**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.
* **Most Common Word**

Given a string paragraph and a string array of the banned words banned, return *the most frequent word that is not banned*. It is **guaranteed** there is **at least one word** that is not banned, and that the answer is **unique**.

The words in paragraph are **case-insensitive** and the answer should be returned in **lowercase**.

**Example 1:**

**Input:** paragraph = "Bob hit a ball, the hit BALL flew far after it was hit.", banned = ["hit"]

**Output:** "ball"

**Explanation:**

"hit" occurs 3 times, but it is a banned word.

"ball" occurs twice (and no other word does), so it is the most frequent non-banned word in the paragraph.

Note that words in the paragraph are not case sensitive,

that punctuation is ignored (even if adjacent to words, such as "ball,"),

and that "hit" isn't the answer even though it occurs more because it is banned.

**Example 2:**

**Input:** paragraph = "a.", banned = []

**Output:** "a"

**Constraints:**

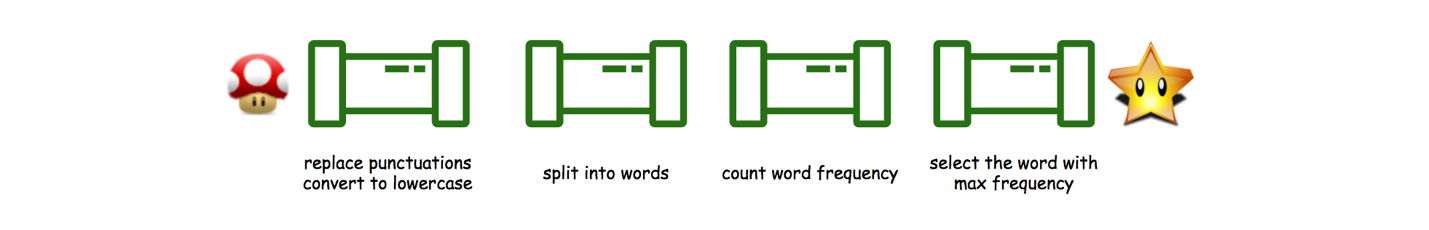
* 1 <= paragraph.length <= 1000
* paragraph consists of English letters, space ' ', or one of the symbols: "!?',;.".
* 0 <= banned.length <= 100
* 1 <= banned[i].length <= 10
* banned[i] consists of only lowercase English letters.

#### Approach 1: String Processing in Pipeline

**Intuition**

We can solve the problem by breaking it into a series of sequential tasks. Each task functions like a stage in a pipeline, which takes the input from the previous stage and then channels its output to the next stage.

More specifically, for this problem, we could break it down into the following stages:



1. We replace all the punctuations with spaces and at the same time convert each letter to its lowercase. One could also accomplish this in two stages. Here we merge them together in one stage.
2. We split the output in the above step into words, with the separator of spaces.
3. We then iterate through the words to count the appearance of each unique word, excluding the words from the banned list.
4. With the hashmap of {word->count}, we then walk through all the items to find the word with the highest frequency.

**Algorithm**

Following the stages we explained before, here are some sample implementations.

class Solution:

def mostCommonWord(self, paragraph: str, banned: List[str]) -> str:

#1). replace the punctuations with spaces,

# and put all letters in lower case

normalized\_str = ''.join([c.lower() if c.isalnum() else ' ' for c in paragraph])

#2). split the string into words

words = normalized\_str.split()

word\_count = defaultdict(int)

banned\_words = set(banned)

#3). count the appearance of each word, excluding the banned words

for word in words:

if word not in banned\_words:

word\_count[word] += 1

#4). return the word with the highest frequency

return max(word\_count.items(), key=operator.itemgetter(1))[0]

**Complexity Analysis**

Let N*N* be the number of characters in the input string and M*M* be the number of characters in the banned list.

