**1. Two Sum**

Given an array of integers nums and an integer target, return indices of the two numbers such that they add up to *target*.

You may assume that each input would have **exactly one solution**, and you may not use the same element twice.

You can return the answer in any order.

**Example 1:**

**Input:** nums = [2,7,11,15], target = 9

**Output:** [0,1]

**Output:** Because nums[0] + nums[1] == 9, we return [0, 1].

**Example 2:**

**Input:** nums = [3,2,4], target = 6

**Output:** [1,2]

**Example 3:**

**Input:** nums = [3,3], target = 6

**Output:** [0,1]

**Constraints:**

* 2 <= nums.length <= 104
* -109 <= nums[i] <= 109
* -109 <= target <= 109
* **Only one valid answer exists.**
* Approach 1: Brute Force
* **Algorithm**
* The brute force approach is simple. Loop through each element x*x* and find if there is another value that equals to target - x*target*−*x*.

class Solution:

def twoSum(self, nums: List[int], target: int) -> List[int]:

for i in range(len(nums)):

for j in range(i + 1, len(nums)):

if nums[j] == target - nums[i]:

return [i, j]

**Complexity Analysis**

* Time complexity: O(n^2)*O*(*n*2). For each element, we try to find its complement by looping through the rest of the array which takes O(n)*O*(*n*) time. Therefore, the time complexity is O(n^2)*O*(*n*2).
* Space complexity: O(1)*O*(1). The space required does not depend on the size of the input array, so only constant space is used.

#### Approach 2: Two-pass Hash Table

**Intuition**

To improve our runtime complexity, we need a more efficient way to check if the complement exists in the array. If the complement exists, we need to get its index. What is the best way to maintain a mapping of each element in the array to its index? A hash table.

We can reduce the lookup time from O(n)*O*(*n*) to O(1)*O*(1) by trading space for speed. A hash table is well suited for this purpose because it supports fast lookup in *near* constant time. I say "near" because if a collision occurred, a lookup could degenerate to O(n)*O*(*n*) time. However, lookup in a hash table should be amortized O(1)*O*(1) time as long as the hash function was chosen carefully.

**Algorithm**

A simple implementation uses two iterations. In the first iteration, we add each element's value as a key and its index as a value to the hash table. Then, in the second iteration, we check if each element's complement (target - nums[i]*target*−*nums*[*i*]) exists in the hash table. If it does exist, we return current element's index and its complement's index. Beware that the complement must not be nums[i]*nums*[*i*] itself!

class Solution:

def twoSum(self, nums: List[int], target: int) -> List[int]:

hashmap = {}

for i in range(len(nums)):

hashmap[nums[i]] = i

for i in range(len(nums)):

complement = target - nums[i]

if complement in hashmap and hashmap[complement] != i:

return [i, hashmap[complement]]

**Complexity Analysis**

* Time complexity: O(n)*O*(*n*). We traverse the list containing n*n* elements exactly twice. Since the hash table reduces the lookup time to O(1)*O*(1), the overall time complexity is O(n)*O*(*n*).
* Space complexity: O(n)*O*(*n*). The extra space required depends on the number of items stored in the hash table, which stores exactly n*n* elements.

#### Approach 3: One-pass Hash Table

**Algorithm**

It turns out we can do it in one-pass. While we are iterating and inserting elements into the hash table, we also look back to check if current element's complement already exists in the hash table. If it exists, we have found a solution and return the indices immediately.

class Solution:

def twoSum(self, nums: List[int], target: int) -> List[int]:

hashmap = {}

for i in range(len(nums)):

complement = target - nums[i]

if complement in hashmap:

return [i, hashmap[complement]]

hashmap[nums[i]] = i

**Complexity Analysis**

* Time complexity: O(n)*O*(*n*). We traverse the list containing n*n* elements only once. Each lookup in the table costs only O(1)*O*(1) time.
* Space complexity: O(n)*O*(*n*). The extra space required depends on the number of items stored in the hash table, which stores at most n*n* elements.
* **Longest Substring Without Repeating Characters**

**Example 1:**

**Input:** s = "abcabcbb"

**Output:** 3

**Explanation:** The answer is "abc", with the length of 3.

**Example 2:**

**Input:** s = "bbbbb"

**Output:** 1

**Explanation:** The answer is "b", with the length of 1.

**Example 3:**

**Input:** s = "pwwkew"

**Output:** 3

**Explanation:** The answer is "wke", with the length of 3.

Notice that the answer must be a substring, "pwke" is a subsequence and not a substring.

**Example 4:**

**Input:** s = ""

**Output:** 0

**Constraints:**

* 0 <= s.length <= 5 \* 104
* s consists of English letters, digits, symbols and spaces.

#### Approach 2: Sliding Window

**Algorithm**

The naive approach is very straightforward. But it is too slow. So how can we optimize it?

In the naive approaches, we repeatedly check a substring to see if it has duplicate character. But it is unnecessary. If a substring s\_{ij}*sij*​ from index i*i* to j - 1*j*−1 is already checked to have no duplicate characters. We only need to check if s[j]*s*[*j*] is already in the substring s\_{ij}*sij*​.

To check if a character is already in the substring, we can scan the substring, which leads to an O(n^2)*O*(*n*2) algorithm. But we can do better.

By using HashSet as a sliding window, checking if a character in the current can be done in O(1)*O*(1).

A sliding window is an abstract concept commonly used in array/string problems. A window is a range of elements in the array/string which usually defined by the start and end indices, i.e. [i, j)[*i*,*j*) (left-closed, right-open). A sliding window is a window "slides" its two boundaries to the certain direction. For example, if we slide [i, j)[*i*,*j*) to the right by 11 element, then it becomes [i+1, j+1)[*i*+1,*j*+1) (left-closed, right-open).

Back to our problem. We use HashSet to store the characters in current window [i, j)[*i*,*j*) (j = i*j*=*i* initially). Then we slide the index j*j* to the right. If it is not in the HashSet, we slide j*j* further. Doing so until s[j] is already in the HashSet. At this point, we found the maximum size of substrings without duplicate characters start with index i*i*. If we do this for all i*i*, we get our answer.

class Solution:

def lengthOfLongestSubstring(self, s: str) -> int:

chars = [0] \* 128

left = right = 0

res = 0

while right < len(s):

r = s[right]

chars[ord(r)] += 1

while chars[ord(r)] > 1:

l = s[left]

chars[ord(l)] -= 1

left += 1

res = max(res, right - left + 1)

right += 1

return res

**Complexity Analysis**

* Time complexity : O(2n) = O(n)*O*(2*n*)=*O*(*n*). In the worst case each character will be visited twice by i*i* and j*j*.
* Space complexity : O(min(m, n))*O*(*min*(*m*,*n*)). Same as the previous approach. We need O(k)*O*(*k*) space for the sliding window, where k*k* is the size of the Set. The size of the Set is upper bounded by the size of the string n*n* and the size of the charset/alphabet m*m*.

#### Approach 3: Sliding Window Optimized

The above solution requires at most 2n steps. In fact, it could be optimized to require only n steps. Instead of using a set 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.

The reason is that if s[j]*s*[*j*] have a duplicate in the range [i, j)[*i*,*j*) with index j'*j*′, we don't need to increase i*i* little by little. We can skip all the elements in the range [i, j'][*i*,*j*′] and let i*i* to be j' + 1*j*′+1 directly.

class Solution:

def lengthOfLongestSubstring(self, s: str) -> int:

chars = [None] \* 128

left = right = 0

res = 0

while right < len(s):

r = s[right]

index = chars[ord(r)]

if index != None and index >= left and index < right:

left = index + 1

res = max(res, right - left + 1)

chars[ord(r)] = right

right += 1

return res

**Complexity Analysis**

* Time complexity : O(n)*O*(*n*). Index j*j* will iterate n*n* times.
* Space complexity (HashMap) : O(min(m, n))*O*(*min*(*m*,*n*)). Same as the previous approach.
* Space complexity (Table): O(m)*O*(*m*). m*m* is the size of the charset.

**String to Integer (atoi)**

Implement the myAtoi(string s) function, which converts a string to a 32-bit signed integer (similar to C/C++'s atoi function).

The algorithm for myAtoi(string s) is as follows:

1. Read in and ignore any leading whitespace.
2. Check if the next character (if not already at the end of the string) is '-' or '+'. Read this character in if it is either. This determines if the final result is negative or positive respectively. Assume the result is positive if neither is present.
3. Read in next the characters until the next non-digit character or the end of the input is reached. The rest of the string is ignored.
4. Convert these digits into an integer (i.e. "123" -> 123, "0032" -> 32). If no digits were read, then the integer is 0. Change the sign as necessary (from step 2).
5. If the integer is out of the 32-bit signed integer range [-231, 231 - 1], then clamp the integer so that it remains in the range. Specifically, integers less than -231 should be clamped to -231, and integers greater than 231 - 1 should be clamped to 231 - 1.
6. Return the integer as the final result.

**Note:**

* Only the space character ' ' is considered a whitespace character.
* **Do not ignore** any characters other than the leading whitespace or the rest of the string after the digits.

**Example 1:**

**Input:** s = "42"

**Output:** 42

**Explanation:** The underlined characters are what is read in, the caret is the current reader position.

Step 1: "42" (no characters read because there is no leading whitespace)

^

Step 2: "42" (no characters read because there is neither a '-' nor '+')

^

Step 3: "42" ("42" is read in)

^

The parsed integer is 42.

