Edition 1

LEETCODE

50 COMMON INTERVIEW QUESTIONS

Clean Code Handbook

LEETCODE

Clean Code Handbook

---

admin@leetcode.com

CHAPTER 0: FOREWORD 4

CHAPTER 1: ARRAY/STRING 5

1. TWO SUM 5
2. TWO SUM II – INPUT ARRAY IS SORTED 6
3. TWO SUM III – DATA STRUCTURE DESIGN 8
4. VALID PALINDROME 10
5. IMPLEMENT STRSTR() 11
6. REVERSE WORDS IN A STRING 12
7. REVERSE WORDS IN A STRING II 13
8. STRING TO INTEGER (ATOI) 14
9. VALID NUMBER 16
10. LONGEST SUBSTRING WITHOUT REPEATING CHARACTERS 19
11. LONGEST SUBSTRING WITH AT MOST TWO DISTINCT CHARACTERS 21
12. MISSING RANGES 23
13. LONGEST PALINDROMIC SUBSTRING 24
14. ONE EDIT DISTANCE 27
15. READ N CHARACTERS GIVEN READ4 29
16. READ N CHARACTERS GIVEN READ4 – CALL MULTIPLE TIMES 30

CHAPTER 2: MATH 32

17. REVERSE INTEGER 32
18. PLUS ONE 34
19. PALINDROME NUMBER 35

CHAPTER 3: LINKED LIST 37

20. MERGE TWO SORTED LISTS 37
21. ADD TWO NUMBERS 38
22. SWAP NODES IN PAIRS 39
23. MERGE K SORTED LINKED LISTS 40
24. COPY LIST WITH RANDOM POINTER 43

CHAPTER 4: BINARY TREE 46

25. VALIDATE BINARY SEARCH TREE 46
26. MAXIMUM DEPTH OF BINARY TREE 49
27. MINIMUM DEPTH OF BINARY TREE 50
28. BALANCED BINARY TREE 52
29. CONVERT SORTED ARRAY TO BALANCED BINARY SEARCH TREE 54
30. CONVERT SORTED LIST TO BALANCED BINARY SEARCH TREE 56
31. BINARY TREE MAXIMUM PATH SUM 58
32. BINARY TREE UPSIDE DOWN 60

CHAPTER 5: BIT MANIPULATION 62

33. SINGLE NUMBER 62
34. SINGLE NUMBER II 64

CHAPTER 6: MISC 66

35. Spiral Matrix 66
36. Integer to Roman 68
37. Roman to Integer 70
38. Clone graph 72

CHAPTER 7: STACK 74

39. Min Stack 74
40. Evaluate Reverse Polish Notation 76
41. Valid Parentheses 80

CHAPTER 8: DYNAMIC PROGRAMMING 81

42. Climbing Stairs 81
43. Unique Paths 83
44. Unique Paths II 86
45. Maximum Sum Subarray 87
46. Maximum Product Subarray 89
47. Coins in a Line 90

CHAPTER 9: BINARY SEARCH 95

48. Search Insert Position 95
49. Find Minimum in Sorted Rotated Array 97
50. Find Minimum in Rotated Sorted Array II – with duplicates 99

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.

```
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: 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:

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, …
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.

```
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.

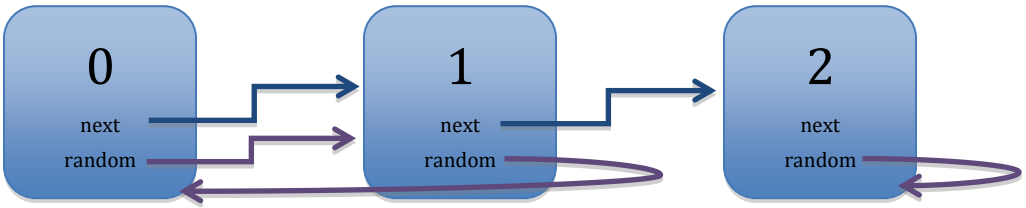


Figure 1: Linked list that has both next and random pointers.

$O(n^2)$ 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:

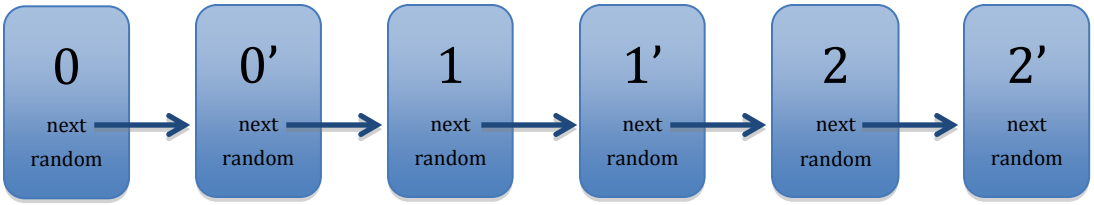
It is now natural to lead to a mapping so we can quickly lookup the node to connect. We can easily build the map of indices to cloned nodes. Therefore, we have reduced the complexity to $O(1)$ when connecting the random nodes.

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.

```
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.

```
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:

```
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 +∞. 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.

```
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.

```
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*).

```
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;
}
```

27. Minimum Depth of Binary Tree

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

Question:

Given a binary tree, find its minimum depth.

The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.

Similar to Question [26. Maximum Depth of Binary Tree], here we need to find the *minimum* depth instead.

Solution:

$O(n)$ runtime, $O(\log n)$ space – Depth-first traversal:

Similar to the [Recursion] approach to find the maximum depth, but make sure you consider these cases:

i. The node itself is a leaf node. The minimum depth is one.

ii. Node that has one empty sub-tree while the other one is non-empty. Return the minimum depth of that non-empty sub-tree.

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

$O(n)$ runtime, $O(n)$ space – Breadth-first traversal:

Note that the previous approach traverses all the nodes even for a highly unbalanced tree. In fact, we could optimize this scenario by doing a breadth-first traversal (also known as level-order traversal). When we encounter the first leaf node, we immediately stop the traversal.

We also keep track of the current depth and increment it when we reach the end of level. We know that we have reached the end of level when the current node is the right-most node.

Compared to the recursion approach, the breadth-first traversal works well for highly unbalanced tree because it does not need to traverse all nodes. The worst case is when the tree is a full binary tree with a total of n nodes. In this case, we have to traverse all nodes. The worst case space complexity is $O(n)$, due to the extra space needed to store current level nodes in the queue.

```java
public int minDepth(TreeNode root) {
   if (root == null) return 0;
   Queue<TreeNode> q = new LinkedList<>();
   q.add(root);
   TreeNode rightMost = root;
   int depth = 1;
   while (!q.isEmpty()) {
      TreeNode node = q.poll();
      if (node.left == null && node.right == null) break;
      if (node.left != null) q.add(node.left);
      if (node.right != null) q.add(node.right);
      if (node == rightMost) {
         depth++;
         rightMost = (node.right != null) ? node.right : node.left;
      }
   }
   return depth;
}
```

28. Balanced Binary Tree

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

Question:

Given a binary tree, determine if it is height-balanced.

For this problem, a height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of every node never differs by more than 1.

Solution:

$O(n^2)$ runtime, $O(n)$ stack space – Brute force top-down recursion:

We could devise a brute force algorithm directly based on the above definition. We also reused the [Recursion] approach to find the maximum depth of a subtree. The brute force algorithm worst case runtime complexity is $O(n^2)$ when the input tree is degenerated.

```
public boolean isBalanced(TreeNode root) {
  if (root == null) return true;
  return Math.abs(maxDepth(root.left) - maxDepth(root.right)) <= 1
      && isBalanced(root.left)
      && isBalanced(root.right);
}

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

$O(n)$ runtime, $O(n)$ stack space – Bottom-up recursion:

It seems that the above approach is recalculating max depth repeatedly for each node. We could avoid the recalculation by passing the depth *bottom-up*. We use a sentinel value –1 to represent that the tree is unbalanced so we could avoid unnecessary calculations.

In each step, we look at the left subtree's depth (L), and ask: "Is the left subtree unbalanced?" If it is indeed unbalanced, we return –1 right away. Otherwise, L represents the left subtree's depth. We then repeat the same process for the right subtree's depth (R).

We calculate the absolute difference between L and R. If the subtrees' depth difference is less than one, we could return the height of the current node, otherwise return –1 meaning the current tree is unbalanced.

```java
public boolean isBalanced(TreeNode root) {
   return maxDepth(root) != -1;
}

private int maxDepth(TreeNode root) {
   if (root == null) return 0;
   int L = maxDepth(root.left);
   if (L == -1) return -1;
   int R = maxDepth(root.right);
   if (R == -1) return -1;
   return (Math.abs(L - R) <= 1) ? (Math.max(L, R) + 1) : -1;
}
```

29. Convert Sorted Array to Balanced Binary Search Tree

Code it now: https://oj.leetcode.com/problems/convert-sorted-array-to-binary-search-tree/ Difficulty: Medium, Frequency: Low

Question:

Given an array where elements are sorted in ascending order, convert it to a height balanced BST.

Hint:

This question is highly recursive in nature. Think of how binary search works.

