# ﻿1,Two Sum\*\*\*:

**Python\_solution:**

Here is a Python solution in O(n) time

class Solution(object):

def twoSum(self, nums, target):

"""

:type nums: List[int]

:type target: int

:rtype: List[int]

"""

if len(nums) <= 1:

return False

buff\_dict = {}

for i in range(len(nums)):

if nums[i] in buff\_dict:

return [buff\_dict[nums[i]], i+1]

else:

buff\_dict[target - nums[i]] = i+1

# 2,Add Two Numbers\*\*\*:

**Python\_solution:**

Clear python code, straight forward

class Solution:

# @return a ListNode

def addTwoNumbers(self, l1, l2):

carry = 0

root = n = ListNode(0)

while l1 or l2 or carry:

v1 = v2 = 0

if l1:

v1 = l1.val

l1 = l1.next

if l2:

v2 = l2.val

l2 = l2.next

carry, val = divmod(v1+v2+carry, 10)

n.next = ListNode(val)

n = n.next

return root.next

# 3,Longest Substring Without Repeating Characters\*\*\*:

**Python\_solution:**

A Python solution - 85ms - O(n)

class Solution:

# @return an integer

def lengthOfLongestSubstring(self, s):

start = maxLength = 0

usedChar = {}

for i in range(len(s)):

if s[i] in usedChar and start <= usedChar[s[i]]:

start = usedChar[s[i]] + 1

else:

maxLength = max(maxLength, i - start + 1)

usedChar[s[i]] = i

return maxLength

# 4,Median of Two Sorted Arrays\*\*\*:

**Python\_solution:**

Intuitive Python O(log (m+n)) solution, by kth smallest in the two sorted arrays, 252ms

The idea is in the comment:

def findMedianSortedArrays(self, A, B):

l = len(A) + len(B)

if l % 2 == 1:

return self.kth(A, B, l // 2)

else:

return (self.kth(A, B, l // 2) + self.kth(A, B, l // 2 - 1)) / 2.

def kth(self, a, b, k):

if not a:

return b[k]

if not b:

return a[k]

ia, ib = len(a) // 2 , len(b) // 2

ma, mb = a[ia], b[ib]

# when k is bigger than the sum of a and b's median indices

if ia + ib < k:

# if a's median is bigger than b's, b's first half doesn't include k

if ma > mb:

return self.kth(a, b[ib + 1:], k - ib - 1)

else:

return self.kth(a[ia + 1:], b, k - ia - 1)

# when k is smaller than the sum of a and b's indices

else:

# if a's median is bigger than b's, a's second half doesn't include k

if ma > mb:

return self.kth(a[:ia], b, k)

else:

return self.kth(a, b[:ib], k)

# 5,Longest Palindromic Substring\*\*\*:

**Python\_solution:**

Python O(n^2) method with some optimization, 88ms.

Basic thought is simple. when you increase s by 1 character, you could only increase maxPalindromeLen by 1 or 2, and that new maxPalindrome includes this new character. Proof: if on adding 1 character, maxPalindromeLen increased by 3 or more, say the new maxPalindromeLen is Q, and the old maxPalindromeLen is P, and Q>=P+3. Then it would mean, even without this new character, there would be a palindromic substring ending in the last character, whose length is at least Q-2. Since Q-2 would be >P, this contradicts the condition that P is the maxPalindromeLen without the additional character.

So, it becomes simple, you only need to scan from beginning to the end, adding one character at a time, keeping track of maxPalindromeLen, and for each added character, you check if the substrings ending with this new character, with length P+1 or P+2, are palindromes, and update accordingly.

Now, this is O(n^2) as taking substrings and checking palindromicity seem O(n) time. We can speed up it by realizing that strings are immutable, and there are memory slicing tricks will help to speed these operations up. comparing string equality with "==" is O(1), and using slicing to substring and reverse is ̶a̶l̶s̶o̶ ̶O̶(̶1̶)̶ ̶(̶n̶o̶t̶ ̶t̶o̶t̶a̶l̶l̶y̶ ̶s̶u̶r̶e̶ ̶a̶b̶o̶u̶t̶ ̶t̶h̶e̶ ̶s̶l̶i̶c̶i̶n̶g̶ ̶t̶h̶o̶u̶g̶h̶.̶ ̶ ̶I̶ ̶t̶h̶i̶n̶k̶ ̶i̶t̶ ̶i̶s̶ ̶O̶(̶1̶)̶,̶ ̶b̶u̶t̶ ̶c̶o̶u̶l̶d̶ ̶n̶o̶t̶ ̶f̶i̶n̶d̶ ̶a̶n̶y̶ ̶s̶o̶l̶i̶d̶ ̶l̶i̶t̶e̶r̶a̶t̶u̶r̶e̶ ̶a̶b̶o̶u̶t̶ ̶i̶t̶.̶ O(n) (thanks to ChuntaoLu). But as slicing is optimized by the interpreter's C code, it should run pretty fast. I'm pretty new to Python. Would appreciate you would give more insights or further optimization.

Thus, here is the O(n) method:

class Solution:

# @return a string

def longestPalindrome(self, s):

if len(s)==0:

return 0

maxLen=1

start=0

for i in xrange(len(s)):

if i-maxLen >=1 and s[i-maxLen-1:i+1]==s[i-maxLen-1:i+1][::-1]:

start=i-maxLen-1

maxLen+=2

continue

if i-maxLen >=0 and s[i-maxLen:i+1]==s[i-maxLen:i+1][::-1]:

start=i-maxLen

maxLen+=1

return s[start:start+maxLen]

# 6,ZigZag Conversion\*\*\*:

**Python\_solution:**

Python O(n) Solution in 96ms (99.43%)

class Solution(object):

def convert(self, s, numRows):

"""

:type s: str

:type numRows: int

:rtype: str

"""

if numRows == 1 or numRows >= len(s):

return s

L = [''] \* numRows

index, step = 0, 1

for x in s:

L[index] += x

if index == 0:

step = 1

elif index == numRows -1:

step = -1

index += step

return ''.join(L)

# 7,Reverse Integer\*\*\*:

**Python\_solution:**

Golfing in Python

Get the `s`ign, get the `r`eversed absolute integer, and return their product if `r` didn't "overflow".

def reverse(self, x):

s = cmp(x, 0)

r = int(`s\*x`[::-1])

return s\*r \* (r < 2\*\*31)

As compressed one-liner, for potential comparison:

def reverse(self, x):

s=cmp(x,0);r=int(`s\*x`[::-1]);return(r<2\*\*31)\*s\*r

Anybody got something shorter?

# 8,String to Integer (atoi)\*\*\*:

**Python\_solution:**

Python solution based on RegEx

class Solution:

# @return an integer

def atoi(self, str):

str = str.strip()

str = re.findall('(^[\+\-0]\*\d+)\D\*', str)

try:

result = int(''.join(str))

MAX\_INT = 2147483647

MIN\_INT = -2147483648

if result > MAX\_INT > 0:

return MAX\_INT

elif result < MIN\_INT < 0:

return MIN\_INT

else:

return result

except:

return 0

# 9,Palindrome Number\*\*\*:

**Python\_solution:**

Python solution based on the algorithm in leetcode blog

class Solution:

# @param x, an integer

# @return a boolean

def isPalindrome(self, x):

if x < 0:

return False

ranger = 1

while x / ranger >= 10:

ranger \*= 10

while x:

left = x / ranger

right = x % 10

if left != right:

return False

x = (x % ranger) / 10

ranger /= 100

return True

# 10,Regular Expression Matching\*\*\*:

**Python\_solution:**

My DP approach in Python with comments and unittest

I shared my DP approach with comments and provided some unit tests for it. Some statements in the approach directly affect some corner cases, for example, comment out line 22-23, then the unittest `test\_symbol\_0` will fail. Hope this script helps us better understand the problem.

import unittest

class Solution(object):

def isMatch(self, s, p):

# The DP table and the string s and p use the same indexes i and j, but

# table[i][j] means the match status between p[:i] and s[:j], i.e.

# table[0][0] means the match status of two empty strings, and

# table[1][1] means the match status of p[0] and s[0]. Therefore, when

# refering to the i-th and the j-th characters of p and s for updating

# table[i][j], we use p[i - 1] and s[j - 1].

# Initialize the table with False. The first row is satisfied.

table = [[False] \* (len(s) + 1) for \_ in range(len(p) + 1)]

# Update the corner case of matching two empty strings.

table[0][0] = True

# Update the corner case of when s is an empty string but p is not.

# Since each '\*' can eliminate the charter before it, the table is

# vertically updated by the one before previous. [test\_symbol\_0]

for i in range(2, len(p) + 1):

table[i][0] = table[i - 2][0] and p[i - 1] == '\*'

for i in range(1, len(p) + 1):

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

if p[i - 1] != "\*":

# Update the table by referring the diagonal element.

table[i][j] = table[i - 1][j - 1] and \

(p[i - 1] == s[j - 1] or p[i - 1] == '.')

else:

# Eliminations (referring to the vertical element)

# Either refer to the one before previous or the previous.

# I.e. \* eliminate the previous or count the previous.

# [test\_symbol\_1]

table[i][j] = table[i - 2][j] or table[i - 1][j]

# Propagations (referring to the horizontal element)

# If p's previous one is equal to the current s, with

# helps of \*, the status can be propagated from the left.

# [test\_symbol\_2]

if p[i - 2] == s[j - 1] or p[i - 2] == '.':

table[i][j] |= table[i][j - 1]

return table[-1][-1]

class TestSolution(unittest.TestCase):

def test\_none\_0(self):

s = ""

p = ""

self.assertTrue(Solution().isMatch(s, p))

def test\_none\_1(self):

s = ""

p = "a"

self.assertFalse(Solution().isMatch(s, p))

def test\_no\_symbol\_equal(self):

s = "abcd"

p = "abcd"

self.assertTrue(Solution().isMatch(s, p))

def test\_no\_symbol\_not\_equal\_0(self):

s = "abcd"

p = "efgh"

self.assertFalse(Solution().isMatch(s, p))

def test\_no\_symbol\_not\_equal\_1(self):

s = "ab"

p = "abb"

self.assertFalse(Solution().isMatch(s, p))

def test\_symbol\_0(self):

s = ""

p = "a\*"

self.assertTrue(Solution().isMatch(s, p))

def test\_symbol\_1(self):

s = "a"

p = "ab\*"

self.assertTrue(Solution().isMatch(s, p))

def test\_symbol\_2(self):

# E.g.

# s a b b

# p 1 0 0 0

# a 0 1 0 0

# b 0 0 1 0

# \* 0 1 1 1

s = "abb"

p = "ab\*"

self.assertTrue(Solution().isMatch(s, p))

if \_\_name\_\_ == "\_\_main\_\_":

unittest.main()

# 11,Container With Most Water\*\*\*:

**Best\_solution:**

Yet another way to see what happens in the O(n) algorithm

The O(n) solution with proof by contradiction doesn't look intuitive enough to me. Before moving on, read [the algorithm][1] first if you don't know it yet.

Here's another way to see what happens in a matrix representation:

Draw a matrix where the row is the first line, and the column is the second line. For example, say `n=6`.

In the figures below, `x` means we don't need to compute the volume for that case: (1) On the diagonal, the two lines are overlapped; (2) The lower left triangle area of the matrix is symmetric to the upper right area.

We start by computing the volume at `(1,6)`, denoted by `o`. Now if the left line is shorter than the right line, then all the elements left to `(1,6)` on the first row have smaller volume, so we don't need to compute those cases (crossed by `---`).

1 2 3 4 5 6

1 x ------- o

2 x x

3 x x x

4 x x x x

5 x x x x x

6 x x x x x x

Next we move the left line and compute `(2,6)`. Now if the right line is shorter, all cases below `(2,6)` are eliminated.

1 2 3 4 5 6

1 x ------- o

2 x x o

3 x x x |

4 x x x x |

5 x x x x x |

6 x x x x x x

And no matter how this `o` path goes, we end up only need to find the max value on this path, which contains `n-1` cases.

1 2 3 4 5 6

1 x ------- o

2 x x - o o o

3 x x x o | |

4 x x x x | |

5 x x x x x |

6 x x x x x x

Hope this helps. I feel more comfortable seeing things this way.

[1]: https://oj.leetcode.com/discuss/1074/anyone-who-has-a-o-n-algorithm

# 12,Integer to Roman\*\*\*:

**Python\_solution:**

Share My Python Solution 96ms

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

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

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

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

return M[num/1000] + C[(num%1000)/100] + X[(num%100)/10] + I[num%10];

# 13,Roman to Integer\*\*\*:

**Python\_solution:**

My Straightforward Python Solution

class Solution:

# @param {string} s

# @return {integer}

def romanToInt(self, s):

roman = {'M': 1000,'D': 500 ,'C': 100,'L': 50,'X': 10,'V': 5,'I': 1}

z = 0

for i in range(0, len(s) - 1):

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

z -= roman[s[i]]

else:

z += roman[s[i]]

return z + roman[s[-1]]

\*Note: The trick is that the last letter is always added. Except the last one, if one letter is less than its latter one, this letter is subtracted.

# 14,Longest Common Prefix\*\*\*:

**Python\_solution:**

Simple Python solution

Might be a bit slow, but here's my relatively elegant Python solution:

class Solution:

# @return a string

def longestCommonPrefix(self, strs):

if not strs:

return ""

for i, letter\_group in enumerate(zip(\*strs)):

if len(set(letter\_group)) > 1:

return strs[0][:i]

else:

return min(strs)

# 15,3Sum\*\*\*:

**Python\_solution:**

Python easy to understand solution (O(n\*n) time).

def threeSum(self, nums):

res = []

nums.sort()

for i in xrange(len(nums)-2):

if i > 0 and nums[i] == nums[i-1]:

continue

l, r = i+1, len(nums)-1

while l < r:

s = nums[i] + nums[l] + nums[r]

if s < 0:

l +=1

elif s > 0:

r -= 1

else:

res.append((nums[i], nums[l], nums[r]))

while l < r and nums[l] == nums[l+1]:

l += 1

while l < r and nums[r] == nums[r-1]:

r -= 1

l += 1; r -= 1

return res

# 16,3Sum Closest\*\*\*:

**Python\_solution:**

Python O(N^2) solution

class Solution:

# @return an integer

def threeSumClosest(self, num, target):

num.sort()

result = num[0] + num[1] + num[2]

for i in range(len(num) - 2):

j, k = i+1, len(num) - 1

while j < k:

sum = num[i] + num[j] + num[k]

if sum == target:

return sum

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

result = sum

if sum < target:

j += 1

elif sum > target:

k -= 1

return result

# 17,Letter Combinations of a Phone Number\*\*\*:

**Python\_solution:**

One line python solution

class Solution:

# @return a list of strings, [s1, s2]

def letterCombinations(self, digits):

if '' == digits: return []

kvmaps = {

'2': 'abc',

'3': 'def',

'4': 'ghi',

'5': 'jkl',

'6': 'mno',

'7': 'pqrs',

'8': 'tuv',

'9': 'wxyz'

}

return reduce(lambda acc, digit: [x + y for x in acc for y in kvmaps[digit]], digits, [''])

# 18,4Sum\*\*\*:

**Python\_solution:**

Python 140ms beats 100%, and works for N-sum (N>=2)

The core is to implement a fast 2-pointer to solve 2-sum, and recursion to reduce the N-sum to 2-sum. Some optimization was be made knowing the list is sorted.

def fourSum(self, nums, target):

nums.sort()

results = []

self.findNsum(nums, target, 4, [], results)

return results

def findNsum(self, nums, target, N, result, results):

if len(nums) < N or N < 2: return

# solve 2-sum

if N == 2:

l,r = 0,len(nums)-1

while l < r:

if nums[l] + nums[r] == target:

results.append(result + [nums[l], nums[r]])

l += 1

r -= 1

while l < r and nums[l] == nums[l - 1]:

l += 1

while r > l and nums[r] == nums[r + 1]:

r -= 1

elif nums[l] + nums[r] < target:

l += 1

else:

r -= 1

else:

for i in range(0, len(nums)-N+1): # careful about range

if target < nums[i]\*N or target > nums[-1]\*N: # take advantages of sorted list

break

if i == 0 or i > 0 and nums[i-1] != nums[i]: # recursively reduce N

self.findNsum(nums[i+1:], target-nums[i], N-1, result+[nums[i]], results)

return

Just revisited and clean the code

def fourSum(self, nums, target):

def findNsum(nums, target, N, result, results):

if len(nums) < N or N < 2 or target < nums[0]\*N or target > nums[-1]\*N: # early termination

return

if N == 2: # two pointers solve sorted 2-sum problem

l,r = 0,len(nums)-1

while l < r:

s = nums[l] + nums[r]

if s == target:

results.append(result + [nums[l], nums[r]])

l += 1

while l < r and nums[l] == nums[l-1]:

l += 1

elif s < target:

l += 1

else:

r -= 1

else: # recursively reduce N

for i in range(len(nums)-N+1):

if i == 0 or (i > 0 and nums[i-1] != nums[i]):

findNsum(nums[i+1:], target-nums[i], N-1, result+[nums[i]], results)

results = []

findNsum(sorted(nums), target, 4, [], results)

return results

# 19,Remove Nth Node From End of List\*\*\*:

**Python\_solution:**

3 short Python solutions

\*\*Value-Shifting - AC in 64 ms\*\*

My first solution is "cheating" a little. Instead of really removing the nth \*node\*, I remove the nth \*value\*. I recursively determine the indexes (counting from back), then shift the values for all indexes larger than n, and then always drop the head.

class Solution:

def removeNthFromEnd(self, head, n):

def index(node):

if not node:

return 0

i = index(node.next) + 1

if i > n:

node.next.val = node.val

return i

index(head)

return head.next

---

\*\*Index and Remove - AC in 56 ms\*\*

In this solution I recursively determine the indexes again, but this time my helper function removes the nth node. It returns two values. The index, as in my first solution, and the possibly changed head of the remaining list.

class Solution:

def removeNthFromEnd(self, head, n):

def remove(head):

if not head:

return 0, head

i, head.next = remove(head.next)

return i+1, (head, head.next)[i+1 == n]

return remove(head)[1]

---

\*\*n ahead - AC in 48 ms\*\*

The standard solution, but without a dummy extra node. Instead, I simply handle the special case of removing the head right after the fast cursor got its head start.

class Solution:

def removeNthFromEnd(self, head, n):

fast = slow = head

for \_ in range(n):

fast = fast.next

if not fast:

return head.next

while fast.next:

fast = fast.next

slow = slow.next

slow.next = slow.next.next

return head

# 20,Valid Parentheses\*\*\*:

**Python\_solution:**

Simple Python solution with stack

class Solution:

# @return a boolean

def isValid(self, s):

stack = []

dict = {"]":"[", "}":"{", ")":"("}

for char in s:

if char in dict.values():

stack.append(char)

elif char in dict.keys():

if stack == [] or dict[char] != stack.pop():

return False

else:

return False

return stack == []

It's quite obvious.

# 21,Merge Two Sorted Lists\*\*\*:

**Python\_solution:**

Python solutions (iteratively, recursively, iteratively in-place).

# iteratively

def mergeTwoLists1(self, l1, l2):

dummy = cur = ListNode(0)

while l1 and l2:

if l1.val < l2.val:

cur.next = l1

l1 = l1.next

else:

cur.next = l2

l2 = l2.next

cur = cur.next

cur.next = l1 or l2

return dummy.next

# recursively

def mergeTwoLists2(self, l1, l2):

if not l1 or not l2:

return l1 or l2

if l1.val < l2.val:

l1.next = self.mergeTwoLists(l1.next, l2)

return l1

else:

l2.next = self.mergeTwoLists(l1, l2.next)

return l2

# in-place, iteratively

def mergeTwoLists(self, l1, l2):

if None in (l1, l2):

return l1 or l2

dummy = cur = ListNode(0)

dummy.next = l1

while l1 and l2:

if l1.val < l2.val:

l1 = l1.next

else:

nxt = cur.next

cur.next = l2

tmp = l2.next

l2.next = nxt

l2 = tmp

cur = cur.next

cur.next = l1 or l2

return dummy.next

# 22,Generate Parentheses\*\*\*:

**Python\_solution:**

4-7 lines Python

`p` is the parenthesis-string built so far, `left` and `right` tell the number of left and right parentheses still to add, and `parens` collects the parentheses.