Since 42 is in the range [-231, 231 - 1], the final result is 42.

**Example 2:**

**Input:** s = " -42"

**Output:** -42

**Explanation:**

Step 1: " -42" (leading whitespace is read and ignored)

^

Step 2: " -42" ('-' is read, so the result should be negative)

^

Step 3: " -42" ("42" is read in)

^

The parsed integer is -42.

Since -42 is in the range [-231, 231 - 1], the final result is -42.

**Example 3:**

**Input:** s = "4193 with words"

**Output:** 4193

**Explanation:**

Step 1: "4193 with words" (no characters read because there is no leading whitespace)

^

Step 2: "4193 with words" (no characters read because there is neither a '-' nor '+')

^

Step 3: "4193 with words" ("4193" is read in; reading stops because the next character is a non-digit)

^

The parsed integer is 4193.

Since 4193 is in the range [-231, 231 - 1], the final result is 4193.

**Example 4:**

**Input:** s = "words and 987"

**Output:** 0

**Explanation:**

Step 1: "words and 987" (no characters read because there is no leading whitespace)

^

Step 2: "words and 987" (no characters read because there is neither a '-' nor '+')

^

Step 3: "words and 987" (reading stops immediately because there is a non-digit 'w')

^

The parsed integer is 0 because no digits were read.

Since 0 is in the range [-231, 231 - 1], the final result is 0.

**Example 5:**

**Input:** s = "-91283472332"

**Output:** -2147483648

**Explanation:**

Step 1: "-91283472332" (no characters read because there is no leading whitespace)

^

Step 2: "-91283472332" ('-' is read, so the result should be negative)

^

Step 3: "-91283472332" ("91283472332" is read in)

^

The parsed integer is -91283472332.

Since -91283472332 is less than the lower bound of the range [-231, 231 - 1], the final result is clamped to -231 = -2147483648.

**Constraints:**

* 0 <= s.length <= 200
* s consists of English letters (lower-case and upper-case), digits (0-9), ' ', '+', '-', and '.'.

def myAtoi(self, s: str):

s = s.strip() #Removes any trailing/leading spaces

if not s: #Check if it is an empty string

return 0

char\_to\_num = {str(i):i for i in range(10)} #create a dictionary for each number 0-9

signs = {'+':1, '-':-1}

final\_number = 0

final\_sign = 1 #default positive

if s[0] in signs: #Check first letter is a sign

final\_sign = signs[s[0]]

s = s[1:] #if first letter is sign, ignore first letter

for letter in s:

if letter in char\_to\_num: #This ensures only integers are read

final\_number = final\_number\*10+char\_to\_num[letter]

else:

break #The loop stops at first non-integer

lower\_limit = -(2\*\*31)

upper\_limit = (2\*\*31)-1

final\_number = final\_number \* final\_sign

if final\_number < lower\_limit:

return lower\_limit

elif final\_number > upper\_limit:

return upper\_limit

else:

return final\_number

**Container With Most Water**

Given n non-negative integers a1, a2, ..., an, where each represents a point at coordinate (i, ai). n vertical lines are drawn such that the two endpoints of the line i is at (i, ai) and (i, 0). Find two lines, which, together with the x-axis forms a container, such that the container contains the most water.

**Notice** that you may not slant the container.

**Example 1:**

A picture containing icon

Description automatically generated

**Input:** height = [1,8,6,2,5,4,8,3,7]

**Output:** 49

**Explanation:** The above vertical lines are represented by array [1,8,6,2,5,4,8,3,7]. In this case, the max area of water (blue section) the container can contain is 49.

**Example 2:**

**Input:** height = [1,1]

**Output:** 1

**Example 3:**

**Input:** height = [4,3,2,1,4]

**Output:** 16

**Example 4:**

**Input:** height = [1,2,1]

**Output:** 2

**Constraints:**

* n == height.length
* 2 <= n <= 105
* 0 <= height[i] <= 104

def maxArea(self, height: List[int]) -> int:

area = 0

head = 0

tail = len(height) - 1

for width in reversed(range(len(height))):

if height[head] < height[tail]:

res = width \* height[head]

head += 1

else:

res = width \* height[tail]

tail -= 1

if res > area:

area = res

return area

**Integer to Roman**

Roman numerals are represented by seven different symbols: I, V, X, L, C, D and M.

**Symbol** **Value**

I 1

V 5

X 10

L 50

C 100

D 500

M 1000

For example, 2 is written as II in Roman numeral, just two one's added together. 12 is written as XII, which is simply X + II. The number 27 is written as XXVII, which is XX + V + II.

Roman numerals are usually written largest to smallest from left to right. However, the numeral for four is not IIII. Instead, the number four is written as IV. Because the one is before the five we subtract it making four. The same principle applies to the number nine, which is written as IX. There are six instances where subtraction is used:

* I can be placed before V (5) and X (10) to make 4 and 9.
* X can be placed before L (50) and C (100) to make 40 and 90.
* C can be placed before D (500) and M (1000) to make 400 and 900.

Given an integer, convert it to a roman numeral.

**Example 1:**

**Input:** num = 3

**Output:** "III"

**Example 2:**

**Input:** num = 4

**Output:** "IV"

**Example 3:**

**Input:** num = 9

**Output:** "IX"

**Example 4:**

**Input:** num = 58

**Output:** "LVIII"

**Explanation:** L = 50, V = 5, III = 3.

**Example 5:**

**Input:** num = 1994

**Output:** "MCMXCIV"

**Explanation:** M = 1000, CM = 900, XC = 90 and IV = 4.

**Constraints:**

* 1 <= num <= 3999

## **Solution**

#### Overview

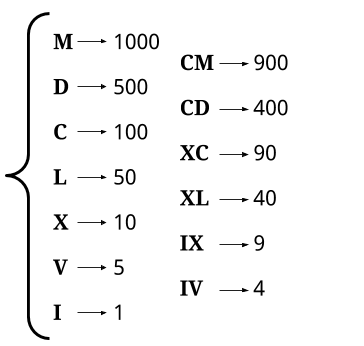
In a lot of countries, Roman Numerals are taught in elementary school-level math. This has made them a somewhat popular "easy" interview question. Unfortunately though, this ignores the fact that not everybody learned them in school, and therefore a big advantage has been given to those who did. I suspect it's also difficult for a lot of us who have learned them previously to fully appreciate how much easier prior experience makes this question. While this is very unfair, and possibly very frustrating, keep in mind that the best thing you can do is work through this question and the related question [Roman to Integer](https://leetcode.com/problems/roman-to-integer/) so that you don't get caught out by it in a real interview. In short, if you're here reading this, you've saved yourself from getting caught out by it! Thankfully, questions that rely on this kind of prior knowledge are few and far between.

**Have a go at Roman to Integer first**

The problem of converting a [Roman Numeral to an Integer](https://leetcode.com/problems/roman-to-integer/) is simpler. Therefore, we suggest that you have a go at it first if you're finding this question difficult. This will allow you to become more familiar with the concept of Roman Numerals without the "ambiguity" issue that comes up in converting an integer to a Roman Numeral. When converting a Roman Numeral to an integer, there's only one sensible conversion.

**Roman Numeral Symbols**

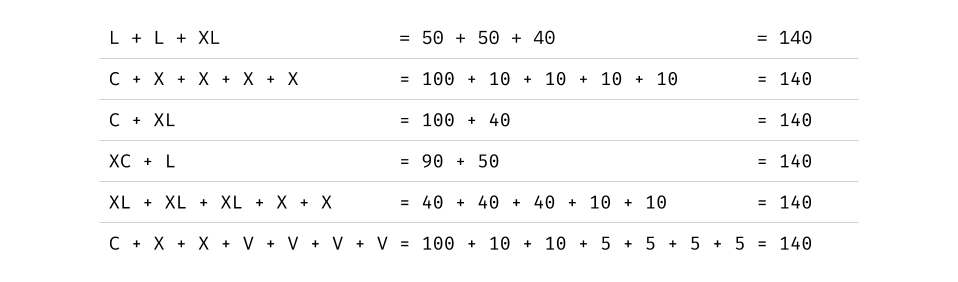
Roman Numerals are made with 7 single-letter symbols, each with its own value. Additionally, the subtractive rules (as explained in the problem description) give an additional 6 symbols. This gives us a total of 13 unique symbols (each symbol is made of either 1 letter or 2).



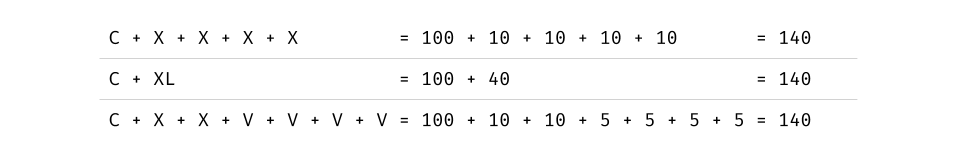
An integer is represented as a Roman Numeral by finding symbols that add to its value.

**Handling Ambiguity**

One thing that can be a bit confusing if you're not familiar with Roman Numerals is knowing which representation is the "correct" one for a particular integer. For example, consider these possible ways of representing 140. Which of these is correct?



**The system we use to decide** is to select the representation with the largest possible symbols, working from left to right. For example, the representations above with the largest symbol at the start are the ones starting with C.