* Time Complexity: \mathcal{O}(N + M)O(*N*+*M*).
  + It would take \mathcal{O}(N)O(*N*) time to process each stage of the pipeline as we built.
  + In addition, we built a set out of the list of banned words, which would take \mathcal{O}(M)O(*M*) time.
  + Hence, the overall time complexity of the algorithm is \mathcal{O}(N + M)O(*N*+*M*).
* Space Complexity: \mathcal{O}(N + M)O(*N*+*M*).
  + We built a hashmap to count the frequency of each unique word, whose space would be of \mathcal{O}(N)O(*N*).
  + Similarly, we built a set out of the banned word list, which would consume additional \mathcal{O}(M)O(*M*) space.
  + Therefore, the overall space complexity of the algorithm is \mathcal{O}(N + M)O(*N*+*M*).

#### Approach 2: Character Processing in One-Pass

**Intuition**

With the approach of String manipulation pipeline, it is clear and easy to debug, since we could locate and inspect each stage if anything goes wrong.

However, one might argue that it is probably not the most efficient way to solve the problem, since we scan the input string multiple times.

Indeed, it is possible to process the input string once and only once to accomplish the tasks.

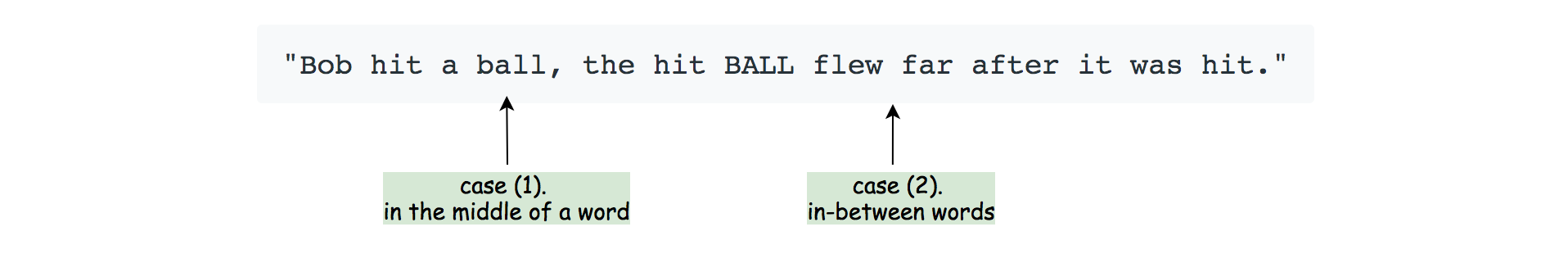
We could iterate through the string character by character, and do the processing ***on-the-fly***, rather than delaying the processing to the latter stages of the pipeline.

The idea is that we consume the input string on the character base. At the moment we reach the end of one word, we can then start to perform the word-based logics such as checking if the word is in the banned list, updating the frequency of the word and also updating the most frequent word we've seen so far etc.

**Algorithm**

We could implement the algorithm in one single loop, over the characters of the input string.

* At each iteration, the character is either of letter (maybe digit), or punctuation or space in other cases.



* Further more, we could divide it into the following two cases:
  + **Case (1):** we are in the middle of a word.
  + **Case (2):** we in in-between the words, e.g. punctuations between the words or at the end of the paragraph.
* We then can organize the logics into the above two cases.
  + In case (1), we simply append the character into the word buffer.
  + In case (2), we do the rest of the logics, as follows:
    - check if the word is enlisted in the banned list.
    - if not, update the frequency of the word.
    - update the most common word that we've seen so far.

class Solution:

def mostCommonWord(self, paragraph: str, banned: List[str]) -> str:

banned\_words = set(banned)

ans = ""

max\_count = 0

word\_count = defaultdict(int)

word\_buffer = []

for p, char in enumerate(paragraph):

#1). consume the characters in a word

if char.isalnum():

word\_buffer.append(char.lower())

if p != len(paragraph)-1:

continue

#2). at the end of one word or at the end of paragraph

if len(word\_buffer) > 0:

word = "".join(word\_buffer)

if word not in banned\_words:

word\_count[word] +=1

if word\_count[word] > max\_count:

max\_count = word\_count[word]

ans = word

# reset the buffer for the next word

word\_buffer = []

return ans

**Complexity Analysis**

Let N*N* be the number of characters in the input string and M*M* be the number of characters in the banned list.

