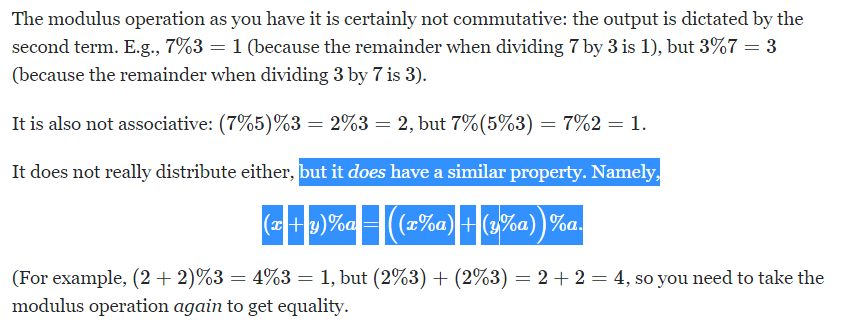
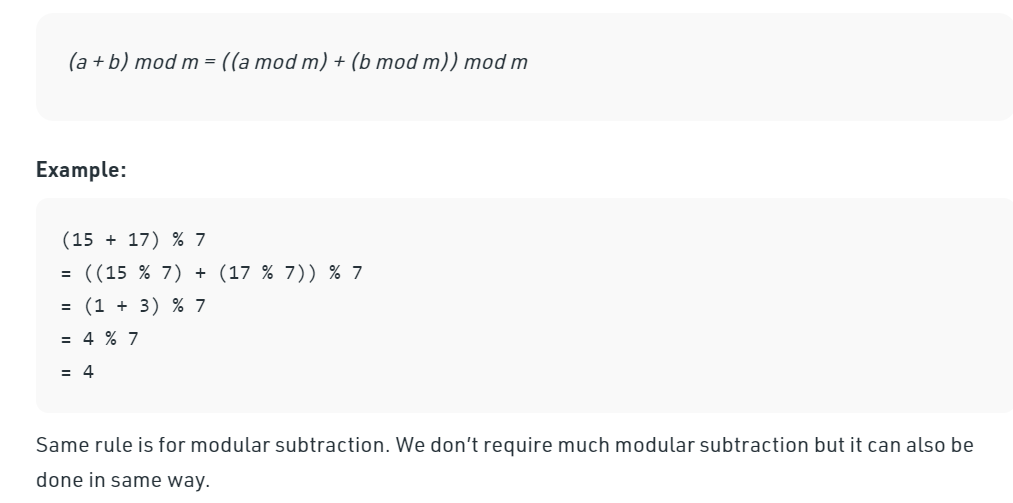
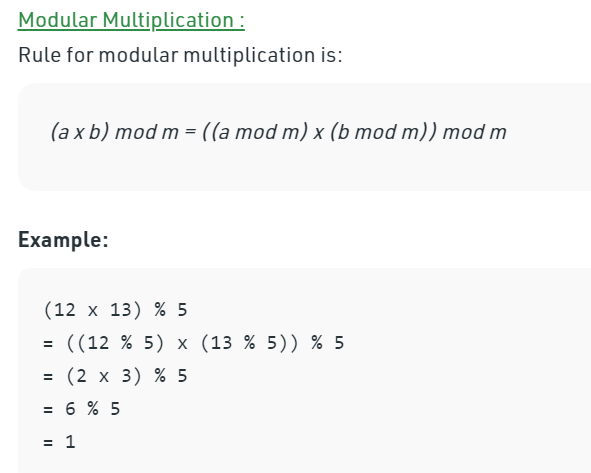
6-4-21

1. **Binary Search**
2. **Priority Q**
3. **Add to dict, and then linear scanning**
4. **DP: longest increasing sub-sequence**
5. **Sum of subarrays == target value**