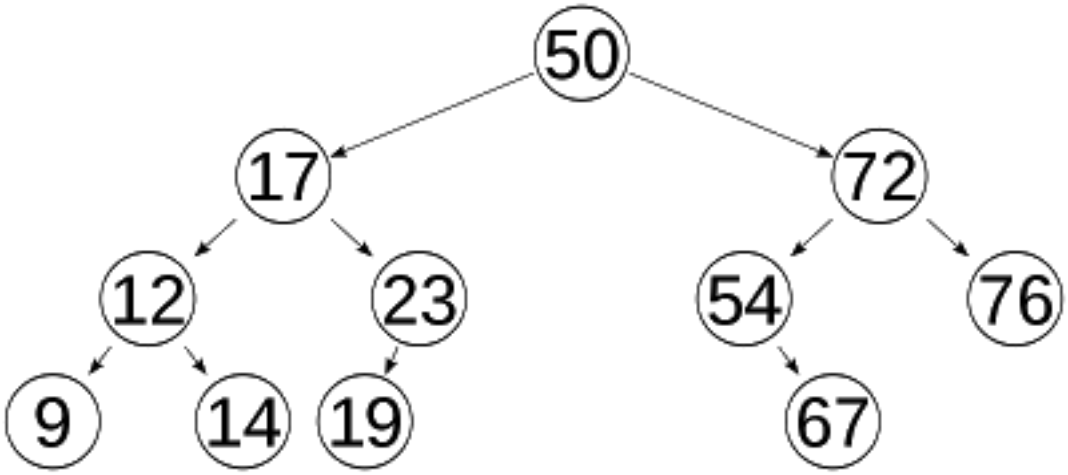


Figure 2: An example of a height-balanced tree. A height-balanced tree is a tree whose subtrees differ in height by no more than one and the subtrees are height-balanced, too.

Solution:

$O(n)$ runtime, $O(\log n)$ stack space – Divide and conquer:

If you would have to choose an array element to be the root of a balanced BST, which element would you pick? The root of a balanced BST should be the middle element from the sorted array.

You would pick the middle element from the sorted array in each iteration. You then create a node in the tree initialized with this element. After the element is chosen, what is left? Could you identify the sub-problems within the problem?

There are two arrays left — The one on its left and the one on its right. These two arrays are the sub-problems of the original problem, since both of them are sorted. Furthermore, they are subtrees of the current node's left and right child.

The code below creates a balanced BST from the sorted array in $O(n)$ time (n is the number of elements in the array). Compare how similar the code is to a binary search algorithm. Both are using the divide and conquer methodology. Because the input array could be subdivided in at most $\log(n)$ times, the extra stack space used by the recursion is in $O(\log n)$.

```
public TreeNode sortedArrayToBST(int[] num) {
   return sortedArrayToBST(num, 0, num.length-1);
}

private TreeNode sortedArrayToBST(int[] arr, int start, int end) {
   if (start > end) return null;
   int mid = (start + end) / 2;
   TreeNode node = new TreeNode(arr[mid]);
   node.left = sortedArrayToBST(arr, start, mid-1);
   node.right = sortedArrayToBST(arr, mid+1, end);
   return node;
}
```

Further Thoughts:

Consider changing the problem statement to “Converting a singly linked list to a balanced BST”. How would your implementation change from the above? See Question [30. Convert Sorted List to Balanced Binary Search Tree].

30. Convert Sorted List to Balanced Binary Search Tree

Code it now: https://oj.leetcode.com/problems/convert-sorted-list-to-binary-search-tree/ Difficulty: Hard, Frequency: Low

Question:

Given a singly linked list where elements are sorted in ascending order, convert it to a height balanced BST.

Hint:

Things get a little more complicated when you have a singly linked list instead of an array. Please note that in linked list, you no longer have random access to an element in $O(1)$ time.

How about inserting nodes following the list's order? If we can achieve this, we no longer need to find the middle element, as we are able to traverse the list while inserting nodes to the tree.

***O*(*n* log *n*) runtime, O(log *n*) stack space – Brute force:**

A naive way is to apply the previous solution from Question [29. Convert Sorted Array to Balanced Binary Search Tree] directly. In each recursive call, you would have to traverse half of the list's length to find the middle element. The run time complexity is clearly $O(n \log n)$, where n is the total number of elements in the list. This is because each level of recursive call requires a total of $\frac{n}{2}$ traversal steps in the list, and there are a total of $\log(n)$ number of levels (ie, the height of the balanced tree).

***O*(*n*) runtime, *O*(log *n*) stack space – Bottom-up recursion:**

As usual, the best solution requires you to think from another perspective. In other words, we no longer create nodes in the tree using the top-down approach. We create nodes bottom-up, and assign them to its parents. The bottom-up approach enables us to access the list in its order while creating nodes.

Isn't the bottom-up approach neat? Each time you are stuck with the top-down approach, give bottom-up a try. Although bottom-up approach is not the most natural way we think, it is extremely helpful in some cases. However, you should prefer top-down instead of bottom-up in general, since the latter is more difficult to verify in correctness.

Below is the code for converting a singly linked list to a balanced BST. Please note that the algorithm requires the list's length to be passed in as the function's parameters. The list's length could be found in $O(n)$ time by traversing the entire list's once. The recursive calls traverse the list and create tree's nodes by the list's order, which also takes $O(n)$ time. Therefore, the overall run time complexity is still $O(n)$.

```
private ListNode list;

private TreeNode sortedListToBST(int start, int end) {
   if (start > end) return null;
   int mid = (start + end) / 2;
   TreeNode leftChild = sortedListToBST(start, mid-1);
   TreeNode parent = new TreeNode(list.val);
   parent.left = leftChild;
   list = list.next;
   parent.right = sortedListToBST(mid+1, end);
   return parent;
}

public TreeNode sortedListToBST(ListNode head) {
   int n = 0;
   ListNode p = head;
   while (p != null) {
      p = p.next;
      n++;
   }
   list = head;
   return sortedListToBST(0, n - 1);
}
```

31. Binary Tree Maximum Path Sum

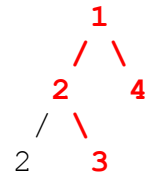
Code it now: https://oj.leetcode.com/problems/binary-tree-maximum-path-sum/ Difficulty: Hard, Frequency: Medium

Question:

Given a binary tree, find the maximum path sum.

The path may start and end at any node in the tree.

For example, given the below binary tree,



The highlighted path yields the maximum sum 10.

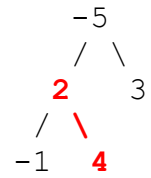
Example Questions Candidate Might Ask:

Q: What if the tree is empty?
A: Assume the tree is non-empty.

Q: How about a tree that contains only a single node?
A: Then the maximum path sum starts and ends at the same node.

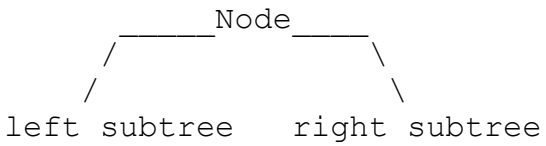
Q: What if every node contains negative value?
A: Then you should return the single node value that is the least negative.

Q: Does the maximum path have to go through the root node?
A: Not necessarily. For example, the below tree yield 6 as the maximum path sum and does not go through root.



Hint:

Anytime when you found that doing top down approach uses a lot of repeated calculation, bottom up approach usually is able to be more efficient.



Try the bottom up approach. At each node, the potential maximum path could be one of these cases:

i. max(left subtree) + node

ii. max(right subtree) + node

iii. max(left subtree) + max(right subtree) + node

iv. the node itself

Then, we need to return the maximum path sum that goes through this node and to either one of its left or right subtree to its parent. There's a little trick here: If this maximum happens to be negative, we should return 0, which means: "Do not include this subtree as part of the maximum path of the parent node", which greatly simplifies our code.

```
private int maxSum;

public int maxPathSum(TreeNode root) {
   maxSum = Integer.MIN_VALUE;
   findMax(root);
   return maxSum;
}

private int findMax(TreeNode p) {
   if (p == null) return 0;
   int left = findMax(p.left);
   int right = findMax(p.right);
   maxSum = Math.max(p.val + left + right, maxSum);
   int ret = p.val + Math.max(left, right);
   return ret > 0 ? ret : 0;
}
```

32. Binary Tree Upside Down

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

Question:

Given a binary tree where all the right nodes are either leaf nodes with a sibling (a left node that shares the same parent node) or empty, flip it upside down and turn it into a tree where the original right nodes turned into left leaf nodes. Return the new root.

Solution:

At each node you want to assign:

```
p.left = parent.right;
p.right = parent;
```

Top down approach:

We need to be very careful when reassigning current node's left and right children. Besides making a copy of the parent node, you would also need to make a copy of the parent's right child too. The reason is the current node becomes the parent node in the next iteration.

```
public TreeNode UpsideDownBinaryTree(TreeNode root) {
   TreeNode p = root, parent = null, parentRight = null;
   while (p != null) {
      TreeNode left = p.left;
      p.left = parentRight;
      parentRight = p.right;
      p.right = parent;
      parent = p;
      p = left;
   }
   return parent;
}
```

The above code is actually very similar to the algorithm in reversing a linked list.

Bottom up approach:

Although the code for the top-down approach seems concise, it is actually subtle and there are a lot of hidden traps if you are not careful. The other approach is thinking recursively in a bottom-up fashion. If we reassign the bottom-level nodes before the upper ones, we won't have to make copies and worry about overwriting something. We know the new root will be the left-most leaf node, so we begin the reassignment here.

```
public TreeNode UpsideDownBinaryTree(TreeNode root) {
   return dfsBottomUp(root, null);
}

private TreeNode dfsBottomUp(TreeNode p, TreeNode parent) {
   if (p == null) return parent;
   TreeNode root = dfsBottomUp(p.left, p);
   p.left = (parent == null) ? parent : parent.right;
   p.right = parent;
   return root;
}
```

Chapter 5: Bit Manipulation

33. Single Number

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

Question:

Given an array of integers, every element appears twice except for one. Find that single one.

