode head) {
    RandomListNode p = head;
    while (p != null) {
        RandomListNode next = p.next;
        RandomListNode copy = new RandomListNode(p.label);
        p.next = copy;
        copy.next = next;
        p = next;
    }
    p = head;
    while (p != null) {
        p.next.random = (p.random != null) ? p.random.next : null;
        p = p.next.next;
    }
    p = head;
    RandomListNode headCopy = (p != null) ? p.next : null;
    while (p != null) {
        RandomListNode copy = p.next;
        p.next = copy.next;
        p = p.next;
        copy.next = (p != null) ? p.next : null;
    }
    return headCopy;
}
```

Chapter 4: Binary Tree

25. Validate Binary Search Tree

Code it now: https://oj.leetcode.com/problems/validate-binary-search-tree/ Difficulty: Medium, Frequency: High

Question:

Given a binary tree, determine if it is a valid Binary Search Tree (BST).

Solution:

First, you must understand the difference between Binary Tree and BST. Binary tree is a tree data structure in which each node has at most two child nodes. A BST is based on binary tree, but with the following additional properties:

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.

People who do not understand the definition of BST may give the following algorithm:

Assume that the current node's value is *k*. Then for each node, check if the left node's value is less than *k* and the right node's value is greater than *k*. If all of the nodes satisfy this property, then it is a BST.

It sounds correct and convincing, but look at this counter example below: A sample tree that we name it as binary tree (1).

```
  10
 /  \
5   15   -------- binary tree (1)
   /  \
  6   20
```

It's obvious that this is not a valid BST, since (6) could never be on the right of (10).

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

Based on BST's definition, we can easily devise a brute force solution:

Assume that the current node's value is *k*. Then for each node, check if all nodes of left subtree contain values that are less than *k*. Also check if all nodes of right subtree contain values that are greater than *k*. If all of the nodes satisfy this property, then it must be a BST.

Below is the brute force code that is inefficient:

```java
public boolean isValidBST(TreeNode root) {
  if (root == null) return true;
  return isSubtreeLessThan(root.left, root.val)
      && isSubtreeGreaterThan(root.right, root.val)
      && isValidBST(root.left) && isValidBST(root.right);
}

private boolean isSubtreeLessThan(TreeNode p, int val) {
  if (p == null) return true;
  return p.val < val
      && isSubtreeLessThan(p.left, val)
      && isSubtreeLessThan(p.right, val);
}

private boolean isSubtreeGreaterThan(TreeNode p, int val) {
  if (p == null) return true;
  return p.val > val
      && isSubtreeGreaterThan(p.left, val)
      && isSubtreeGreaterThan(p.right, val);
}
```

The worst case runtime complexity is $O(n^2)$ for the brute force algorithm, when the tree degenerates into a linked list with n nodes.

$O(n)$ runtime, $O(n)$ stack space – Top-down recursion:

Here is the much better solution. We can avoid examining all nodes of both subtrees in each pass by passing down the *low* and *high* limits from the parent to its children.

Refer back to the binary tree (1) above. As we traverse down the tree from node (10) to right node (15), we know for sure that the right node's value fall between 10 and $+\infty$. Then, as we traverse further down from node (15) to left node (6), we know for sure that the left node's value fall between 10 and 15. And since (6) does not satisfy the above requirement, we can quickly determine it is not a valid BST.

```java
public boolean isValidBST(TreeNode root) {
   return valid(root, Integer.MIN_VALUE, Integer.MAX_VALUE);
}

private boolean valid(TreeNode p, int low, int high) {
   if (p == null) return true;
   return p.val > low && p.val < high
         && valid(p.left, low, p.val)
         && valid(p.right, p.val, high);
}
```

This algorithm runs in $O(n)$ time, where n is the number of nodes of the binary tree.

Sharp readers may notice that the above code does not work if the tree contains the smallest or the largest integer value. How could we fix this? One way to fix this is to use *null* to represent the infinity.

```java
public boolean isValidBST(TreeNode root) {
   return valid(root, null, null);
}

private boolean valid(TreeNode p, Integer low, Integer high) {
   if (p == null) return true;
   return (low == null || p.val > low) && (high == null || p.val < high)
         && valid(p.left, low, p.val)
         && valid(p.right, p.val, high);
}
```

$O(n)$ runtime, $O(n)$ stack space – In-order traversal:

Another solution is to do an in-order traversal of the binary tree, and verify that its in-order elements follow a strict monotonic increasing order. During the in-order traversal, we verify that the previous value is less than the current node's value. The runtime complexity is also $O(n)$.

```java
private TreeNode prev;
public boolean isValidBST(TreeNode root) {
   prev = null;
   return isMonotonicIncreasing(root);
}

private boolean isMonotonicIncreasing(TreeNode p) {
   if (p == null) return true;
   if (isMonotonicIncreasing(p.left)) {
      if (prev != null && p.val <= prev.val) return false;
      prev = p;
      return isMonotonicIncreasing(p.right);
   }
   return false;
}
```

26. Maximum Depth of Binary Tree

Code it now: https://oj.leetcode.com/problems/maximum-depth-of-binary-tree/ Difficulty: Easy, Frequency: High

Question:

Given a binary tree, find its maximum depth.

The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

Solution:

The maximum height of a binary tree is defined as the number of nodes along the path from the root node to the deepest leaf node. Note that the maximum depth of an empty tree is 0.

$O(n)$ runtime, $O(\log n)$ space – Recursion:

We could solve this easily using recursion, because each of the left child and right child of a node is a sub-tree itself. We first compute the max height of left sub-tree, and then compute the max height of right sub-tree. The maximum depth of the current node is the greater of the two maximums plus one. For the base case, we look at a tree that is empty, which we return 0.

Assume that n is the total number of nodes in the tree. The runtime complexity is $O(n)$ because it traverse each node exactly once. As the maximum depth of a binary tree is $O(\log n)$, the extra space cost is $O(\log n)$ due to the extra stack space used by the recursion.

```
public int maxDepth(TreeNode root) {
   if (root == null) return 0;
   return Math.max(maxDepth(root.left), maxDepth(root.right)) + 1;
}
```