\*\*Solution 1\*\*

I used a few "tricks"... how many can you find? :-)

def generateParenthesis(self, n):

def generate(p, left, right, parens=[]):

if left: generate(p + '(', left-1, right)

if right > left: generate(p + ')', left, right-1)

if not right: parens += p,

return parens

return generate('', n, n)

\*\*Solution 2\*\*

Here I wrote an actual Python generator. I allow myself to put the `yield q` at the end of the line because it's not that bad and because in "real life" I use Python 3 where I just say `yield from generate(...)`.

def generateParenthesis(self, n):

def generate(p, left, right):

if right >= left >= 0:

if not right:

yield p

for q in generate(p + '(', left-1, right): yield q

for q in generate(p + ')', left, right-1): yield q

return list(generate('', n, n))

\*\*Solution 3\*\*

Improved version of [this](https://leetcode.com/discuss/25725/7-lines-in-python-44-ms). Parameter `open` tells the number of "already opened" parentheses, and I continue the recursion as long as I still have to open parentheses (`n > 0`) and I haven't made a mistake yet (`open >= 0`).

def generateParenthesis(self, n, open=0):

if n > 0 <= open:

return ['(' + p for p in self.generateParenthesis(n-1, open+1)] + \

[')' + p for p in self.generateParenthesis(n, open-1)]

return [')' \* open] \* (not n)

# 23,Merge k Sorted Lists\*\*\*:

**Python\_solution:**

10-line python solution with priority queue

from Queue import PriorityQueue

class Solution(object):

def mergeKLists(self, lists):

dummy = ListNode(None)

curr = dummy

q = PriorityQueue()

for node in lists:

if node: q.put((node.val,node))

while q.qsize()>0:

curr.next = q.get()[1]

curr=curr.next

if curr.next: q.put((curr.next.val, curr.next))

return dummy.next

# 24,Swap Nodes in Pairs\*\*\*:

**Python\_solution:**

7-8 lines C++ / Python / Ruby

Three different implementations of the same algorithm, taking advantage of different strengths of the three languages. I suggest reading all three, even if you don't know all three languages.

All three of course work swap the current node with the next node by rearranging pointers, then move on to the next pair, and repeat until the end of the list.

---

\*\*C++\*\*

Pointer-pointer `pp` points to the pointer to the current node. So at first, `pp` points to `head`, and later it points to the `next` field of ListNodes. Additionally, for convenience and clarity, pointers `a` and `b` point to the current node and the next node.

We need to go from `\*pp == a -> b -> (b->next)` to `\*pp == b -> a -> (b->next)`. The first three lines inside the loop do that, setting those three pointers (from right to left). The fourth line moves `pp` to the next pair.

ListNode\* swapPairs(ListNode\* head) {

ListNode \*\*pp = &head, \*a, \*b;

while ((a = \*pp) && (b = a->next)) {

a->next = b->next;

b->next = a;

\*pp = b;

pp = &(a->next);

}

return head;

}

---

\*\*Python\*\*

Here, `pre` is the previous node. Since the head doesn't have a previous node, I just use `self` instead. Again, `a` is the current node and `b` is the next node.

To go from `pre -> a -> b -> b.next` to `pre -> b -> a -> b.next`, we need to change those three references. Instead of thinking about in what order I change them, I just change all three at once.

def swapPairs(self, head):

pre, pre.next = self, head

while pre.next and pre.next.next:

a = pre.next

b = a.next

pre.next, b.next, a.next = b, a, b.next

pre = a

return self.next

---

\*\*Ruby\*\*

Again, `pre` is the previous node, but here I create a dummy as previous node of the head. And again, `a` is the current node and `b` is the next node. This time I go one node further and call it `c`.

To go from `pre -> a -> b -> c` to `pre -> b -> a -> c`, we need to change those three references. Here I chain the assignments, pretty much directly saying "`pre` points to `b`, which points to `a`, which points to `c`".

def swap\_pairs(head)

pre = dummy = ListNode.new 0

pre.next = head

while a = pre.next and b = a.next

c = b.next

((pre.next = b).next = a).next = c

pre = a

end

dummy.next

end

# 25,Reverse Nodes in k-Group\*\*\*:

**Python\_solution:**

Succinct iterative Python, O(n) time O(1) space

Use a dummy head, and

l, r : define reversing range

pre, cur : used in reversing, standard reverse linked linked list method

jump : used to connect last node in previous k-group to first node in following k-group

def reverseKGroup(self, head, k):

dummy = jump = ListNode(0)

dummy.next = l = r = head

while True:

count = 0

while r and count < k: # use r to locate the range

r = r.next

count += 1

if count == k: # if size k satisfied, reverse the inner linked list

pre, cur = r, l

for \_ in range(k):

cur.next, cur, pre = pre, cur.next, cur # standard reversing

jump.next, jump, l = pre, l, r # connect two k-groups

else:

return dummy.next

# 26,Remove Duplicates from Sorted Array\*\*\*:

**Python\_solution:**

Simple Python solution - O(n)

class Solution:

# @param a list of integers

# @return an integer

def removeDuplicates(self, A):

if not A:

return 0

newTail = 0

for i in range(1, len(A)):

if A[i] != A[newTail]:

newTail += 1

A[newTail] = A[i]

return newTail + 1

# 27,Remove Element\*\*\*:

**Python\_solution:**

Simple Python O(n) two pointer in place solution

Starting from the left every time we find a value that is the target value we swap it out with an item starting from the right. We decrement end each time as we know that the final item is the target value and only increment start once we know the value is ok. Once start reaches end we know all items after that point are the target value so we can stop there.

def removeElement(self, nums, val):

start, end = 0, len(nums) - 1

while start <= end:

if nums[start] == val:

nums[start], nums[end], end = nums[end], nums[start], end - 1

else:

start +=1

return start

# 28,Implement strStr()\*\*\*:

**Python\_solution:**

My answer by Python

class Solution(object):

def strStr(self, haystack, needle):

"""

:type haystack: str

:type needle: str

:rtype: int

"""

for i in range(len(haystack) - len(needle)+1):

if haystack[i:i+len(needle)] == needle:

return i

return -1

# 29,Divide Two Integers\*\*\*:

**Python\_solution:**

Clear python code

class Solution:

# @return an integer

def divide(self, dividend, divisor):

positive = (dividend < 0) is (divisor < 0)

dividend, divisor = abs(dividend), abs(divisor)

res = 0

while dividend >= divisor:

temp, i = divisor, 1

while dividend >= temp:

dividend -= temp

res += i

i <<= 1

temp <<= 1

if not positive:

res = -res

return min(max(-2147483648, res), 2147483647)

# 30,Substring with Concatenation of All Words\*\*\*:

**Python\_solution:**

AC Python 80ms solution, dictionary and two pointers

def \_findSubstring(self, l, r, n, k, t, s, req, ans):

curr = {}

while r + k <= n:

w = s[r:r + k]

r += k

if w not in req:

l = r

curr.clear()

else:

curr[w] = curr[w] + 1 if w in curr else 1

while curr[w] > req[w]:

curr[s[l:l + k]] -= 1

l += k

if r - l == t:

ans.append(l)

def findSubstring(self, s, words):

if not s or not words or not words[0]:

return []

n = len(s)

k = len(words[0])

t = len(words) \* k

req = {}

for w in words:

req[w] = req[w] + 1 if w in req else 1

ans = []

for i in xrange(min(k, n - t + 1)):

self.\_findSubstring(i, i, n, k, t, s, req, ans)

return ans

# 169 / 169 test cases passed.

# Status: Accepted

# Runtime: 80 ms

# 98.60%

First of all consider s as several series of words with length k starting at [0, k-1]. For example "barfoothe" with k = 3, can be view as ["bar", "foo", "the"] for i=0 and ["arf", "oot"] for i = 1 and ["rfo", "oth"] for i = 2.

Thus we need to check each of these series and find out the valid index by definition.

For each series, we just need to check if there exist a range [l, r) where the occurrence or "spectrum" of the words in the range is the same as our given word list's "spectrum". We use dictionary to store the spectrum and maintain it as we loop through s.

collections.Counter class may save a bit of code on updating the counts of the dictionary. However plain dict wins on the speed.

# 31,Next Permutation\*\*\*:

**Python\_solution:**

Two-pointer solution in python with detail expalanation

Credit goes to http://blog.csdn.net/m6830098/article/details/17291259

class Solution(object):

def nextPermutation(self, nums):

"""

:type nums: List[int]

:rtype: void Do not return anything, modify nums in-place instead.

"""

# Use two-pointers: two pointers start from back

# first pointer j stop at descending point

# second pointer i stop at value > nums[j]

# swap and sort rest

if not nums: return None

i = len(nums)-1

j = -1 # j is set to -1 for case `4321`, so need to reverse all in following step

while i > 0:

if nums[i-1] < nums[i]: # first one violates the trend

j = i-1

break

i-=1

for i in xrange(len(nums)-1, -1, -1):

if nums[i] > nums[j]: #

nums[i], nums[j] = nums[j], nums[i] # swap position

nums[j+1:] = sorted(nums[j+1:]) # sort rest

return

# 32,Longest Valid Parentheses\*\*\*:

**Python\_solution:**

Pure 1D-DP without using stack (python) with detailed explanation

class Solution(object):

def longestValidParentheses(self, s):

"""

:type s: str

:rtype: int

"""

# use 1D DP

# dp[i] records the longestValidParenthese EXACTLY ENDING at s[i]

dp = [0 for x in xrange(len(s))]

max\_to\_now = 0

for i in xrange(1,len(s)):

if s[i] == ')':

# case 1: ()()

if s[i-1] == '(':

# add nearest parentheses pairs + 2

dp[i] = dp[i-2] + 2

# case 2: (())

# i-dp[i-1]-1 is the index of last "(" not paired until this ")"

elif i-dp[i-1]-1 >= 0 and s[i-dp[i-1]-1] == '(':

if dp[i-1] > 0: # content within current matching pair is valid

# add nearest parentheses pairs + 2 + parentheses before last "("

dp[i] = dp[i-1] + 2 + dp[i-dp[i-1]-2]

else:

# otherwise is 0

dp[i] = 0

max\_to\_now = max(max\_to\_now, dp[i])

return max\_to\_now

# 33,Search in Rotated Sorted Array\*\*\*:

**Python\_solution:**

Pretty short C++/Java/Ruby/Python

Explanation below the codes.

\*\*Ruby:\*\*

def search(nums, target)

i = (0...nums.size).bsearch { |i|

(nums[0] <= target) ^ (nums[0] > nums[i]) ^ (target > nums[i])

}

nums[i || 0] == target ? i : -1

end

\*\*Ruby Golf\*\*, just once for fun:

def search(n, t)

i=(0...n.size).bsearch{|i|(n[0]<=t)^(n[0]>n[i])^(t>n[i])};n[i||0]==t ?i:-1

end

\*\*Python:\*\*

def search(self, nums, target):

lo, hi = 0, len(nums) - 1

while lo < hi:

mid = (lo + hi) / 2

if (nums[0] > target) ^ (nums[0] > nums[mid]) ^ (target > nums[mid]):

lo = mid + 1

else:

hi = mid

return lo if target in nums[lo:lo+1] else -1

\*\*Python using `bisect`:\*\*

class Solution:

def search(self, nums, target):

self.\_\_getitem\_\_ = lambda i: \

(nums[0] <= target) ^ (nums[0] > nums[i]) ^ (target > nums[i])

i = bisect.bisect\_left(self, True, 0, len(nums))

return i if target in nums[i:i+1] else -1

\*\*C++:\*\*

int search(vector<int>& nums, int target) {

int lo = 0, hi = int(nums.size()) - 1;

while (lo < hi) {

int mid = (lo + hi) / 2;

if ((nums[0] > target) ^ (nums[0] > nums[mid]) ^ (target > nums[mid]))

lo = mid + 1;

else

hi = mid;

}

return lo == hi && nums[lo] == target ? lo : -1;

}

\*\*Java:\*\*

public int search(int[] nums, int target) {

int lo = 0, hi = nums.length - 1;

while (lo < hi) {

int mid = (lo + hi) / 2;

if ((nums[0] > target) ^ (nums[0] > nums[mid]) ^ (target > nums[mid]))

lo = mid + 1;

else

hi = mid;

}

return lo == hi && nums[lo] == target ? lo : -1;

}

---

Explanation

-

My solutions use binary search guided by the following thoughts:

Remember the array is sorted, except it might drop at one point.

- \*\*If nums[0] <= nums[i]\*\*, then nums[0..i] is sorted (in case of "\*\*==\*\*" it's just one element, and in case of "\*\*<\*\*" there must be a drop elsewhere). So we should keep searching in nums[0..i] if the target lies in this sorted range, i.e., if `nums[0] <= target <= nums[i]`.

- \*\*If nums[i] < nums[0]\*\*, then nums[0..i] contains a drop, and thus nums[i+1..end] is sorted and lies strictly between nums[i] and nums[0]. So we should keep searching in nums[0..i] if the target \*doesn't\* lie strictly between them, i.e., if `target <= nums[i] < nums[0]` or `nums[i] < nums[0] <= target`

Those three cases look cyclic:

nums[0] <= target <= nums[i]

target <= nums[i] < nums[0]

nums[i] < nums[0] <= target

So I have the three checks `(nums[0] <= target)`, `(target <= nums[i])` and `(nums[i] < nums[0])`, and I want to know whether exactly two of them are true. They can't all be true or all be false (check it), so I just need to distinguish between "two true" and "one true". Parity is enough for that, so instead of adding them I xor them, which is a bit shorter and particularly helpful in Java and Ruby, because those don't let me add booleans but do let me xor them.

(Actually while developing this I thought of permutations of nums[0], target and nums[i] and the permutation parity and saw those three checks as representing inversions, but I had trouble putting that into words and now find the above explanation much better. But it helped me get there, so I wanted to mention it here.)

# 34,Search for a Range\*\*\*:

**Python\_solution:**

Search for the position target-0.5 and target+0.5, a simple python code with a little trick

class Solution:

# @param A, a list of integers

# @param target, an integer to be searched

# @return a list of length 2, [index1, index2]

def searchRange(self, arr, target):

start = self.binary\_search(arr, target-0.5)

if arr[start] != target:

return [-1, -1]

arr.append(0)

end = self.binary\_search(arr, target+0.5)-1

return [start, end]

def binary\_search(self, arr, target):

start, end = 0, len(arr)-1

while start < end:

mid = (start+end)//2

if target < arr[mid]:

end = mid

else:

start = mid+1

return start

for search the target+0.5 position we add something whatever to the list end

to get the right position for the edge case

take ([0,1,2,3,4,5], 5) for example:

we append 0 to the list end

[0,1,2,3,4,5,0]

[4,5,0]# start now is 4, end is 6, mid is 5, start = mid+1 = 6, end the while loop

finally we get the 5.5 position == start == 6

# 35,Search Insert Position\*\*\*:

**Python\_solution:**

Python beats 98%

class Solution(object):

def searchInsert(self, nums, key):

if key > nums[len(nums) - 1]:

return len(nums)

if key < nums[0]:

return 0

l, r = 0, len(nums) - 1

while l <= r:

m = (l + r)/2

if nums[m] > key:

r = m - 1

if r >= 0:

if nums[r] < key:

return r + 1

else:

return 0

elif nums[m] < key:

l = m + 1

if l < len(nums):

if nums[l] > key:

return l

else:

return len(nums)

else:

return m

Once the left border is larger than key, than return index. Once the right border is less than key, then return index.

# 36,Valid Sudoku\*\*\*:

**Python\_solution:**

1-7 lines Python, 4 solutions

\*\*Idea\*\*

Just go through all you see (like "7 in row 3") and check for duplicates.

\*\*Solution 1\*\*

Using `Counter`. One logical line, seven physical lines.

def isValidSudoku(self, board):

return 1 == max(collections.Counter(

x

for i, row in enumerate(board)

for j, c in enumerate(row)

if c != '.'

for x in ((c, i), (j, c), (i/3, j/3, c))

).values() + [1])

The ` + [1]` is only for the empty board, where `max` would get an empty list and complain. It's not necessary to get it accepted here, as the empty board isn't among the test cases, but it's good to have.

\*\*Solution 2\*\*

Using `len(set)`.

def isValidSudoku(self, board):

seen = sum(([(c, i), (j, c), (i/3, j/3, c)]

for i, row in enumerate(board)

for j, c in enumerate(row)

if c != '.'), [])

return len(seen) == len(set(seen))

\*\*Solution 3\*\*

Using `any`.

def isValidSudoku(self, board):

seen = set()

return not any(x in seen or seen.add(x)

for i, row in enumerate(board)

for j, c in enumerate(row)

if c != '.'

for x in ((c, i), (j, c), (i/3, j/3, c)))

\*\*Solution 4\*\*

Iterating a different way.

def isValidSudoku(self, board):

seen = sum(([(c, i), (j, c), (i/3, j/3, c)]

for i in range(9) for j in range(9)

for c in [board[i][j]] if c != '.'), [])

return len(seen) == len(set(seen))

# 37,Sudoku Solver\*\*\*:

**Python\_solution:**

48ms straitforward python DFS solution with explanations

It's similar to how human solve Sudoku. <br>

1. create a hash table (dictionary) `val` to store possible values in every location.<br>

2. Each time, start from the location with fewest possible values, choose one value from it and then update the board and possible values at other locations. If this update is valid, keep solving (DFS). If this update is invalid (leaving zero possible values at some locations) or this value doesn't lead to the solution, undo the updates and then choose the next value. <br>

Since we calculated `val` at the beginning and start filling the board from the location with fewest possible values, the amount of calculation and thus the runtime can be significantly reduced: <br><br>

The run time is 48-68 ms on LeetCode OJ, which seems to be among the fastest python solutions here.<br><br>

The `PossibleVals` function may be further simplified/optimized, but it works just fine for now. (it would look less lengthy if we are allowed to use numpy array for the board lol). <br>

<br>

def solveSudoku(self, board):

self.board = board

self.val = self.PossibleVals()

self.Solver()

def PossibleVals(self):

a = "123456789"

d, val = {}, {}

for i in xrange(9):

for j in xrange(9):

ele = self.board[i][j]

if ele != ".":

d[("r", i)] = d.get(("r", i), []) + [ele]

d[("c", j)] = d.get(("c", j), []) + [ele]

d[(i//3, j//3)] = d.get((i//3, j//3), []) + [ele]

else:

val[(i,j)] = []

for (i,j) in val.keys():

inval = d.get(("r",i),[])+d.get(("c",j),[])+d.get((i/3,j/3),[])

val[(i,j)] = [n for n in a if n not in inval ]

return val

def Solver(self):

if len(self.val)==0:

return True

kee = min(self.val.keys(), key=lambda x: len(self.val[x]))

nums = self.val[kee]

for n in nums:

update = {kee:self.val[kee]}

if self.ValidOne(n, kee, update): # valid choice

if self.Solver(): # keep solving

return True

self.undo(kee, update) # invalid choice or didn't solve it => undo

return False

def ValidOne(self, n, kee, update):

self.board[kee[0]][kee[1]] = n

del self.val[kee]

i, j = kee

for ind in self.val.keys():

if n in self.val[ind]:

if ind[0]==i or ind[1]==j or (ind[0]/3,ind[1]/3)==(i/3,j/3):

update[ind] = n

self.val[ind].remove(n)

if len(self.val[ind])==0:

return False

return True

def undo(self, kee, update):

self.board[kee[0]][kee[1]]="."

for k in update:

if k not in self.val:

self.val[k]= update[k]

else:

self.val[k].append(update[k])

return None

# 38,Count and Say\*\*\*:

**Python\_solution:**

4-5 lines Python solutions

\*\*Solution 1\*\* ... using a regular expression

def countAndSay(self, n):

s = '1'

for \_ in range(n - 1):

s = re.sub(r'(.)\1\*', lambda m: str(len(m.group(0))) + m.group(1), s)

return s

---

\*\*Solution 2\*\* ... using a regular expression

def countAndSay(self, n):

s = '1'

for \_ in range(n - 1):

s = ''.join(str(len(group)) + digit

for group, digit in re.findall(r'((.)\2\*)', s))

return s

---

\*\*Solution 3\*\* ... using `groupby`

def countAndSay(self, n):

s = '1'

for \_ in range(n - 1):

s = ''.join(str(len(list(group))) + digit

for digit, group in itertools.groupby(s))

return s

# 39,Combination Sum\*\*\*:

**Python\_solution:**

Python dfs solution.

def combinationSum(self, candidates, target):

res = []

candidates.sort()

self.dfs(candidates, target, 0, [], res)

return res

def dfs(self, nums, target, index, path, res):

if target < 0:

return # backtracking

if target == 0:

res.append(path)

return

for i in xrange(index, len(nums)):

self.dfs(nums, target-nums[i], i, path+[nums[i]], res)

# 40,Combination Sum II\*\*\*:

**Python\_solution:**

DP solution in Python

I also did it with recursion, turns out the DP solution is 3~4 times faster.

def combinationSum2(self, candidates, target):

candidates.sort()

table = [None] + [set() for i in range(target)]

for i in candidates:

if i > target:

break

for j in range(target - i, 0, -1):

table[i + j] |= {elt + (i,) for elt in table[j]}

table[i].add((i,))

return map(list, table[target])

# 41,First Missing Positive\*\*\*:

**Python\_solution:**

Python O(1) space, O(n) time solution with explanation

def firstMissingPositive(self, nums):

"""

:type nums: List[int]

:rtype: int

Basic idea:

1. for any array whose length is l, the first missing positive must be in range [1,...,l+1],

so we only have to care about those elements in this range and remove the rest.