Example Questions Candidate Might Ask:

Q: Does the array contain both positive and negative integers?
A: Yes.

Q: Could any element appear more than twice?
A: No.

Solution:

We could use a map to keep track of the number of times an element appears. In a second pass, we could extract the single number by consulting the hash map. As a hash map provides constant time lookup, the overall complexity is O(n), where n is the total number of elements.

```
public int singleNumber(int[] A) {
   Map<Integer, Integer> map = new HashMap<>();
   for (int x : A) {
      int count = map.containsKey(x) ? map.get(x) : 0;
      map.put(x, count + 1);
   }
   for (int x : A) {
      if (map.get(x) == 1) {
         return x;
      }
   }
   throw new IllegalArgumentException("No single element");
}
```

Although the map approach works, we are not taking advantage of the "every elements appears twice except one" property. Could we do better in one pass?

How about inserting the elements into a set instead? If an element already exists, we discard the element from the set knowing that it will not appear again. After the first pass, the set must contain only the single element.

```
public int singleNumber(int[] A) {
   Set<Integer> set = new HashSet<>();
   for (int x : A) {
      if (set.contains(x)) {
         set.remove(x);
      } else {
         set.add(x);
      }
   }
   return set.iterator().next();
}
```

The set is pretty efficient and runs in one pass. However, it uses extra space of O(n).

XOR-ing a number with itself is zero. If we XOR all numbers together, it would effectively cancel out all elements that appear twice leaving us with only the single number. As the XOR operation is both commutative and associative, the order in how you XOR them does not matter.

```
public int singleNumber(int[] A) {
   int num = 0;
   for (int x : A) {
      num ^= x;
   }
   return num;
}
```

Further Thoughts:

Let us change the question a little: “If every element appears even number of times except for one element that appears odd number of times, find that one element”, would the XOR approach work? Why?

34. Single Number II

Code it now: https://oj.leetcode.com/problems/single-number-ii/ Difficulty: Hard, Frequency: Medium

Question:

Given an array of integers, every element appears *three* times except for one. Find that single one.

Note:

Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory?

Solution:

To solve this problem using only constant space, you have to rethink how the numbers are being represented in computers – using bits.

If you sum the i^{th} bit of all numbers and mod 3, it must be either 0 or 1 due to the constraint of this problem where each number must appear either three times or once. This will be the i^{th} bit of that "single number".

A straightforward implementation is to use an array of size 32 to keep track of the total count of i^{th} bit.

```
int singleNumber(int A[], int n) {
    int count[32] = {0};
    int result = 0;
    for (int i = 0; i < 32; i++) {
        for (int j = 0; j < n; j++) {
            if ((A[j] >> i) & 1) {
                count[i]++;
            }
        }
        result |= ((count[i] % 3) << i);
    }
    return result;
}
```

We can improve this based on the previous solution using three bitmask variables:

1. ones as a bitmask to represent the i^{th} bit had appeared once.
2. twos as a bitmask to represent the i^{th} bit had appeared twice.
3. threes as a bitmask to represent the i^{th} bit had appeared three times.

When the i^{th} bit had appeared for the third time, clear the i^{th} bit of both ones and twos to 0. The final answer will be the value of ones.

```
int singleNumber(int A[], int n) {
    int ones = 0, twos = 0, threes = 0;
    for (int i = 0; i < n; i++) {
        twos |= ones & A[i];
        ones ^= A[i];
        threes = ones & twos;
        ones &= ~threes;
        twos &= ~threes;
    }
    return ones;
}
```

Further Thoughts:

If we extend the problem to:

> Given an array of integers, every element appears *k* times except for one. Find that single one which appears *l* times.

How would you solve it?

Please see the excellent answer by @ranmocy in LeetCode Discuss:

https://oj.leetcode.com/discuss/857/constant-space-solution?show=2542#a2542

Chapter 6: Misc

35. Spiral Matrix

Code it now: https://oj.leetcode.com/problems/spiral-matrix/ Difficulty: Medium, Frequency: Medium

Question:

Given a matrix of m ×n elements (m rows, n columns), return all elements of the matrix in spiral order.

For example, given the following matrix:

```
[
  [ 1, 2, 3 ],
  [ 4, 5, 6 ],
  [ 7, 8, 9 ]
]
```

You should return [1,2,3,6,9,8,7,4,5].

Solution:

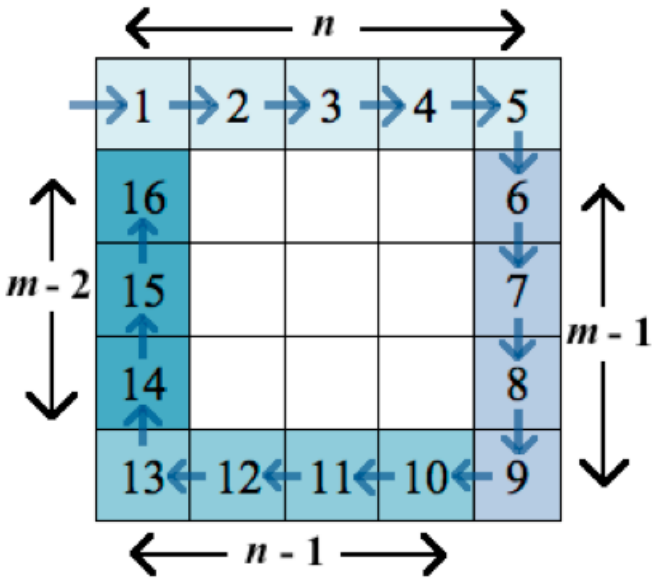


Figure 3: A m ×n matrix. The arrows show the direction of traversal in spiral order.

We simulate walking the matrix from the top left corner in a spiral manner. In the outmost level, we traverse n steps right, m – 1 steps down, n – 1 steps left, and m – 2 steps up, then we continue traverse into its next inner level.

As the traversal spiral toward the matrix's center, we stop by determining if we have reached the "center". However, defining the "center" is difficult since the matrix is not

necessarily a square. Think of edge cases such as 1×1, 1×10 and 10×1 matrices, where is the "center"? These cases had to be dealt separately and are messy.

A cleaner solution is to keep track of our current position and the number of steps in both horizontal and vertical directions. As we change direction we decrement the steps in that direction. When the number of steps in a direction becomes zero, we know that we have finished traversing the entire matrix.

```
public List<Integer> spiralOrder(int[][] matrix) {
   List<Integer> elements = new ArrayList<>();
   if (matrix.length == 0) return elements;
   int m = matrix.length, n = matrix[0].length;
   int row = 0, col = -1;
   while (true) {
      for (int i = 0; i < n; i++) {
         elements.add(matrix[row][++col]);
      }
      if (--m == 0) break;
      for (int i = 0; i < m; i++) {
         elements.add(matrix[++row][col]);
      }
      if (--n == 0) break;
      for (int i = 0; i < n; i++) {
         elements.add(matrix[row][--col]);
      }
      if (--m == 0) break;
      for (int i = 0; i < m; i++) {
         elements.add(matrix[--row][col]);
      }
      if (--n == 0) break;
   }
   return elements;
}
```

36. Integer to Roman

Code it now: https://oj.leetcode.com/problems/integer-to-roman/ Difficulty: Medium, Frequency: Low

Question:

Given an integer, convert it to a roman numeral.

Input is guaranteed to be within the range from 1 to 3999.

Hint:

What is the range of the numbers?

Solution:

Roman Literal	Decimal
I	**1**
V	**5**
X	**10**
L	**50**
C	**100**
D	**500**
M	**1000**

Table 1: Roman literals and its decimal representations.

First, let's understand how to read roman numerals. The rule of roman numerals is simple: Symbols are placed from left to right starting with the largest, and we add the values according to the additive notation. However, there is an exception to avoid four symbols being repeated in succession, also known as the subtractive notation.

The additive notation:

We combine the symbols and add the values. For example, III is three ones, which is 3. Another example XV means ten followed by a five, which is 15.

The subtractive notation:

Four characters are avoided being repeated in succession (such as IIII). Instead, the symbol I could appear before V and X to signify 4 (IV) and 9 (IX) respectively. Using the same pattern, we observe that X could appear before L and C to signify 40 (XL) and 90 (XC) respectively. The same pattern could be applied to C that is placed before D and M.

With our understanding of roman numerals, we have to decide how to extract the digits from the integer. Should we extract from right to left (from the least significant digit) or from left to right (from the most significant digit)?

If digits are extracted from right to left, we have to append the symbols in reversed order. Extracting digits from left to right seem more natural. It is also slightly trickier but not if we know the maximum number of digits could the number have in advanced, which we do – The number is within the range from 1 to 3999.