* Modulus operation







1. **79. Word Search**

Approach 1: Backtracking

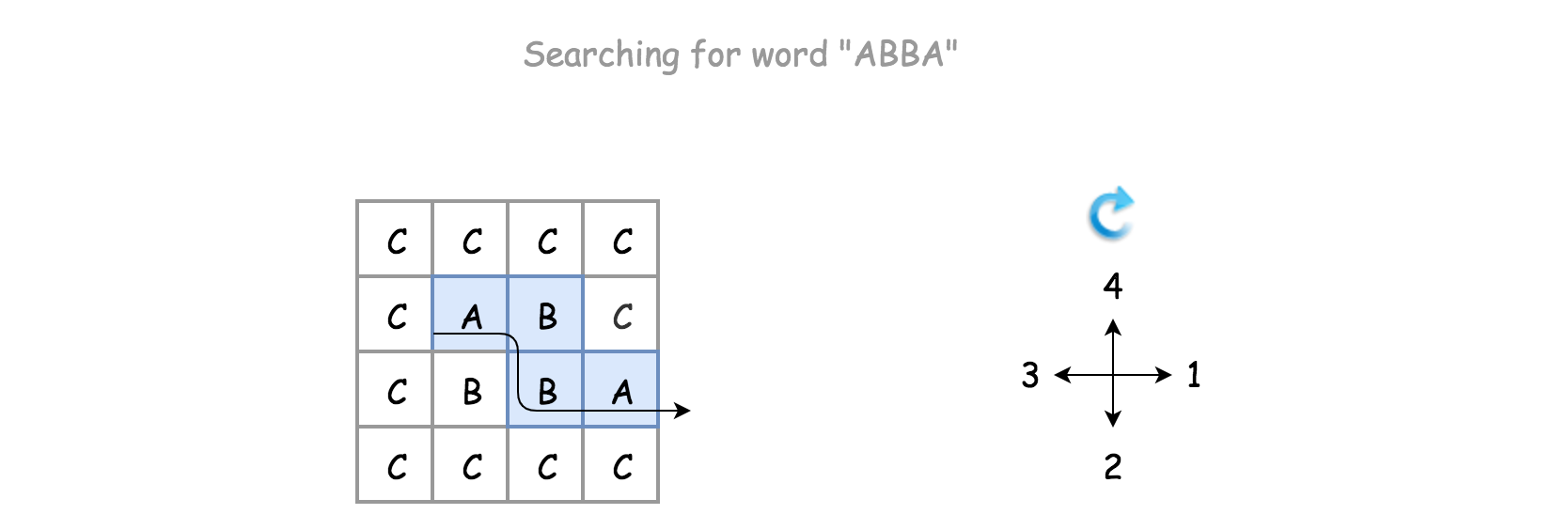
**Intuition**

This problem is yet another 2D grid traversal problem, which is similar with another problem called [489. Robot Room Cleaner](https://leetcode.com/articles/robot-room-cleaner/).

Many people in the [discussion forum](https://leetcode.com/problems/word-search/discuss/) claimed that the solution is of ***DFS*** (Depth-First Search). Although it is true that we would explore the 2D grid with the DFS strategy for this problem, it does not capture the entire nature of the solution.

We argue that a more accurate term to summarize the solution would be ***backtracking***, which is a methodology where we mark the current path of exploration, if the path does not lead to a solution, we then revert the change (*i.e.* backtracking) and try another path.

As the general idea for the solution, we would walk around the 2D grid, at each step we *mark* our choice before jumping into the next step. And at the end of each step, we would also revert our marking, so that we could have a *clean slate* to try another *direction*. In addition, the exploration is done via the *DFS* strategy, where we go as further as possible before we try the next direction.



**Algorithm**

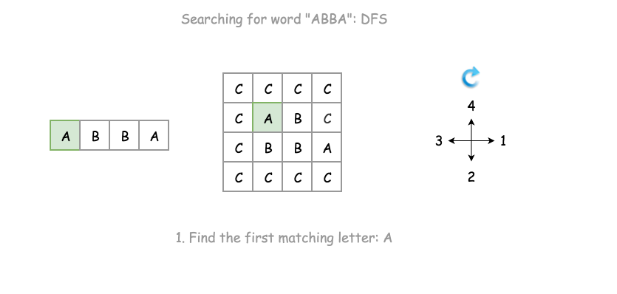
There is a certain code pattern for all the algorithms of backtracking. For example, one can find one template in our [Explore card of Recursion II](https://leetcode.com/explore/learn/card/recursion-ii/472/backtracking/2793/).

The skeleton of the algorithm is a loop that iterates through each cell in the grid. For each cell, we invoke the *backtracking* function (*i.e.* backtrack()) to check if we would obtain a solution, starting from this very cell.

For the backtracking function backtrack(row, col, suffix), as a DFS algorithm, it is often implemented as a *recursive* function. The function can be broke down into the following four steps:

* Step 1). At the beginning, first we check if we reach the bottom case of the recursion, where the word to be matched is empty, *i.e.* we have already found the match for each prefix of the word.
* Step 2). We then check if the current state is invalid, either the position of the cell is out of the boundary of the board or the letter in the current cell does not match with the first letter of the word.
* Step 3). If the current step is valid, we then start the exploration of backtracking with the strategy of DFS. First, we mark the current cell as *visited*, *e.g.* any non-alphabetic letter will do. Then we iterate through the four possible directions, namely *up*, *right*, *down* and *left*. The order of the directions can be altered, to one's preference.
* Step 4). At the end of the exploration, we revert the cell back to its original state. Finally we return the result of the exploration.

We demonstrate how it works with an example in the following animation.



1 / 6

**Notes**

There are a few choices that we made for our backtracking algorithm, here we elaborate some thoughts that are behind those choices.

Instead of returning directly once we find a match, we simply *break* out of the loop and do the cleanup before returning.

Here is what the alternative solution might look like.

As once notices, we simply return True if the result of recursive call to backtrack() is positive. Though this minor modification would have no impact on the time or space complexity, it would however leave with some "side-effect", *i.e.* the matched letters in the original board would be altered to #.