2. we can use the array index as the hash to restore the frequency of each number within

the range [1,...,l+1]

"""

nums.append(0)

n = len(nums)

for i in range(len(nums)): #delete those useless elements

if nums[i]<0 or nums[i]>=n:

nums[i]=0

for i in range(len(nums)): #use the index as the hash to record the frequency of each number

nums[nums[i]%n]+=n

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

if nums[i]/n==0:

return i

return n

# 42,Trapping Rain Water\*\*\*:

**Python\_solution:**

Share my one pass Python solution with explaination

For index i, the water volume of i: `vol\_i = min(left\_max\_i, right\_max\_i) - bar\_i`.

The left\_max array from left to right is always non-descending, the right\_max is non-ascending.

Having such observation, we can say:

Given i < j, if left\_max\_i <= right\_max\_j: `vol\_i = left\_max\_i - bar\_i`, otherwise, `vol\_j = right\_max\_j - bar\_j`

because, if left\_max\_i <= right\_max\_j: `left\_max\_i <= right\_max\_j <= right\_max\_j-1 <= ... <= right\_max\_i`, then `min(left\_max\_i, right\_max\_i)` is always `left\_max\_i`

Code is pasted.

def trap(self, bars):

if not bars or len(bars) < 3:

return 0

volume = 0

left, right = 0, len(bars) - 1

l\_max, r\_max = bars[left], bars[right]

while left < right:

l\_max, r\_max = max(bars[left], l\_max), max(bars[right], r\_max)

if l\_max <= r\_max:

volume += l\_max - bars[left]

left += 1

else:

volume += r\_max - bars[right]

right -= 1

return volume

# 43,Multiply Strings\*\*\*:

**Python\_solution:**

Simple Python solution, 18 lines

def multiply(num1, num2):

product = [0] \* (len(num1) + len(num2))

pos = len(product)-1

for n1 in reversed(num1):

tempPos = pos

for n2 in reversed(num2):

product[tempPos] += int(n1) \* int(n2)

product[tempPos-1] += product[tempPos]/10

product[tempPos] %= 10

tempPos -= 1

pos -= 1

pt = 0

while pt < len(product)-1 and product[pt] == 0:

pt += 1

return ''.join(map(str, product[pt:]))

# 44,Wildcard Matching\*\*\*:

**Python\_solution:**

Python DP solution

class Solution:

# @return a boolean

def isMatch(self, s, p):

length = len(s)

if len(p) - p.count('\*') > length:

return False

dp = [True] + [False]\*length

for i in p:

if i != '\*':

for n in reversed(range(length)):

dp[n+1] = dp[n] and (i == s[n] or i == '?')

else:

for n in range(1, length+1):

dp[n] = dp[n-1] or dp[n]

dp[0] = dp[0] and i == '\*'

return dp[-1]

dp[n] means the substring s[:n] if match the pattern i

dp[0] means the empty string '' or s[:0] which only match the pattern '\*'

use the reversed builtin because for every dp[n+1] we use the previous 'dp'

add Java O(m\*n) version code

public boolean isMatch(String s, String p) {

int count = 0;

for (char c : p.toCharArray()) {

if (c == '\*')

count++;

}

if (p.length() - count > s.length())

return false;

boolean[][] dp = new boolean[p.length() + 1][s.length() + 1];

dp[0][0] = true;

for (int j = 1; j <= p.length(); j++) {

char pattern = p.charAt(j - 1);

dp[j][0] = dp[j - 1][0] && pattern == '\*';

for (int i = 1; i <= s.length(); i++) {

char letter = s.charAt(i - 1);

if (pattern != '\*') {

dp[j][i] = dp[j - 1][i - 1] && (pattern == '?' || pattern == letter);

} else

dp[j][i] = dp[j][i - 1] || dp[j - 1][i];

}

}

return dp[p.length()][s.length()];

}

# 45,Jump Game II\*\*\*:

**Python\_solution:**

10-lines C++ (16ms) / Python BFS Solutions with Explanations

This problem has a nice BFS structure. Let's illustrate it using the example `nums = [2, 3, 1, 1, 4]` in the problem statement. We are initially at position `0`. Then we can move at most `nums[0]` steps from it. So, after one move, we may reach `nums[1] = 3` or `nums[2] = 1`. So these nodes are reachable in `1` move. From these nodes, we can further move to `nums[3] = 1` and `nums[4] = 4`. Now you can see that the target `nums[4] = 4` is reachable in `2` moves.

Putting these into codes, we keep two pointers `start` and `end` that record the current range of the starting nodes. Each time after we make a move, update `start` to be `end + 1` and `end` to be the farthest index that can be reached in `1` move from the current `[start, end]`.

To get an accepted solution, it is important to handle all the edge cases. And the following codes handle all of them in a unified way without using the unclean `if` statements :-)

----------

\*\*C++\*\*

class Solution {

public:

int jump(vector<int>& nums) {

int n = nums.size(), step = 0, start = 0, end = 0;

while (end < n - 1) {

step++;

int maxend = end + 1;

for (int i = start; i <= end; i++) {

if (i + nums[i] >= n - 1) return step;

maxend = max(maxend, i + nums[i]);

}

start = end + 1;

end = maxend;

}

return step;

}

};

----------

\*\*Python\*\*

class Solution:

# @param {integer[]} nums

# @return {integer}

def jump(self, nums):

n, start, end, step = len(nums), 0, 0, 0

while end < n - 1:

step += 1

maxend = end + 1

for i in range(start, end + 1):

if i + nums[i] >= n - 1:

return step

maxend = max(maxend, i + nums[i])

start, end = end + 1, maxend

return step

# 46,Permutations\*\*\*:

**Python\_solution:**

My AC simple iterative java/python solution

the basic idea is, to permute n numbers, we can add the nth number into the resulting `List<List<Integer>>` from the n-1 numbers, in every possible position.

For example, if the input num[] is {1,2,3}: First, add 1 into the initial `List<List<Integer>>` (let's call it "answer").

Then, 2 can be added in front or after 1. So we have to copy the List<Integer> in answer (it's just {1}), add 2 in position 0 of {1}, then copy the original {1} again, and add 2 in position 1. Now we have an answer of {{2,1},{1,2}}. There are 2 lists in the current answer.

Then we have to add 3. first copy {2,1} and {1,2}, add 3 in position 0; then copy {2,1} and {1,2}, and add 3 into position 1, then do the same thing for position 3. Finally we have 2\*3=6 lists in answer, which is what we want.

public List<List<Integer>> permute(int[] num) {

List<List<Integer>> ans = new ArrayList<List<Integer>>();

if (num.length ==0) return ans;

List<Integer> l0 = new ArrayList<Integer>();

l0.add(num[0]);

ans.add(l0);

for (int i = 1; i< num.length; ++i){

List<List<Integer>> new\_ans = new ArrayList<List<Integer>>();

for (int j = 0; j<=i; ++j){

for (List<Integer> l : ans){

List<Integer> new\_l = new ArrayList<Integer>(l);

new\_l.add(j,num[i]);

new\_ans.add(new\_l);

}

}

ans = new\_ans;

}

return ans;

}

-------------------------------------------------------------------------

python version is more concise:

def permute(self, nums):

perms = [[]]

for n in nums:

new\_perms = []

for perm in perms:

for i in xrange(len(perm)+1):

new\_perms.append(perm[:i] + [n] + perm[i:]) ###insert n

perms = new\_perms

return perms

# 47,Permutations II\*\*\*:

**Python\_solution:**

9-line python solution with 1 line to handle duplication, beat 99% of others :-)

Very similar to Permutation I, see explanations in https://leetcode.com/discuss/19510/my-ac-simple-iterative-java-python-solution. To handle duplication, just avoid inserting a number before any of its duplicates.

def permuteUnique(self, nums):

ans = [[]]

for n in nums:

new\_ans = []

for l in ans:

for i in xrange(len(l)+1):

new\_ans.append(l[:i]+[n]+l[i:])

if i<len(l) and l[i]==n: break #handles duplication

ans = new\_ans

return ans

# 48,Rotate Image\*\*\*:

**Python\_solution:**

1 line in Python

class Solution(object):

def rotate(self, matrix):

"""

:type matrix: List[List[int]]

:rtype: void Do not return anything, modify matrix in-place instead.

"""

matrix[::] = zip(\*matrix[::-1])

# 49,Group Anagrams\*\*\*:

**Python\_solution:**

2-line Python solution, AC with 350ms (some useful Python tricks)

def anagrams(self, strs):

count = collections.Counter([tuple(sorted(s)) for s in strs])

return filter(lambda x: count[tuple(sorted(x))]>1, strs)

- collections.Counter creates a counter object. A counter object is like a specific kind of dictionary where it is build for counting (objects that hashes to same value)

- tuple(sorted(s)) is used here so that anagrams will be hashed to the same value. tuple is used because sorted returns a list which cannot be hashed but tuples can be hashed

- filter: selects some elements of the list based on given function (first argument - a lambda function is given here)

- lambda function defined here returns True if number of anagrams of that elements is greater than 1

# 50,Pow(x, n)\*\*\*:

**Python\_solution:**

Shortest Python - Guaranteed

[Surprisingly](http://stackoverflow.com/questions/30693639/why-does-class-x-mypow-pow-work-what-about-self), I can just use Python's existing `pow` like this:

class Solution:

myPow = pow

That's even shorter than the other more obvious "cheat":

class Solution:

def myPow(self, x, n):

return x \*\* n

And to calm down the haters, here's me \*"doing it myself"\*:

Recursive:

class Solution:

def myPow(self, x, n):

if not n:

return 1

if n < 0:

return 1 / self.myPow(x, -n)

if n % 2:

return x \* self.myPow(x, n-1)

return self.myPow(x\*x, n/2)

Iterative:

class Solution:

def myPow(self, x, n):

if n < 0:

x = 1 / x

n = -n

pow = 1

while n:

if n & 1:

pow \*= x

x \*= x

n >>= 1

return pow

# 51,N-Queens\*\*\*:

**Python\_solution:**

Fast, short, and easy-to-understand python solution, 11 lines, 76ms

ideas: <br>

Use the `DFS` helper function to find solutions recursively. A solution will be found when the length of `queens` is equal to `n` ( `queens` is a list of the indices of the queens).<br><br>

In this problem, whenever a location `(x, y`) is occupied, any other locations `(p, q )` where `p + q == x + y` or `p - q == x - y` would be invalid. We can use this information to keep track of the indicators (`xy\_dif` and `xy\_sum` ) of the invalid positions and then call DFS recursively with valid positions only. <br><br>

At the end, we convert the result (a list of lists; each sublist is the indices of the queens) into the desire format.

def solveNQueens(self, n):

def DFS(queens, xy\_dif, xy\_sum):

p = len(queens)

if p==n:

result.append(queens)

return None

for q in range(n):

if q not in queens and p-q not in xy\_dif and p+q not in xy\_sum:

DFS(queens+[q], xy\_dif+[p-q], xy\_sum+[p+q])

result = []

DFS([],[],[])

return [ ["."\*i + "Q" + "."\*(n-i-1) for i in sol] for sol in result]

# 52,N-Queens II\*\*\*:

**Python\_solution:**

Python recursive dfs solution.

The idea here is quite similar to [N-Queens ][1] while we don't need to record the path, and as the return value is a number not a list, it's better to use a global variable to record the result.

def totalNQueens(self, n):

self.res = 0

self.dfs([-1]\*n, 0)

return self.res

def dfs(self, nums, index):

if index == len(nums):

self.res += 1

return

for i in xrange(len(nums)):

nums[index] = i

if self.valid(nums, index):

self.dfs(nums, index+1)

def valid(self, nums, n):

for i in xrange(n):

if nums[i] == nums[n] or abs(nums[n]-nums[i]) == n-i:

return False

return True

[1]: https://leetcode.com/discuss/53764/python-recursive-dfs-solution-with-comments

# 53,Maximum Subarray\*\*\*:

**Python\_solution:**

A Python solution

class Solution:

# @param A, a list of integers

# @return an integer

# 6:57

def maxSubArray(self, A):

if not A:

return 0

curSum = maxSum = A[0]

for num in A[1:]:

curSum = max(num, curSum + num)

maxSum = max(maxSum, curSum)

return maxSum

# 54,Spiral Matrix\*\*\*:

**Python\_solution:**

1-liner in Python

Inefficient for large matrices, but here I got it accepted in 40 ms, one of the fastest Python submissions.

def spiralOrder(self, matrix):

return matrix and list(matrix.pop(0)) + self.spiralOrder(zip(\*matrix)[::-1])

# 55,Jump Game\*\*\*:

**Best\_solution:**

Linear and simple solution in C++

I just iterate and update the maximal index that I can reach

bool canJump(int A[], int n) {

int i = 0;

for (int reach = 0; i < n && i <= reach; ++i)

reach = max(i + A[i], reach);

return i == n;

}

# 56,Merge Intervals\*\*\*:

**Python\_solution:**

7 lines, easy, Python

Just go through the intervals sorted by start coordinate and either combine the current interval with the previous one if they overlap, or add it to the output by itself if they don't.

def merge(self, intervals):

out = []

for i in sorted(intervals, key=lambda i: i.start):

if out and i.start <= out[-1].end:

out[-1].end = max(out[-1].end, i.end)

else:

out += i,

return out

# 57,Insert Interval\*\*\*:

**Python\_solution:**

O(n) Python solution

class Solution:

# @param intervals, a list of Intervals

# @param newInterval, a Interval

# @return a list of Interval

def insert(self, intervals, newInterval):

start = newInterval.start

end = newInterval.end

result = []

i = 0

while i < len(intervals):

if start <= intervals[i].end:

if end < intervals[i].start:

break

start = min(start, intervals[i].start)

end = max(end, intervals[i].end)

else:

result.append(intervals[i])

i += 1

result.append(Interval(start, end))

result += intervals[i:]

return result

# 58,Length of Last Word\*\*\*:

**Python\_solution:**

One line Python solution

def lengthOfLastWord(self, s):

return len(s.rstrip(' ').split(' ')[-1])

I know this is not the solution that the question wants. Just for fun.

# 59,Spiral Matrix II\*\*\*:

**Python\_solution:**

4-9 lines Python solutions

\*\*Solution 1: \*Build it inside-out\*\*\* - 44 ms, 5 lines

Start with the empty matrix, add the numbers in reverse order until we added the number 1. Always rotate the matrix clockwise and add a top row:

|| => |9| => |8| |6 7| |4 5| |1 2 3|

|9| => |9 8| => |9 6| => |8 9 4|

|8 7| |7 6 5|

The code:

def generateMatrix(self, n):

A, lo = [], n\*n+1

while lo > 1:

lo, hi = lo - len(A), lo

A = [range(lo, hi)] + zip(\*A[::-1])

return A

While this isn't O(n^2), it's actually quite fast, presumably due to me not doing much in Python but relying on `zip` and `range` and `+` being fast. I got it accepted in 44 ms, matching the fastest time for recent Python submissions (according to the submission detail page).

---

\*\*Solution 2: \*Ugly inside-out\*\*\* - 48 ms, 4 lines

Same as solution 1, but without helper variables. Saves a line, but makes it ugly. Also, because I access A[0][0], I had to handle the n=0 case differently.

def generateMatrix(self, n):

A = [[n\*n]]

while A[0][0] > 1:

A = [range(A[0][0] - len(A), A[0][0])] + zip(\*A[::-1])

return A \* (n>0)

---

\*\*Solution 3: \*Walk the spiral\*\*\* - 52 ms, 9 lines

Initialize the matrix with zeros, then walk the spiral path and write the numbers 1 to n\*n. Make a right turn when the cell ahead is already non-zero.

def generateMatrix(self, n):

A = [[0] \* n for \_ in range(n)]

i, j, di, dj = 0, 0, 0, 1

for k in xrange(n\*n):

A[i][j] = k + 1

if A[(i+di)%n][(j+dj)%n]:

di, dj = dj, -di

i += di

j += dj

return A

# 60,Permutation Sequence\*\*\*:

**Python\_solution:**

Share my Python solution with detailed explanation

The idea is as follow:

For permutations of n, the first (n-1)! permutations start with 1, next (n-1)! ones start with 2, ... and so on. And in each group of (n-1)! permutations, the first (n-2)! permutations start with the smallest remaining number, ...

take n = 3 as an example, the first 2 (that is, (3-1)! ) permutations start with 1, next 2 start with 2 and last 2 start with 3. For the first 2 permutations (123 and 132), the 1st one (1!) starts with 2, which is the smallest remaining number (2 and 3). So we can use a loop to check the region that the sequence number falls in and get the starting digit. Then we adjust the sequence number and continue.

import math

class Solution:

# @param {integer} n

# @param {integer} k

# @return {string}

def getPermutation(self, n, k):

numbers = range(1, n+1)

permutation = ''

k -= 1

while n > 0:

n -= 1

# get the index of current digit

index, k = divmod(k, math.factorial(n))

permutation += str(numbers[index])

# remove handled number

numbers.remove(numbers[index])

return permutation

# 61,Rotate List\*\*\*:

**Python\_solution:**

97.63% Python Solution

class Solution(object):

def rotateRight(self, head, k):

"""

:type head: ListNode

:type k: int

:rtype: ListNode

"""

if not head:

return None

if head.next == None:

return head

pointer = head

length = 1

while pointer.next:

pointer = pointer.next

length += 1

rotateTimes = k%length

if k == 0 or rotateTimes == 0:

return head

fastPointer = head

slowPointer = head

for a in range (rotateTimes):

fastPointer = fastPointer.next

while fastPointer.next:

slowPointer = slowPointer.next

fastPointer = fastPointer.next

temp = slowPointer.next

slowPointer.next = None

fastPointer.next = head

head = temp

return head

# 62,Unique Paths\*\*\*:

**Python\_solution:**

1 Line Math Solution (Python)

class Solution(object):

def uniquePaths(self, m, n):

"""

:type m: int

:type n: int

:rtype: int

"""

return math.factorial(m+n-2)/math.factorial(m-1)/math.factorial(n-1)

# 63,Unique Paths II\*\*\*:

**Python\_solution:**

Accepted simple Python in-place solution

As below. Any comments on how to make it shorter? Thx!

class Solution:

# @param obstacleGrid, a list of lists of integers

# @return an integer

def uniquePathsWithObstacles(self, obstacleGrid):

m = len(obstacleGrid)

n = len(obstacleGrid[0])

obstacleGrid[0][0] = 1 - obstacleGrid[0][0]

for i in range(1, n):

if not obstacleGrid[0][i]:

obstacleGrid[0][i] = obstacleGrid[0][i-1]

else:

obstacleGrid[0][i] = 0

for i in range(1, m):

if not obstacleGrid[i][0]:

obstacleGrid[i][0] = obstacleGrid[i-1][0]

else:

obstacleGrid[i][0] = 0

for i in range(1, m):

for j in range(1, n):

if not obstacleGrid[i][j]:

obstacleGrid[i][j] = obstacleGrid[i][j-1]+obstacleGrid[i-1][j]

else:

obstacleGrid[i][j] = 0

return obstacleGrid[-1][-1]

# 64,Minimum Path Sum\*\*\*:

**Python\_solution:**

Simple python dp 70ms

def minPathSum(self, grid):

m = len(grid)

n = len(grid[0])

for i in range(1, n):

grid[0][i] += grid[0][i-1]

for i in range(1, m):

grid[i][0] += grid[i-1][0]

for i in range(1, m):

for j in range(1, n):

grid[i][j] += min(grid[i-1][j], grid[i][j-1])

return grid[-1][-1]

# 65,Valid Number\*\*\*:

**Python\_solution:**

A simple solution in Python based on DFA

I was asked in the interview of linkedIn, writing it directly can be extremely complicated, for there are many special cases we have to deal with, and the code I wrote was messy. Then I failed to pass the interview.