To decide which of these to go with, we look at the next symbol. Two of them have an X, which is worth 10, and one of them has an XL, which is worth 40. Because the XL is worth more, we go with that representation. Therefore, the representation for 140 is CXL.

This definition of Roman Numerals is, these days, the "most accepted". Interestingly, it still isn't an absolute standard, and throughout history, there have been many variants. If you're interested in math and history, we recommend checking out the [Wikipedia article](https://en.wikipedia.org/wiki/Roman_numerals) for your own interest.

class Solution:

def intToRoman(self, num: int) -> str:

digits = [(1000, "M"), (900, "CM"), (500, "D"), (400, "CD"), (100, "C"),

(90, "XC"), (50, "L"), (40, "XL"), (10, "X"), (9, "IX"),

(5, "V"), (4, "IV"), (1, "I")]

roman\_digits = []

# Loop through each symbol.

for value, symbol in digits:

# We don't want to continue looping if we're done.

if num == 0: break

count, num = divmod(num, value)

# Append "count" copies of "symbol" to roman\_digits.

roman\_digits.append(symbol \* count)

return "".join(roman\_digits)

**Complexity Analysis**

* Time complexity : O(1)*O*(1).

As there is a finite set of roman numerals, there is a hard upper limit on how many times the loop can iterate. This upper limit is 15 times, and it occurs for the number 3888, which has a representation of MMMDCCCLXXXVIII. Therefore, we say the time complexity is constant, i.e. O(1)*O*(1).

* Space complexity : O(1)*O*(1).

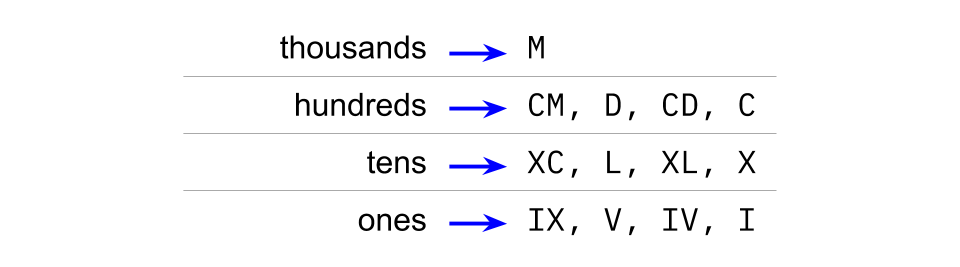
The amount of memory used does not change with the size of the input integer, and is therefore constant.

#### Approach 2: Hardcode Digits

**Intuition**

Please don't panic and assume you need to memorize the values in this approach. The first approach should be fine, and in-fact has the added bonus of being more flexible if we were to extend the Roman Numeral symbol set to have symbols over 1000. This second approach is only included for completeness. Do try to understand how we derived this approach, though.

An interesting observation that can be made is that each of the digits in the integer's decimal representation can be treated independently when converting the integer into a Roman Numeral. Notice that all of the symbols can be split into groups based on their highest factor out of 1000, 100, 10, and 1.

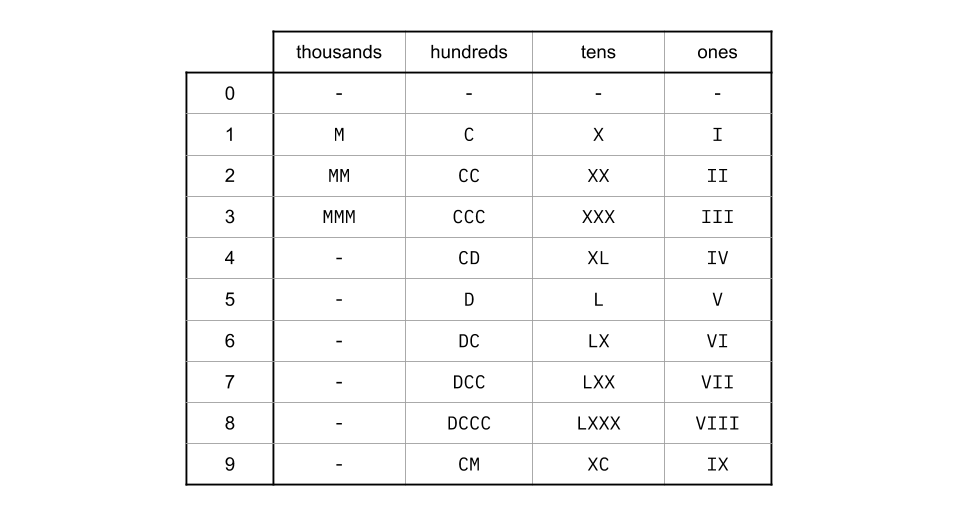


While the number is at least 1000, an M (1000) will be appended to the output and 1000 will be subtracted from the integer. The other symbols won't even be considered until the number is below 1000. Additionally, the M (1000)s cannot represent any lower part of the number. Therefore, we can represent the thousands digit of the integer entirely with M (1000)s.

Now, assume we have a remainder of between 100 and 999. The next symbols considered are those in the hundreds row. The highest symbol that could fit in right now is CM (900), and the lowest is C (100). None of the symbols in this range can possibly modify the tens or ones. As long as the remainder is still above 100, we can still take at least C (100) out of it. This means that we'll only be subtracting symbols from the hundreds row for as long as the number is at least 100.

The same argument applies for the tens, and then the ones.

We can, therefore, work out what the representation for each digit, in each place, is. There are only 34 of them; 0, 1, 2, 3 and 4 for the thousands column, and 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9 for each of the hundreds, tens, and ones. So with a pencil, paper, and some patience, you can hopefully work out the representation for each of these possibilities and hardcode them. Then, converting an integer to a Roman Numeral will require breaking the integer into digits and appending the relevant representation for each digit.



Getting each digit of the number can be done using the modulus and division operators. The division operator removes the digits below the place we want, and the modulus operator removes the digits from above. This simply leaves the digit we want.

thousands\_digit = integer / 1000

hundreds\_digit = (integer % 1000) / 100

tens\_digit = (integer % 100) / 10

ones\_digit = integer % 10

Then, we can simply look these up in the hardcoded table, and append the results together!

**Algorithm**

The cleanest way to go about it in code is to have 4 separate arrays; one for each place value. Then, extract the digits, look up their symbols in the relevant array, and append them all together.

class Solution:

def intToRoman(self, num: int) -> str:

thousands = ["", "M", "MM", "MMM"]

hundreds = ["", "C", "CC", "CCC", "CD", "D", "DC", "DCC", "DCCC", "CM"]

tens = ["", "X", "XX", "XXX", "XL", "L", "LX", "LXX", "LXXX", "XC"]

ones = ["", "I", "II", "III", "IV", "V", "VI", "VII", "VIII", "IX"]

return (thousands[num // 1000] + hundreds[num % 1000 // 100]

+ tens[num % 100 // 10] + ones[num % 10])

**Complexity Analysis**

* Time complexity : O(1)*O*(1).

The same number of operations is done, regardless of the size of the input. Therefore, the time complexity is constant.

* Space complexity : O(1)*O*(1).

While we have Arrays, they are the same size, *regardless of the size of the input*. Therefore, they are constant for the purpose of space-complexity analysis.

The downside of this approach is that it is inflexible if Roman Numerals were to be extended (which is an interesting follow-up question). For example, what if we said the symbol H now represents 5000, and P now represents 10000, allowing us to represent numbers up to 39999? Approach 1 will be a lot quicker to modify, as you simply need to add these 2 values to the code without doing any calculations. But for Approach 2, you'll need to calculate and hardcode ten new representations. What if we then added symbols to be able to go up to 399,999,999? Approach 2 becomes more and more difficult to manage, the more symbols we add.

**Roman to Integer**

Roman numerals are represented by seven different symbols: I, V, X, L, C, D and M.

**Symbol** **Value**

I 1

V 5

X 10

L 50

C 100

D 500

M 1000

For example, 2 is written as II in Roman numeral, just two one's added together. 12 is written as XII, which is simply X + II. The number 27 is written as XXVII, which is XX + V + II.

Roman numerals are usually written largest to smallest from left to right. However, the numeral for four is not IIII. Instead, the number four is written as IV. Because the one is before the five we subtract it making four. The same principle applies to the number nine, which is written as IX. There are six instances where subtraction is used:

* I can be placed before V (5) and X (10) to make 4 and 9.
* X can be placed before L (50) and C (100) to make 40 and 90.
* C can be placed before D (500) and M (1000) to make 400 and 900.

Given a roman numeral, convert it to an integer.

**Example 1:**

**Input:** s = "III"

**Output:** 3

**Example 2:**

**Input:** s = "IV"

**Output:** 4

**Example 3:**

**Input:** s = "IX"

**Output:** 9

**Example 4:**

**Input:** s = "LVIII"

**Output:** 58

**Explanation:** L = 50, V= 5, III = 3.

**Example 5:**

**Input:** s = "MCMXCIV"

**Output:** 1994

**Explanation:** M = 1000, CM = 900, XC = 90 and IV = 4.

**Constraints:**

* 1 <= s.length <= 15
* s contains only the characters ('I', 'V', 'X', 'L', 'C', 'D', 'M').
* It is **guaranteed** that s is a valid roman numeral in the range [1, 3999].