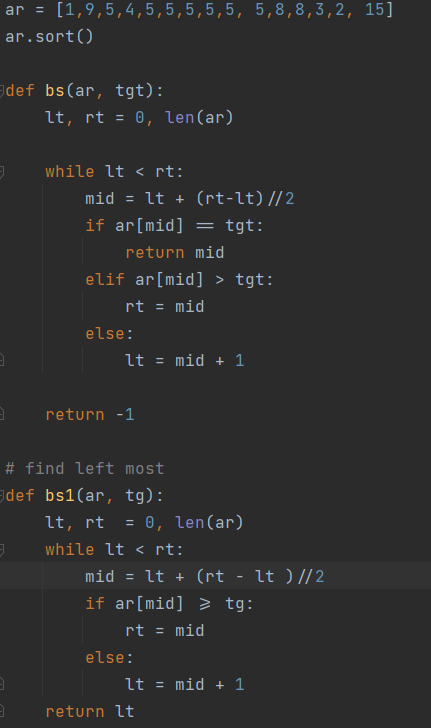
Instead of doing the boundary checks before the recursive call on the backtrack() function, we do it within the function.

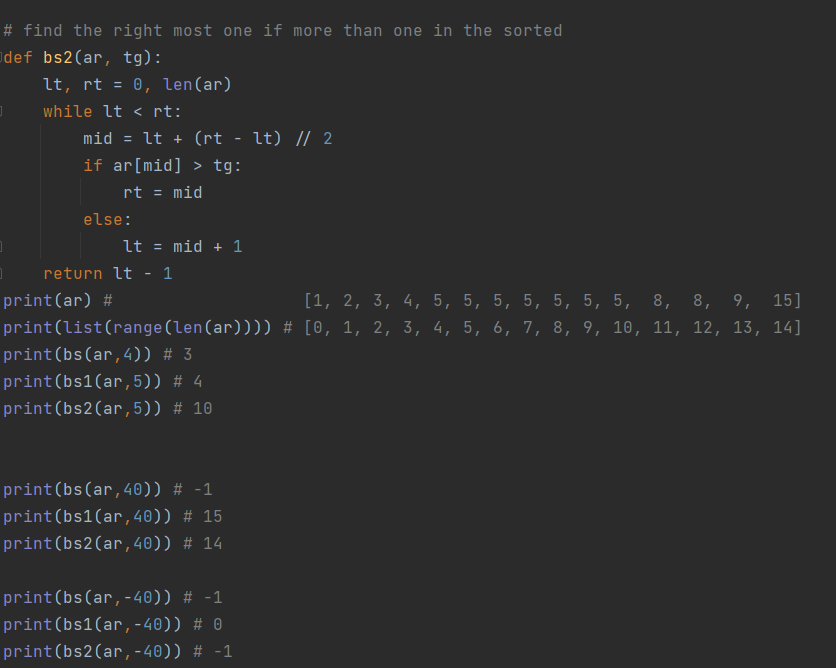
This is an important choice though. Doing the boundary check within the function would allow us to reach the bottom case, for the test case where the board contains only a single cell, since either of neighbor indices would not be valid.

**Complexity**

* Time Complexity: \mathcal{O}(N \cdot 4 ^ L)O(*N*⋅4*L*) where N*N* is the number of cells in the board and L*L* is the length of the word to be matched.  
  + For the backtracking function, its execution trace would be visualized as a 4-ary tree, each of the branches represent a potential exploration in the corresponding direction. Therefore, in the worst case, the total number of invocation would be the number of nodes in a full 4-nary tree, which is about 4^L4*L*.
  + We iterate through the board for backtracking, *i.e.* there could be N*N* times invocation for the backtracking function in the worst case.
  + As a result, overall the time complexity of the algorithm would be \mathcal{O}(N \cdot 4 ^ L)O(*N*⋅4*L*).
* Space Complexity: \mathcal{O}(L)O(*L*) where L*L* is the length of the word to be matched.  
  + The main consumption of the memory lies in the recursion call of the backtracking function. The maximum length of the call stack would be the length of the word. Therefore, the space complexity of the algorithm is \mathcal{O}(L)O(*L*).