Here's a clear solution. With DFA we can easily get our idea into shape and then debug, and the source code is clear and simple.

class Solution(object):

def isNumber(self, s):

"""

:type s: str

:rtype: bool

"""

#define a DFA

state = [{},

{'b': 1, 's': 2, 'd':3, '.':4},

{'d':3, '.':4},

{'d':3, '.':5, 'e':6, 'b':9},

{'d':5},

{'d':5, 'e':6, 'b':9},

{'s':7, 'd':8},

{'d':8},

{'d':8, 'b':9},

{'b':9}]

currentState = 1

for c in s:

if c >= '0' and c <= '9':

c = 'd'

if c == ' ':

c = 'b'

if c in ['+', '-']:

c = 's'

if c not in state[currentState].keys():

return False

currentState = state[currentState][c]

if currentState not in [3,5,8,9]:

return False

return True

![enter image description here][1]

[1]: http://normanyahq.github.io/static/files/valid\_number\_dfa.svg

# 66,Plus One\*\*\*:

**Python\_solution:**

Simple Python solution with explanation (Plus One)

def plusOne(digits):

num = 0

for i in range(len(digits)):

num += digits[i] \* pow(10, (len(digits)-1-i))

return [int(i) for i in str(num+1)]

We're given a list of digits, and the idea here is to convert that list to an integer, \*num\*. So each digit is multiplied by the proper place value and added to \*num\*. For example, if \*digits\* = [3, 8, 2, 5] then on the first iteration 3 is multiplied by 10 to the power of 4-1-0 = 3, so this results in 3000, which is added to \*num\*. Then 8 is multiplied by 10^2 and added to \*num\*, and so on.

The last step is to add 1 to \*num\*, convert it to a list and return that list.

# 67,Add Binary\*\*\*:

**Python\_solution:**

An accepted concise Python recursive solution 10 lines

#add two binary from back to front, I think it is very self explained, when 1+1 we need a carry.

class Solution:

def addBinary(self, a, b):

if len(a)==0: return b

if len(b)==0: return a

if a[-1] == '1' and b[-1] == '1':

return self.addBinary(self.addBinary(a[0:-1],b[0:-1]),'1')+'0'

if a[-1] == '0' and b[-1] == '0':

return self.addBinary(a[0:-1],b[0:-1])+'0'

else:

return self.addBinary(a[0:-1],b[0:-1])+'1'

# 68,Text Justification\*\*\*:

**Python\_solution:**

Concise python solution, 10 lines.

def fullJustify(self, words, maxWidth):

res, cur, num\_of\_letters = [], [], 0

for w in words:

if num\_of\_letters + len(w) + len(cur) > maxWidth:

for i in range(maxWidth - num\_of\_letters):

cur[i%(len(cur)-1 or 1)] += ' '

res.append(''.join(cur))

cur, num\_of\_letters = [], 0

cur += [w]

num\_of\_letters += len(w)

return res + [' '.join(cur).ljust(maxWidth)]

How does it work? Well in the question statement, the sentence "Extra spaces between words should be distributed as evenly as possible. If the number of spaces on a line do not divide evenly between words, the empty slots on the left will be assigned more spaces than the slots on the right" was just a really long and awkward way to say \*round robin\*. The following line implements the round robin logic:

for i in range(maxWidth - num\_of\_letters):

cur[i%(len(cur)-1 or 1)] += ' '

What does this line do? Once you determine that there are only k words that can fit on a given line, you know what the total length of those words is num\_of\_letters. Then the rest are spaces, and there are (maxWidth - num\_of\_letters) of spaces. The "or 1" part is for dealing with the edge case len(cur) == 1.

--------------------------------------------

The following is my older solution for reference, longer and less clear. The idea is the same, but I did not figure out the nice way to distribute the space at the time.

def fullJustify(self, words, maxWidth):

res, cur, num\_of\_letters = [], [], 0

for w in words:

if num\_of\_letters + len(w) + len(cur) > maxWidth:

if len(cur) == 1:

res.append( cur[0] + ' '\*(maxWidth - num\_of\_letters) )

else:

num\_spaces = maxWidth - num\_of\_letters

space\_between\_words, num\_extra\_spaces = divmod( num\_spaces, len(cur)-1)

for i in range(num\_extra\_spaces):

cur[i] += ' '

res.append( (' '\*space\_between\_words).join(cur) )

cur, num\_of\_letters = [], 0

cur += [w]

num\_of\_letters += len(w)

res.append( ' '.join(cur) + ' '\*(maxWidth - num\_of\_letters - len(cur) + 1) )

return res

# 69,Sqrt(x)\*\*\*:

**Python\_solution:**

Python binary search solution (O(lgn)).

# Binary search

def mySqrt(self, x):

l, r = 0, x

while l <= r:

mid = l + (r-l)//2

if mid \* mid <= x < (mid+1)\*(mid+1):

return mid

elif x < mid \* mid:

r = mid

else:

l = mid + 1

# 70,Climbing Stairs\*\*\*:

**Python\_solution:**

Python different solutions (bottom up, top down).

# Top down - TLE

def climbStairs1(self, n):

if n == 1:

return 1

if n == 2:

return 2

return self.climbStairs(n-1)+self.climbStairs(n-2)

# Bottom up, O(n) space

def climbStairs2(self, n):

if n == 1:

return 1

res = [0 for i in xrange(n)]

res[0], res[1] = 1, 2

for i in xrange(2, n):

res[i] = res[i-1] + res[i-2]

return res[-1]

# Bottom up, constant space

def climbStairs3(self, n):

if n == 1:

return 1

a, b = 1, 2

for i in xrange(2, n):

tmp = b

b = a+b

a = tmp

return b

# Top down + memorization (list)

def climbStairs4(self, n):

if n == 1:

return 1

dic = [-1 for i in xrange(n)]

dic[0], dic[1] = 1, 2

return self.helper(n-1, dic)

def helper(self, n, dic):

if dic[n] < 0:

dic[n] = self.helper(n-1, dic)+self.helper(n-2, dic)

return dic[n]

# Top down + memorization (dictionary)

def \_\_init\_\_(self):

self.dic = {1:1, 2:2}

def climbStairs(self, n):

if n not in self.dic:

self.dic[n] = self.climbStairs(n-1) + self.climbStairs(n-2)

return self.dic[n]

# 71,Simplify Path\*\*\*:

**Python\_solution:**

9 lines of Python code

class Solution(object):

def simplifyPath(self, path):

places = [p for p in path.split("/") if p!="." and p!=""]

stack = []

for p in places:

if p == "..":

if len(stack) > 0:

stack.pop()

else:

stack.append(p)

return "/" + "/".join(stack)

# 72,Edit Distance\*\*\*:

**Python\_solution:**

Python solutions (O(m\*n), O(n) space).

# O(m\*n) space

def minDistance1(self, word1, word2):

l1, l2 = len(word1)+1, len(word2)+1

dp = [[0 for \_ in xrange(l2)] for \_ in xrange(l1)]

for i in xrange(l1):

dp[i][0] = i

for j in xrange(l2):

dp[0][j] = j

for i in xrange(1, l1):

for j in xrange(1, l2):

dp[i][j] = min(dp[i-1][j]+1, dp[i][j-1]+1, dp[i-1][j-1]+(word1[i-1]!=word2[j-1]))

return dp[-1][-1]

# O(n) space with rolling array

def minDistance(self, word1, word2):

l1, l2 = len(word1)+1, len(word2)+1

pre = [0 for \_ in xrange(l2)]

for j in xrange(l2):

pre[j] = j

for i in xrange(1, l1):

cur = [i]\*l2

for j in xrange(1, l2):

cur[j] = min(cur[j-1]+1, pre[j]+1, pre[j-1]+(word1[i-1]!=word2[j-1]))

pre = cur[:]

return pre[-1]

# 73,Set Matrix Zeroes\*\*\*:

**Python\_solution:**

O(1) space solution in Python

class Solution:

# @param {integer[][]} matrix

# @return {void} Do not return anything, modify matrix in-place instead.

def setZeroes(self, matrix):

m = len(matrix)

if m == 0:

return

n = len(matrix[0])

row\_zero = False

for i in range(m):

if matrix[i][0] == 0:

row\_zero = True

col\_zero = False

for j in range(n):

if matrix[0][j] == 0:

col\_zero = True

for i in range(1, m):

for j in range(1, n):

if matrix[i][j] == 0:

matrix[i][0] = 0

matrix[0][j] = 0

for i in range(1, m):

if matrix[i][0] == 0:

for j in range(1, n):

matrix[i][j] = 0

for j in range(1, n):

if matrix[0][j] == 0:

for i in range(1, m):

matrix[i][j] = 0

if col\_zero:

for j in range(n):

matrix[0][j] = 0

if row\_zero:

for i in range(m):

matrix[i][0] = 0

# 74,Search a 2D Matrix\*\*\*:

**Python\_solution:**

A Python binary search solution - O(logn)

It is basically an advanced version of the binary search

class Solution:

# @param matrix, a list of lists of integers

# @param target, an integer

# @return a boolean

# 8:21

def searchMatrix(self, matrix, target):

if not matrix or target is None:

return False

rows, cols = len(matrix), len(matrix[0])

low, high = 0, rows \* cols - 1

while low <= high:

mid = (low + high) / 2

num = matrix[mid / cols][mid % cols]

if num == target:

return True

elif num < target:

low = mid + 1

else:

high = mid - 1

return False

# 75,Sort Colors\*\*\*:

**Python\_solution:**

AC Python in place one pass solution O(n) time O(1) space, no swap no count

def sortColors(self, nums):

i = j = 0

for k in xrange(len(nums)):

v = nums[k]

nums[k] = 2

if v < 2:

nums[j] = 1

j += 1

if v == 0:

nums[i] = 0

i += 1

# 86 / 86 test cases passed.

# Status: Accepted

# Runtime: 44 ms

# 84.03%

Just like the Lomuto partition algorithm usually used in quick sort. We keep a loop invariant that [0,i) [i, j) [j, k) are 0s, 1s and 2s sorted in place for [0,k). Here ")" means exclusive. We don't need to swap because we know the values we want.

# 76,Minimum Window Substring\*\*\*:

**Python\_solution:**

12 lines Python

The current window is `s[i:j]` and the result window is `s[I:J]`. In `need[c]` I store how many times I need character `c` (can be negative) and `missing` tells how many characters are still missing. In the loop, first add the new character to the window. Then, if nothing is missing, remove as much as possible from the window start and then update the result.

def minWindow(self, s, t):

need, missing = collections.Counter(t), len(t)

i = I = J = 0

for j, c in enumerate(s, 1):

missing -= need[c] > 0

need[c] -= 1

if not missing:

while i < j and need[s[i]] < 0:

need[s[i]] += 1

i += 1

if not J or j - i <= J - I:

I, J = i, j

return s[I:J]

# 77,Combinations\*\*\*:

**Python\_solution:**

AC Python backtracking iterative solution 60 ms

def combine(self, n, k):

ans = []

stack = []

x = 1

while True:

l = len(stack)

if l == k:

ans.append(stack[:])

if l == k or x > n - k + l + 1:

if not stack:

return ans

x = stack.pop() + 1

else:

stack.append(x)

x += 1

# 26 / 26 test cases passed.

# Status: Accepted

# Runtime: 60 ms

# 98.51%

Combinations is typical application for backtracking. Two conditions for back track: (1) the stack length is already k (2) the current value is too large for the rest slots to fit in since we are using ascending order to make sure the uniqueness of each combination.

# 78,Subsets\*\*\*:

**Python\_solution:**

Python easy to understand solutions (DFS recursively, Bit Manipulation, Iteratively).

# DFS recursively

def subsets1(self, nums):

res = []

self.dfs(sorted(nums), 0, [], res)

return res

def dfs(self, nums, index, path, res):

res.append(path)

for i in xrange(index, len(nums)):

self.dfs(nums, i+1, path+[nums[i]], res)

# Bit Manipulation

def subsets2(self, nums):

res = []

nums.sort()

for i in xrange(1<<len(nums)):

tmp = []

for j in xrange(len(nums)):

if i & 1 << j: # if i >> j & 1:

tmp.append(nums[j])

res.append(tmp)

return res

# Iteratively

def subsets(self, nums):

res = [[]]

for num in sorted(nums):

res += [item+[num] for item in res]

return res

# 79,Word Search\*\*\*:

**Python\_solution:**

Python dfs solution with comments.

def exist(self, board, word):

if not board:

return False

for i in xrange(len(board)):

for j in xrange(len(board[0])):

if self.dfs(board, i, j, word):

return True

return False

# check whether can find word, start at (i,j) position

def dfs(self, board, i, j, word):

if len(word) == 0: # all the characters are checked

return True

if i<0 or i>=len(board) or j<0 or j>=len(board[0]) or word[0]!=board[i][j]:

return False

tmp = board[i][j] # first character is found, check the remaining part

board[i][j] = "#" # avoid visit agian

# check whether can find "word" along one direction

res = self.dfs(board, i+1, j, word[1:]) or self.dfs(board, i-1, j, word[1:]) \

or self.dfs(board, i, j+1, word[1:]) or self.dfs(board, i, j-1, word[1:])

board[i][j] = tmp

return res

# 80,Remove Duplicates from Sorted Array II\*\*\*:

**Python\_solution:**

3-6 easy lines, C++, Java, Python, Ruby

Same simple solution written in several languages. Just go through the numbers and include those in the result that haven't been included twice already.

\*\*C++\*\*

int removeDuplicates(vector<int>& nums) {

int i = 0;

for (int n : nums)

if (i < 2 || n > nums[i-2])

nums[i++] = n;

return i;

}

\*\*Java\*\*

public int removeDuplicates(int[] nums) {

int i = 0;

for (int n : nums)

if (i < 2 || n > nums[i-2])

nums[i++] = n;

return i;

}

\*\*Python\*\*

def removeDuplicates(self, nums):

i = 0

for n in nums:

if i < 2 or n > nums[i-2]:

nums[i] = n

i += 1

return i

\*\*Ruby\*\*

def remove\_duplicates(nums)

i = 0

nums.each { |n| nums[(i+=1)-1] = n if i < 2 || n > nums[i-2] }

i

end

# 81,Search in Rotated Sorted Array II\*\*\*:

**Python\_solution:**

Python easy to understand solution (with comments).

def search(self, nums, target):

l, r = 0, len(nums)-1

while l <= r:

mid = l + (r-l)//2

if nums[mid] == target:

return True

while l < mid and nums[l] == nums[mid]: # tricky part

l += 1

# the first half is ordered

if nums[l] <= nums[mid]:

# target is in the first half

if nums[l] <= target < nums[mid]:

r = mid - 1

else:

l = mid + 1

# the second half is ordered

else:

# target is in the second half

if nums[mid] < target <= nums[r]:

l = mid + 1

else:

r = mid - 1

return False

# 82,Remove Duplicates from Sorted List II\*\*\*:

**Python\_solution:**

Python in-place solution with dummy head node.

def deleteDuplicates(self, head):

dummy = pre = ListNode(0)

dummy.next = head

while head and head.next:

if head.val == head.next.val:

while head and head.next and head.val == head.next.val:

head = head.next

head = head.next

pre.next = head

else:

pre = pre.next

head = head.next

return dummy.next

# 83,Remove Duplicates from Sorted List\*\*\*:

**Python\_solution:**

Simple iterative Python 6 lines, 60 ms

def deleteDuplicates(self, head):

cur = head

while cur:

while cur.next and cur.next.val == cur.val:

cur.next = cur.next.next # skip duplicated node

cur = cur.next # not duplicate of current node, move to next node

return head

# 84,Largest Rectangle in Histogram\*\*\*:

**Python\_solution:**

AC Python clean solution using stack 76ms

def largestRectangleArea(self, height):

height.append(0)

stack = [-1]

ans = 0

for i in xrange(len(height)):

while height[i] < height[stack[-1]]:

h = height[stack.pop()]

w = i - stack[-1] - 1

ans = max(ans, h \* w)

stack.append(i)

height.pop()

return ans

# 94 / 94 test cases passed.

# Status: Accepted

# Runtime: 76 ms

# 97.34%

The stack maintain the indexes of buildings with ascending height. Before adding a new building pop the building who is taller than the new one. The building popped out represent the height of a rectangle with the new building as the right boundary and the current stack top as the left boundary. Calculate its area and update ans of maximum area. Boundary is handled using dummy buildings.

# 85,Maximal Rectangle\*\*\*:

**Python\_solution:**

AC Python DP solutioin 120ms based on largest rectangle in histogram

def maximalRectangle(self, matrix):

if not matrix or not matrix[0]:

return 0

n = len(matrix[0])

height = [0] \* (n + 1)

ans = 0

for row in matrix:

for i in xrange(n):

height[i] = height[i] + 1 if row[i] == '1' else 0

stack = [-1]

for i in xrange(n + 1):

while height[i] < height[stack[-1]]:

h = height[stack.pop()]

w = i - 1 - stack[-1]

ans = max(ans, h \* w)

stack.append(i)

return ans

# 65 / 65 test cases passed.

# Status: Accepted

# Runtime: 120 ms

# 100%

The solution is based on [largest rectangle in histogram][1] solution. Every row in the matrix is viewed as the ground with some buildings on it. The building height is the count of consecutive 1s from that row to above rows. The rest is then the same as [this solution for largest rectangle in histogram][2]

[1]: https://leetcode.com/problems/largest-rectangle-in-histogram/

[2]: https://leetcode.com/discuss/65647/ac-python-clean-solution-using-stack-76ms

# 86,Partition List\*\*\*:

**Python\_solution:**

Python concise solution with dummy nodes.

def partition(self, head, x):

h1 = l1 = ListNode(0)

h2 = l2 = ListNode(0)

while head:

if head.val < x:

l1.next = head

l1 = l1.next

else:

l2.next = head

l2 = l2.next

head = head.next

l2.next = None

l1.next = h2.next

return h1.next

# 87,Scramble String\*\*\*:

**Python\_solution:**

Python recursive solution

class Solution:

# @return a boolean

def isScramble(self, s1, s2):

n, m = len(s1), len(s2)

if n != m or sorted(s1) != sorted(s2):

return False

if n < 4 or s1 == s2:

return True

f = self.isScramble

for i in range(1, n):

if f(s1[:i], s2[:i]) and f(s1[i:], s2[i:]) or \

f(s1[:i], s2[-i:]) and f(s1[i:], s2[:-i]):

return True

return False

# 88,Merge Sorted Array\*\*\*:

**Python\_solution:**

Beautiful Python Solution

def merge(self, nums1, m, nums2, n):

while m > 0 and n > 0:

if nums1[m-1] >= nums2[n-1]:

nums1[m+n-1] = nums1[m-1]

m -= 1

else:

nums1[m+n-1] = nums2[n-1]

n -= 1

if n > 0:

nums1[:n] = nums2[:n]

# 89,Gray Code\*\*\*:

**Python\_solution:**

One-liner Python solution (with demo in comments)

All you need is a bit of careful thought.