## **Solution**

#### Overview

In a lot of countries, Roman Numerals are taught in elementary school-level math. This has made them a somewhat popular "easy" interview question. Unfortunately though, this ignores the fact that not everybody learned them in school, and therefore a big advantage has been given to those who did. I suspect it's also difficult for a lot of us who have learned them previously to fully appreciate how much easier prior experience makes this question. While this is very unfair, and possibly very frustrating, keep in mind that the best thing you can do is work through this question and the related question [Integer to Roman](https://leetcode.com/problems/integer-to-roman/) so that you don't get caught out by it in a real interview.

**Can we assume the input is valid?**

Yes. Here on Leetcode, you can make that assumption because you haven't been told what to do if it isn't.

In a real interview, this is a question you should ask the interviewer. Don't ever assume without asking in a real interview that the input has to be valid.

**Is there only one valid representation for each number?**

This is more relevant to the other question, [Integer to Roman](https://leetcode.com/problems/integer-to-roman/), however we'll still briefly look at it now.

Given that the representation for 3 is III, it could seem natural that the representation for 15 is VVV, because that would be 5 + 5 + 5. However, it's actually XV, which is 10 + 5. How are you even supposed to know which is correct?

The trick is to use the "biggest" symbols you can. Because X is bigger than V, we should use an X first and then make up the remainder with a single V, giving XV.

We'll talk more about this in the [Integer to Roman](https://leetcode.com/problems/integer-to-roman/) article. This question is a lot simpler because there's only one logical way of converting from a Roman Numeral to an Integer. This is also why this question is labelled as "easy", whereas the other is labelled as "medium".

**A few more examples**

If you're not very familiar with Roman Numerals, work through these examples and then have another go at writing your own algorithm before reading the rest of this solution article.

What is *CXVII* as an integer?

Recall that C = 100, X = 10, V = 5, and I = 1. Because the symbols are ordered from most significant to least, we can simply add the symbols, i.e. C + X + V + I + I = 100 + 10 + 5 + 1 + 1 = 117.

What is *DXCI* as an integer?

Recall that D = 500.

Now, notice that this time the symbols are not ordered from most significant to least—the X and C are out of numeric order. Because of this, we subtract the value of X (10) from the value of C (100) to get 90.

So, going from left to right, we have D + (C - X) + I = 500 + 90 + 1 = 591.

What is *CMXCIV* as an integer?

Recall that M = 1000.

The symbols barely look sorted at all here—from left-to-right we have 100, 1000, 10, 100, 1, 5. Do not panic though, we just need to look for each occurrence of a smaller symbols preceding a bigger symbol. The first, third, and fifth symbols are all smaller than their next symbol. Therefore they are all going to be subtracted from their next.

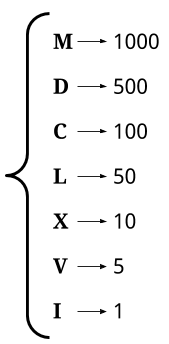
* The first two symbols are CM. This is M - C = 1000 - 100 = 900
* The second two symbols are XC. This is C - X = 100 - 10 = 90.
* The final two symbols are IV. This is V - I = 5 - 1 = 4.

Like we did above, we add these together. (M - C) + (C - X) + (V - I) = 900 + 90 + 4 = 994.

#### Approach 1: Left-to-Right Pass

**Intuition**

Let's hard-code a mapping with the value of each symbol so that we can easily look them up.



Now, recall that each symbol adds its own value, except for when a smaller valued symbol is before a larger valued symbol. In those cases, instead of adding both symbols to the total, we need to subtract the large from the small, adding that instead.

Therefore, the simplest algorithm is to use a pointer to scan through the string, at each step deciding whether to add the current symbol and go forward 1 place, or add the difference of the next 2 symbols and go forward 2 places. Here is this algorithm in pseudocode.

total = 0

i = 0

while i < s.length:

if at least 2 symbols remaining AND value of s[i] < value of s[i + 1]:

total = total + (value of s[i + 1]) - (value of s[i])

i = i + 2

else:

total = total + (value of s[i])

i = i + 1

return total

values = {

"I": 1,

"V": 5,

"X": 10,

"L": 50,

"C": 100,

"D": 500,

"M": 1000,

}

class Solution:

def romanToInt(self, s: str) -> int:

total = 0

i = 0

while i < len(s):

# If this is the subtractive case.

if i + 1 < len(s) and values[s[i]] < values[s[i + 1]]:

total += values[s[i + 1]] - values[s[i]]

i += 2

# Else this is NOT the subtractive case.

else:

total += values[s[i]]

i += 1

return total

**Complexity Analysis**

Let n*n* be the length of the input string (the total number of symbols in it).

* Time complexity : O(1)*O*(1).

As there is a finite set of roman numerals, the maximum number possible number can be 3999, which in roman numerals is MMMCMXCIX. As such the time complexity is O(1)*O*(1).

If roman numerals had an arbitrary number of symbols, then the time complexity would be proportional to the length of the input, i.e. O(n)*O*(*n*). This is assuming that looking up the value of each symbol is O(1)*O*(1).

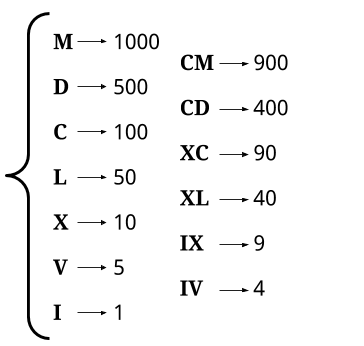
* Space complexity : O(1)*O*(1).

Because only a constant number of single-value variables are used, the space complexity is O(1)*O*(1).

#### Approach 2: Left-to-Right Pass Improved

**Intuition**

Instead of viewing a Roman Numeral as having 7 unique symbols, we could instead view it as having 13 unique symbols—some of length-1 and some of length-2.



For example, here is the Roman Numeral MMCMLXXXIX broken into its symbols using this definition:

Splitting the numeral into parts

We can then look up the value of each symbol and add them together.

Adding up the sum of the numeral

After making a Map of String -> Integer with the 13 "symbols", we need to work our way down the string in the same way as before (we'll do left-to-right, however right-to-left will work okay too), firstly checking if we're at a length-2 symbol, and if not, then treating it as a length-1 symbol.

total = 0

i = 0

while i < s.length:

if at least 2 characters remaining and s.substing(i, i + 1) is in values:

total = total + (value of s.substring(i, i + 1))

i = i + 2

else:

total = total + (value of s[i])

i = i + 1

return total

values = {

"I": 1,

"V": 5,

"X": 10,

"L": 50,

"C": 100,

"D": 500,

"M": 1000,

"IV": 4,

"IX": 9,

"XL": 40,

"XC": 90,

"CD": 400,

"CM": 900

}

class Solution:

def romanToInt(self, s: str) -> int:

total = 0

i = 0

while i < len(s):

# This is the subtractive case.

if i < len(s) - 1 and s[i:i+2] in values:

total += values[s[i:i+2]]

i += 2

else:

total += values[s[i]]

i += 1

return total

#### Approach 3: Right-to-Left Pass

**Intuition**

This approach is a more elegant variant of Approach 1. Just to be clear though, Approach 1 and Approach 2 are probably sufficient for an interview. This approach is still well worth understanding though.

In the "subtraction" cases, such as XC, we've been updating our running sum as follows:

sum += value(C) - value(X)

However, notice that this is mathematically equivalent to the following:

sum += value(C)

sum -= value(X)

Utilizing this means that we can process one symbol each time we go around the main loop. We still need to determine whether or not our current symbol should be added or subtracted by looking at the neighbour though.

In Approach 1, we had to be careful when inspecting the next symbol to not go over the end of the string. This check wasn't difficult to do, but it increased the code complexity a bit, and it turns out we can avoid it with this approach!

Observe the following:

1. Without looking at the next symbol, we don't know whether or not the left-most symbol should be added or subtracted.
2. The right-most symbol is always added. It is either by itself, or the additive part of a pair.

So, what we can do is initialise sum to be the value of the right-most (last) symbol. Then, we work backwards through the string, starting from the second-to-last-symbol. We check the symbol after (i + 1) to determine whether the current symbol should be "added" or "subtracted".

last = s.length - 1

total = value(last)

`

for i from last - 1 down to 0:

if value(s[i]) < value(s[i+1]):

total -= value(s[i])

else:

total += value(s[i])

return sum

Because we're starting at the second-to-last-index, we know that index i + 1 always exists. We no longer need to handle its potential non-existence as a special case, and additionally we're able to (cleanly) use a for loop, as we're always moving along by 1 index at at time, unlike before where it could have been 1 or 2.

values = {

"I": 1,

"V": 5,

"X": 10,

"L": 50,

"C": 100,

"D": 500,

"M": 1000,

}

class Solution:

def romanToInt(self, s: str) -> int:

total = values.get(s[-1])

for i in reversed(range(len(s) - 1)):

if values[s[i]] < values[s[i + 1]]:

total -= values[s[i]]

else:

total += values[s[i]]

return total

**Complexity Analysis**

* Time complexity : O(1)*O*(1).

Same as Approach 1.

* Space complexity : O(1)*O*(1).

Same as Approach 1.

**3Sum**

Given an integer array nums, return all the triplets [nums[i], nums[j], nums[k]] such that i != j, i != k, and j != k, and nums[i] + nums[j] + nums[k] == 0.

Notice that the solution set must not contain duplicate triplets.