Using the additive notation, we convert to roman numerals by breaking it so each chunk can be represented by the symbol entity. For example, 11 = 10 + 1 = “X” + “I”. Similarly, 6 = 5 + 1 = “V” + “I”. Let’s take a look of an example which uses the subtractive notation: 49 = 40 + 9 = “XL” + “IX”. Note that we treat “XL” and “IX” as one single entity to avoid dealing with these special cases to greatly simplify the code.

```
private static final int[] values = {
   1000, 900, 500, 400,
   100,  90,  50,  40,
   10,   9,   5,   4,
   1
};
private static final String[] symbols = {
   "M", "CM", "D", "CD",
   "C", "XC", "L", "XL",
   "X", "IX", "V", "IV",
   "I"
};

public String intToRoman(int num) {
   StringBuilder roman = new StringBuilder();
   int i = 0;
   while (num > 0) {
      int k = num / values[i];
      for (int j = 0; j < k; j++) {
         roman.append(symbols[i]);
         num -= values[i];
      }
      i++;
   }
   return roman.toString();
}
```

Follow up:

Implement Roman to Integer. See Question [37. Roman to Integer].

37. Roman to Integer

Code it now: https://oj.leetcode.com/problems/roman-to-integer/ Difficulty: Medium, Frequency: Low

Question:

Given a roman numeral, convert it to an integer.

Input is guaranteed to be within the range from 1 to 3999.

Solution:

Roman Literal	Decimal
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

Table 2: Roman literals and its decimal representations.

Let's work through some examples. Assume the input is "VII", using the [**additive notation**], we could simply add up each roman literal, 'V' + 'I' + 'I' = 5 + 1 + 1 = 7.

Now let's look at another example input "IV". Now we need to use the [**subtractive notation**]. We first look at 'I', and we add 1 to it. Then we look at 'V' and since a *smaller* roman literal 'I' appears before it, we need to subtract 'I' from 'V'. Remember that we already added another 'I' before it, so we need to subtract a total of two one's from it.

Below is a more complex example that involves both additive and subtractive notation: "MXCVI".

Roman literals from left to right	Accumulated total
M	**1000**
M**X**	1000 + **10** = 1010
MX**C**	1010 + (**100 – 2 * 10**) = 1010 + 80 = 1090
MXC**V**	1090 + **5** = 1095
MXCV**I**	1095 + **1** = 1096

Table 3: Step by step calculation of roman numeral "MXCVI".

```
private Map<Character, Integer> map =
      new HashMap<Character, Integer>() {{
            put('I', 1);  put('V', 5);   put('X', 10);
            put('L', 50); put('C', 100); put('D', 500);
            put('M', 1000);
      }};

public int romanToInt(String s) {
   int prev = 0, total = 0;
   for (char c : s.toCharArray()) {
      int curr = map.get(c);
      total += (curr > prev) ? (curr - 2 * prev) : curr;
      prev = curr;
   }
   return total;
}
```

38. Clone graph

Code it now: https://oj.leetcode.com/problems/clone-graph/ Difficulty: Medium, Frequency: Medium

Question:

Clone an undirected graph. Each node in the graph contains a label and a list of its neighbors.

Solution:

There are two main ways to traverse a graph: *breadth-first* or *depth-first*. Let's try the depth-first approach first, which is a recursion algorithm. Then we will look at the breadth-first approach, which is an iterative algorithm that uses a queue.

O(*n*) runtime, *O*(*n*) space – Depth-first traversal:

A graph is simply represented by a graph node that serves as its starting point. In fact, the starting point could be any other graph nodes and it does not affect the cloning algorithm.

As each of its neighbors is a graph node too, we could recursively clone each of its neighbors and assign it to each neighbor of the cloned node. We can easily see that it is doing a depth-first traversal of each node.

Note that the graph could contain cycles, for example a node could have a neighbor that points back to it. Therefore, we should use a map that records each node's copy to avoid infinite recursion.

```
public UndirectedGraphNode cloneGraph(UndirectedGraphNode graph) {
   if (graph == null) return null;
   Map<UndirectedGraphNode, UndirectedGraphNode> map = new HashMap<>();
   return DFS(graph, map);
}

private UndirectedGraphNode DFS(UndirectedGraphNode graph,
      Map<UndirectedGraphNode, UndirectedGraphNode> map) {
   if (map.containsKey(graph)) {
      return map.get(graph);
   }
   UndirectedGraphNode graphCopy = new UndirectedGraphNode(graph.label);
   map.put(graph, graphCopy);
   for (UndirectedGraphNode neighbor : graph.neighbors) {
      graphCopy.neighbors.add(DFS(neighbor, map));
   }
   return graphCopy;
}
```

O(*n*) runtime, *O*(*n*) space – Breadth-first traversal:

How does the breadth-first traversal works? Easy, as we pop a node off the queue, we copy each of its neighbors, and push them to the queue.

A straight forward breadth-first traversal seemed to work. But some details are still missing. For example, how do we connect the nodes of the cloned graph?

The fact that B can traverse back to A implies that the graph may contain a cycle. You must take extra care to handle this case or else your code could have an infinite loop.

Let's analyze this further by using the below example:

Figure 4: A simple graph

Assume that the starting point of the graph is A. First, you make a copy of node A (A2), and found that A has only one neighbor B. You make a copy of B (B2) and connect A2→B2 by pushing B2 as A2's neighbor. Next, you find that B has A as neighbor, which you have already made a copy of. Here, we have to be careful not to make a copy of A again, but to connect B2→A2 by pushing A2 as B2's neighbor. But, how do we know if a node has already been copied?

Easy, we could use a hash table! As we copy a node, we insert it into the table. If we later find that one of a node's neighbors is already in the table, we do not make a copy of that neighbor, but to push its neighbor's copy to its copy instead. Therefore, the hash table would need to store a mapping of key-value pairs, where the key is a node in the original graph and its value is the node's copy.

```
public UndirectedGraphNode cloneGraph(UndirectedGraphNode graph) {
   if (graph == null) return null;
   Map<UndirectedGraphNode, UndirectedGraphNode> map = new HashMap<>();
   Queue<UndirectedGraphNode> q = new LinkedList<>();
   q.add(graph);
   UndirectedGraphNode graphCopy = new UndirectedGraphNode(graph.label);
   map.put(graph, graphCopy);
   while (!q.isEmpty()) {
      UndirectedGraphNode node = q.poll();
      for (UndirectedGraphNode neighbor : node.neighbors) {
         if (map.containsKey(neighbor)) {
            map.get(node).neighbors.add(map.get(neighbor));
         } else {
            UndirectedGraphNode neighborCopy =
                  new UndirectedGraphNode(neighbor.label);
            map.get(node).neighbors.add(neighborCopy);
            map.put(neighbor, neighborCopy);
            q.add(neighbor);
         }
      }
   }
   return graphCopy;
}
```

Chapter 7: Stack

39. Min Stack

Code it now: https://oj.leetcode.com/problems/min-stack/ Difficulty: Easy, Frequency: N/A

Question:

Design a stack that supports push, pop, top, and retrieving the minimum element in *constant* time.

- *push*(*x*) – Push element *x* onto stack.
- *pop*() – Removes the element on top of the stack.
- *top*() – Get the top element.
- *getMin*() – Retrieve the minimum element in the stack.

Hints:

- Consider space-time tradeoff. How would you keep track of the minimums using extra space?
- Make sure to consider duplicate elements.

Solution:

***O*(*n*) runtime, *O*(*n*) space – Extra stack:**

Consider using an extra stack to keep track of the current minimum value. During the push operation we choose the new element or the current minimum, whichever that is smaller to push onto the min stack.

For the pop operation, we would pop from both stacks. *getMin*() is then reflected by the top element of min stack.

To illustrate this idea, we push the elements 1, 4, 3, 0, 3 in that order.

Main stack	Min stack
3	0
0	0
3	1
4	1
1	1

After popping two elements from the stack it becomes:

Main stack	Min stack
3	1
4	1
1	1

O(*n*) runtime, *O*(*n*) space – Minor space optimization:

If a new element is larger than the current minimum, we do not need to push it on to the min stack. When we perform the pop operation, check if the popped element is the same as the current minimum. If it is, pop it off the min stack too.

```
class MinStack {
   private Stack<Integer> stack = new Stack<>();
   private Stack<Integer> minStack = new Stack<>();

   public void push(int x) {
      stack.push(x);
      if (minStack.isEmpty() || x <= minStack.peek()) {
         minStack.push(x);
      }
   }

   public void pop() {
      if (stack.pop().equals(minStack.peek())) minStack.pop();
   }

   public int top() {
      return stack.peek();
   }

   public int getMin() {
      return minStack.peek();
   }
}
```

40. Evaluate Reverse Polish Notation

Code it now: https://oj.leetcode.com/problems/evaluate-reverse-polish-notation/ Difficulty: Medium, Frequency: Low

Question:

Evaluate the value of an arithmetic expression in Reverse Polish Notation.

Valid operators are +, -, *, /. Each operand may be an integer or another expression.

Some examples:

["2", "1", "+", "3", "*"] -> ((2 + 1) * 3) -> 9

["4", "13", "5", "/", "+"] -> (4 + (13 / 5)) -> 6

Example Questions Candidate Might Ask:

Q: Is an empty array a valid input?
A: No.

Solution:

The Reverse Polish Notation (RPN) is also known as the postfix notation, because each operator appears *after* its operands. For example, the infix notation "3 + 4" is expressed as "3 4 +" in RPN.

The brute force approach:

We look for the simplest RPN sequence that could be evaluated immediately, that is: Two successive operands followed by an operator such as "4 2 +". We replace this sequence with the expression's value and repeat until there is only one operand left.

For example,

["4", **"13", "5", "/"**, "+"]

➔ **["4", "2", "+"]**

➔ ["6"]

How would you do the replacement? You could do it in-place with the input array, but it would result in quadratic runtime as elements have to be shifted every time a replacement occurs.