Btw, it's extremely useful to write down your thought/demo in comments before you actually start to write the code, especially during interview.

Even if you do not solve the problem finally, the interviewer at least get to know what you're thinking.

And if you don't get the problem right, he/she will have a chance to correct you.

class Solution:

# @return a list of integers

'''

from up to down, then left to right

0 1 11 110

10 111

101

100

start: [0]

i = 0: [0, 1]

i = 1: [0, 1, 3, 2]

i = 2: [0, 1, 3, 2, 6, 7, 5, 4]

'''

def grayCode(self, n):

results = [0]

for i in range(n):

results += [x + pow(2, i) for x in reversed(results)]

return results

# 90,Subsets II\*\*\*:

**Python\_solution:**

Simple python solution without extra space.

class Solution:

# @param num, a list of integer

# @return a list of lists of integer

def subsetsWithDup(self, S):

res = [[]]

S.sort()

for i in range(len(S)):

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

l = len(res)

for j in range(len(res) - l, len(res)):

res.append(res[j] + [S[i]])

return res

if S[i] is same to S[i - 1], then it needn't to be added to all of the subset, just add it to the last l subsets which are created by adding S[i - 1]

# 91,Decode Ways\*\*\*:

**Python\_solution:**

Accpeted Python DP solution

class Solution:

# @param s, a string

# @return an integer

def numDecodings(self, s):

#dp[i] = dp[i-1] if s[i] != "0"

# +dp[i-2] if "09" < s[i-1:i+1] < "27"

if s == "": return 0

dp = [0 for x in range(len(s)+1)]

dp[0] = 1

for i in range(1, len(s)+1):

if s[i-1] != "0":

dp[i] += dp[i-1]

if i != 1 and s[i-2:i] < "27" and s[i-2:i] > "09": #"01"ways = 0

dp[i] += dp[i-2]

return dp[len(s)]

# 92,Reverse Linked List II\*\*\*:

**Python\_solution:**

Python one pass iterative solution

The idea is simple and intuitive: find linkedlist [m, n], reverse it, then connect m with n+1, connect n with m-1

class Solution:

# @param head, a ListNode

# @param m, an integer

# @param n, an integer

# @return a ListNode

def reverseBetween(self, head, m, n):

if m == n:

return head

dummyNode = ListNode(0)

dummyNode.next = head

pre = dummyNode

for i in range(m - 1):

pre = pre.next

# reverse the [m, n] nodes

reverse = None

cur = pre.next

for i in range(n - m + 1):

next = cur.next

cur.next = reverse

reverse = cur

cur = next

pre.next.next = cur

pre.next = reverse

return dummyNode.next

# 93,Restore IP Addresses\*\*\*:

**Python\_solution:**

Adding a python solution, also requesting for improvement

class Solution:

# @param s, a string

# @return a list of strings

def restoreIpAddresses(self,s):

answer = []

s\_len = len(s)

for i in [1,2,3]:

for j in [i+1,i+2,i+3]:

for k in [j+1,j+2,j+3]:

if k >= s\_len:

continue

s1 = s[:i]

s2 = s[i:j]

s3 = s[j:k]

s4 = s[k:]

if self.check\_valid([s1,s2,s3,s4]):

new\_string = s1 + "." + s2 + "." + s3 + "." + s4

answer.append(new\_string)

return answer

def check\_valid(self,str\_list):

for s in str\_list:

if s[0] == "0" and s != "0":

return False

if int(s) > 255:

return False

return True

# 94,Binary Tree Inorder Traversal\*\*\*:

**Python\_solution:**

Python recursive and iterative solutions.

# recursively

def inorderTraversal1(self, root):

res = []

self.helper(root, res)

return res

def helper(self, root, res):

if root:

self.helper(root.left, res)

res.append(root.val)

self.helper(root.right, res)

# iteratively

def inorderTraversal(self, root):

res, stack = [], []

while True:

while root:

stack.append(root)

root = root.left

if not stack:

return res

node = stack.pop()

res.append(node.val)

root = node.right

# 95,Unique Binary Search Trees II\*\*\*:

**Python\_solution:**

Recursive python solution

class Solution(object):

def generateTrees(self, n):

"""

:type n: int

:rtype: List[TreeNode]

"""

if n == 0:

return [[]]

return self.dfs(1, n+1)

def dfs(self, start, end):

if start == end:

return None

result = []

for i in xrange(start, end):

for l in self.dfs(start, i) or [None]:

for r in self.dfs(i+1, end) or [None]:

node = TreeNode(i)

node.left, node.right = l, r

result.append(node)

return result

Use start/end instead of actual nodes to bosst the program.

# 96,Unique Binary Search Trees\*\*\*:

**Python\_solution:**

Python solutions (DP + Catalan number)

# DP

def numTrees1(self, n):

res = [0] \* (n+1)

res[0] = 1

for i in xrange(1, n+1):

for j in xrange(i):

res[i] += res[j] \* res[i-1-j]

return res[n]

# Catalan Number (2n)!/((n+1)!\*n!)

def numTrees(self, n):

return math.factorial(2\*n)/(math.factorial(n)\*math.factorial(n+1))

# 97,Interleaving String\*\*\*:

**Python\_solution:**

Python DP solutions (O(m\*n), O(n) space), BFS, DFS.

# O(m\*n) space

def isInterleave1(self, s1, s2, s3):

r, c, l= len(s1), len(s2), len(s3)

if r+c != l:

return False

dp = [[True for \_ in xrange(c+1)] for \_ in xrange(r+1)]

for i in xrange(1, r+1):

dp[i][0] = dp[i-1][0] and s1[i-1] == s3[i-1]

for j in xrange(1, c+1):

dp[0][j] = dp[0][j-1] and s2[j-1] == s3[j-1]

for i in xrange(1, r+1):

for j in xrange(1, c+1):

dp[i][j] = (dp[i-1][j] and s1[i-1] == s3[i-1+j]) or \

(dp[i][j-1] and s2[j-1] == s3[i-1+j])

return dp[-1][-1]

# O(2\*n) space

def isInterleave2(self, s1, s2, s3):

l1, l2, l3 = len(s1)+1, len(s2)+1, len(s3)+1

if l1+l2 != l3+1:

return False

pre = [True for \_ in xrange(l2)]

for j in xrange(1, l2):

pre[j] = pre[j-1] and s2[j-1] == s3[j-1]

for i in xrange(1, l1):

cur = [pre[0] and s1[i-1] == s3[i-1]] \* l2

for j in xrange(1, l2):

cur[j] = (cur[j-1] and s2[j-1] == s3[i+j-1]) or \

(pre[j] and s1[i-1] == s3[i+j-1])

pre = cur[:]

return pre[-1]

# O(n) space

def isInterleave3(self, s1, s2, s3):

r, c, l= len(s1), len(s2), len(s3)

if r+c != l:

return False

dp = [True for \_ in xrange(c+1)]

for j in xrange(1, c+1):

dp[j] = dp[j-1] and s2[j-1] == s3[j-1]

for i in xrange(1, r+1):

dp[0] = (dp[0] and s1[i-1] == s3[i-1])

for j in xrange(1, c+1):

dp[j] = (dp[j] and s1[i-1] == s3[i-1+j]) or (dp[j-1] and s2[j-1] == s3[i-1+j])

return dp[-1]

# DFS

def isInterleave4(self, s1, s2, s3):

r, c, l= len(s1), len(s2), len(s3)

if r+c != l:

return False

stack, visited = [(0, 0)], set((0, 0))

while stack:

x, y = stack.pop()

if x+y == l:

return True

if x+1 <= r and s1[x] == s3[x+y] and (x+1, y) not in visited:

stack.append((x+1, y)); visited.add((x+1, y))

if y+1 <= c and s2[y] == s3[x+y] and (x, y+1) not in visited:

stack.append((x, y+1)); visited.add((x, y+1))

return False

# BFS

def isInterleave(self, s1, s2, s3):

r, c, l= len(s1), len(s2), len(s3)

if r+c != l:

return False

queue, visited = [(0, 0)], set((0, 0))

while queue:

x, y = queue.pop(0)

if x+y == l:

return True

if x+1 <= r and s1[x] == s3[x+y] and (x+1, y) not in visited:

queue.append((x+1, y)); visited.add((x+1, y))

if y+1 <= c and s2[y] == s3[x+y] and (x, y+1) not in visited:

queue.append((x, y+1)); visited.add((x, y+1))

return False

# 98,Validate Binary Search Tree\*\*\*:

**Python\_solution:**

Python version based on inorder traversal

# Definition for a binary tree node

# class TreeNode:

# def \_\_init\_\_(self, x):

# self.val = x

# self.left = None

# self.right = None

class Solution:

# @param root, a tree node

# @return a boolean

# 7:38

def isValidBST(self, root):

output = []

self.inOrder(root, output)

for i in range(1, len(output)):

if output[i-1] >= output[i]:

return False

return True

def inOrder(self, root, output):

if root is None:

return

self.inOrder(root.left, output)

output.append(root.val)

self.inOrder(root.right, output)

# 99,Recover Binary Search Tree\*\*\*:

**Python\_solution:**

Tree Deserializer and Visualizer for Python

Wrote some tools for my own local testing. For example `deserialize('[1,2,3,null,null,4,null,null,5]')` will turn that into a tree and return the root [as explained in the FAQ](https://leetcode.com/faq/). I also wrote a visualizer. Two examples:

`drawtree(deserialize('[1,2,3,null,null,4,null,null,5]'))`:

![enter image description here][1]

`drawtree(deserialize('[2,1,3,0,7,9,1,2,null,1,0,null,null,8,8,null,null,null,null,7]'))`:

![enter image description here][2]

Here's the code. If you save it as a Python script and run it, it should as a demo show the above two pictures in turtle windows (one after the other). And you can of course import it from other scripts and then it will only provide the class/functions and not show the demo.

class TreeNode:

def \_\_init\_\_(self, val, left=None, right=None):

self.val = val

self.left = left

self.right = right

def \_\_repr\_\_(self):

return 'TreeNode({})'.format(self.val)

def deserialize(string):

if string == '{}':

return None

nodes = [None if val == 'null' else TreeNode(int(val))

for val in string.strip('[]{}').split(',')]

kids = nodes[::-1]

root = kids.pop()

for node in nodes:

if node:

if kids: node.left = kids.pop()

if kids: node.right = kids.pop()

return root

def drawtree(root):

def height(root):

return 1 + max(height(root.left), height(root.right)) if root else -1

def jumpto(x, y):

t.penup()

t.goto(x, y)

t.pendown()

def draw(node, x, y, dx):

if node:

t.goto(x, y)

jumpto(x, y-20)

t.write(node.val, align='center', font=('Arial', 12, 'normal'))

draw(node.left, x-dx, y-60, dx/2)

jumpto(x, y-20)

draw(node.right, x+dx, y-60, dx/2)

import turtle

t = turtle.Turtle()

t.speed(0); turtle.delay(0)

h = height(root)

jumpto(0, 30\*h)

draw(root, 0, 30\*h, 40\*h)

t.hideturtle()

turtle.mainloop()

if \_\_name\_\_ == '\_\_main\_\_':

drawtree(deserialize('[1,2,3,null,null,4,null,null,5]'))

drawtree(deserialize('[2,1,3,0,7,9,1,2,null,1,0,null,null,8,8,null,null,null,null,7]'))

[1]: http://pochmann.org/leetcode/images/tree1.png

[2]: http://pochmann.org/leetcode/images/tree2.png

# 100,Same Tree\*\*\*:

**Python\_solution:**

Shortest+simplest Python

The "proper" way:

class Solution:

def isSameTree(self, p, q):

if p and q:

return p.val == q.val and self.isSameTree(p.left, q.left) and self.isSameTree(p.right, q.right)

return p == q

The "tupleify" way:

class Solution:

def isSameTree(self, p, q):

def t(n):

return n and (n.val, t(n.left), t(n.right))

return t(p) == t(q)

# 101,Symmetric Tree\*\*\*:

**Python\_solution:**

Recursively and iteratively solution in Python

Basically, this question is recursively. Or we can say, the tree structure is recursively, so the recursively solution maybe easy to write:

TC: O(b) SC: O(log n)

class Solution:

def isSymmetric(self, root):

if root is None:

return True

else:

return self.isMirror(root.left, root.right)

def isMirror(self, left, right):

if left is None and right is None:

return True

if left is None or right is None:

return False

if left.val == right.val:

outPair = self.isMirror(left.left, right.right)

inPiar = self.isMirror(left.right, right.left)

return outPair and inPiar

else:

return False

The essence of recursively is Stack, so we can use our own stack to rewrite it into iteratively:

class Solution2:

def isSymmetric(self, root):

if root is None:

return True

stack = [[root.left, root.right]]

while len(stack) > 0:

pair = stack.pop(0)

left = pair[0]

right = pair[1]

if left is None and right is None:

continue

if left is None or right is None:

return False

if left.val == right.val:

stack.insert(0, [left.left, right.right])

stack.insert(0, [left.right, right.left])

else:

return False

return True

# 102,Binary Tree Level Order Traversal\*\*\*:

**Python\_solution:**

5-6 lines fast python solution (48 ms)

`level` is a list of the nodes in the current level. Keep appending a list of the values of these nodes to `ans` and then updating `level` with all the nodes in the next level (kids) until it reaches an empty level. Python's list comprehension makes it easier to deal with many conditions in a concise manner.

<br>

Solution 1, (6 lines)

def levelOrder(self, root):

ans, level = [], [root]

while root and level:

ans.append([node.val for node in level])

LRpair = [(node.left, node.right) for node in level]

level = [leaf for LR in LRpair for leaf in LR if leaf]

return ans

<br>

Solution 2, (5 lines), same idea but use only one list comprehension in while loop to get the next level

def levelOrder(self, root):

ans, level = [], [root]

while root and level:

ans.append([node.val for node in level])

level = [kid for n in level for kid in (n.left, n.right) if kid]

return ans

<br>

Solution 3 (10 lines), just an expansion of solution 1&2 for better understanding.

def levelOrder(self, root):

if not root:

return []

ans, level = [], [root]

while level:

ans.append([node.val for node in level])

temp = []

for node in level:

temp.extend([node.left, node.right])

level = [leaf for leaf in temp if leaf]

return ans

# 103,Binary Tree Zigzag Level Order Traversal\*\*\*:

**Python\_solution:**

Python simple BFS

Simple straightforward solution using flag to decide whether from left to right or from right to left

class Solution(object):

def zigzagLevelOrder(self, root):

"""

:type root: TreeNode

:rtype: List[List[int]]

"""

if not root: return []

res, temp, stack, flag=[], [], [root], 1

while stack:

for i in xrange(len(stack)):

node=stack.pop(0)

temp+=[node.val]

if node.left: stack+=[node.left]

if node.right: stack+=[node.right]

res+=[temp[::flag]]

temp=[]

flag\*=-1

return res

# 104,Maximum Depth of Binary Tree\*\*\*:

**Python\_solution:**

1 line Ruby and Python

Just a bit shorter/different than previous solutions.

Ruby:

def max\_depth(root)

root ? 1 + [max\_depth(root.left), max\_depth(root.right)].max : 0

end

Python:

def maxDepth(self, root):

return 1 + max(map(self.maxDepth, (root.left, root.right))) if root else 0

# 105,Construct Binary Tree from Preorder and Inorder Traversal\*\*\*:

**Python\_solution:**

Python short recursive solution.

def buildTree(self, preorder, inorder):

if inorder:

ind = inorder.index(preorder.pop(0))

root = TreeNode(inorder[ind])

root.left = self.buildTree(preorder, inorder[0:ind])

root.right = self.buildTree(preorder, inorder[ind+1:])

return root

# 106,Construct Binary Tree from Inorder and Postorder Traversal\*\*\*:

**Python\_solution:**

A Python recursive solution

# Definition for a binary tree node

# class TreeNode:

# def \_\_init\_\_(self, x):

# self.val = x

# self.left = None

# self.right = None

class Solution:

# @param inorder, a list of integers

# @param postorder, a list of integers

# @return a tree node

# 12:00

def buildTree(self, inorder, postorder):

if not inorder or not postorder:

return None

root = TreeNode(postorder.pop())

inorderIndex = inorder.index(root.val)

root.right = self.buildTree(inorder[inorderIndex+1:], postorder)

root.left = self.buildTree(inorder[:inorderIndex], postorder)

return root

# 107,Binary Tree Level Order Traversal II\*\*\*:

**Python\_solution:**

Python solutions (dfs recursively, dfs+stack, bfs+queue).

# dfs recursively

def levelOrderBottom1(self, root):

res = []

self.dfs(root, 0, res)

return res

def dfs(self, root, level, res):

if root:

if len(res) < level + 1:

res.insert(0, [])

res[-(level+1)].append(root.val)

self.dfs(root.left, level+1, res)

self.dfs(root.right, level+1, res)

# dfs + stack

def levelOrderBottom2(self, root):

stack = [(root, 0)]

res = []

while stack:

node, level = stack.pop()

if node:

if len(res) < level+1:

res.insert(0, [])

res[-(level+1)].append(node.val)

stack.append((node.right, level+1))

stack.append((node.left, level+1))

return res

# bfs + queue

def levelOrderBottom(self, root):

queue, res = collections.deque([(root, 0)]), []

while queue:

node, level = queue.popleft()

if node:

if len(res) < level+1:

res.insert(0, [])

res[-(level+1)].append(node.val)

queue.append((node.left, level+1))

queue.append((node.right, level+1))

return res

# 108,Convert Sorted Array to Binary Search Tree\*\*\*:

**Python\_solution:**

An easy Python solution

The idea is to find the root first, then recursively build each left and right subtree

# Definition for a binary tree node

# class TreeNode:

# def \_\_init\_\_(self, x):

# self.val = x

# self.left = None

# self.right = None

class Solution:

# @param num, a list of integers

# @return a tree node

# 12:37

def sortedArrayToBST(self, num):

if not num:

return None

mid = len(num) // 2

root = TreeNode(num[mid])

root.left = self.sortedArrayToBST(num[:mid])

root.right = self.sortedArrayToBST(num[mid+1:])

return root

# 109,Convert Sorted List to Binary Search Tree\*\*\*:

**Python\_solution:**

Python recursive solution with detailed comments (operate linked-list directly).