**Example 1:**

**Input:** nums = [-1,0,1,2,-1,-4]

**Output:** [[-1,-1,2],[-1,0,1]]

**Example 2:**

**Input:** nums = []

**Output:** []

**Example 3:**

**Input:** nums = [0]

**Output:** []

**Constraints:**

* 0 <= nums.length <= 3000
* -105 <= nums[i] <= 105

## **Solution**

This problem is a follow-up of Two Sum, and it is a good idea to first take a look at [Two Sum](https://leetcode.com/articles/two-sum/) and [Two Sum II](https://leetcode.com/articles/two-sum-ii-input-array-is-sorted/). An interviewer may ask to solve Two Sum first, and then throw 3Sum at you. Pay attention to subtle differences in problem description and try to re-use existing solutions!

Two Sum, Two Sum II and 3Sum share a similarity that the sum of elements must match the target exactly. A difference is that, instead of exactly one answer, we need to find all unique triplets that sum to zero.

Before jumping in, let's check the existing solutions and determine the best conceivable runtime (BCR) for 3Sum:

1. [Two Sum](https://leetcode.com/articles/two-sum/) uses a hashmap to find complement values, and therefore achieves \mathcal{O}(N)O(*N*) time complexity.
2. [Two Sum II](https://leetcode.com/articles/two-sum-ii-input-array-is-sorted/) uses the two pointers pattern and also has \mathcal{O}(N)O(*N*) time complexity for a sorted array. We can use this approach for any array if we sort it first, which bumps the time complexity to \mathcal{O}(n\log{n})O(*n*log*n*).

Considering that there is one more dimension in 3Sum, it sounds reasonable to shoot for \mathcal{O}(n^2)O(*n*2) time complexity as our BCR.

#### Approach 1: Two Pointers

We will follow the same two pointers pattern as in [Two Sum II](https://leetcode.com/articles/two-sum-ii-input-array-is-sorted/). It requires the array to be sorted, so we'll do that first. As our BCR is \mathcal{O}(n^2)O(*n*2), sorting the array would not change the overall time complexity.

To make sure the result contains unique triplets, we need to skip duplicate values. It is easy to do because repeating values are next to each other in a sorted array.

If you are wondering how to solve this problem without sorting the array, go over the ["No-Sort"](https://leetcode.com/problems/3sum/solution/#approach3) approach below. There are cases when that approach is preferable, and your interviewer may probe your knowledge there.

After sorting the array, we move our pivot element nums[i] and analyze elements to its right. We find all pairs whose sum is equal -nums[i] using the two pointers pattern, so that the sum of the pivot element (nums[i]) and the pair (-nums[i]) is equal to zero.

As a quick refresher, the pointers are initially set to the first and the last element respectively. We compare the sum of these two elements to the target. If it is smaller, we increment the lower pointer lo. Otherwise, we decrement the higher pointer hi. Thus, the sum always moves toward the target, and we "prune" pairs that would move it further away. Again, this works only if the array is sorted. Head to the [Two Sum II](https://leetcode.com/articles/two-sum-ii-input-array-is-sorted/) solution for the detailed explanation.

**Algorithm**

The implementation is straightforward - we just need to modify twoSumII to produce triplets and skip repeating values.

1. For the main function:
   * Sort the input array nums.
   * Iterate through the array:
     + If the current value is greater than zero, break from the loop. Remaining values cannot sum to zero.
     + If the current value is the same as the one before, skip it.
     + Otherwise, call twoSumII for the current position i.
2. For twoSumII function:
   * Set the low pointer lo to i + 1, and high pointer hi to the last index.
   * While low pointer is smaller than high:
     + If sum of nums[i] + nums[lo] + nums[hi] is less than zero, increment lo.
     + If sum is greater than zero, decrement hi.
     + Otherwise, we found a triplet:
       - Add it to the result res.
       - Decrement hi and increment lo.
       - Increment lo while the next value is the same as before to avoid duplicates in the result.
3. Return the result res.

class Solution:

def threeSum(self, nums: List[int]) -> List[List[int]]:

res = []

nums.sort()

for i in range(len(nums)):

if nums[i] > 0:

break

if i == 0 or nums[i - 1] != nums[i]:

self.twoSumII(nums, i, res)

return res

def twoSumII(self, nums: List[int], i: int, res: List[List[int]]):

lo, hi = i + 1, len(nums) - 1

while (lo < hi):

sum = nums[i] + nums[lo] + nums[hi]

if sum < 0:

lo += 1

elif sum > 0:

hi -= 1

else:

res.append([nums[i], nums[lo], nums[hi]])

lo += 1

hi -= 1

while lo < hi and nums[lo] == nums[lo - 1]:

lo += 1

**Complexity Analysis**

* Time Complexity: \mathcal{O}(n^2)O(*n*2). twoSumII is \mathcal{O}(n)O(*n*), and we call it n*n* times.

Sorting the array takes \mathcal{O}(n\log{n})O(*n*log*n*), so overall complexity is \mathcal{O}(n\log{n} + n^2)O(*n*log*n*+*n*2). This is asymptotically equivalent to \mathcal{O}(n^2)O(*n*2).

* Space Complexity: from \mathcal{O}(\log{n})O(log*n*) to \mathcal{O}(n)O(*n*), depending on the implementation of the sorting algorithm. For the purpose of complexity analysis, we ignore the memory required for the output.

### **Approach 2: Hashset**

Since triplets must sum up to the target value, we can try the hash table approach from the [Two Sum](https://leetcode.com/articles/two-sum/) solution. This approach won't work, however, if the sum is not necessarily equal to the target, like in [3Sum Smaller](https://leetcode.com/problems/3sum-smaller/) and [3Sum Closest](https://leetcode.com/problems/3sum-closest/).

We move our pivot element nums[i] and analyze elements to its right. We find all pairs whose sum is equal -nums[i] using the [Two Sum: One-pass Hash Table](https://leetcode.com/articles/two-sum/#approach-3-one-pass-hash-table) approach, so that the sum of the pivot element (nums[i]) and the pair (-nums[i]) is equal to zero.

To do that, we process each element nums[j] to the right of the pivot, and check whether a complement -nums[i] - nums[j] is already in the hashset. If it is, we found a triplet. Then, we add nums[j] to the hashset, so it can be used as a complement from that point on.

Like in the approach above, we will also sort the array so we can skip repeated values. We provide a different way to avoid duplicates in the ["No-Sort"](https://leetcode.com/problems/3sum/solution/#approach3) approach below.

**Algorithm**

The main function is the same as in the [Two Pointers](https://leetcode.com/problems/3sum/solution/#approach1) approach above. Here, we use twoSum (instead of twoSumII), modified to produce triplets and skip repeating values.

1. For the main function:
   * Sort the input array nums.
   * Iterate through the array:
     + If the current value is greater than zero, break from the loop. Remaining values cannot sum to zero.
     + If the current value is the same as the one before, skip it.
     + Otherwise, call twoSum for the current position i.
2. For twoSum function:
   * For each index j > i in A:
     + Compute complement value as -nums[i] - nums[j].
     + If complement exists in hashset seen:
       - We found a triplet - add it to the result res.
       - Increment j while the next value is the same as before to avoid duplicates in the result.
     + Add nums[j] to hashset seen
3. Return the result res.

class Solution:

def threeSum(self, nums: List[int]) -> List[List[int]]:

res = []

nums.sort()

for i in range(len(nums)):

if nums[i] > 0:

break

if i == 0 or nums[i - 1] != nums[i]:

self.twoSum(nums, i, res)

return res

def twoSum(self, nums: List[int], i: int, res: List[List[int]]):

seen = set()

j = i + 1

while j < len(nums):

complement = -nums[i] - nums[j]

if complement in seen:

res.append([nums[i], nums[j], complement])

while j + 1 < len(nums) and nums[j] == nums[j + 1]:

j += 1

seen.add(nums[j])

j += 1

* Time Complexity: \mathcal{O}(n^2)O(*n*2). twoSum is \mathcal{O}(n)O(*n*), and we call it n*n* times.

Sorting the array takes \mathcal{O}(n\log{n})O(*n*log*n*), so overall complexity is \mathcal{O}(n\log{n} + n^2)O(*n*log*n*+*n*2). This is asymptotically equivalent to \mathcal{O}(n^2)O(*n*2).

* Space Complexity: \mathcal{O}(n)O(*n*) for the hashset.

#### Approach 3: "No-Sort"

What if you cannot modify the input array, and you want to avoid copying it due to memory constraints?

We can adapt the hashset approach above to work for an unsorted array. We can put a combination of three values into a hashset to avoid duplicates. Values in a combination should be ordered (e.g. ascending). Otherwise, we can have results with the same values in the different positions.

**Algorithm**

The algorithm is similar to the hashset approach above. We just need to add few optimizations so that it works efficiently for repeated values:

1. Use another hashset dups to skip duplicates in the outer loop.
   * Without this optimization, the submission will time out for the test case with 3,000 zeroes. This case is handled naturally when the array is sorted.
2. Instead of re-populating a hashset every time in the inner loop, we can use a hashmap and populate it once. Values in the hashmap will indicate whether we have encountered that element in the current iteration. When we process nums[j] in the inner loop, we set its hashmap value to i. This indicates that we can now use nums[j] as a complement for nums[i].
   * This is more like a trick to compensate for container overheads. The effect varies by language, e.g. for C++ it cuts the runtime in half. Without this trick the submission may time out.

class Solution:

def threeSum(self, nums: List[int]) -> List[List[int]]:

res, dups = set(), set()

seen = {}

for i, val1 in enumerate(nums):

if val1 not in dups:

dups.add(val1)

for j, val2 in enumerate(nums[i+1:]):

complement = -val1 - val2

if complement in seen and seen[complement] == i:

res.add(tuple(sorted((val1, val2, complement))))

seen[val2] = i

return res

**Complexity Analysis**

* Time Complexity: \mathcal{O}(n^2)O(*n*2). We have outer and inner loops, each going through n*n* elements.