1. **Binary Search**

****

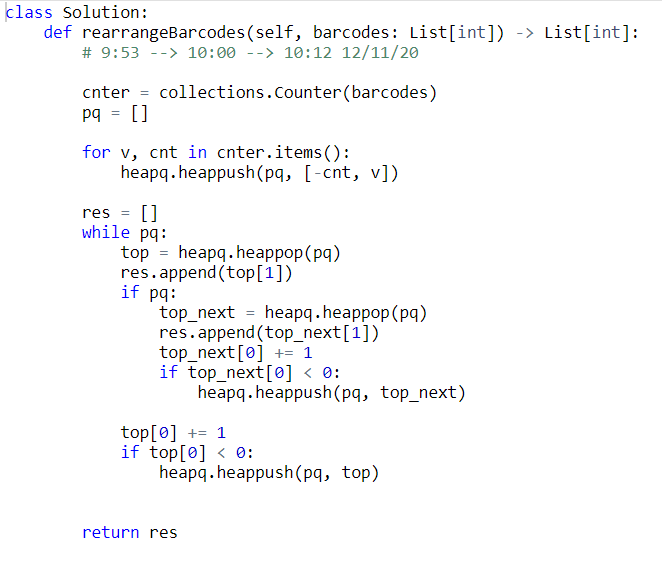
****

ar = [1,9,5,4,5,5,5,5,5, 5,8,8,3,2, 15]  
ar.sort()  
  
def bs(ar, tgt):  
 lt, rt = 0, len(ar)  
  
 while lt < rt:  
 mid = lt + (rt-lt)//2  
 if ar[mid] == tgt:  
 return mid  
 elif ar[mid] > tgt:  
 rt = mid  
 else:  
 lt = mid + 1  
  
 return -1  
  
# find left most  
def bs1(ar, tg):  
 lt, rt = 0, len(ar)  
 while lt < rt:  
 mid = lt + (rt - lt )//2  
 if ar[mid] >= tg:  
 rt = mid  
 else:  
 lt = mid + 1  
 return lt  
  
# find the right most one if more than one in the sorted  
def bs2(ar, tg):  
 lt, rt = 0, len(ar)  
 while lt < rt:  
 mid = lt + (rt - lt) // 2  
 if ar[mid] > tg:  
 rt = mid  
 else:  
 lt = mid + 1  
 return lt - 1  
print(ar) # [1, 2, 3, 4, 5, 5, 5, 5, 5, 5, 5, 8, 8, 9, 15]  
print(list(range(len(ar)))) # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]  
print(bs(ar,4)) # 3  
print(bs1(ar,5)) # 4  
print(bs2(ar,5)) # 10  
  
  
print(bs(ar,40)) # -1  
print(bs1(ar,40)) # 15  
print(bs2(ar,40)) # 14  
  
print(bs(ar,-40)) # -1  
print(bs1(ar,-40)) # 0  
print(bs2(ar,-40)) # -1

1. **Priority Q**

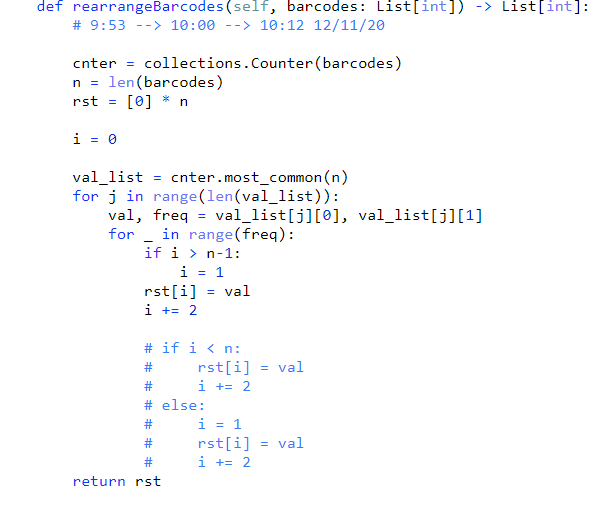
**Heapq, heapq.heappush, heapq.heappop(),nsmallest, nlargest**