# recursively

def sortedListToBST(self, head):

if not head:

return

if not head.next:

return TreeNode(head.val)

# here we get the middle point,

# even case, like '1234', slow points to '2',

# '3' is root, '12' belongs to left, '4' is right

# odd case, like '12345', slow points to '2', '12'

# belongs to left, '3' is root, '45' belongs to right

slow, fast = head, head.next.next

while fast and fast.next:

fast = fast.next.next

slow = slow.next

# tmp points to root

tmp = slow.next

# cut down the left child

slow.next = None

root = TreeNode(tmp.val)

root.left = self.sortedListToBST(head)

root.right = self.sortedListToBST(tmp.next)

return root

# 110,Balanced Binary Tree\*\*\*:

**Python\_solution:**

VERY SIMPLE Python solution, beats 90%

class Solution(object):

def isBalanced(self, root):

def check(root):

if root is None:

return 0

left = check(root.left)

right = check(root.right)

if left == -1 or right == -1 or abs(left - right) > 1:

return -1

return 1 + max(left, right)

return check(root) != -1

# 111,Minimum Depth of Binary Tree\*\*\*:

**Python\_solution:**

My solution in python

The idea is to use recursion, the accepted short python code looks like follows:

class Solution:

# @param root, a tree node

# @return an integer

def minDepth(self, root):

if root == None:

return 0

if root.left==None or root.right==None:

return self.minDepth(root.left)+self.minDepth(root.right)+1

return min(self.minDepth(root.right),self.minDepth(root.left))+1

# 112,Path Sum\*\*\*:

**Python\_solution:**

Short Python recursive solution - O(n)

# Definition for a binary tree node

# class TreeNode:

# def \_\_init\_\_(self, x):

# self.val = x

# self.left = None

# self.right = None

class Solution:

# @param root, a tree node

# @param sum, an integer

# @return a boolean

# 1:27

def hasPathSum(self, root, sum):

if not root:

return False

if not root.left and not root.right and root.val == sum:

return True

sum -= root.val

return self.hasPathSum(root.left, sum) or self.hasPathSum(root.right, sum)

# 113,Path Sum II\*\*\*:

**Python\_solution:**

Short python solution

class Solution:

def pathSum(self, root, sum):

if not root: return []

if root.left == None and root.right == None:

if sum == root.val:

return [[root.val]]

else:

return []

a = self.pathSum(root.left, sum - root.val) + \

self.pathSum(root.right, sum - root.val)

return [[root.val] + i for i in a]

# 114,Flatten Binary Tree to Linked List\*\*\*:

**Python\_solution:**

An inorder python solution

class Solution:

# @param root, a tree node

# @return nothing, do it in place

prev = None

def flatten(self, root):

if not root:

return

self.prev = root

self.flatten(root.left)

temp = root.right

root.right, root.left = root.left, None

self.prev.right = temp

self.flatten(temp)

\*

/

n

/ \

left right

\

\*

\*

\

p

The idea is very simple. Suppose n is the current visiting node, and p is the previous node of preorder traversal to n.right.

We just need to do the inorder replacement:

n.left -> NULL

n.right - > n.left

p->right -> n.right

# 115,Distinct Subsequences\*\*\*:

**Python\_solution:**

Python dp solutions (O(m\*n), O(n) space).

# O(m\*n) space

def numDistinct1(self, s, t):

l1, l2 = len(s)+1, len(t)+1

dp = [[1] \* l2 for \_ in xrange(l1)]

for j in xrange(1, l2):

dp[0][j] = 0

for i in xrange(1, l1):

for j in xrange(1, l2):

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

return dp[-1][-1]

# O(n) space

def numDistinct(self, s, t):

l1, l2 = len(s)+1, len(t)+1

cur = [0] \* l2

cur[0] = 1

for i in xrange(1, l1):

pre = cur[:]

for j in xrange(1, l2):

cur[j] = pre[j] + pre[j-1]\*(s[i-1] == t[j-1])

return cur[-1]

# 116,Populating Next Right Pointers in Each Node\*\*\*:

**Python\_solution:**

Python accepted code

def connect(self, root):

if not root: return

while root.left:

cur = root.left

prev = None

while root:

if prev: prev.next = root.left

root.left.next = root.right

prev = root.right

root = root.next

root = cur

# 117,Populating Next Right Pointers in Each Node II\*\*\*:

**Python\_solution:**

AC Python O(1) space solution 12 lines and easy to understand

The algorithm is a BFS or level order traversal. We go through the tree level by level. node is the pointer in the parent level, tail is the tail pointer in the child level.

The parent level can be view as a singly linked list or queue, which we can traversal easily with a pointer.

Connect the tail with every one of the possible nodes in child level, update it only if the connected node is not nil.

Do this one level by one level. The whole thing is quite straightforward.

\*\*Python\*\*

def connect(self, node):

tail = dummy = TreeLinkNode(0)

while node:

tail.next = node.left

if tail.next:

tail = tail.next

tail.next = node.right

if tail.next:

tail = tail.next

node = node.next

if not node:

tail = dummy

node = dummy.next

# 61 / 61 test cases passed.

# Status: Accepted

# Runtime: 100 ms

# 95.26%

# 118,Pascal's Triangle\*\*\*:

**Python\_solution:**

Python 4 lines short solution using map.

def generate(self, numRows):

res = [[1]]

for i in range(1, numRows):

res += [map(lambda x, y: x+y, res[-1] + [0], [0] + res[-1])]

return res[:numRows]

explanation: Any row can be constructed using the offset sum of the previous row. Example:

1 3 3 1 0

+ 0 1 3 3 1

= 1 4 6 4 1

# 119,Pascal's Triangle II\*\*\*:

**Python\_solution:**

Very simple Python solution

class Solution:

# @param {integer} rowIndex

# @return {integer[]}

def getRow(self, rowIndex):

row = [1]

for i in range(1, rowIndex+1):

row = list(map(lambda x,y: x+y, [0]+row, row + [0]))

return row

# 120,Triangle\*\*\*:

**Python\_solution:**

Python easy to understand solutions (top-down, bottom-up).

# O(n\*n/2) space, top-down

def minimumTotal1(self, triangle):

if not triangle:

return

res = [[0 for i in xrange(len(row))] for row in triangle]

res[0][0] = triangle[0][0]

for i in xrange(1, len(triangle)):

for j in xrange(len(triangle[i])):

if j == 0:

res[i][j] = res[i-1][j] + triangle[i][j]

elif j == len(triangle[i])-1:

res[i][j] = res[i-1][j-1] + triangle[i][j]

else:

res[i][j] = min(res[i-1][j-1], res[i-1][j]) + triangle[i][j]

return min(res[-1])

# Modify the original triangle, top-down

def minimumTotal2(self, triangle):

if not triangle:

return

for i in xrange(1, len(triangle)):

for j in xrange(len(triangle[i])):

if j == 0:

triangle[i][j] += triangle[i-1][j]

elif j == len(triangle[i])-1:

triangle[i][j] += triangle[i-1][j-1]

else:

triangle[i][j] += min(triangle[i-1][j-1], triangle[i-1][j])

return min(triangle[-1])

# Modify the original triangle, bottom-up

def minimumTotal3(self, triangle):

if not triangle:

return

for i in xrange(len(triangle)-2, -1, -1):

for j in xrange(len(triangle[i])):

triangle[i][j] += min(triangle[i+1][j], triangle[i+1][j+1])

return triangle[0][0]

# bottom-up, O(n) space

def minimumTotal(self, triangle):

if not triangle:

return

res = triangle[-1]

for i in xrange(len(triangle)-2, -1, -1):

for j in xrange(len(triangle[i])):

res[j] = min(res[j], res[j+1]) + triangle[i][j]

return res[0]

# 121,Best Time to Buy and Sell Stock\*\*\*:

**Python\_solution:**

Easy O(n) Python solution

def maxProfit(prices):

max\_profit, min\_price = 0, float('inf')

for price in prices:

min\_price = min(min\_price, price)

profit = price - min\_price

max\_profit = max(max\_profit, profit)

return max\_profit

# 122,Best Time to Buy and Sell Stock II\*\*\*:

**Python\_solution:**

Clear 1-line Python Solution

Basically, if tomorrow's price is higher than today's, we buy it today and sell tomorrow. Otherwise, we don't. Here is the code:

class Solution(object):

def maxProfit(self, prices):

return sum(max(prices[i + 1] - prices[i], 0) for i in range(len(prices) - 1))

# 123,Best Time to Buy and Sell Stock III\*\*\*:

**Best\_solution:**

Is it Best Solution with O(n), O(1).

The thinking is simple and is inspired by the best solution from Single Number II (I read through the discussion after I use DP).

Assume we only have 0 money at first;

4 Variables to maintain some interested 'ceilings' so far:

The maximum of if we've just buy 1st stock, if we've just sold 1nd stock, if we've just buy 2nd stock, if we've just sold 2nd stock.

Very simple code too and work well. I have to say the logic is simple than those in Single Number II.

public class Solution {

public int maxProfit(int[] prices) {

int hold1 = Integer.MIN\_VALUE, hold2 = Integer.MIN\_VALUE;

int release1 = 0, release2 = 0;

for(int i:prices){ // Assume we only have 0 money at first

release2 = Math.max(release2, hold2+i); // The maximum if we've just sold 2nd stock so far.

hold2 = Math.max(hold2, release1-i); // The maximum if we've just buy 2nd stock so far.

release1 = Math.max(release1, hold1+i); // The maximum if we've just sold 1nd stock so far.

hold1 = Math.max(hold1, -i); // The maximum if we've just buy 1st stock so far.

}

return release2; ///Since release1 is initiated as 0, so release2 will always higher than release1.

}

}

# 124,Binary Tree Maximum Path Sum\*\*\*:

**Python\_solution:**

12 lines of Python code, fast and easy to understand

class Solution(object):

def maxPathSum(self, root):

def dfs(node): # returns: max one side path sum, max path sum

l = r = 0

ls = rs = None

if node.left:

l, ls = dfs(node.left)

l = max(l, 0)

if node.right:

r, rs = dfs(node.right)

r = max(r, 0)

return node.val + max(l, r), max(node.val + l + r, ls, rs)

if root:

return dfs(root)[1]

return 0

# 125,Valid Palindrome\*\*\*:

**Python\_solution:**

Python in-place two-pointer solution.

def isPalindrome(self, s):

l, r = 0, len(s)-1

while l < r:

while l < r and not s[l].isalnum():

l += 1

while l <r and not s[r].isalnum():

r -= 1

if s[l].lower() != s[r].lower():

return False

l +=1; r -= 1

return True

# 126,Word Ladder II\*\*\*:

**Python\_solution:**

Use defaultdict for traceback and easy writing, 20 lines python code

class Solution:

# @param start, a string

# @param end, a string

# @param dict, a set of string

# @return a list of lists of string

def findLadders(self, start, end, dic):

dic.add(end)

level = {start}

parents = collections.defaultdict(set)

while level and end not in parents:

next\_level = collections.defaultdict(set)

for node in level:

for char in string.ascii\_lowercase:

for i in range(len(start)):

n = node[:i]+char+node[i+1:]

if n in dic and n not in parents:

next\_level[n].add(node)

level = next\_level

parents.update(next\_level)

res = [[end]]

while res and res[0][0] != start:

res = [[p]+r for r in res for p in parents[r[0]]]

return res

Every level we use the defaultdict to get rid of the duplicates

# 127,Word Ladder\*\*\*:

**Python\_solution:**

Share my two Python solutions: a very concise one (12 lines, ~160ms) and an optimized solution(~100ms)

The idea behind the first solution is to use character flopping plus bidirectional BFS. Use set operations as much as possible.

class Solution:

# @param {string} beginWord

# @param {string} endWord

# @param {set<string>} wordDict

# @return {integer}

def ladderLength(self, beginWord, endWord, wordDict):

length = 2

front, back = set([beginWord]), set([endWord])

wordDict.discard(beginWord)

while front:

# generate all valid transformations

front = wordDict & (set(word[:index] + ch + word[index+1:] for word in front

for index in range(len(beginWord)) for ch in 'abcdefghijklmnopqrstuvwxyz'))

if front & back:

# there are common elements in front and back, done

return length

length += 1

if len(front) > len(back):

# swap front and back for better performance (fewer choices in generating nextSet)

front, back = back, front

# remove transformations from wordDict to avoid cycle

wordDict -= front

return 0

The optimizations:

-- Generating next set

An alternative is to immediately add a candidate to next set if it is in the dictionary.

Another way to generate next set: for each word in the current set, check if a word in the dictionary can be transformed to it. If it can, add it to the next set. The time complexity of the two methods is analyzed below, assuming word length is L, size of current set and dictionary are M and N, respectively.

a. character flopping:

Loop over current set, character of word, alphabet, the flopping itself is O(L). Time complexity is O(26ML^2)

b. verify transformation:

Loop over current set, dictionary, character of word. Time complexity is O(MNL)

For b) to be faster, the switching point is N = 26L. This scale can be adjusted.

Since the size of dictionary shrinks during the process, it is beneficial to switch to b) in the late stage, or use it for a small dictionary.

-- Removing current word set from dictionary

It seems natural to use difference\_update for this job since size of dictionary is bigger than that of current word set. But is it so? Note that here we are sure that every word in current set does exist in the dictionary.

a. S.difference\_update(T) or S -= T

For every key (entry) in T, if it is in S, remove it from T. There are len(T) removes and len(T) peeks.

b. S.difference(T) or S - T

Create a new empty set. For every key (entry) in S, if it is not in T, add it to new set. There are len(S)-len(T) adds and len(S) peeks.

If the sizes of current word set and dictionary are close, using difference\_update means we will remove almost everything from dictionary. If we use difference, only a handful of adds. I use size of dictionary is twice that of current word set as the switching point. This threshold can be adjusted too.

The following optimized code takes ~100ms.

class Solution:

# @param {string} beginWord

# @param {string} endWord

# @param {set<string>} wordDict

# @return {integer}

def ladderLength(self, beginWord, endWord, wordDict):

def generateNextSet1(current, wordDict, wordLen):

nextSet = set()

for word in current:

for index in range(wordLen):

for ch in 'abcdefghijklmnopqrstuvwxyz':

nextWord = word[:index] + ch + word[index+1:]

if nextWord in wordDict:

nextSet.add(nextWord)

return nextSet

def generateNextSet2(current, wordDict):

nextSet = set()

for word in current:

for nextWord in wordDict:

index = 0

try:

while word[index] == nextWord[index]:

index += 1

if word[index+1:] == nextWord[index+1:]:

nextSet.add(nextWord)

except:

continue

return nextSet

steps, wordLen = 2, len(beginWord)

front, back = set([beginWord]), set([endWord])

wordDict.discard(beginWord)

switchThreshold = 26\*wordLen

while front:

# get all valid transformations

if len(wordDict) >= switchThreshold:

front = generateNextSet1(front, wordDict, wordLen)

else:

front = generateNextSet2(front, wordDict)

if front & back:

# there are common elements in front and back, done

return steps

steps += 1

if len(front) >= len(back):

# swap front and back for better performance (smaller nextSet)

front, back = back, front

# remove transformations from wordDict to avoid cycles

if (len(wordDict)>>1) >= len(front):

# s.difference\_update(t): O(len(t))

wordDict -= front

else:

# s.difference(t): O(len(s))

wordDict = wordDict - front

return 0

# 128,Longest Consecutive Sequence\*\*\*:

**Python\_solution:**

Python O(n) solution using sets

class Solution:

# @param num, a list of integer

# @return an integer

def longestConsecutive(self, num):

num=set(num)

maxLen=0

while num:

n=num.pop()

i=n+1

l1=0

l2=0

while i in num:

num.remove(i)

i+=1

l1+=1

i=n-1

while i in num:

num.remove(i)

i-=1

l2+=1

maxLen=max(maxLen,l1+l2+1)

return maxLen

# 129,Sum Root to Leaf Numbers\*\*\*:

**Python\_solution:**

Python solutions (dfs+stack, bfs+queue, dfs recursively).

# dfs + stack

def sumNumbers1(self, root):

if not root:

return 0

stack, res = [(root, root.val)], 0

while stack:

node, value = stack.pop()

if node:

if not node.left and not node.right:

res += value

if node.right:

stack.append((node.right, value\*10+node.right.val))

if node.left:

stack.append((node.left, value\*10+node.left.val))

return res

# bfs + queue

def sumNumbers2(self, root):

if not root:

return 0

queue, res = collections.deque([(root, root.val)]), 0

while queue:

node, value = queue.popleft()

if node:

if not node.left and not node.right:

res += value

if node.left:

queue.append((node.left, value\*10+node.left.val))

if node.right:

queue.append((node.right, value\*10+node.right.val))

return res

# recursively

def sumNumbers(self, root):

self.res = 0

self.dfs(root, 0)

return self.res

def dfs(self, root, value):

if root:

#if not root.left and not root.right:

# self.res += value\*10 + root.val

self.dfs(root.left, value\*10+root.val)

#if not root.left and not root.right:

# self.res += value\*10 + root.val

self.dfs(root.right, value\*10+root.val)

if not root.left and not root.right:

self.res += value\*10 + root.val

# 130,Surrounded Regions\*\*\*:

**Python\_solution:**

9 lines, Python 148 ms

Phase 1: "Save" every O-region touching the border, changing its cells to 'S'.

Phase 2: Change every 'S' on the board to 'O' and everything else to 'X'.

def solve(self, board):

if not any(board): return

m, n = len(board), len(board[0])

save = [ij for k in range(m+n) for ij in ((0, k), (m-1, k), (k, 0), (k, n-1))]

while save:

i, j = save.pop()

if 0 <= i < m and 0 <= j < n and board[i][j] == 'O':

board[i][j] = 'S'

save += (i, j-1), (i, j+1), (i-1, j), (i+1, j)

board[:] = [['XO'[c == 'S'] for c in row] for row in board]

In case you don't like my last line, you could do this instead:

for row in board:

for i, c in enumerate(row):

row[i] = 'XO'[c == 'S']

# 131,Palindrome Partitioning\*\*\*:

**Python\_solution:**

1-liner Python, Ruby

Python:

Broken into several physical lines for readability, but still one logical line and just one simple statement.

def partition(self, s):

return [[s[:i]] + rest

for i in xrange(1, len(s)+1)

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

for rest in self.partition(s[i:])] or [[]]

Ruby:

def partition(s)

s == '' ? [[]] : s.size.times.flat\_map { |i| s[0..i] != s[0..i].reverse ? [] :

partition(s[i+1..-1]).map { |rest| [s[0..i]] + rest }

}

end

# 132,Palindrome Partitioning II\*\*\*:

**Python\_solution:**

56 ms python with explanation

Algorithm (460 ms) credits go to:

[https://leetcode.com/discuss/9476/solution-does-not-need-table-palindrome-right-uses-only-space][1]

The main algorithm idea is if s[i,j] is a palindrome, then the minCut(s[:j]) is \*\*at most\*\* minCut(s[:i-1])+1. This literally needs to find out all possible palindromes in the list. The above post provides an efficient search algorithm. O(n) space and O(n^2) time complexity.