While the asymptotic complexity is the same, this algorithm is noticeably slower than the previous approach. Lookups in a hashset, though requiring a constant time, are expensive compared to the direct memory access.

* Space Complexity: \mathcal{O}(n)O(*n*) for the hashset/hashmap.

For the purpose of complexity analysis, we ignore the memory required for the output. However, in this approach we also store output in the hashset for deduplication. In the worst case, there could be \mathcal{O}(n^2)O(*n*2) triplets in the output, like for this example: [-k, -k + 1, ..., -1, 0, 1, ... k - 1, k]. Adding a new number to this sequence will produce n / 3 new triplets.

**3Sum Closest**

Given an integer array nums of length n and an integer target, find three integers in nums such that the sum is closest to target.

Return *the sum of the three integers*.

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

**Example 1:**

**Input:** nums = [-1,2,1,-4], target = 1

**Output:** 2

**Explanation:** The sum that is closest to the target is 2. (-1 + 2 + 1 = 2).

**Example 2:**

**Input:** nums = [0,0,0], target = 1

**Output:** 0

**Constraints:**

* 3 <= nums.length <= 1000
* -1000 <= nums[i] <= 1000
* -104 <= target <= 104

## **Solution**

This problem is a variation of [3Sum](https://leetcode.com/articles/3sum/). The main difference is that the sum of a triplet is not necessarily equal to the target. Instead, the sum is in some relation with the target, which is closest to the target for this problem. In that sense, this problem shares similarities with [3Sum Smaller](https://leetcode.com/articles/3sum-smaller/).

Before jumping in, let's check solutions for the similar problems:

1. [3Sum](https://leetcode.com/articles/3sum/) fixes one number and uses either the two pointers pattern or a hash set to find complementary pairs. Thus, the time complexity is \mathcal{O}(n^2)O(*n*2).
2. [3Sum Smaller](https://leetcode.com/articles/3sum-smaller/), similarly to 3Sum, uses the two pointers pattern to enumerate smaller pairs. Note that we cannot use a hash set here because we do not have a specific value to look up.

For the same reason as for 3Sum Smaller, we cannot use a hash set approach here. So, we will focus on the two pointers pattern and shoot for \mathcal{O}(n^2)O(*n*2) time complexity as the best conceivable runtime (BCR).

#### Approach 1: Two Pointers

The two pointers pattern requires the array to be sorted, so we do that first. As our BCR is \mathcal{O}(n^2)O(*n*2), the sort operation would not change the overall time complexity.

In the sorted array, we process each value from left to right. For value v, we need to find a pair which sum, ideally, is equal to target - v. We will follow the same two pointers approach as for 3Sum, however, since this 'ideal' pair may not exist, we will track the smallest absolute difference between the sum and the target. The two pointers approach naturally enumerates pairs so that the sum moves toward the target.

**Algorithm**

1. Initialize the minimum difference diff with a large value.
2. Sort the input array nums.
3. Iterate through the array:
   * For the current position i, set lo to i + 1, and hi to the last index.
   * While the lo pointer is smaller than hi:
     + Set sum to nums[i] + nums[lo] + nums[hi].
     + If the absolute difference between sum and target is smaller than the absolute value of diff:
       - Set diff to target - sum.
     + If sum is less than target, increment lo.
     + Else, decrement hi.
   * If diff is zero, break from the loop.
4. Return the value of the closest triplet, which is target - diff.

class Solution:

def threeSumClosest(self, nums: List[int], target: int) -> int:

diff = float('inf')

nums.sort()

for i in range(len(nums)):

lo, hi = i + 1, len(nums) - 1

while (lo < hi):

sum = nums[i] + nums[lo] + nums[hi]

if abs(target - sum) < abs(diff):

diff = target - sum

if sum < target:

lo += 1

else:

hi -= 1

if diff == 0:

break

return target - diff

**Complexity Analysis**

* Time Complexity: \mathcal{O}(n^2)O(*n*2). We have outer and inner loops, each going through n*n* elements.

Sorting the array takes \mathcal{O}(n\log{n})O(*n*log*n*), so overall complexity is \mathcal{O}(n\log{n} + n^2)O(*n*log*n*+*n*2). This is asymptotically equivalent to \mathcal{O}(n^2)O(*n*2).

* Space Complexity: from \mathcal{O}(\log{n})O(log*n*) to \mathcal{O}(n)O(*n*), depending on the implementation of the sorting algorithm.

#### Approach 2: Binary Search

We can adapt the [3Sum Smaller: Binary Search](https://leetcode.com/articles/3sum-smaller/#approach-2-binary-search-accepted) approach to this problem.

In the two pointers approach, we fix one number and use two pointers to enumerate pairs. Here, we fix two numbers, and use a binary search to find the third complement number. This is less efficient than the two pointers approach, however, it could be more intuitive to come up with.

Note that we may not find the exact complement number, so we check the difference between the complement and two numbers: the next higher and the previous lower. For example, if the complement is 42, and our array is [-10, -4, 15, 30, 60], the next higher is 60 (so the difference is -18), and the previous lower is 30 (and the difference is 12).

**Algorithm**

1. Initialize the minimum difference diff with a large value.
2. Sort the input array nums.
3. Iterate through the array (outer loop):
   * For the current position i, iterate through the array starting from j = i + 1 (inner loop):
     + Binary-search for complement (target - nums[i] - nums[j]) in the rest of the array.
     + For the next higher value, check its absolute difference with complement against diff.
     + For the previous lower value, check its absolute difference with complement against diff.
     + Update diff based on the smallest absolute difference.
   * If diff is zero, break from the loop.
4. Return the value of the closest triplet, which is target - diff.

class Solution:

def threeSumClosest(self, nums: List[int], target: int) -> int:

diff = float('inf')

nums.sort()

for i in range(len(nums)):

for j in range(i + 1, len(nums)):

complement = target - nums[i] - nums[j]

hi = bisect\_right(nums, complement, j + 1)

lo = hi - 1

if hi < len(nums) and abs(complement - nums[hi]) < abs(diff):

diff = complement - nums[hi]

if lo > j and abs(complement - nums[lo]) < abs(diff):

diff = complement - nums[lo]

if diff == 0:

break

return target – diff

**Complexity Analysis**

* Time Complexity: \mathcal{O}(n^2\log{n})O(*n*2log*n*). Binary search takes \mathcal{O}(\log{n})O(log*n*), and we do it n*n* times in the inner loop. Since we are going through n*n* elements in the outer loop, the overall complexity is \mathcal{O}(n^2\log{n})O(*n*2log*n*).
* Space Complexity: from \mathcal{O}(\log{n})O(log*n*) to \mathcal{O}(n)O(*n*), depending on the implementation of the sorting algorithm.

**Implement strStr()**

Implement [strStr()](http://www.cplusplus.com/reference/cstring/strstr/" \t "_blank).

Return the index of the first occurrence of needle in haystack, or -1 if needle is not part of haystack.

**Clarification:**

What should we return when needle is an empty string? This is a great question to ask during an interview.

For the purpose of this problem, we will return 0 when needle is an empty string. This is consistent to C's [strstr()](http://www.cplusplus.com/reference/cstring/strstr/" \t "_blank) and Java's [indexOf()](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html" \l "indexOf(java.lang.String)" \t "_blank).

**Example 1:**

**Input:** haystack = "hello", needle = "ll"

**Output:** 2

**Example 2:**

**Input:** haystack = "aaaaa", needle = "bba"

**Output:** -1

**Example 3:**

**Input:** haystack = "", needle = ""

**Output:** 0

**Constraints:**

* 0 <= haystack.length, needle.length <= 5 \* 104
* haystack and needle consist of only lower-case English characters.

def strStr(self, haystack: str, needle: str) -> int:

needleLen = len(needle)

if needleLen == 0:

return 0

index = -1

hIndex = 0

hStackLen = len(haystack)

while index < 0 and hIndex <= (hStackLen - needleLen):

if haystack[hIndex: hIndex + needleLen] == needle:

index = hIndex

else:

hIndex += 1

return index

**Rotate Image**

You are given an n x n 2D matrix representing an image, rotate the image by **90** degrees (clockwise).

You have to rotate the image [**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm), which means you have to modify the input 2D matrix directly. **DO NOT** allocate another 2D matrix and do the rotation.