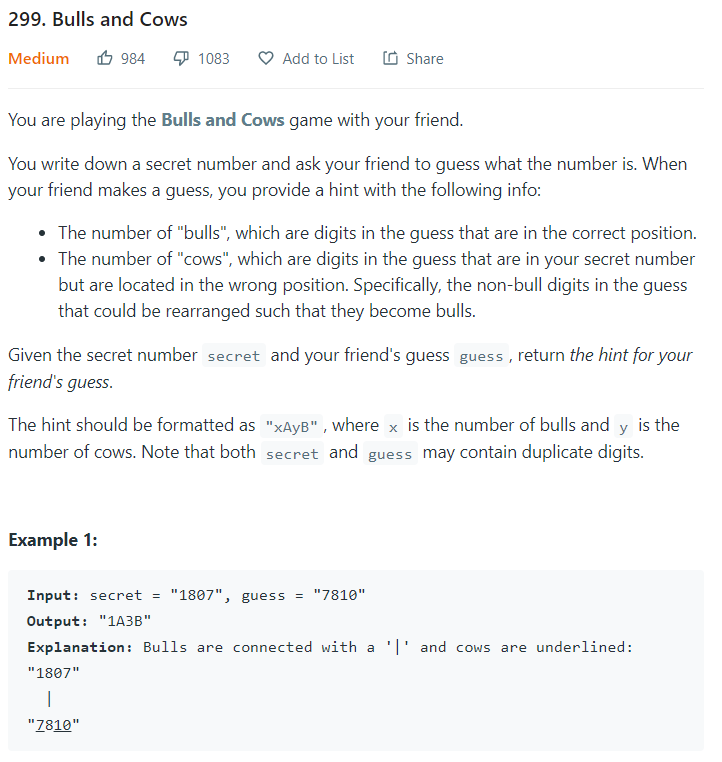


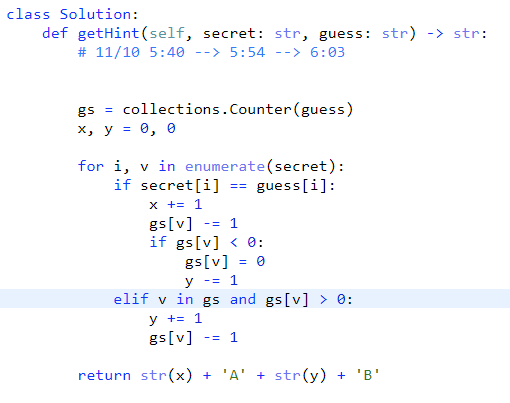
Element of heapq can be a tuple or list and **sorted based on first\_element, then 2nd element and so on**.

Heapq.heappush(-cnt) to sort decreasingly.

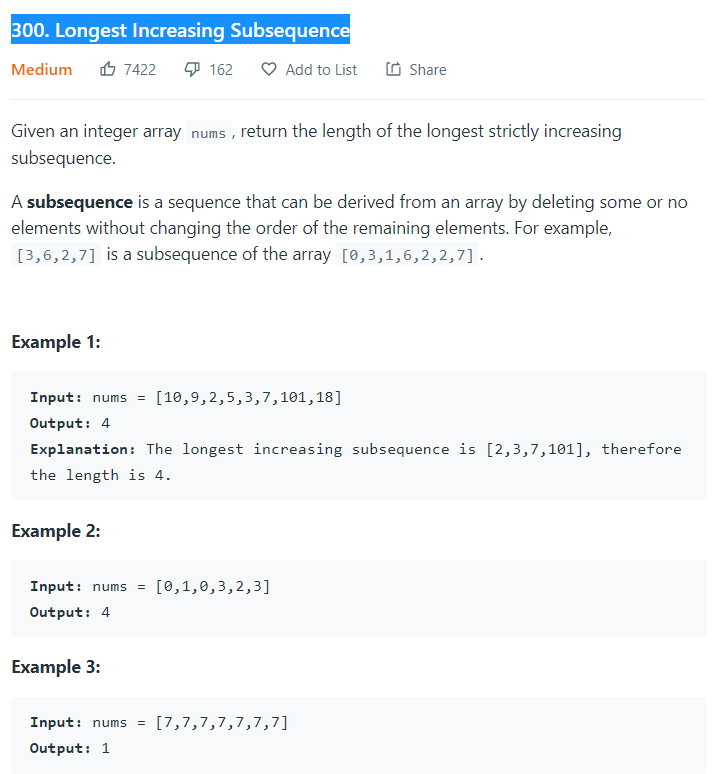


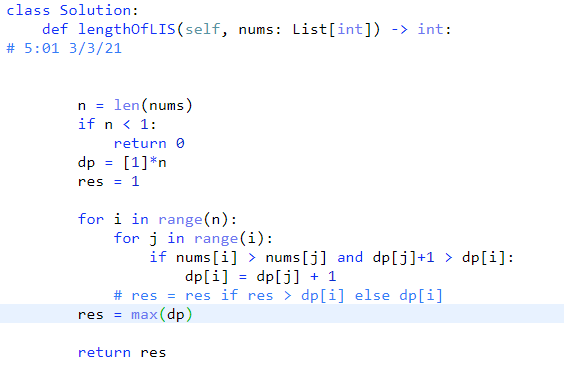
1. **Add to dict, and then linear scanning**
2. **299. Bulls and Cows**





1. **DP: longest increasing sub-sequence**
2. **300. Longest Increasing Subsequence**