Further acceleration (460 ms -> 56 ms) credits go to:

[https://leetcode.com/discuss/43950/python-100ms-extra-dealing-super-cases-reduces-576ms-100ms][2]

The main idea for acceleration is to quickly check and exclude a few long palindrome tests..

def minCut(self, s):

# acceleration

if s == s[::-1]: return 0

for i in range(1, len(s)):

if s[:i] == s[:i][::-1] and s[i:] == s[i:][::-1]:

return 1

# algorithm

cut = [x for x in range(-1,len(s))] # cut numbers in worst case (no palindrome)

for i in range(len(s)):

r1, r2 = 0, 0

# use i as origin, and gradually enlarge radius if a palindrome exists

# odd palindrome

while i-r1 >= 0 and i+r1 < len(s) and s[i-r1] == s[i+r1]:

cut[i+r1+1] = min(cut[i+r1+1], cut[i-r1]+1)

r1 += 1

# even palindrome

while i-r2 >= 0 and i+r2+1 < len(s) and s[i-r2] == s[i+r2+1]:

cut[i+r2+2] = min(cut[i+r2+2], cut[i-r2]+1)

r2 += 1

return cut[-1]

The following code simply implements the algorithm without any optimization (1800 ms), and should be easier to understand. O(n) space and O(n^3) time complexity.

def minCut(self, s):

cut = [x for x in range(-1,len(s))]

for i in range(0,len(s)):

for j in range(i,len(s)):

if s[i:j] == s[j:i:-1]:

cut[j+1] = min(cut[j+1],cut[i]+1)

return cut[-1]

[1]: https://leetcode.com/discuss/9476/solution-does-not-need-table-palindrome-right-uses-only-space

[2]: https://leetcode.com/discuss/43950/python-100ms-extra-dealing-super-cases-reduces-576ms-100ms

# 133,Clone Graph\*\*\*:

**Python\_solution:**

Python DFS short solution

Use a dictionary to store the UndirectedGraphNode

def cloneGraph(self, node):

if not node:

return node

root = UndirectedGraphNode(node.label)

stack = [node]

visit = {}

visit[node.label] = root

while stack:

top = stack.pop()

for n in top.neighbors:

if n.label not in visit:

stack.append(n)

visit[n.label] = UndirectedGraphNode(n.label)

visit[top.label].neighbors.append(visit[n.label])

return root

# 134,Gas Station\*\*\*:

**Python\_solution:**

Possibly the MOST easiest approach, O(N), one variable, Python

def canCompleteCircuit(self, gas, cost):

"""

:type gas: List[int]

:type cost: List[int]

:rtype: int

"""

if len(gas) == 0 or len(cost) == 0 or sum(gas) < sum(cost):

return -1

position = 0

balance = 0 # current tank balance

for i in range(len(gas)):

balance += gas[i] - cost[i] # update balance

if balance < 0: # balance drops to negative, reset the start position

balance = 0

position = i+1

return position

# 135,Candy\*\*\*:

**Python\_solution:**

Simple python solution with two passes

class Solution:

# @param {integer[]} ratings

# @return {integer}

def candy(self, ratings):

# use two pass scan from left to right and vice versa to keep the candy level up to now

# similar to like the Trapping Rain Water question

res = [1]\*len(ratings) # also compatable with [] input

lbase = rbase = 1

# left scan

for i in xrange(1, len(ratings)):

lbase = lbase + 1 if ratings[i] > ratings[i-1] else 1

res[i] = lbase

# right scan

for i in xrange(len(ratings)-2, -1, -1):

rbase = rbase + 1 if ratings[i] > ratings[i+1] else 1

res[i] = max(rbase, res[i])

return sum(res)

# 136,Single Number\*\*\*:

**Python\_solution:**

Python different solutions.

def singleNumber1(self, nums):

dic = {}

for num in nums:

dic[num] = dic.get(num, 0)+1

for key, val in dic.items():

if val == 1:

return key

def singleNumber2(self, nums):

res = 0

for num in nums:

res ^= num

return res

def singleNumber3(self, nums):

return 2\*sum(set(nums))-sum(nums)

def singleNumber4(self, nums):

return reduce(lambda x, y: x ^ y, nums)

def singleNumber(self, nums):

return reduce(operator.xor, nums)

# 137,Single Number II\*\*\*:

**Python\_solution:**

Python bitwise solution

class Solution:

# @param A, a list of integer

# @return an integer

def singleNumber(self, A):

ans = 0

for i in xrange(0,32):

count = 0

for a in A:

if ((a >> i) & 1):

count+=1

ans |= ((count%3) << i)

return self.convert(ans)

def convert(self,x):

if x >= 2\*\*31:

x -= 2\*\*32

return x

# 138,Copy List with Random Pointer\*\*\*:

**Python\_solution:**

Clear and short python O(2n) and O(n) solution

class Solution:

# @param head, a RandomListNode

# @return a RandomListNode

def copyRandomList(self, head):

dic = dict()

m = n = head

while m:

dic[m] = RandomListNode(m.label)

m = m.next

while n:

dic[n].next = dic.get(n.next)

dic[n].random = dic.get(n.random)

n = n.next

return dic.get(head)

O(n)

class Solution:

# @param head, a RandomListNode

# @return a RandomListNode

def copyRandomList(self, head):

dic = collections.defaultdict(lambda: RandomListNode(0))

dic[None] = None

n = head

while n:

dic[n].label = n.label

dic[n].next = dic[n.next]

dic[n].random = dic[n.random]

n = n.next

return dic[head]

# 139,Word Break\*\*\*:

**Python\_solution:**

Simple DP solution in Python with description

The idea is the following:

- d is an array that contains booleans

- d[i] is True if there is a word in the dictionary that \*ends\* at ith index of s AND d is also True at the beginning of the word

Example:

- s = "leetcode"

- words = ["leet", "code"]

- d[3] is True because there is "leet" in the dictionary that ends at 3rd index of "leetcode"

- d[7] is True because there is "code" in the dictionary that ends at the 7th index of "leetcode" AND d[3] is True

The result is the last index of d.

def word\_break(s, words):

d = [False] \* len(s)

for i in range(len(s)):

for w in words:

if w == s[i-len(w)+1:i+1] and (d[i-len(w)] or i-len(w) == -1):

d[i] = True

return d[-1]

# 140,Word Break II\*\*\*:

**Python\_solution:**

9 lines Python, 10 lines C++

\*\*Python:\*\*

def wordBreak(self, s, wordDict):

memo = {len(s): ['']}

def sentences(i):

if i not in memo:

memo[i] = [s[i:j] + (tail and ' ' + tail)

for j in range(i+1, len(s)+1)

if s[i:j] in wordDict

for tail in sentences(j)]

return memo[i]

return sentences(0)

\*\*C++:\*\*

vector<string> wordBreak(string s, unordered\_set<string>& wordDict) {

unordered\_map<int, vector<string>> memo {{s.size(), {""}}};

function<vector<string>(int)> sentences = [&](int i) {

if (!memo.count(i))

for (int j=i+1; j<=s.size(); j++)

if (wordDict.count(s.substr(i, j-i)))

for (string tail : sentences(j))

memo[i].push\_back(s.substr(i, j-i) + (tail=="" ? "" : ' ' + tail));

return memo[i];

};

return sentences(0);

}

# 141,Linked List Cycle\*\*\*:

**Python\_solution:**

Except-ionally fast Python

Took 88 ms and the "Accepted Solutions Runtime Distribution" doesn't show any faster Python submissions. The "trick" is to not check all the time whether we have reached the end but to handle it via an exception. ["Easier to ask for forgiveness than permission."](https://docs.python.org/3/glossary.html#term-eafp)

The algorithm is of course [Tortoise and hare](https://en.wikipedia.org/wiki/Cycle\_detection#Tortoise\_and\_hare).

def hasCycle(self, head):

try:

slow = head

fast = head.next

while slow is not fast:

slow = slow.next

fast = fast.next.next

return True

except:

return False

# 142,Linked List Cycle II\*\*\*:

**Python\_solution:**

Share my python solution with detailed explanation

My solution consists of two parts. The first one checks if a cycle exists or not. The second one determines the entry of the cycle if it exists.

The first part is inspired by [this post][1]. about Linked List Cycle I

The logic behind the 2nd part is like this:

Consider the following linked list, where E is the cylce entry and X, the crossing point of fast and slow.

H: distance from head to cycle entry E

D: distance from E to X

L: cycle length

\_\_\_\_\_

/ \

head\_\_\_\_\_H\_\_\_\_\_\_E \

\ /

X\_\_\_\_\_/

If fast and slow both start at head, when fast catches slow, slow has traveled H+D and fast 2(H+D).

Assume fast has traveled n loops in the cycle, we have:

2H + 2D = H + D + L --> H + D = nL --> H = nL - D

Thus if two pointers start from head and X, respectively, one first reaches E, the other also reaches E.

In my solution, since fast starts at head.next, we need to move slow one step forward in the beginning of part 2

class Solution:

# @param head, a ListNode

# @return a list node

def detectCycle(self, head):

try:

fast = head.next

slow = head

while fast is not slow:

fast = fast.next.next

slow = slow.next

except:

# if there is an exception, we reach the end and there is no cycle

return None

# since fast starts at head.next, we need to move slow one step forward

slow = slow.next

while head is not slow:

head = head.next

slow = slow.next

return head

[1]: https://leetcode.com/discuss/40120/except-ionally-fast-python

# 143,Reorder List\*\*\*:

**Python\_solution:**

A python solution O(n) time, O(1) space

# Splits in place a list in two halves, the first half is >= in size than the second.

# @return A tuple containing the heads of the two halves

def \_splitList(head):

fast = head

slow = head

while fast and fast.next:

slow = slow.next

fast = fast.next

fast = fast.next

middle = slow.next

slow.next = None

return head, middle

# Reverses in place a list.

# @return Returns the head of the new reversed list

def \_reverseList(head):

last = None

currentNode = head

while currentNode:

nextNode = currentNode.next

currentNode.next = last

last = currentNode

currentNode = nextNode

return last

# Merges in place two lists

# @return The newly merged list.

def \_mergeLists(a, b):

tail = a

head = a

a = a.next

while b:

tail.next = b

tail = tail.next

b = b.next

if a:

a, b = b, a

return head

class Solution:

# @param head, a ListNode

# @return nothing

def reorderList(self, head):

if not head or not head.next:

return

a, b = \_splitList(head)

b = \_reverseList(b)

head = \_mergeLists(a, b)

# 144,Binary Tree Preorder Traversal\*\*\*:

**Python\_solution:**

Very simple iterative Python solution

Classical usage of stack's LIFO feature, very easy to grasp:

def preorderTraversal(self, root):

ret = []

stack = [root]

while stack:

node = stack.pop()

if node:

ret.append(node.val)

stack.append(node.right)

stack.append(node.left)

return ret

# 145,Binary Tree Postorder Traversal\*\*\*:

**Python\_solution:**

Share my two Python iterative solutions, post-order and modified preorder then reverse

The first is by postorder using a flag to indicate whether the node has been visited or not.

class Solution:

# @param {TreeNode} root

# @return {integer[]}

def postorderTraversal(self, root):

traversal, stack = [], [(root, False)]

while stack:

node, visited = stack.pop()

if node:

if visited:

# add to result if visited

traversal.append(node.val)

else:

# post-order

stack.append((node, True))

stack.append((node.right, False))

stack.append((node.left, False))

return traversal

The 2nd uses modified preorder (right subtree first). Then reverse the result.

class Solution:

# @param {TreeNode} root

# @return {integer[]}

def postorderTraversal(self, root):

traversal, stack = [], [root]

while stack:

node = stack.pop()

if node:

# pre-order, right first

traversal.append(node.val)

stack.append(node.left)

stack.append(node.right)

# reverse result

return traversal[::-1]

# 146,LRU Cache\*\*\*:

**Python\_solution:**

Python Dict + Double LinkedList

class Node:

def \_\_init\_\_(self, k, v):

self.key = k

self.val = v

self.prev = None

self.next = None

class LRUCache:

def \_\_init\_\_(self, capacity):

self.capacity = capacity

self.dic = dict()

self.head = Node(0, 0)

self.tail = Node(0, 0)

self.head.next = self.tail

self.tail.prev = self.head

def get(self, key):

if key in self.dic:

n = self.dic[key]

self.\_remove(n)

self.\_add(n)

return n.val

return -1

def set(self, key, value):

if key in self.dic:

self.\_remove(self.dic[key])

n = Node(key, value)

self.\_add(n)

self.dic[key] = n

if len(self.dic) > self.capacity:

n = self.head.next

self.\_remove(n)

del self.dic[n.key]

def \_remove(self, node):

p = node.prev

n = node.next

p.next = n

n.prev = p

def \_add(self, node):

p = self.tail.prev

p.next = node

self.tail.prev = node

node.prev = p

node.next = self.tail

# 147,Insertion Sort List\*\*\*:

**Python\_solution:**

Python time limit is too tight

I have basically the same code in python and java (see below). python got TLE, but java was accepted. I propose to relax the python time limit a little bit.

\*\*Python\*\*

class Solution:

# @param head, a ListNode

# @return a ListNode

def insertionSortList(self, head):

srt = None

while head:

node = head

head = head.next

node.next = None

srt = self.insertTo(srt, node)

return srt

def insertTo(self, head, node):

node.next = head

head = node

while node.next and node.val > node.next.val:

node.val, node.next.val = node.next.val, node.val

node = node.next

return head

\*\*java\*\*

public class Solution {

public ListNode insertionSortList(ListNode head) {

ListNode srt = null;

while (head != null) {

ListNode node = head;

head = head.next;

node.next = null;

srt = insertTo(srt, node);

}

return srt;

}

public ListNode insertTo(ListNode head, ListNode node) {

node.next = head;

head = node;

while (node.next != null && node.val > node.next.val) {

node.val = node.val ^ node.next.val;

node.next.val = node.val ^ node.next.val;

node.val = node.val ^ node.next.val;

node = node.next;

}

return head;

}

}

# 148,Sort List\*\*\*:

**Python\_solution:**

Clean python code

class Solution(object):

def merge(self, h1, h2):

dummy = tail = ListNode(None)

while h1 and h2:

if h1.val < h2.val:

tail.next, tail, h1 = h1, h1, h1.next

else:

tail.next, tail, h2 = h2, h2, h2.next

tail.next = h1 or h2

return dummy.next

def sortList(self, head):

if not head or not head.next:

return head

pre, slow, fast = None, head, head

while fast and fast.next:

pre, slow, fast = slow, slow.next, fast.next.next

pre.next = None

return self.merge(\*map(self.sortList, (head, slow)))

# 149,Max Points on a Line\*\*\*:

**Python\_solution:**

Python 68 ms code

def maxPoints(self, points):

l = len(points)

m = 0

for i in range(l):

dic = {'i': 1}

same = 0

for j in range(i+1, l):

tx, ty = points[j].x, points[j].y

if tx == points[i].x and ty == points[i].y:

same += 1

continue

if points[i].x == tx: slope = 'i'

else:slope = (points[i].y-ty) \* 1.0 /(points[i].x-tx)

if slope not in dic: dic[slope] = 1

dic[slope] += 1

m = max(m, max(dic.values()) + same)

return m

# 150,Evaluate Reverse Polish Notation\*\*\*:

**Python\_solution:**

Python solution with comments (don't use eval() function).

def evalRPN(self, tokens):

stack = []

for t in tokens:

if t not in ["+", "-", "\*", "/"]:

stack.append(int(t))

else:

r, l = stack.pop(), stack.pop()

if t == "+":

stack.append(l+r)

elif t == "-":

stack.append(l-r)

elif t == "\*":

stack.append(l\*r)

else:

# here take care of the case like "1/-22",

# in Python 2.x, it returns -1, while in

# Leetcode it should return 0

if l\*r < 0 and l % r != 0:

stack.append(l/r+1)

else:

stack.append(l/r)

return stack.pop()

# 151,Reverse Words in a String\*\*\*:

**Python\_solution:**

My Accept Answer of Python with one line

My Python code using the function of array and string. Both time and memory is O(n).

class Solution:

# @param s, a string

# @return a string

def reverseWords(self, s):

return " ".join(s.strip().split()[::-1])

# 152,Maximum Product Subarray\*\*\*:

**Python\_solution:**

In Python, can it be more concise?

def maxProduct(nums):

maximum=big=small=nums[0]

for n in nums[1:]:

big, small=max(n, n\*big, n\*small), min(n, n\*big, n\*small)

maximum=max(maximum, big)

return maximum

# 153,Find Minimum in Rotated Sorted Array\*\*\*:

**Python\_solution:**

9-line python clean code

Just use binary search

class Solution(object):

def findMin(self, nums):

"""

:type nums: List[int]

:rtype: int

"""

i = 0

j = len(nums) - 1

while i < j:

m = i + (j - i) / 2

if nums[m] > nums[j]:

i = m + 1

else:

j = m

return nums[i]

# 154,Find Minimum in Rotated Sorted Array II\*\*\*:

**Python\_solution:**

Python solution. Worst case O(N)

def findMin(self, nums):

beg = 0

end = len(nums)-1

while beg <= end:

while beg < end and nums[beg] == nums[beg + 1]:

beg += 1

while end > beg and nums[end] == nums[end - 1]:

end -= 1

if beg == end:

return nums[beg]

mid = (beg+end)/2

if nums[mid] > nums[end]:

beg = mid + 1

else:

end = mid

return nums[beg]

# 155,Min Stack\*\*\*:

**Python\_solution:**

My Python solution

class MinStack:

def \_\_init\_\_(self):

self.q = []

# @param x, an integer

# @return an integer

def push(self, x):

curMin = self.getMin()

if curMin == None or x < curMin:

curMin = x

self.q.append((x, curMin));

# @return nothing

def pop(self):

self.q.pop()

# @return an integer

def top(self):

if len(self.q) == 0:

return None

else:

return self.q[len(self.q) - 1][0]

# @return an integer

def getMin(self):

if len(self.q) == 0:

return None

else:

return self.q[len(self.q) - 1][1]

# 160,Intersection of Two Linked Lists\*\*\*:

**Python\_solution:**

Concise python code with comments

class Solution:

# @param two ListNodes

# @return the intersected ListNode

def getIntersectionNode(self, headA, headB):

if headA is None or headB is None:

return None

pa = headA # 2 pointers

pb = headB

while pa is not pb:

# if either pointer hits the end, switch head and continue the second traversal,

# if not hit the end, just move on to next

pa = headB if pa is None else pa.next

pb = headA if pb is None else pb.next

return pa # only 2 ways to get out of the loop, they meet or the both hit the end=None

# the idea is if you switch head, the possible difference between length would be countered.

# On the second traversal, they either hit or miss.

# if they meet, pa or pb would be the node we are looking for,

# if they didn't meet, they will hit the end at the same iteration, pa == pb == None, return either one of them is the same,None

# 162,Find Peak Element\*\*\*:

**Python\_solution:**

My clean and readable python solution

Basic Idea: Binary search

Elaboration:

if an element(not the right-most one) is smaller than its right neighbor, then there must be a peak element on its right, because the elements on its right is either

1. always increasing -> the right-most element is the peak

2. always decreasing -> the left-most element is the peak

3. first increasing then decreasing -> the pivot point is the peak

4. first decreasing then increasing -> the left-most element is the peak

Therefore, we can find the peak only on its right elements( cut the array to half)

The same idea applies to that an element(not the left-most one) is smaller than its left neighbor.