**Example 1:**



**Input:** matrix = [[1,2,3],[4,5,6],[7,8,9]]

**Output:** [[7,4,1],[8,5,2],[9,6,3]]

**Example 2:**

A picture containing text, shoji, crossword puzzle, clipart

Description automatically generated

**Input:** matrix = [[5,1,9,11],[2,4,8,10],[13,3,6,7],[15,14,12,16]]

**Output:** [[15,13,2,5],[14,3,4,1],[12,6,8,9],[16,7,10,11]]

**Example 3:**

**Input:** matrix = [[1]]

**Output:** [[1]]

**Example 4:**

**Input:** matrix = [[1,2],[3,4]]

**Output:** [[3,1],[4,2]]

**Constraints:**

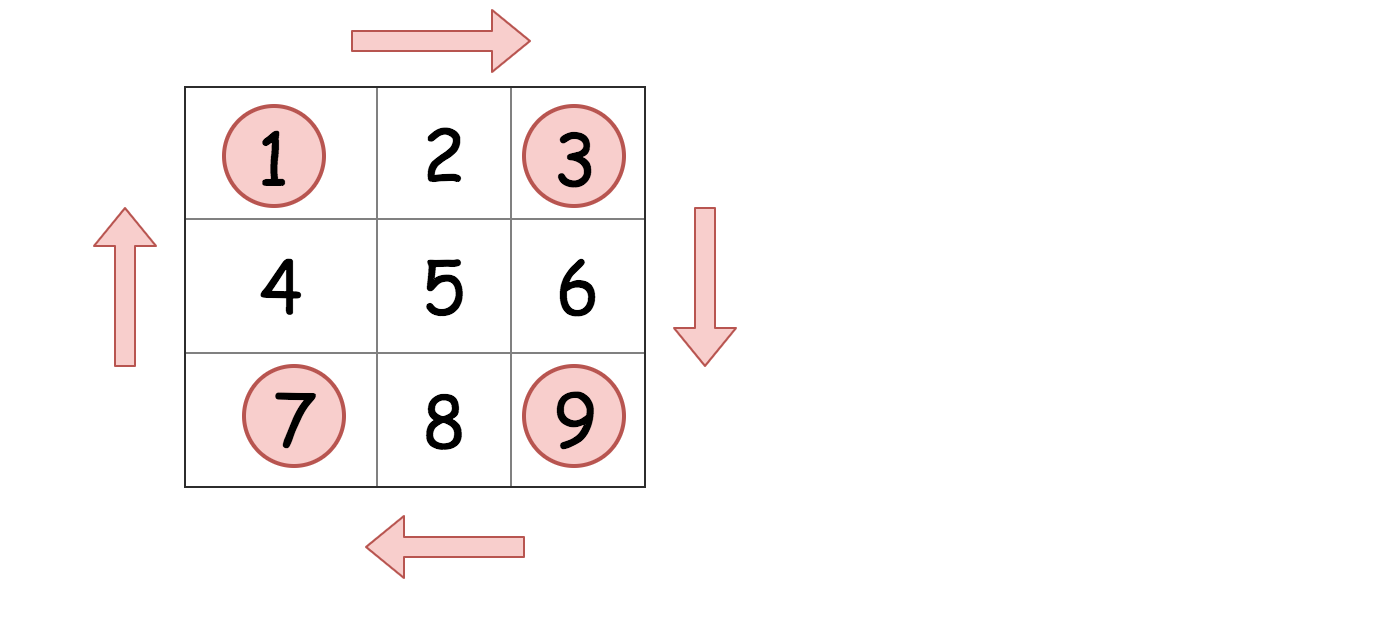
* matrix.length == n
* matrix[i].length == n
* 1 <= n <= 20
* -1000 <= matrix[i][j] <= 1000

## **Solution**

#### Approach 1: Rotate Groups of Four Cells

**Intuition**

Observe how the cells move in groups when we rotate the image.



We can iterate over each group of four cells and rotate them.

class Solution:

def rotate(self, matrix: List[List[int]]) -> None:

n = len(matrix[0])

for i in range(n // 2 + n % 2):

for j in range(n // 2):

tmp = matrix[n - 1 - j][i]

matrix[n - 1 - j][i] = matrix[n - 1 - i][n - j - 1]

matrix[n - 1 - i][n - j - 1] = matrix[j][n - 1 -i]

matrix[j][n - 1 - i] = matrix[i][j]

matrix[i][j] = tmp

**Complexity Analysis**

Let M*M* be the number of cells in the matrix.

* Time complexity : \mathcal{O}(M)O(*M*), as each cell is getting read once and written once.
* Space complexity : \mathcal{O}(1)O(1) because we do not use any other additional data structures.

#### Approach 2: Reverse on Diagonal and then Reverse Left to Right

**Intuition**

The most elegant solution for rotating the matrix is to firstly reverse the matrix around the main diagonal, and then reverse it from left to right. These operations are called **transpose** and **reflect** in linear algebra.

class Solution:

def rotate(self, matrix: List[List[int]]) -> None:

self.transpose(matrix)

self.reflect(matrix)

def transpose(self, matrix):

n = len(matrix)

for i in range(n):

for j in range(i, n):

matrix[j][i], matrix[i][j] = matrix[i][j], matrix[j][i]

def reflect(self, matrix):

n = len(matrix)

for i in range(n):

for j in range(n // 2):

matrix[i][j], matrix[i][-j - 1] = matrix[i][-j - 1], matrix[i][j]

**Complexity Analysis**

Let M*M* be the number of cells in the grid.

* Time complexity : \mathcal{O}(M)O(*M*). We perform two steps; transposing the matrix, and then reversing each row. Transposing the matrix has a cost of \mathcal{O}(M)O(*M*) because we're moving the value of each cell once. Reversing each row also has a cost of \mathcal{O}(M)O(*M*), because again we're moving the value of each cell once.
* Space complexity : \mathcal{O}(1)O(1) because we do not use any other additional data structures.

**Group Anagrams**

Given an array of strings strs, group **the anagrams** together. You can return the answer in **any order**.

An **Anagram** is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

**Example 1:**

**Input:** strs = ["eat","tea","tan","ate","nat","bat"]

**Output:** [["bat"],["nat","tan"],["ate","eat","tea"]]

**Example 2:**

**Input:** strs = [""]

**Output:** [[""]]

**Example 3:**

**Input:** strs = ["a"]

**Output:** [["a"]]

**Constraints:**

* 1 <= strs.length <= 104
* 0 <= strs[i].length <= 100
* strs[i] consists of lowercase English letters.

## **Solution Article**

#### Approach 1: Categorize by Sorted String

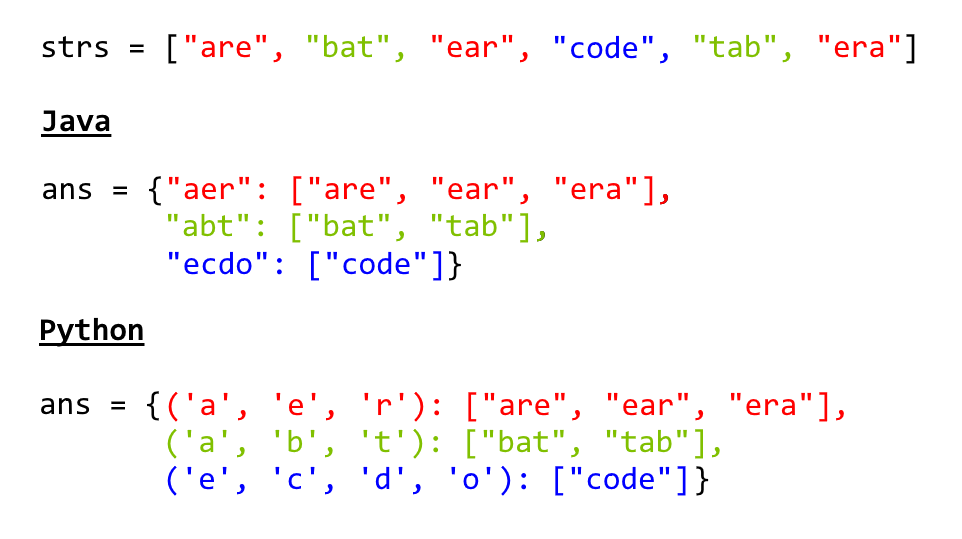
**Intuition**

Two strings are anagrams if and only if their sorted strings are equal.

**Algorithm**

Maintain a map ans : {String -> List} where each key \text{K}K is a sorted string, and each value is the list of strings from the initial input that when sorted, are equal to \text{K}K.

In Java, we will store the key as a string, eg. code. In Python, we will store the key as a hashable tuple, eg. ('c', 'o', 'd', 'e').



class Solution(object):

def groupAnagrams(self, strs):

ans = collections.defaultdict(list)

for s in strs:

ans[tuple(sorted(s))].append(s)

return ans.values()

**Complexity Analysis**

* Time Complexity: O(NK \log K)*O*(*NK*log*K*), where N*N* is the length of strs, and K*K* is the maximum length of a string in strs. The outer loop has complexity O(N)*O*(*N*) as we iterate through each string. Then, we sort each string in O(K \log K)*O*(*K*log*K*) time.
* Space Complexity: O(NK)*O*(*NK*), the total information content stored in ans.