* Time Complexity: \mathcal{O}(N + M)O(*N*+*M*).
  + We traverse each character in the input string once and only once. At each iteration, it takes constant time to perform the operations, except the operation that we build a new string out of the buffer. Excluding the cost of string-building out of the iteration, we can consider the cost of iterations as \mathcal{O}(N)O(*N*).
  + If we combine all the string-building operations all together, in total it would take another \mathcal{O}(N)O(*N*) time.
  + In addition, we built a set out of the list of banned words, which would take \mathcal{O}(M)O(*M*) time.
  + Hence, the overall time complexity of the algorithm is \mathcal{O}(N) + \mathcal{O}(N) + \mathcal{O}(M) = \mathcal{O}(N + M)O(*N*)+O(*N*)+O(*M*)=O(*N*+*M*).
* Space Complexity: \mathcal{O}(N + M)O(*N*+*M*).
  + We built a hashmap to count the frequency of each unique word, whose space would be of \mathcal{O}(N)O(*N*).
  + Similarly, we built a set out of the banned word list, which would consume additional \mathcal{O}(M)O(*M*) space.
  + Therefore, the overall space complexity of the algorithm is \mathcal{O}(N + M)O(*N*+*M*).

class Solution:

def mostCommonWord(self, paragraph: str, banned: List[str]) -> str:

# hash set of banned list

set\_banned = set(banned)

# paragraph lowercase and without punctuation symbols

# replace it with white space

para\_lower = paragraph.lower()

para\_lower = re.sub(r"[!?',;.]+",' ',para\_lower)

#print(para\_lower)

# get the word list now by split method

# from the word list, which occurs how much time

# a dictionary, easy way is using a counter

word\_list = para\_lower.split()

counter = Counter(word\_list).most\_common()

# now return the first most common that is not in the

# banned set

for word,\_ in counter:

if word not in set\_banned:

result = word

break

return result

**Reorder Log Files**

You are given an array of logs. Each log is a space-delimited string of words, where the first word is the **identifier**.

There are two types of logs:

* **Letter-logs**: All words (except the identifier) consist of lowercase English letters.
* **Digit-logs**: All words (except the identifier) consist of digits.

Reorder these logs so that:

1. The **letter-logs** come before all **digit-logs**.
2. The **letter-logs** are sorted lexicographically by their contents. If their contents are the same, then sort them lexicographically by their identifiers.
3. The **digit-logs** maintain their relative ordering.

Return *the final order of the logs*.

**Example 1:**

**Input:** logs = ["dig1 8 1 5 1","let1 art can","dig2 3 6","let2 own kit dig","let3 art zero"]

**Output:** ["let1 art can","let3 art zero","let2 own kit dig","dig1 8 1 5 1","dig2 3 6"]

**Explanation:**

The letter-log contents are all different, so their ordering is "art can", "art zero", "own kit dig".

The digit-logs have a relative order of "dig1 8 1 5 1", "dig2 3 6".

**Example 2:**

**Input:** logs = ["a1 9 2 3 1","g1 act car","zo4 4 7","ab1 off key dog","a8 act zoo"]

**Output:** ["g1 act car","a8 act zoo","ab1 off key dog","a1 9 2 3 1","zo4 4 7"]

**Constraints:**

* 1 <= logs.length <= 100
* 3 <= logs[i].length <= 100
* All the tokens of logs[i] are separated by a **single** space.
* logs[i] is guaranteed to have an identifier and at least one word after the identifier.

#### Overview

First of all, let us put aside the debate whether this problem is an easy or medium one. The problem is a good exercise to practice the technique of **custom sort** in different languages.

The idea of custom sort is that we don't have to rewrite a sorting algorithm every time we have a different ***sorting criteria*** among the elements.

Each language provides certain interface that allows us to **customize** the sorting criteria of the sorting functions, so that we can reuse the implementation of sorting in different scenarios.

In this article, we will present two ways to specify the sorting order, namely by **comparator** and by **sorting key**.

#### Approach 2: Sorting by Keys

**Intuition**

Rather than defining pairwise relationships among all elements in a collection, the order of the elements can also be defined with **sorting keys**.