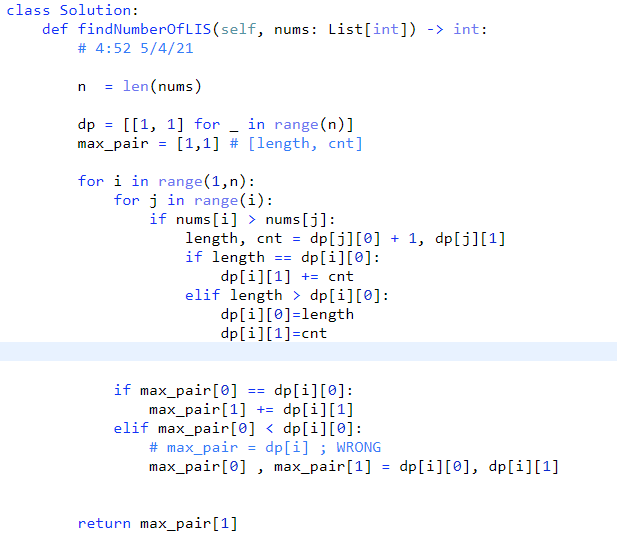


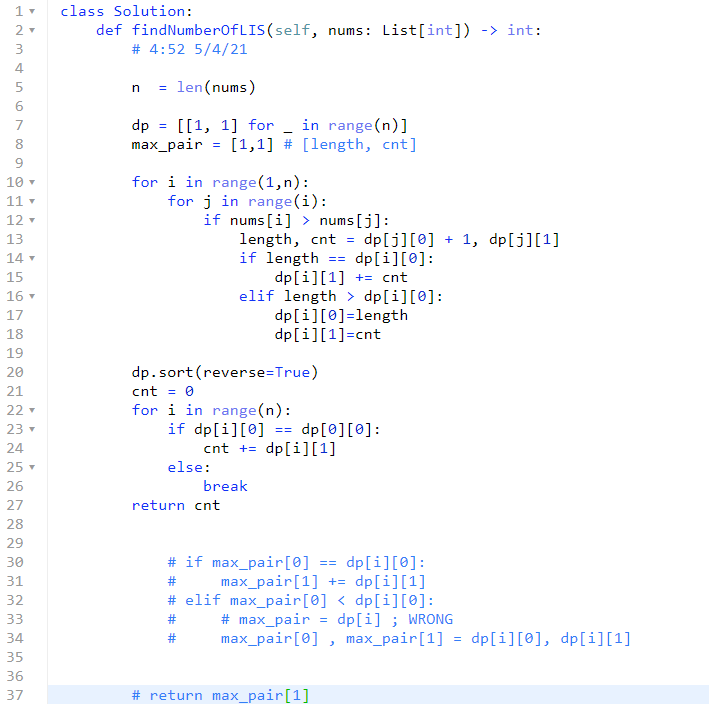
1. **673. Number of Longest Increasing Subsequence**



**# DP with trace back.**

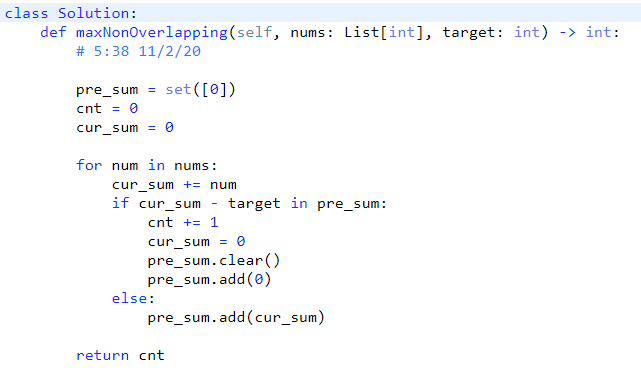
**# each element is a pair of [length, cnt]**

****

****

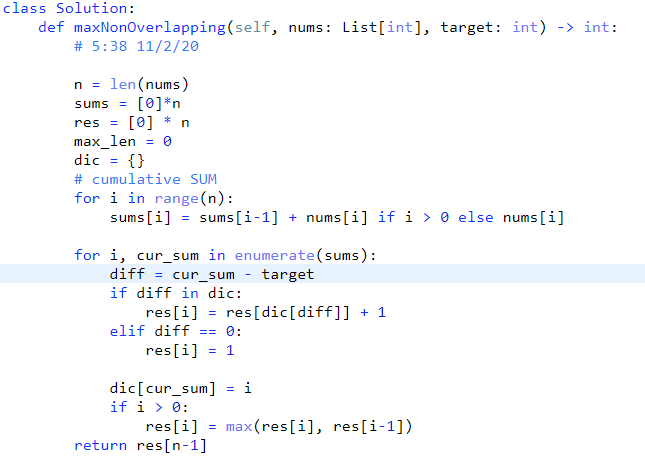
1. **Sum of subarrays == target value**
2. **1546. Maximum Number of Non-Overlapping Subarrays With Sum Equals Target**