A workaround is to copy the input array to a doubly linked list. The replacement is efficient and the next scan begins with the replaced value's previous element. This results in an algorithm with linear runtime and linear space. Although this works, the implementation is complex and is far from ideal in an interview session.

The optimal approach:

Observe that every time we see an operator, we need to evaluate the last two operands. Stack fits perfectly as it is a Last-In-First-Out (LIFO) data structure.

We evaluate the expression left-to-right and push operands onto the stack until we encounter an operator, which we pop the top two values from the stack. We then evaluate the operator, with the values as arguments and push the result back onto the stack.

For example, the infix expression "8 – ((1 + 2) * 2)" in RPN is:

8 1 2 + 2 * –

Input	Operation	Stack	Notes
8	Push operand	8	
1	Push operand	1 8	
2	Push operand	2 1 8	
+	Add	3 8	Pop two values 1, 2 and push result 3
2	Push operand	2 3 8	
*	Multiply	6 8	Pop two values 3, 2 and push result 6
–	Subtract	2	Pop two values 8, 6 and push result 2

After the algorithm finishes, the stack contains only one value which is the RPN expression's result; in this case, 2.

```
private static final Set<String> OPERATORS =
      new HashSet<>(Arrays.asList("+", "-", "*", "/"));

public int evalRPN(String[] tokens) {
   Stack<Integer> stack = new Stack<>();
   for (String token : tokens) {
      if (OPERATORS.contains(token)) {
         int y = stack.pop();
         int x = stack.pop();
         stack.push(eval(x, y, token));
      } else {
         stack.push(Integer.parseInt(token));
      }
   }
   return stack.pop();
}

private int eval(int x, int y, String operator) {
   switch (operator) {
      case "+": return x + y;
      case "-": return x - y;
      case "*": return x * y;
      default:  return x / y;
   }
}
```

Further Thoughts:

The above code contains duplication. For example, if we decide to add a new operator, we would need to update the code in two places – in the set's initialization and the switch statement. Could you refactor the code so it is more extensible?

You are probably not expected to write this refactored code during an interview session. However, it will make you a stronger candidate if you could make this observation and point this out, as it shows to the interviewer that you care about clean code.

In Java, create an interface called Operator and map each operator string to an implementation of the Operator interface. For other languages such as C++, each operator will be mapped to a function pointer instead.

```
interface Operator {
   int eval(int x, int y);
}

private static final Map<String, Operator> OPERATORS =
      new HashMap<String, Operator>() {{
            put("+", new Operator() {
               public int eval(int x, int y) { return x + y; }
            });
            put("-", new Operator() {
               public int eval(int x, int y) { return x - y; }
            });
            put("*", new Operator() {
               public int eval(int x, int y) { return x * y; }
            });
            put("/", new Operator() {
               public int eval(int x, int y) { return x / y; }
            });
      }};

public int evalRPN(String[] tokens) {
   Stack<Integer> stack = new Stack<>();
   for (String token : tokens) {
      if (OPERATORS.containsKey(token)) {
         int y = stack.pop();
         int x = stack.pop();
         stack.push(OPERATORS.get(token).eval(x, y));
      } else {
         stack.push(Integer.parseInt(token));
      }
   }
   return stack.pop();
}
```

41. Valid Parentheses

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

Question:

Given a string containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.

The brackets must close in the correct order, "()" and "()[]{}" are all valid but "(]" and "([)]" are not.

Example Questions Candidate Might Ask:

Q: Is the empty string valid?
A: Yes.

Solution:

To validate the parentheses, we need to match each closing parenthesis with its opening counterpart. A Last-In-First-Out (LIFO) data structure such as stack is the perfect fit.

As we see an opening parenthesis, we push it onto the stack. On the other hand, when we encounter a closing parenthesis, we pop the last inserted opening parenthesis from the stack and check if the pair is a valid match.

It would be wise to avoid writing multiple if statements when matching parentheses, as your interviewer may think that you are writing sloppy code. You could use a map, which is more maintainable.

```
private static final Map<Character, Character> map =
      new HashMap<Character, Character>() {{
            put('(', ')');
            put('{', '}');
            put('[', ']');
      }};

public boolean isValid(String s) {
   Stack<Character> stack = new Stack<>();
   for (char c : s.toCharArray()) {
      if (map.containsKey(c)) {
         stack.push(c);
      } else if (stack.isEmpty() || map.get(stack.pop()) != c) {
         return false;
      }
   }
   return stack.isEmpty();
}
```

Chapter 8: Dynamic Programming

42. Climbing Stairs

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

Question:

You are climbing a staircase. It takes *n* steps to reach to the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

Solution:

***O*(*n*) runtime, *O*(1) space – Dynamic programming:**

This is a classic Dynamic Programming problem.

Define:

f(*n*) = number of ways you can climb to the nth step.

To reach to the n^{th} step, you have only two choices:

1. Advance one step from the $n - 1^{th}$ step.
2. Advance two steps from the $n - 2^{th}$ step.

Therefore, *f*(*n*) = *f*(*n* – 1) + *f*(*n* – 2), which is the exact same recurrence formula defined by the Fibonacci sequence (with different base cases, though).

Set base cases *f*(1) = 1, *f*(2) = 2 and you are almost done.

Now, we could calculate *f*(n) easily by storing previous values in an one dimension array and work our way up to n. Heck, we can even optimize this further by storing just the previous two values.

```
public int climbStairs(int n) {
    int p = 1, q = 1;
    for (int i = 2; i <= n; i++) {
        int temp = q;
        q += p;
        p = temp;
    }
    return q;
}
```

Combinatorics:

Interestingly, this problem could also be solved using combinatorics.

Warning: Math-y stuff ahead, feel free to skip this section if you are not interested.

For example, let's assume $n = 6$.

Let:

x = number of 1's,
y = number of 2's.

We could reach the top using one of the four combinations below:

x	*y*	
6	0	=> 1) Six single steps.
4	1	=> 2) Four single steps and one double step.
2	2	=> 3) Two single steps and two double steps.
0	3	=> 4) Three double steps.

For the first combination pair $(x,y) = (6,0)$, there's obviously only one way of arranging six single steps.

For the second combination pair (4,1), there's five ways of arranging (think of it as slotting the double step between the single steps).

Similarly, there are six ways C(4,2) and one way $\binom{3}{3}$ of arranging the third and fourth combination pairs respectively.

Generally, for pair (x,y), there are a total of $\binom{x+y}{y} = \frac{(x+y)!}{x!y!}$ ways of arranging the 1's and 2's.

The total number of possible ways is the sum of all individual terms,

$f(6) = 1 + 5 + 6 + 1 = 13$.

Generalizing for all n's (including odd n),

$$f(\mathrm{n}) = \binom{n}{0} + \binom{n-1}{1} + \binom{n-2}{2} + \cdots + \binom{ceil\left(\frac{n}{2}\right)}{floor\left(\frac{n}{2}\right)}$$

43. Unique Paths

Code it now: https://oj.leetcode.com/problems/unique-paths/ Difficulty: Medium, Frequency: Low

Question:

A robot is located at the top-left corner of a $m \times n$ grid (marked 'Start' in the diagram below). The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below). How many possible unique paths are there?

Figure 5: The grid above is 7×3, and is used to illustrate the problem.

$O\left(\binom{m+n}{m}\right)$ runtime, $O(m+n)$ space – Backtracking:

The most direct way is to write code that traverses each possible path, which can be done using backtracking. When you reach *row* = *m* and *col* = *n*, you know you've reached the bottom-right corner, and there is one additional unique path to it. However, when you reach *row* > *m* or *col* > *n*, then it's an invalid path and you should stop traversing. For any grid at *row* = *r* and *col* = *c*, you have two choices: Traverse to the right or traverse to the bottom. Therefore, the total unique paths at grid (*r*, *c*) are equal to the sum of total unique paths from the grid to the right and the grid below.

Deriving the runtime complexity is slightly tricky. Observe that the robot must go right exactly *m* times and go down exactly *n* times. Assume that the right movement is 0 and the down movement is 1. We could then represent the robot's path as a binary string of length = *m* + *n*, where the string contains *m* zeros and *n* ones. Since the backtracking algorithm is just trying to explore all possibilities, its runtime complexity is equivalent to the total permutations of a binary string that contains *m* zeros and *n* ones, which is $\binom{m+n}{m}$.

On the other hand, the space complexity is $O(m+n)$ due to the recursion that goes at most *m* + *n* level deep.

Below is the backtracking code in just five lines of code:

```
private int backtrack(int r, int c, int m, int n) {
   if (r == m - 1 && c == n - 1)
      return 1;
   if (r >= m || c >= n)
      return 0;

   return backtrack(r + 1, c, m, n) + backtrack(r, c + 1, m, n);
}

public int uniquePaths(int m, int n) {
   return backtrack(0, 0, m, n);
}
```

Improved Backtracking Solution using Memoization:

Although the above backtracking solution is easy to code, it is very inefficient in the sense that it recalculates the same solution for a grid over and over again. By caching the results, we prevent recalculation and only calculate when necessary. Here, we are using a dynamic programming (DP) technique called memoization.

```
private int backtrack(int r, int c, int m, int n, int[][] mat) {
   if (r == m - 1 && c == n - 1)
      return 1;
   if (r >= m || c >= n)
      return 0;

   if (mat[r + 1][c] == -1)
      mat[r + 1][c] = backtrack(r + 1, c, m, n, mat);
   if (mat[r][c+1] == -1)
      mat[r][c + 1] = backtrack(r, c + 1, m, n, mat);

   return mat[r + 1][c] + mat[r][c + 1];
}

public int uniquePaths(int m, int n) {
   int[][] mat = new int[m + 1][n + 1];
   for (int i = 0; i < m + 1; i++) {
      for (int j = 0; j < n + 1; j++) {
         mat[i][j] = -1;
      }
   }
   return backtrack(0, 0, m, n, mat);
}
```

$O(mn)$ runtime, $O(mn)$ space – Bottom-up dynamic programming:

If you notice closely, the above DP solution is using a *top-down* approach. Now let's try a *bottom-up* approach. Remember this important relationship that is necessary for this DP solution to work:

The total unique paths at grid (r, c) are equal to the sum of total unique paths from grid to the right (r, $c + 1$) and the grid below ($r + 1$, c).