To illustrate the idea, let us first define a Student object as follows, which has three properties: name, grade, age.

class Student:

def \_\_init\_\_(self, name, grade, age):

self.name = name

self.grade = grade

self.age = age

student\_objects = [

Student('john', 'A', 15),

Student('jane', 'B', 12),

Student('dave', 'B', 10),

]

Now, if we are asked to sort the list of students by age in ascending order, we could simply use the age property of each student as the sorting key, as follows:

>>> sorted(student\_objects, key=lambda student: student.age)

[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]

Furthermore, the key could be a tuple of multiple keys, i.e. tuple(key\_1, key\_2, ... key\_n).

If two elements have the same value on key\_1, the comparison will carry on for the following keys, i.e. key\_2 ... key\_n.

As a result, if we are asked to sort the students first by the grade, then by the age, we can simply return the compound key (stduent.grade, student.age), as follows:

>>> sorted(student\_objects, key=lambda student: (student.grade, student.age))

[('john', 'A', 15), ('dave', 'B', 10), ('jane', 'B', 12)]

**Algorithm**

Given the above intuition, it should be clear that all we need is to translate the rules we defined before into a tuple of keys.

As a reminder, here are a list of the rules that we defined before, concerning the order of logs:

* 1). The letter-logs should be prioritized above all digit-logs.
* 2). Among the letter-logs, we should further sort them based on firstly on their **contents**, and then on their **identifiers** if the contents are identical.
* 3). Among the digit-logs, they should remain in the same order as they are in the collection.

To ensure the above order, we could define a tuple of 3 keys, (key\_1, key\_2, key\_3), as follows:

* key\_1: this key serves as a indicator for the type of logs. For the letter-logs, we could assign its key\_1 with 0, and for the digit-logs, we assign its key\_1 with 1. As we can see, thanks to the assigned values, the letter-logs would take the priority above the digit-logs.
* key\_2: for this key, we use the **content** of the letter-logs as its value, so that among the letter-logs, they would be further ordered based on their content, as required in the Rule (2).
* key\_3: similarly with the key\_2, this key serves to further order the letter-logs. We will use the **identifier** of the letter-logs as its value, so that for the letter-logs with the same content, we could further sort the logs based on its identifier, as required in the Rule (2).

**Note:** for the digit-logs, we don't need the key\_2 and key\_3. We can simply assign the None value to these two keys. As a result, the key value for all the digit-logs would be (1, None, None).

Finally, thanks to the **stability** of sorting algorithms, the elements with the same key value would remain the same order as in the original input. Therefore, the Rule (3) is ensured.

class Solution:

def reorderLogFiles(self, logs: List[str]) -> List[str]:

def get\_key(log):

\_id, rest = log.split(" ", maxsplit=1)

return (0, rest, \_id) if rest[0].isalpha() else (1, )

return sorted(logs, key=get\_key)

**Complexity Analysis**

Let N*N* be the number of logs in the list and M*M* be the maximum length of a single log.

* Time Complexity: \mathcal{O}(M \cdot N \cdot \log N)O(*M*⋅*N*⋅log*N*)
  + The sorted() in Python is implemented with the [Timsort](https://en.wikipedia.org/wiki/Timsort) algorithm whose time complexity is \mathcal{O}(N \cdot \log N)O(*N*⋅log*N*).
  + Since the keys of the elements are basically the logs itself, the comparison between two keys can take up to \mathcal{O}(M)O(*M*) time.
  + Therefore, the overall time complexity of the algorithm is \mathcal{O}(M \cdot N \cdot \log N)O(*M*⋅*N*⋅log*N*).
* Space Complexity: \mathcal{O}(M \cdot N)O(*M*⋅*N*)
  + First, we need \mathcal{O}(M \cdot N)O(*M*⋅*N*) space to keep the keys for the log.
  + In addition, the worst space complexity of the [Timsort](https://en.wikipedia.org/wiki/Timsort) algorithm is \mathcal{O}(N)O(*N*), assuming that the space for each element is \mathcal{O}(1)O(1). Hence we would need \mathcal{O}(M \cdot N)O(*M*⋅*N*) space to hold the intermediate values for sorting.
  + In total, the overall space complexity of the algorithm is \mathcal{O}(M \cdot N + M \cdot N) = \mathcal{O}(M \cdot N)O(*M*⋅*N*+*M*⋅*N*)=O(*M*⋅*N*).