****

****

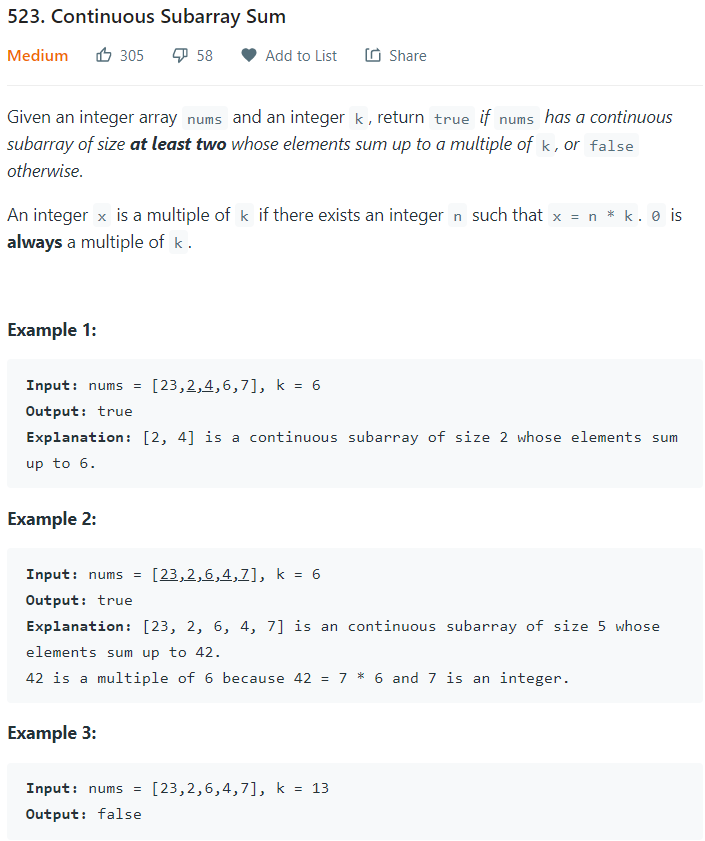
**Calculate cumulative sum**

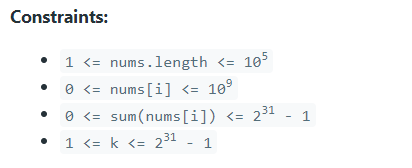
****

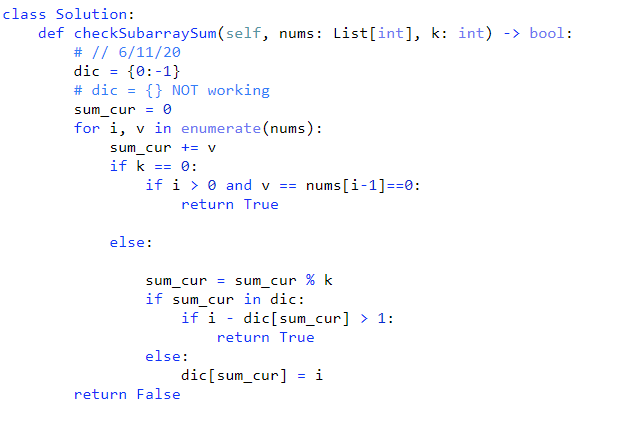
****

**Be careful, the array cumulative sum is NOT strictly increasing since the array element can be 0, or negative.**

1. **523. Continuous Subarray Sum**

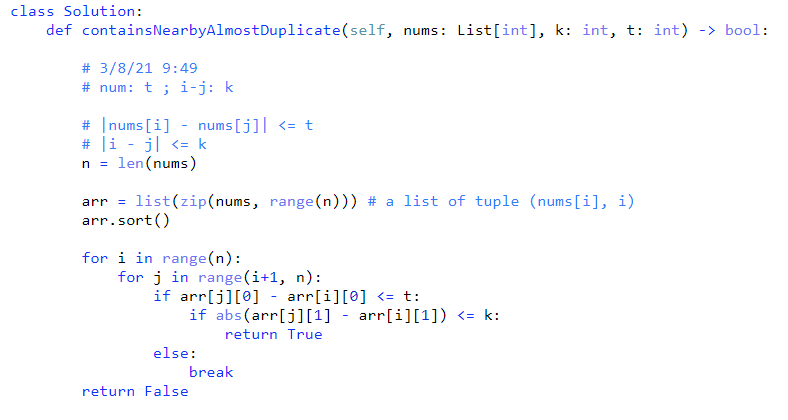