How can this relationship help us solve the problem? We observe that all grids of the bottom edge and right edge must all have only one unique path to the bottom-right

corner. Using this as the base case, we can build our way up to our solution at grid (1, 1) using the relationship above.

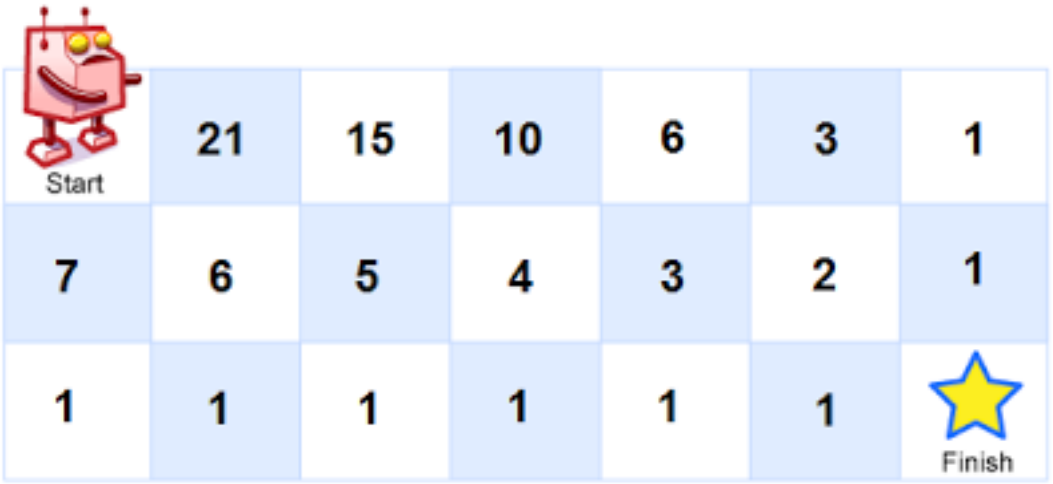


Figure 6: The total unique paths at grid (r, c) are equal to the sum of total unique paths from grid to the right (r, c + 1) and the grid below (r + 1, c).

```
public int uniquePaths(int m, int n) {
    int[][] mat = new int[m + 1][n + 1];
    mat[m - 1][n] = 1;
    for (int r = m - 1; r >= 0; r--) {
        for (int c = n - 1; c >= 0; c--) {
            mat[r][c] = mat[r + 1][c] + mat[r][c + 1];
        }
    }
    return mat[0][0];
}
```

Combinatorial Solution:

It turns out this problem could be solved using combinatorics, which no doubt would be the most efficient solution. In order to see it as a combinatorial problem, there are some necessary observations. Look at the 7 ×3 sample grid in the picture above. Notice that no matter how you traverse the grids, you always traverse a total of 8 steps. To be more exact, you always have to choose 6 steps to the right (R) and 2 steps to the bottom (B). Therefore, the problem can be transformed to a question of how many ways can you choose 6R‘s and 2B‘s in these 8 steps. The answer is $\binom{8}{2}$ or $\binom{8}{6}$. Therefore, the general solution for an $m \times n$ grid is $\binom{m+n-2}{m-1}$.

Further Thoughts:

Now consider if some obstacles are added to the grids marked as ‘X’. How many unique paths would there be? A combinatorial solution is difficult to obtain, but the DP solution can be modified easily to accommodate this constraint. See Question [44. Unique Paths II].

44. Unique Paths II

Code it now: https://oj.leetcode.com/problems/unique-paths-ii/ Difficulty: Medium, Frequency: High

Question:

Similar to Question [43. Unique Paths], but now consider if some obstacles are added to the grids. How many unique paths would there be?

An obstacle and empty space are marked as 1 and 0 respectively in the grid.

For example,

There is one obstacle in the middle of a 3×3 grid as illustrated below.

```
[
  [0,0,0],
  [0,1,0],
  [0,0,0]
]
```

The total number of unique paths is 2.

Solution:

***O*(*mn*) runtime, *O*(*mn*) space – Dynamic programming:**

It turns out to be really easy to extend from the [Bottom-up dynamic programming] approach above. Just set the total paths to 0 when you encounter an obstacle.

```
public int uniquePathsWithObstacles(int[][] obstacleGrid) {
   int m = obstacleGrid.length;
   if (m == 0) return 0;
   int n = obstacleGrid[0].length;
   int[][] mat = new int[m + 1][n + 1];
   mat[m - 1][n] = 1;
   for (int r = m - 1; r >= 0; r--) {
      for (int c = n - 1; c >= 0; c--) {
         mat[r][c] = (obstacleGrid[r][c] == 1) ? 0 : mat[r][c+1] + mat[r+1][c];
      }
   }
   return mat[0][0];
}
```

45. Maximum Sum Subarray

Code it now: https://oj.leetcode.com/problems/maximum-subarray/ Difficulty: Medium, Frequency: High

Question:

Find the contiguous subarray within an array (containing at least one number) that has the largest sum.

For example, given the array [2, 1, –3, 4, –1, 2, 1, –5, 4],

The contiguous array [4, –1, 2, 1] has the largest sum = 6.

Solution:

***O*(*n* log *n*) runtime, *O*(log *n*) stack space – Divide and Conquer:**

Assume we partition the array A into two smaller arrays S and T at the middle index, M. Then, $S = A_1 \ldots A_{M-1}$, and $T = A_{M+1} \ldots A_N$.

The contiguous subarray that has the largest sum could either:

i. Contain the middle element:
   a. The largest sum is the maximum suffix sum of $S + A_M$ + the maximum prefix sum of T.

ii. Does not contain the middle element:
   a. The largest sum is in S, which we could apply the same algorithm to S.
   b. The largest sum is in T, which we could apply the same algorithm to T.

```
public int maxSubArray(int[] A) {
   return maxSubArrayHelper(A, 0, A.length - 1);
}

private int maxSubArrayHelper(int[] A, int L, int R) {
   if (L > R) return Integer.MIN_VALUE;
   int M = (L + R) / 2;
   int leftAns = maxSubArrayHelper(A, L, M - 1);
   int rightAns = maxSubArrayHelper(A, M + 1, R);
   int lMaxSum = 0;
   int sum = 0;
   for (int i = M - 1; i >= L; i--) {
      sum += A[i];
      lMaxSum = Math.max(sum, lMaxSum);
   }
   int rMaxSum = 0;
   sum = 0;
   for (int i = M + 1; i <= R; i++) {
      sum += A[i];
      rMaxSum = Math.max(sum, rMaxSum);
   }
   return Math.max(lMaxSum + A[M] + rMaxSum, Math.max(leftAns, rightAns));
}
```

The runtime complexity could be expressed as $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$, which is $O(n \log n)$. We will not attempt to prove it here; you could read up any advanced algorithm textbooks to learn the proof.

$O(n)$ runtime, $O(1)$ space – Dynamic programming:

To devise a dynamic programming formula, let us assume that we are calculating the maximum sum of subarray that ends at a specific index.

Let us denote that:

f(*k*) = Maximum sum of subarray ending at index *k*.

Then,

f(*k*) = *max*(*f*(*k*-1) + A[*k*], A[*k*])

Using an array of size *n*, We could deduce the final answer by as *f*(*n* – 1), with the initial state of *f*(0) = *A*[0]. Since we only need to access its previous element at each step, two variables are sufficient. Notice the difference between the two: *maxEndingHere* and *maxSoFar*; the former is the maximum sum of subarray that *must* end at index *k*, while the latter is the global maximum subarray sum.

```
public int maxSubArray(int[] A) {
   int maxEndingHere = A[0], maxSoFar = A[0];
   for (int i = 1; i < A.length; i++) {
      maxEndingHere = Math.max(maxEndingHere + A[i], A[i]);
      maxSoFar = Math.max(maxEndingHere, maxSoFar);
   }
   return maxSoFar;
}
```

46. Maximum Product Subarray

Code it now: https://oj.leetcode.com/problems/maximum-product-subarray/ Difficulty: Medium, Frequency: High

Question:

Find the contiguous subarray within an array of integers that has the largest product. For example, given the array [2,3,-2,4], the contiguous subarray [2,3] has the largest product = 6.

Example Questions Candidate Might Ask:

Q: Could the subarray be empty?
A: No, the subarray must contain at least one number.

Solution:

This problem is very similar to Question [45. Maximum Sum Subarray]. There is a slight twist though. Besides keeping track of the largest product, we also need to keep track of the smallest product. Why? The smallest product, which is the largest in the negative sense could become the maximum when being multiplied by a negative number.

Let us denote that:

$f(k)$ = Largest product subarray, from index 0 up to k.