**Trapping Rain Water**

**Solution**

Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it can trap after raining.

**Example 1:**

A picture containing text, clipart

Description automatically generated

**Input:** height = [0,1,0,2,1,0,1,3,2,1,2,1]

**Output:** 6

**Explanation:** The above elevation map (black section) is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are being trapped.

**Example 2:**

**Input:** height = [4,2,0,3,2,5]

**Output:** 9

**Constraints:**

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

class Solution:

def trap(self, height):

"""

:type height: List[int]

:rtype: int

"""

areas = 0

max\_l = max\_r = 0

l = 0

r = len(height)-1

while l < r:

if height[l] < height[r]:

if height[l] > max\_l:

max\_l = height[l]

else:

areas += max\_l - height[l]

l +=1

else:

if height[r] > max\_r:

max\_r = height[r]

else:

areas += max\_r - height[r]

r -=1

return areas

**K Closest Points to Origin**

Given an array of points where points[i] = [xi, yi] represents a point on the **X-Y** plane and an integer k, return the k closest points to the origin (0, 0).

The distance between two points on the **X-Y** plane is the Euclidean distance (i.e., √(x1 - x2)2 + (y1 - y2)2).

You may return the answer in **any order**. The answer is **guaranteed** to be **unique** (except for the order that it is in).

**Example 1:**

Chart, line chart

Description automatically generated

**Input:** points = [[1,3],[-2,2]], k = 1

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

**Explanation:**

The distance between (1, 3) and the origin is sqrt(10).

The distance between (-2, 2) and the origin is sqrt(8).

Since sqrt(8) < sqrt(10), (-2, 2) is closer to the origin.

We only want the closest k = 1 points from the origin, so the answer is just [[-2,2]].

**Example 2:**

**Input:** points = [[3,3],[5,-1],[-2,4]], k = 2

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

**Explanation:** The answer [[-2,4],[3,3]] would also be accepted.

**Constraints:**

* 1 <= k <= points.length <= 104
* -104 < xi, yi < 104

class Solution(object):

def kClosest(self, points, k):

return sorted(points, key=lambda x:x[0]\*\*2 + x[1]\*\*2)[:k]

**Longest Palindromic Substring**

**Solution**

Given a string s, return *the longest palindromic substring* in s.

**Example 1:**

**Input:** s = "babad"

**Output:** "bab"

**Note:** "aba" is also a valid answer.

**Example 2:**

**Input:** s = "cbbd"

**Output:** "bb"

**Example 3:**

**Input:** s = "a"

**Output:** "a"

**Example 4:**

**Input:** s = "ac"

**Output:** "a"

**Constraints:**

* 1 <= s.length <= 1000
* s consist of only digits and English letters.

## Approach 1: Longest Common Substring

Apply Dynamic Programming to find the longest common substring, check the index.

### Idea

1. Reverse the string SS to S′S′, find the longest common substring between SS and S′S′;
2. Check the index of the found common substring in SS and S′S′ to fix counter cases like “abcdefcba” (whose reverse is “abcfedcba” -> common sub is “abc” which is wrong).

Dynamic Update Rule: dp[i][j]=⎧⎩⎨⎪⎪⎪⎪01dp[i−1][j−1]+10(if i == 0 and j == 0)(elif i == 0 or j == 0 and dp[i] == dp[j]))(elif dp[i] == dp[j])otherwisedp[i][j]={0(if i == 0 and j == 0)1(elif i == 0 or j == 0 and dp[i] == dp[j]))dp[i−1][j−1]+1(elif dp[i] == dp[j])0otherwise

Calendar

Description automatically generated with medium confidence

Check index of:

* i - dp[i][j] + 1 == len(s) - 1 - j

**class** **Solution1\_update**:

**def** longestPalindrome(**self**, s: **str**) -> **str**:

**if** s is **""**:

**return** **""**

rev = s[::-**1**]

dp = [**0** **for** i in **range**(**len**(s))]

max\_len = **0**

max\_end = **0**

**for** i in **range**(**len**(s)):

*# Updated the loop order*

**for** j in **range**(**len**(s)-**1**, -**1**, -**1**):

**if** s[i] == rev[j]:

**if** i == **0** or j == **0**:

dp[j] = **1**

**else**:

dp[j] = dp[j-**1**] + **1**

*# Updated here - new columns should be updated on conditions*

**else**:

dp[j] = **0**

**if** dp[j] > max\_len:

**if** i-dp[j]+**1** == **len**(s)-**1**-j:

max\_len = dp[j]

max\_end = i

**return** s[max\_end - max\_len + **1**: max\_end + **1**]

### Complexity

* Time Complexity: O(n2)O(n2)
* Space Complexity: O(n2)

## Approach 2: Brute Force

### Idea

Just pick all possible substrings to verify if it’s a palindrome.

But it cannot be accepted by LC, because the “Time Limit Exceeded!”

### Solution

**class** **Solution2**:

**def** longestPalindrome(**self**, s: **str**) -> **str**:

**def** isPalindrome(s):

**if** s is **""**:

**return** **False**

**for** i in **range**(**len**(s)//**2**):

**if** s[i] != s[-**1**-i]:

**return** **False**

**return** **True**

common\_subs = {}

**if** s is **""**:

**return** **""**

**for** i in **range**(**len**(s)):

**for** j in **range**(**1**, **len**(s)+**1**):

**if** isPalindrome(s[i:j]):

common\_subs[s[i:j]] = **len**(s[i:j])

**return** **max**(common\_subs, key=common\_subs.get)

### Complexity Analysis

* Time Complexity: O(n3)O(n3)
* Space Complexity: O(1)O(1)

## Approach 3: Dynamic Programming (Improving the Brute Force)

### Idea

We can save some repeated computation in the brute force.

First, we define a function PP: P(i, j)={TrueFalse(if s[i:j] is palindromic)(if s[i:j] is not palindromic)P(i, j)={True(if s[i:j] is palindromic)False(if s[i:j] is not palindromic)

Thus,

P(i, j)=(P(i+1, j-1) and s[i] == s[j])P(i, j)=(P(i+1, j-1) and s[i] == s[j])

The base cases are: P(i, i)=TrueP(i, i+1) = (s[i] == s[i+1])P(i, i)=TrueP(i, i+1) = (s[i] == s[i+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…

Table, calendar

Description automatically generated

### Solution

**class** **Solution3**:

**def** longestPalindrome(**self**, s: **str**) -> **str**:

**if** s is **""**:

**return** s

res = **""**

dp = [[**None** **for** i in **range**(**len**(s))] **for** j in **range**(**len**(s))]

**for** j in **range**(**len**(s)):

**for** i in **range**(j+):

**if** i == j:

dp[j][i] = **True**

**elif** j == i+**1**:

dp[j][i] = (s[i] == s[j])

**else**:

dp[j][i] = (dp[j-**1**][i+**1**] and s[i] == s[j])

**if** dp[j][i] and j - i + **1** > **len**(res):

res = s[i:j+**1**]

**return** res

### Complexity Analysis

* Time Complexity: O(n2)O(n2)
* Space Complexity: O(n2)O(n2)

## Solution 3 Update: Improve Space Complexity

### Idea

or each line of j, dp only depends on previous line of j. Thus, we just need a n by 1 array instead of n by n matrix.

### Solution

**class** **Solution3**:

**def** longestPalindrome(**self**, s: **str**) -> **str**:

**if** s is **""**:

**return** s

res = **""**

dp = [**None** **for** i in **range**(**len**(s))]

**for** j in **range**(**len**(s)):

**for** i in **range**(j+**1**):

**if** i == j:

dp[i] = **True**

**elif** j == i+**1**:

dp[i] = (s[i] == s[j])

**else**:

dp[i] = (dp[i+**1**] and s[i] == s[j])

**if** dp[i] and j - i + **1** > **len**(res):

res = s[i:j+**1**]

**return** res

### Complexity Analysis

* Time Complexity: O(n2)O(n2)
* Space Complexity: O(n)O(n)