#### Approach 2: Categorize by Count

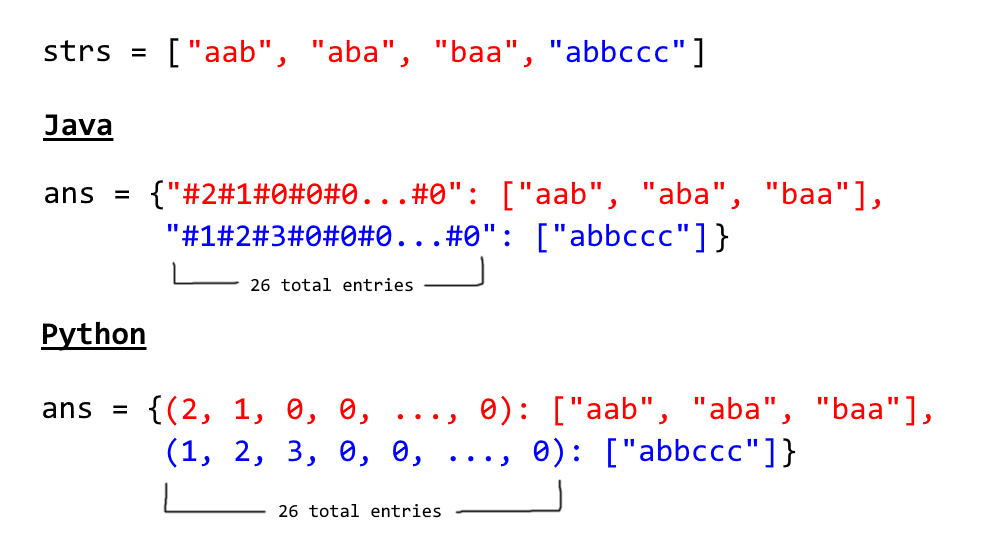
**Intuition**

Two strings are anagrams if and only if their character counts (respective number of occurrences of each character) are the same.

**Algorithm**

We can transform each string \text{s}s into a character count, \text{count}count, consisting of 26 non-negative integers representing the number of \text{a}a's, \text{b}b's, \text{c}c's, etc. We use these counts as the basis for our hash map.

In Java, the hashable representation of our count will be a string delimited with '**#**' characters. For example, abbccc will be #1#2#3#0#0#0...#0 where there are 26 entries total. In python, the representation will be a tuple of the counts. For example, abbccc will be (1, 2, 3, 0, 0, ..., 0), where again there are 26 entries total.



class Solution:

def groupAnagrams(strs):

ans = collections.defaultdict(list)

for s in strs:

count = [0] \* 26

for c in s:

count[ord(c) - ord('a')] += 1

ans[tuple(count)].append(s)

return ans.values()

**Complexity Analysis**

* Time Complexity: O(NK)*O*(*NK*), where N*N* is the length of strs, and K*K* is the maximum length of a string in strs. Counting each string is linear in the size of the string, and we count every string.
* Space Complexity: O(NK)*O*(*NK*), the total information content stored in ans.

**Minimum Window Substring**

Given two strings s and t of lengths m and n respectively, return *the****minimum window substring****of*s*such that every character in*t*(****including duplicates****) is included in the window. If there is no such substring, return the empty string*""*.*

The testcases will be generated such that the answer is **unique**.

A **substring** is a contiguous sequence of characters within the string.

**Example 1:**

**Input:** s = "ADOBECODEBANC", t = "ABC"

**Output:** "BANC"

**Explanation:** The minimum window substring "BANC" includes 'A', 'B', and 'C' from string t.

**Example 2:**

**Input:** s = "a", t = "a"

**Output:** "a"

**Explanation:** The entire string s is the minimum window.

**Example 3:**

**Input:** s = "a", t = "aa"

**Output:** ""

**Explanation:** Both 'a's from t must be included in the window.

Since the largest window of s only has one 'a', return empty string.

**Constraints:**

* m == s.length
* n == t.length
* 1 <= m, n <= 105
* s and t consist of uppercase and lowercase English letters.

**Follow up:** Could you find an algorithm that runs in O(m + n) time?

## **Solution**

#### Approach 1: Sliding Window

**Intuition**

The question asks us to return the minimum window from the string S*S* which has all the characters of the string T*T*. Let us call a window desirable if it has all the characters from T*T*.

We can use a simple sliding window approach to solve this problem.

In any sliding window based problem we have two pointers. One right*right* pointer whose job is to expand the current window and then we have the left*left* pointer whose job is to contract a given window. At any point in time only one of these pointers move and the other one remains fixed.

The solution is pretty intuitive. We keep expanding the window by moving the right pointer. When the window has all the desired characters, we contract (if possible) and save the smallest window till now.

The answer is the smallest desirable window.

For eg. S = "ABAACBAB" T = "ABC". Then our answer window is "ACB" and shown below is one of the possible desirable windows.

Table

Description automatically generated

**Algorithm**

1. We start with two pointers, left*left* and right*right* initially pointing to the first element of the string S*S*.
2. We use the right*right* pointer to expand the window until we get a desirable window i.e. a window that contains all of the characters of T*T*.
3. Once we have a window with all the characters, we can move the left pointer ahead one by one. If the window is still a desirable one we keep on updating the minimum window size.
4. If the window is not desirable any more, we repeat step \; 2*step*2 onwards.

Diagram

Description automatically generated

The above steps are repeated until we have looked at all the windows. The smallest window is returned.

Diagram

Description automatically generated

def minWindow(self, s, t):

"""

:type s: str

:type t: str

:rtype: str

"""

if not t or not s:

return ""

# Dictionary which keeps a count of all the unique characters in t.

dict\_t = Counter(t)

# Number of unique characters in t, which need to be present in the desired window.

required = len(dict\_t)

# left and right pointer

l, r = 0, 0

# formed is used to keep track of how many unique characters in t are present in the current window in its desired frequency.

# e.g. if t is "AABC" then the window must have two A's, one B and one C. Thus formed would be = 3 when all these conditions are met.

formed = 0

# Dictionary which keeps a count of all the unique characters in the current window.

window\_counts = {}

# ans tuple of the form (window length, left, right)

ans = float("inf"), None, None

while r < len(s):

# Add one character from the right to the window

character = s[r]

window\_counts[character] = window\_counts.get(character, 0) + 1

# If the frequency of the current character added equals to the desired count in t then increment the formed count by 1.

if character in dict\_t and window\_counts[character] == dict\_t[character]:

formed += 1

# Try and contract the window till the point where it ceases to be 'desirable'.

while l <= r and formed == required:

character = s[l]

# Save the smallest window until now.

if r - l + 1 < ans[0]:

ans = (r - l + 1, l, r)

# The character at the position pointed by the `left` pointer is no longer a part of the window.

window\_counts[character] -= 1

if character in dict\_t and window\_counts[character] < dict\_t[character]:

formed -= 1

# Move the left pointer ahead, this would help to look for a new window.

l += 1

# Keep expanding the window once we are done contracting.

r += 1

return "" if ans[0] == float("inf") else s[ans[1] : ans[2] + 1]

**Complexity Analysis**

* Time Complexity: O(|S| + |T|)*O*(∣*S*∣+∣*T*∣) where |S| and |T| represent the lengths of strings S*S* and T*T*. In the worst case we might end up visiting every element of string S*S* twice, once by left pointer and once by right pointer. |T|∣*T*∣ represents the length of string T*T*.
* Space Complexity: O(|S| + |T|)*O*(∣*S*∣+∣*T*∣). |S|∣*S*∣ when the window size is equal to the entire string S*S*. |T|∣*T*∣ when T*T* has all unique characters.

#### Approach 2: Optimized Sliding Window

**Intuition**

A small improvement to the above approach can reduce the time complexity of the algorithm to O(2\*|filtered\\_S| + |S| + |T|)*O*(2∗∣*filtered*\_*S*∣+∣*S*∣+∣*T*∣), where filtered\\_S*filtered*\_*S* is the string formed from S by removing all the elements not present in T*T*.

This complexity reduction is evident when |filtered\\_S| <<< |S|∣*filtered*\_*S*∣<<<∣*S*∣.

This kind of scenario might happen when length of string T*T* is way too small than the length of string S*S* and string S*S* consists of numerous characters which are not present in T*T*.

**Algorithm**

We create a list called filtered\\_S*filtered*\_*S* which has all the characters from string S*S* along with their indices in S*S*, but these characters should be present in T*T*.

S = "ABCDDDDDDEEAFFBC" T = "ABC"

filtered\_S = [(0, 'A'), (1, 'B'), (2, 'C'), (11, 'A'), (14, 'B'), (15, 'C')]

Here (0, 'A') means in string S character A is at index 0.

def minWindow(self, s, t):

"""

:type s: str

:type t: str

:rtype: str

"""

if not t or not s:

return ""

dict\_t = Counter(t)

required = len(dict\_t)

# Filter all the characters from s into a new list along with their index.

# The filtering criteria is that the character should be present in t.

filtered\_s = []

for i, char in enumerate(s):

if char in dict\_t:

filtered\_s.append((i, char))

l, r = 0, 0

formed = 0

window\_counts = {}

ans = float("inf"), None, None

# Look for the characters only in the filtered list instead of entire s. This helps to reduce our search.

# Hence, we follow the sliding window approach on as small list.

while r < len(filtered\_s):

character = filtered\_s[r][1]

window\_counts[character] = window\_counts.get(character, 0) + 1

if window\_counts[character] == dict\_t[character]:

formed += 1

# If the current window has all the characters in desired frequencies i.e. t is present in the window

while l <= r and formed == required:

character = filtered\_s[l][1]

# Save the smallest window until now.

end = filtered\_s[r][0]

start = filtered\_s[l][0]

if end - start + 1 < ans[0]:

ans = (end - start + 1, start, end)

window\_counts[character] -= 1

if window\_counts[character] < dict\_t[character]:

formed -= 1

l += 1

r += 1

return "" if ans[0] == float("inf") else s[ans[1] : ans[2] + 1]