Similarly,

$g(k)$ = Smallest product subarray, from index 0 up to k.

Then,

$f(k) = max(f(k\text{-}1) * A[k], A[k], g(k\text{-}1) * A[k])$

$g(k) = min(g(k\text{-}1) * A[k], A[k], f(k\text{-}1) * A[k])$

There we have a dynamic programming formula. Using two arrays of size n, we could deduce the final answer as $f(n\text{-}1)$. Since we only need to access its previous elements at each step, two variables are sufficient.

```
public int maxProduct(int[] A) {
   assert A.length > 0;
   int max = A[0], min = A[0], maxAns = A[0];
   for (int i = 1; i < A.length; i++) {
      int mx = max, mn = min;
      max = Math.max(Math.max(A[i], mx * A[i]), mn * A[i]);
      min = Math.min(Math.min(A[i], mx * A[i]), mn * A[i]);
      maxAns = Math.max(max, maxAns);
   }
   return maxAns;
}
```

47. Coins in a Line

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

Question:

There are *n* coins in a line. (Assume *n* is even). Two players take turns to take a coin from one of the ends of the line until there are no more coins left. The player with the larger amount of money wins.

1. Would you rather go first or second? Does it matter?
2. Assume that you go first, describe an algorithm to compute the maximum amount of money you can win.

Figure 7: U.S. coins in various denominations in a line. Two players take turn to pick a coin from one of the ends until no more coins are left. Whoever with the larger amount of money wins.

Hints:

If you go first, is there a strategy you can follow which prevents you from losing? Try to consider how it matters when the number of coins is odd vs. even.

Solution:

Going first will guarantee that you will not lose. By following the strategy below, you will always win the game (or get a possible tie).

1. Count the sum of all coins that are odd-numbered. (Call this X)
2. Count the sum of all coins that are even-numbered. (Call this Y)
3. If X > Y, take the left-most coin first. Choose all odd-numbered coins in subsequent moves.
4. If X < Y, take the right-most coin first. Choose all even-numbered coins in subsequent moves.
5. If X == Y, you will guarantee to get a tie if you stick with taking only even-numbered/odd-numbered coins.

You might be wondering how you can always choose odd-numbered/even-numbered coins. Let me illustrate this using an example where you have 10 coins:

If you take the coin numbered 1 (the left-most coin), your opponent can only have the choice of taking coin numbered 2 or 10 (which are both even-numbered coins). On the other hand, if you choose to take the coin numbered 10 (the right-most coin), your opponent can only take coin numbered 1 or 9 (which are odd-numbered coins).

Notice that the total number of coins change from even to odd and vice-versa when player takes turn each time. Therefore, by going first and depending on the coin you choose, you are essentially forcing your opponent to take either only even-numbered or odd-numbered coins.

Now that you have found a non-losing strategy, could you compute the maximum amount of money you can win?

Hints:

One misconception is to think that the above non-losing strategy would generate the maximum amount of money as well. This is probably incorrect. Could you find a counter example? (You might need at least 6 coins to find a counter example).

Assume that you are finding the maximum amount of money in a certain range (ie, from coins numbered i to j, inclusive). Could you express it as a recursive formula? Find ways to make it as efficient as possible.

Solution for (2):

Although the simple strategy illustrated in Solution (1) guarantees you not to lose, it does not guarantee that it is optimal in any way.

Here, we use a good counter example to better see why this is so. Assume the coins are laid out as below:

{ 3, 2, 2, 3, 1, 2 }

Following our previous non-losing strategy, we would count the sum of odd-numbered coins, X = 3 + 2 + 1 = 6, and the sum of even-numbered coins, Y = 2 + 3 + 2 = 7. As Y > X, we would take the last coin first and end up winning with the total amount of 7 by taking only even-numbered coins.

However, let us try another way by taking the first coin (valued at 3, denote by (3)) instead. The opponent is left with two possible choices, the left coin (2) and the right coin (2), both valued at 2. No matter which coin the opponent chose, you can always take the other coin (2) next and the configuration of the coins becomes: { 2, 3, 1 }. Now, the coin in the middle (3) would be yours to keep for sure. Therefore, you win the game by a total amount of 3 + 2 + 3 = 8, which proves that the previous non-losing strategy is not necessarily optimal.

To solve this problem in an optimal way, we need to find efficient means in enumerating all possibilities. This is when Dynamic Programming (DP) kicks in and become so powerful that you start to feel magical.

First, we would need some observations to establish a recurrence relation, which is essential as our first step in solving DP problems.

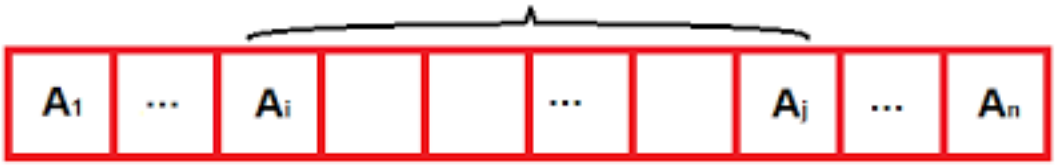


Figure 8: The remaining coins are { Ai ... Aj } and it is your turn. Let P(i, j) denotes the maximum amount of money you can get. Should you choose Ai or Aj?

Assume that P(i, j) denotes the maximum amount of money you can win when the remaining coins are { A_i, …, A_j }, and it is your turn now. You have two choices, either take Ai or Aj. First, let us focus on the case where you take Ai, so that the remaining coins become { A_{i+1} … A_j }. Since the opponent is as smart as you, he must choose the best way that yields the maximum for him, where the maximum amount he can get is denoted by P(i + 1, j).

Therefore, if you choose Ai, the maximum amount you can get is:

$$P_1 = \sum_{k=i}^{j} A_k - P(i+1, j)$$

Similarly, if you choose A_j, the maximum amount you can get is:

$$P_2 = \sum_{k=i}^{j} A_k - P(i, j-1)$$

Therefore,

$$P(i,j) = \max(P_1, P_2)$$
$$= max\left(\sum_{k=i}^{j} A_k - P(i+1,j), \sum_{k=i}^{j} A_k - P(i,j-1)\right)$$

In fact, we are able to simplify the above relation further to (Why?):

$$P(i,j) = \sum_{k=i}^{j} A_k - min\big(P(i+1,j), P(i,j-1)\big)$$

Although the above recurrence relation is easy to understand, we need to compute the value of $\sum_{k=i}^{j} A_k$ in each step, which is not very efficient. To avoid this problem, we can store values of $\sum_{k=i}^{j} A_k$ in a table and avoid re-computations by computing in a certain order. Try to figure this out by yourself. (Hint: You would first compute $P(1,1)$, $P(2,2)$, … $P(n, n)$ and work your way up).

A Better Solution:

There is another solution that does not rely on computing and storing results of $\sum_{k=i}^{j} A_k$, therefore is more efficient in terms of time and space. Let us rewind back to the case where you take Ai, and the remaining coins become { A_{i+1} … A_j }.

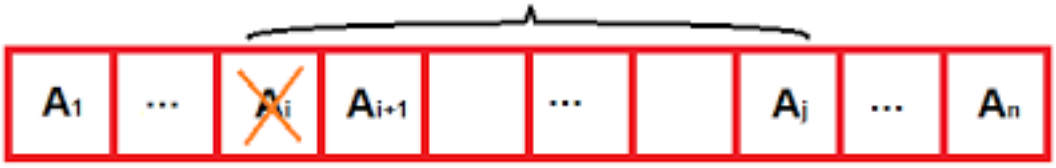


Figure 9: You took Ai from the coins { Ai … Aj }. The opponent will choose either Ai+1 or Aj. Which one would he choose?

Let us look one extra step ahead this time by considering the two coins the opponent will possibly take, A_{i+1} and A_j. If the opponent takes A_{i+1}, the remaining coins are { A_{i+2} … A_j }, which our maximum is denoted by $P(i + 2, j)$. On the other hand, if the opponent takes A_j, our maximum is $P(i + 1, j - 1)$. Since the opponent is as smart as you, he would have chosen the choice that yields the minimum amount to you.

Therefore, the maximum amount you can get when you choose Ai is:

$$P_1 = A_i + min\big(P(i+2,j), P(i+1,j-1)\big)$$

Similarly, the maximum amount you can get when you choose A_j is:

$$P_2 = A_j + min\big(P(i+1,j-1), P(i,j-2)\big)$$

Therefore,

$$P(i,j) = \max(P_1, P_2)$$
$$= max\left(A_i + min\big(P(i+2,j), P(i+1,j-1)\big), A_j + min\big(P(i+1,j-1), P(i,j-2)\big)\right)$$

Although the above recurrence relation could be implemented in few lines of code, its complexity is exponential. The reason is that each recursive call branches into a total of four separate recursive calls, and it could be *n* levels deep from the very first call). Memoization provides an efficient way by avoiding re-computations using intermediate results stored in a table. Below is the code which runs in $O(n^2)$ time and takes $O(n^2)$ space.