****

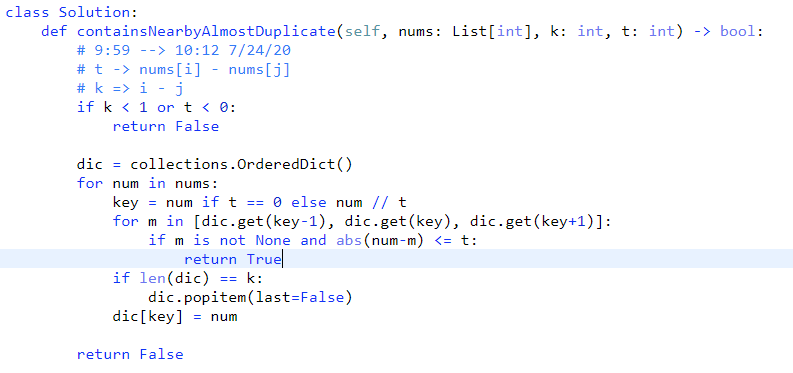
****

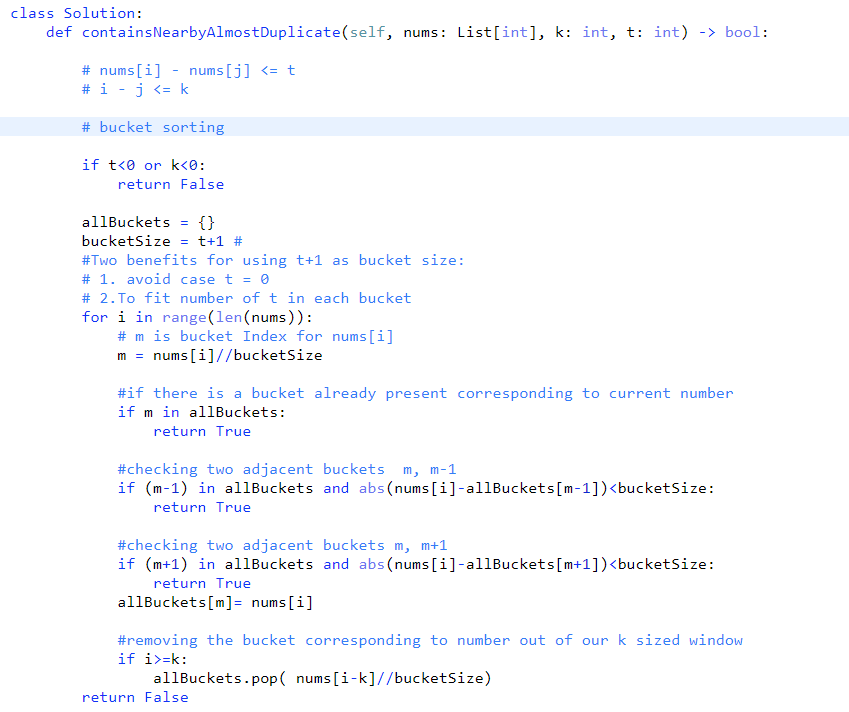
****

1. **Range Check**
2. **220 Contains Duplicate III**

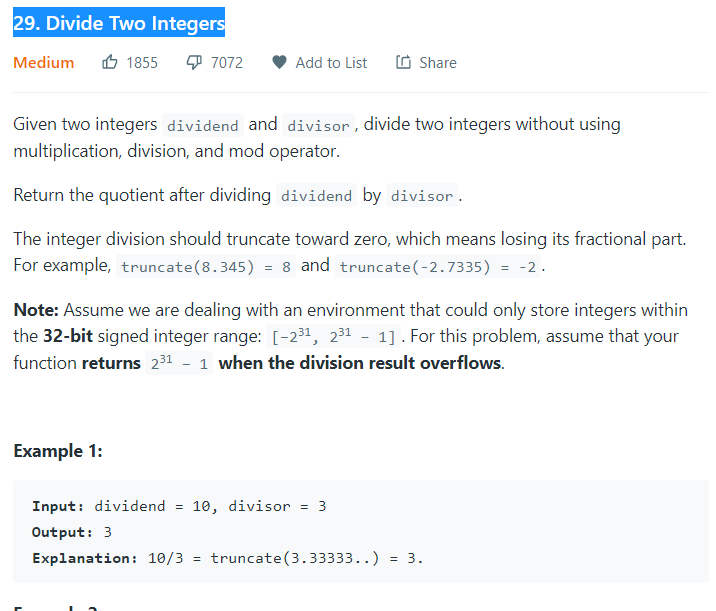
****

****

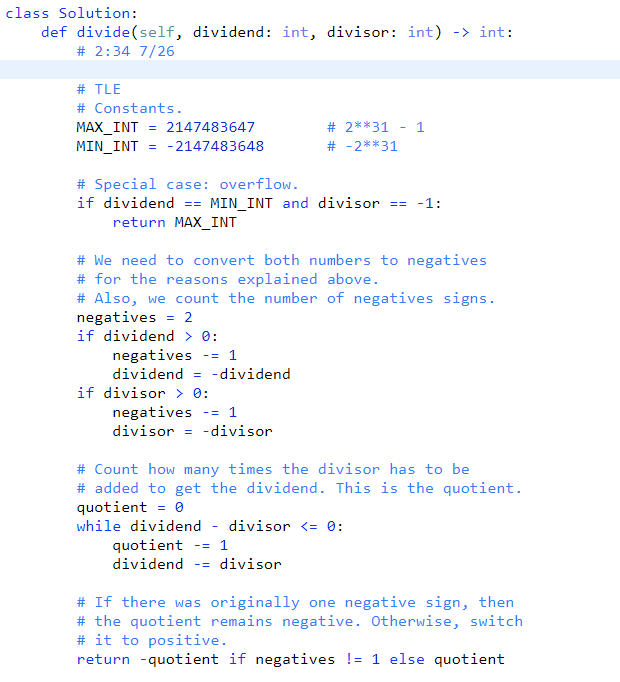
****

****

1. **Tricky problems**
2. **29. Divide Two Integers**

****

* **Repeated Subtractions**

****

#### Repeated Exponential Searches

#### 