The code contains a function *printMoves* which prints out all the moves you and the opponent make (assuming both of you are taking the coins in an optimal way).

```
const int MAX_N = 100;

void printMoves(int P[][MAX_N], int A[], int N) {
  int sum1 = 0, sum2 = 0;
  int m = 0, n = N-1;
  bool myTurn = true;
  while (m <= n) {
    int P1 = P[m+1][n]; // If take A[m], opponent can get...
    int P2 = P[m][n-1]; // If take A[n]
    cout << (myTurn ? "I" : "You") << " take coin no. ";
    if (P1 <= P2) {
      cout << m+1 << " (" << A[m] << ")";
      m++;
    } else {
      cout << n+1 << " (" << A[n] << ")";
      n--;
    }
    cout << (myTurn ? ", " : ".\n");
    myTurn = !myTurn;
  }
  cout << "\nThe total amount of money (maximum) I get is " << P[0][N-1] << ".\n";
}

int maxMoney(int A[], int N) {
  int P[MAX_N][MAX_N] = {0};
  int a, b, c;
  for (int i = 0; i < N; i++) {
    for (int m = 0, n = i; n < N; m++, n++) {
      assert(m < N); assert(n < N);
      a = ((m+2 <= N-1)             ? P[m+2][n] : 0);
      b = ((m+1 <= N-1 && n-1 >= 0) ? P[m+1][n-1] : 0);
      c = ((n-2 >= 0)               ? P[m][n-2] : 0);
      P[m][n] = max(A[m] + min(a,b),
                    A[n] + min(b,c));
    }
  }
  printMoves(P, A, N);
  return P[0][N-1];
}
```

Further Thoughts:

Assume that your opponent is so dumb that you are able to manipulate him into choosing the coins you want him to choose. Now, what is the maximum possible amount of money you can win?

Chapter 9: Binary Search

48. Search Insert Position

Code it now: https://oj.leetcode.com/problems/search-insert-position/ Difficulty: Medium, Frequency: Low

Question:

Given a sorted array and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

You may assume no duplicates in the array.

Here are few examples.
[1,3,5,6], 5 → 2
[1,3,5,6], 2 → 1
[1,3,5,6], 7 → 4
[1,3,5,6], 0 → 0

Solution:

This problem is a direct application of Binary Search, as you can spot it easily by the keywords *sorted* and *finding target*. The requirements seem complex, but let's first start with something we're already familiar with – The raw binary search algorithm.

Let's start with defining two variables, L and R representing its lowest and highest inclusive indices that are searched, which are initialized to 0 and $n - 1$ respectively.

```
int L = 0, R = A.length - 1;
while (L < R) {
   int M = (L + R) / 2;
   // TODO: Implement conditional checks.
}
```

Code 1: Getting started with a Binary Search template.

Now, the key part of the binary search – We look at the middle element, and ask: "Is the middle element smaller than the target element?" If this is true, then it means all elements from L up to M inclusive could be excluded from the search. Otherwise, the middle element is greater or equal to the target element and that means all elements from $M + 1$ up to R could be excluded.

```
int L = 0, R = A.length - 1;
while (L < R) {
   int M = (L + R) / 2;
   if (A[M] < target) {
      L = M + 1;
   } else {
      R = M;
   }
}
```

Code 2: Filling out the key part of the Binary Search algorithm.

A good thing to verify your above binary search does not stuck in an infinite loop is to test with input containing two elements, e.g., [1,3] and test with *target* = 0 and 1. Here, our binary search works properly, but if we were to define *M* as the *upper* middle, that is: *M* = (*L* + *R* + 1) / 2, then it will stuck in an infinite loop.

We've now reached the final step. When the while loop ends, *L* must be equal to *R* and it is a valid index. Obviously, if *A*[*L*] is equal to target, we return *L*. If *A*[*L*] is greater than *target*, that means we are inserting *target* ***before*** *A*[*L*], so we return *L*. If *A*[*L*] is less than *target*, that means we insert *target* ***after*** A[L], so we return *L* + 1.

```
public int searchInsert(int[] A, int target) {
   int L = 0, R = A.length - 1;
   while (L < R) {
      int M = (L + R) / 2;
      if (A[M] < target) {
         L = M + 1;
      } else {
         R = M;
      }
   }
   return (A[L] < target) ? L + 1 : L;
}
```

49. Find Minimum in Sorted Rotated Array

Code it now: https://oj.leetcode.com/problems/find-minimum-in-rotated-sorted-array/ Difficulty: Medium, Frequency: High

Question:

Suppose a sorted array is rotated at some pivot unknown to you beforehand.

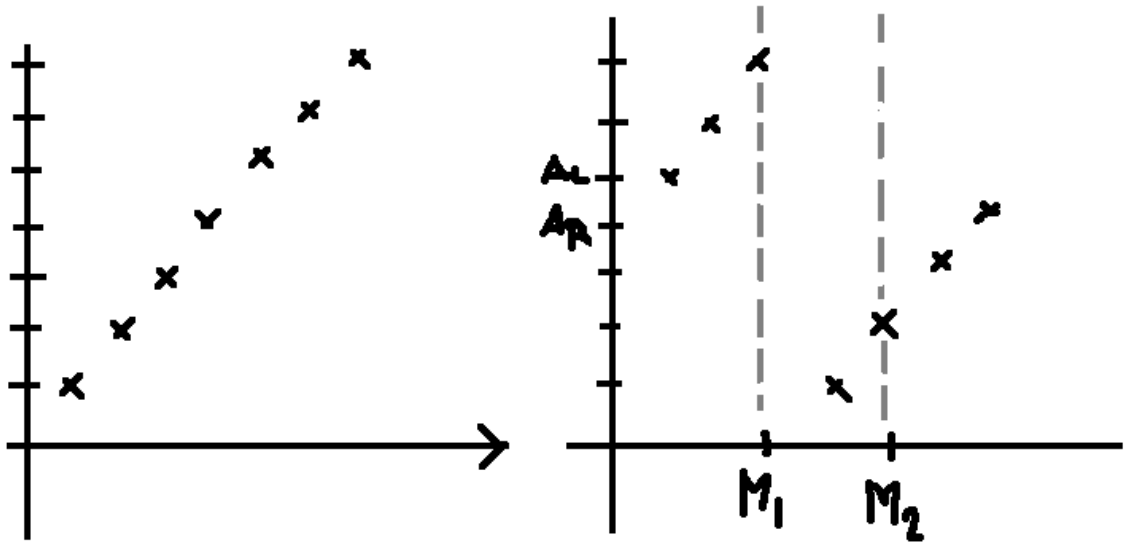
(i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).

Find the minimum element.

You may assume no duplicate exists in the array.

Solution:

The minimum is at A_i where $A_{i-1} > A_i$. Notice that if we subdivide the array into two, one will always be sorted, while the other contains the minimum.



Imagine we have an array [1,2,3,4,5,6,7] (See graph 1) which was being rotated 3 steps to the right [5,6,7,1,2,3,4] (See graph 2). Let's say we subdivide the array at point *k* to two subarrays $[A_L, A_{L+1}, \ldots, A_k]$, $[A_{k+1}, \ldots, A_R]$.

If the sorted array is not rotated, then $A_L < A_R$ then we could return A_L as the minimum immediately.

Otherwise for a sorted array that was rotated at least one step, A_L must always be greater than A_R.

Let's assume we choose M_1 as the dividing point. Since $A_{M1} > A_R$, we know that each element in $[A_L \ldots A_{M1}]$ is greater than A_R (Remember that $A_L > A_R$?). Therefore, the minimum value must locate in $[A_{M1+1} \ldots A_R]$.

On the other hand, let's assume we choose M_2 as the dividing point. Since $A_{M2} \leq A_R$, we know that each element in $[A_{M2+1} \ldots A_R]$ is greater than A_{M2}. Therefore, the minimum point must locate in $[A_L \ldots A_{M2}]$.

As we are discarding half of the elements at each step, the runtime complexity is $O(\log n)$.

To understand the correct terminating condition, we look at two elements. Let us choose the lower median as M = (L + R) / 2. Therefore, if there are two elements, it will choose A_L as the first element.

There are two cases for two elements:

A = [1,2]

B = [2,1]

For A, $1 < 2 \Rightarrow A_M < A_R$, and therefore it will set R = M => R = 0.

For B, $2 > 1 \Rightarrow A_M > A_R$, and therefore it will set L = M + 1 => L = 1.

Therefore, it is clear that when L == R, we have found the minimum element.

```
public int findMin(int[] A) {
   int L = 0, R = A.length - 1;
   while (L < R && A[L] >= A[R]) {
      int M = (L + R) / 2;
      if (A[M] > A[R]) {
         L = M + 1;
      } else {
         R = M;
      }
   }
   return A[L];
}
```

Further Thoughts:

If the rotated sorted array could contain duplicates? Is your algorithm still $O(\log n)$ in runtime complexity?

50. Find Minimum in Rotated Sorted Array II – with duplicates

Code it now: https://oj.leetcode.com/problems/find-minimum-in-rotated-sorted-array-ii/ Difficulty: Hard, Frequency: Medium

Question:

If the rotated sorted array could contain duplicates? Is your algorithm still $O(\log n)$ in runtime complexity?

Solution:

For case where $A_L == A_M == A_R$, the minimum could be on A_M's left or right side (eg, [1, 1, 1, 0, 1] or [1, 0, 1, 1, 1]). In this case, we could not discard either subarrays and therefore such worst case degenerates to the order of $O(n)$.

```
public int findMin(int[] A) {
   int L = 0, R = A.length - 1;
   while (L < R && A[L] >= A[R]) {
      int M = (L + R) / 2;
      if (A[M] > A[R]) {
         L = M + 1;
      } else if (A[M] < A[L]) {
         R = M;
      } else {   // A[L] == A[M] == A[R]
         L = L + 1;
      }
   }
   return A[L];
}
```