Conditions:

1. array length is 1 -> return the only index

2. array length is 2 -> return the bigger number's index

3. array length is bigger than 2 ->

(1) find mid, compare it with its left and right neighbors

(2) return mid if nums[mid] greater than both neighbors

(3) take the right half array if nums[mid] smaller than right neighbor

(4) otherwise, take the left half

Run time: O(logn)

Memory: constant

Test cases:

[1]

[1,2]

[2,1]

[1,2,3]

[3,2,1]

[2,1,3]

def findPeakElement(self, nums):

left = 0

right = len(nums)-1

# handle condition 3

while left < right-1:

mid = (left+right)/2

if nums[mid] > nums[mid+1] and nums[mid] > nums[mid-1]:

return mid

if nums[mid] < nums[mid+1]:

left = mid+1

else:

right = mid-1

#handle condition 1 and 2

return left if nums[left] >= nums[right] else right

# 164,Maximum Gap\*\*\*:

**Python\_solution:**

Python bucket sort from official solution

class Solution:

# @param num, a list of integer

# @return an integer

def maximumGap(self, num):

if len(num) < 2 or min(num) == max(num):

return 0

a, b = min(num), max(num)

size = math.ceil((b-a)/(len(num)-1))

bucket = [[None, None] for \_ in range((b-a)//size+1)]

for n in num:

b = bucket[(n-a)//size]

b[0] = n if b[0] is None else min(b[0], n)

b[1] = n if b[1] is None else max(b[1], n)

bucket = [b for b in bucket if b[0] is not None]

return max(bucket[i][0]-bucket[i-1][1] for i in range(1, len(bucket)))

The python version is 3.4, for 2.7 version rewrite the size

# 165,Compare Version Numbers\*\*\*:

**Python\_solution:**

2-4 lines Python, 3 different ways

\*\*Solution 1: \*Pad with `izip\_longest` with `fillvalue=0`\*\*\*

def compareVersion(self, version1, version2):

splits = (map(int, v.split('.')) for v in (version1, version2))

return cmp(\*zip(\*itertools.izip\_longest(\*splits, fillvalue=0)))

\*\*Solution 2: \*Pad with `[0] \* lengthDifference`\*\*\*

def compareVersion(self, version1, version2):

v1, v2 = (map(int, v.split('.')) for v in (version1, version2))

d = len(v2) - len(v1)

return cmp(v1 + [0]\*d, v2 + [0]\*-d)

\*\*Solution 3: \*Recursive, add zeros on the fly\*\*\*

def compareVersion(self, version1, version2):

main1, \_, rest1 = ('0' + version1).partition('.')

main2, \_, rest2 = ('0' + version2).partition('.')

return cmp(int(main1), int(main2)) or \

len(rest1+rest2) and self.compareVersion(rest1, rest2)

# 166,Fraction to Recurring Decimal\*\*\*:

**Python\_solution:**

Do not use python as cpp, here's a short version python code

Though python is slow, It is easy to write

class Solution:

# @return a string

def fractionToDecimal(self, numerator, denominator):

n, remainder = divmod(abs(numerator), abs(denominator))

sign = '-' if numerator\*denominator < 0 else ''

result = [sign+str(n), '.']

stack = []

while remainder not in stack:

stack.append(remainder)

n, remainder = divmod(remainder\*10, abs(denominator))

result.append(str(n))

idx = stack.index(remainder)

result.insert(idx+2, '(')

result.append(')')

return ''.join(result).replace('(0)', '').rstrip('.')

and there's no overflow

# 168,Excel Sheet Column Title\*\*\*:

**Python\_solution:**

My 1 lines code in Java, C++, and Python

Java:

return n == 0 ? "" : convertToTitle(--n / 26) + (char)('A' + (n % 26));

C++:

return n == 0 ? "" : convertToTitle(n / 26) + (char) (--n % 26 + 'A');

update: because the behavior of different compilers, the safe version should be:

return n == 0 ? "" : convertToTitle((n - 1) / 26) + (char) ((n - 1) % 26 + 'A');

Python:

return "" if num == 0 else self.convertToTitle((num - 1) / 26) + chr((num - 1) % 26 + ord('A'))

# 169,Majority Element\*\*\*:

**Python\_solution:**

One line solution in Python

NOTICE that the majority element \*\*always\*\* exist in the array,so that the middle \*\*always\*\* is the answer

return sorted(num)[len(num)/2]

# 171,Excel Sheet Column Number\*\*\*:

**Python\_solution:**

One line python code using Map/Reduce

def titleToNumber(self, s):

return reduce(lambda x,y:x\*26+y,map(lambda x:ord(x)-ord('A')+1,s))

# 172,Factorial Trailing Zeroes\*\*\*:

**Python\_solution:**

O(log5\_n) solution, python.

1. meet the number that can be dived by 5, the Trailing will have 1 more zero.

2 .meet the number that can be dived by 5\*5, the Trailing will have 2 more zero.

..

..

..

and so on.

so we just find how many number can be dived by 5, can be dived by 5\*5 ... and sum up.

def trailingZeroes(self, n):

r = 0

while n > 0:

n /= 5

r += n

return r

# 173,Binary Search Tree Iterator\*\*\*:

**Python\_solution:**

Two Python solutions, stack and generator

stack solution:

def \_\_init\_\_(self, root):

self.stack = []

while root:

self.stack.append(root)

root = root.left

# @return a boolean, whether we have a next smallest number

def hasNext(self):

return len(self.stack) > 0

# @return an integer, the next smallest number

def next(self):

node = self.stack.pop()

x = node.right

while x:

self.stack.append(x)

x = x.left

return node.val

generator solution:

def \_\_init\_\_(self, root):

self.last = root

while self.last and self.last.right:

self.last = self.last.right

self.current = None

self.g = self.iterate(root)

# @return a boolean, whether we have a next smallest number

def hasNext(self):

return self.current is not self.last

# @return an integer, the next smallest number

def next(self):

return next(self.g)

def iterate(self, node):

if node is None:

return

for x in self.iterate(node.left):

yield x

self.current = node

yield node.val

for x in self.iterate(node.right):

yield x

# 174,Dungeon Game\*\*\*:

**Python\_solution:**

6 lines Python, 8 lines Ruby

Just some DP.

---

\*\*Python\*\*

def calculateMinimumHP(self, dungeon):

n = len(dungeon[0])

need = [2\*\*31] \* (n-1) + [1]

for row in dungeon[::-1]:

for j in range(n)[::-1]:

need[j] = max(min(need[j:j+2]) - row[j], 1)

return need[0]

Got accepted in 52 ms, faster than all other recent Python submissions (best was 56 ms, achieved by 5.7692%).

---

\*\*Ruby\*\*

def calculate\_minimum\_hp(dungeon)

n = dungeon[0].size - 1

need = [1/0.0] \* n + [1]

dungeon.reverse\_each do |row|

n.downto(0) do |j|

need[j] = [need[j..j+1].min - row[j], 1].max

end

end

need[0]

end

# 175,Combine Two Tables\*\*\*:

**Best\_solution:**

Its a simple question of Left Join. My solution attached

SELECT Person.FirstName, Person.LastName, Address.City, Address.State from Person LEFT JOIN Address on Person.PersonId = Address.PersonId;

# 176,Second Highest Salary\*\*\*:

**Best\_solution:**

Simple query which handles the NULL situation

SELECT max(Salary)

FROM Employee

WHERE Salary < (SELECT max(Salary) FROM Employee)

Using max() will return a NULL if the value doesn't exist. So there is no need to UNION a NULL. Of course, if the second highest value is guaranteed to exist, using LIMIT 1,1 will be the best answer.

# 177,Nth Highest Salary\*\*\*:

**Best\_solution:**

Accpted Solution for the Nth Highest Salary

CREATE FUNCTION getNthHighestSalary(N INT) RETURNS INT

BEGIN

DECLARE M INT;

SET M=N-1;

RETURN (

# Write your MySQL query statement below.

SELECT DISTINCT Salary FROM Employee ORDER BY Salary DESC LIMIT M, 1

);

END

# 178,Rank Scores\*\*\*:

**Best\_solution:**

Simple, Short, Fast

These are four different solutions.

---

\*\*With Variables:\*\* 841 ms

First one uses two variables, one for the current rank and one for the previous score.

SELECT

Score,

@rank := @rank + (@prev <> (@prev := Score)) Rank

FROM

Scores,

(SELECT @rank := 0, @prev := -1) init

ORDER BY Score desc

---

\*\*Always Count:\*\* 1322 ms

This one counts, for each score, the number of distinct greater or equal scores.

SELECT

Score,

(SELECT count(distinct Score) FROM Scores WHERE Score >= s.Score) Rank

FROM Scores s

ORDER BY Score desc

---

\*\*Always Count, Pre-uniqued:\*\* 795 ms

Same as the previous one, but faster because I have a subquery that "uniquifies" the scores first. Not entirely sure \*why\* it's faster, I'm guessing MySQL makes `tmp` a temporary table and uses it for every outer Score.

SELECT

Score,

(SELECT count(\*) FROM (SELECT distinct Score s FROM Scores) tmp WHERE s >= Score) Rank

FROM Scores

ORDER BY Score desc

---

\*\*Filter/count Scores^2:\*\* 1414 ms

Inspired by the attempt in wangkan2001's answer. Finally `Id` is good for something :-)

SELECT s.Score, count(distinct t.score) Rank

FROM Scores s JOIN Scores t ON s.Score <= t.score

GROUP BY s.Id

ORDER BY s.Score desc

# 179,Largest Number\*\*\*:

**Python\_solution:**

My 3-lines code in Java and Python

The logic is pretty straightforward. Just compare number by convert it to string.

Thanks for Java 8, it makes code beautiful.

Java:

public class Solution {

public String largestNumber(int[] num) {

String[] array = Arrays.stream(num).mapToObj(String::valueOf).toArray(String[]::new);

Arrays.sort(array, (String s1, String s2) -> (s2 + s1).compareTo(s1 + s2));

return Arrays.stream(array).reduce((x, y) -> x.equals("0") ? y : x + y).get();

}

}

Python:

class Solution:

# @param num, a list of integers

# @return a string

def largestNumber(self, num):

num = [str(x) for x in num]

num.sort(cmp=lambda x, y: cmp(y+x, x+y))

return ''.join(num).lstrip('0') or '0'

# 180,Consecutive Numbers\*\*\*:

**Best\_solution:**

Simple solution

Select DISTINCT l1.Num from Logs l1, Logs l2, Logs l3

where l1.Id=l2.Id-1 and l2.Id=l3.Id-1

and l1.Num=l2.Num and l2.Num=l3.Num

# 181,Employees Earning More Than Their Managers\*\*\*:

**Best\_solution:**

A straightforward method

select E1.Name

from Employee as E1, Employee as E2

where E1.ManagerId = E2.Id and E1.Salary > E2.Salary

# 182,Duplicate Emails\*\*\*:

**Best\_solution:**

I have this Simple Approach, anybody has some other way

select Email

from Person

group by Email

having count(\*) > 1

# 183,Customers Who Never Order\*\*\*:

**Best\_solution:**

Three accepted solutions

SELECT A.Name from Customers A

WHERE NOT EXISTS (SELECT 1 FROM Orders B WHERE A.Id = B.CustomerId)

SELECT A.Name from Customers A

LEFT JOIN Orders B on a.Id = B.CustomerId

WHERE b.CustomerId is NULL

SELECT A.Name from Customers A

WHERE A.Id NOT IN (SELECT B.CustomerId from Orders B)

# 184,Department Highest Salary\*\*\*:

**Best\_solution:**

Three accpeted solutions

SELECT D.Name AS Department ,E.Name AS Employee ,E.Salary

FROM

Employee E,

(SELECT DepartmentId,max(Salary) as max FROM Employee GROUP BY DepartmentId) T,

Department D

WHERE E.DepartmentId = T.DepartmentId

AND E.Salary = T.max

AND E.DepartmentId = D.id

SELECT D.Name,A.Name,A.Salary

FROM

Employee A,

Department D

WHERE A.DepartmentId = D.Id

AND NOT EXISTS

(SELECT 1 FROM Employee B WHERE B.Salary > A.Salary AND A.DepartmentId = B.DepartmentId)

SELECT D.Name AS Department ,E.Name AS Employee ,E.Salary

from

Employee E,

Department D

WHERE E.DepartmentId = D.id

AND (DepartmentId,Salary) in

(SELECT DepartmentId,max(Salary) as max FROM Employee GROUP BY DepartmentId)

# 185,Department Top Three Salaries\*\*\*:

**Best\_solution:**

Accepted solution without group by or order by

select d.Name Department, e1.Name Employee, e1.Salary

from Employee e1

join Department d

on e1.DepartmentId = d.Id

where 3 > (select count(distinct(e2.Salary))

from Employee e2

where e2.Salary > e1.Salary

and e1.DepartmentId = e2.DepartmentId

);

# 187,Repeated DNA Sequences\*\*\*:

**Python\_solution:**

4 lines Python solution

I use a defauldict to initialize as 0 the dictionary of integers, then I check the dictionary for substrings seen more than once.

class Solution:

# @param s, a string

# @return a list of strings

def findRepeatedDnaSequences(self, s):

sequences = collections.defaultdict(int) #set '0' as the default value for non-existing keys

for i in range(len(s)):

sequences[s[i:i+10]] += 1#add 1 to the count

return [key for key, value in sequences.iteritems() if value > 1] #extract the relevant keys

# 188,Best Time to Buy and Sell Stock IV\*\*\*:

**Python\_solution:**

Well explained Python DP with comments

I think the general idea has been thoroughly explained by other brilliant leetcoders. All of the solutions are beautiful and concise. However, most of the them don't look obvious to me, so I wrote this and hope it looks more straight forward.

It's O(kn), apparently not optimal. I name the key variables as local profit and global profit to make things much understandable (well, at least , to me). Performance is not too bad though.

def maxProfit4(self, k, prices):

n = len(prices)

if n < 2:

return 0

# k is big enougth to cover all ramps.

if k >= n / 2:

return sum(i - j

for i, j in zip(prices[1:], prices[:-1]) if i - j > 0)

globalMax = [[0] \* n for \_ in xrange(k + 1)]

for i in xrange(1, k + 1):

# The max profit with i transations and selling stock on day j.

localMax = [0] \* n

for j in xrange(1, n):

profit = prices[j] - prices[j - 1]

localMax[j] = max(

# We have made max profit with (i - 1) transations in

# (j - 1) days.

# For the last transation, we buy stock on day (j - 1)

# and sell it on day j.

globalMax[i - 1][j - 1] + profit,

# We have made max profit with (i - 1) transations in

# (j - 1) days.

# For the last transation, we buy stock on day j and

# sell it on the same day, so we have 0 profit, apparently

# we do not have to add it.

globalMax[i - 1][j - 1], # + 0,

# We have made profit in (j - 1) days.

# We want to cancel the day (j - 1) sale and sell it on

# day j.

localMax[j - 1] + profit)

globalMax[i][j] = max(globalMax[i][j - 1], localMax[j])

return globalMax[k][-1]

# 189,Rotate Array\*\*\*:

**Python\_solution:**

My solution by using Python

class Solution:

# @param nums, a list of integer

# @param k, num of steps

# @return nothing, please modify the nums list in-place.

def rotate(self, nums, k):

n = len(nums)

k = k % n

nums[:] = nums[n-k:] + nums[:n-k]

A little important thing to be cautious:

nums[:] = nums[n-k:] + nums[:n-k]

can't be written as:

nums = nums[n-k:] + nums[:n-k]

on the OJ.

The previous one can truly change the value of \*\*old\*\* \*nums\*, but the following one just changes its reference to a \*\*new\*\* \*nums\* not the value of \*\*old\*\* \*nums\*.

# 190,Reverse Bits\*\*\*:

**Python\_solution:**

Python AC with 63ms, 3lines

class Solution:

# @param n, an integer

# @return an integer

def reverseBits(self, n):

oribin='{0:032b}'.format(n)

reversebin=oribin[::-1]

return int(reversebin,2)

# 191,Number of 1 Bits\*\*\*:

**Python\_solution:**

[Python] 2 solutions. One naive solution with built-in functions. One trick with bit operation

1.Built in solution with built-in function:

def hammingWeight(self, n):

"""

:type n: int

:rtype: int

"""

return bin(n).count('1')

2.Using bit operation to cancel a `1` in each round

Think of a number in binary `n = XXXXXX1000`, `n - 1 is XXXXXX0111`. `n & (n - 1)` will be `XXXXXX0000` which is just cancel the last `1`

def hammingWeight(self, n):

"""

:type n: int

:rtype: int

"""

c = 0

while n:

n &= n - 1

c += 1

return c

# 192,Word Frequency\*\*\*:

**Best\_solution:**

My simple solution (one line with pipe)

```

cat words.txt | tr -s ' ' '\n' | sort | uniq -c | sort -r | awk '{ print $2, $1 }'

```

\*\*tr -s\*\*: truncate the string with target string, but only remaining one instance (e.g. multiple whitespaces)

\*\*sort\*\*: To make the same string successive so that `uniq` could count the same string fully and correctly.

\*\*uniq -c\*\*: uniq is used to filter out the repeated lines which are successive, -c means counting

\*\*sort -r\*\*: -r means sorting in descending order

\*\*awk '{ print $2, $1 }'\*\*: To format the output, see [here][1].

[1]: http://linux.cn/article-3945-1.html

# 193,Valid Phone Numbers\*\*\*:

**Best\_solution:**

Three different solutions using grep, sed, and awk

Using `grep`:

grep -P '^(\d{3}-|\(\d{3}\) )\d{3}-\d{4}$' file.txt

Using `sed`:

sed -n -r '/^([0-9]{3}-|\([0-9]{3}\) )[0-9]{3}-[0-9]{4}$/p' file.txt

Using `awk`:

awk '/^([0-9]{3}-|\([0-9]{3}\) )[0-9]{3}-[0-9]{4}$/' file.txt

# 194,Transpose File\*\*\*:

**Best\_solution:**

AC solution using awk and statement just like C.

Just feel free to use `for` and `if`.

You can append string easily, for example, `s = s a` to append `a` with `s`.

awk '

{

for (i = 1; i <= NF; i++) {

if(NR == 1) {

s[i] = $i;

} else {

s[i] = s[i] " " $i;

}

}

}

END {

for (i = 1; s[i] != ""; i++) {

print s[i];

}

}' file.txt

# 195,Tenth Line\*\*\*:

**Best\_solution:**

Share four different solutions

# Solution 1

cnt=0

while read line && [ $cnt -le 10 ]; do

let 'cnt = cnt + 1'

if [ $cnt -eq 10 ]; then

echo $line

exit 0

fi

done < file.txt

# Solution 2

awk 'FNR == 10 {print }' file.txt

# OR

awk 'NR == 10' file.txt

# Solution 3

sed -n 10p file.txt

# Solution 4

tail -n+10 file.txt|head -1

# 196,Delete Duplicate Emails\*\*\*:

**Best\_solution:**

Simple Solution

> DELETE p1

FROM Person p1, Person p2

WHERE p1.Email = p2.Email AND

> p1.Id > p2.Id

EXPLANATION:

- Take the table in the example

\*\*Id | Email\*\*

\*\*1 | john@example.com\*\*

\*\*2 | bob@example.com\*\*

\*\*3 | john@example.com\*\*

- Join the table on itself by the Email and you'll get:

> FROM Person p1, Person p2 WHERE p1.Email = p2.Email

\*\*p1.Id | p1.Email | p2.Id | p2.Email\*\*

\*\*1 | john@example.com | 1 | john@example.com\*\*

\*\*3 | john@example.com | 1 | john@example.com\*\*

\*\*2 | bob@example.com | 2 | bob@example.com\*\*

\*\*1 | john@example.com | 3 | john@example.com\*\*

\*\*3 | john@example.com | 3 | john@example.com\*\*

- From this results filter the records that have p1.Id>p2.ID, in this case you'll get just one record:

> AND p1.Id > p2.Id

\*\*p1.Id | p1.Email | p2.Id | p2.Email\*\*

\*\*3 | john@example.com | 1 | john@example.com\*\*

- This is the record we need to delete, and by saying

> DELETE p1

in this multiple-table syntax, only matching rows from the tables listed before the FROM clause are deleted, in this case just

\*\*p1.Id | p1.Email\*\*

\*\*3 | john@example.com\*\*

will be deleted

# 197,Rising Temperature\*\*\*:

**Best\_solution:**

Simple Solution

SELECT wt1.Id

FROM Weather wt1, Weather wt2

WHERE wt1.Temperature > wt2.Temperature AND

TO\_DAYS(wt1.DATE)-TO\_DAYS(wt2.DATE)=1;

EXPLANATION:

\*\*TO\_DAYS(wt1.DATE)\*\* return the number of days between from year 0 to date DATE

\*\*TO\_DAYS(wt1.DATE)-TO\_DAYS(wt2.DATE)=1\*\* check if wt2.DATE is yesterday respect to wt1.DATE

We select from the joined tables the rows that have

\*\*wt1.Temperature > wt2.Temperature\*\*

and difference between dates in days of 1 (yesterday):

\*\*TO\_DAYS(wt1.DATE)-TO\_DAYS(wt2.DATE)=1;\*\*

# 198,House Robber\*\*\*:

**Python\_solution:**

Python solution, 3 lines.

Based on the recursive formula:

> f(0) = nums[0]

> f(1) = max(num[0], num[1])

> f(k) = max( f(k-2) + nums[k], f(k-1) )

class Solution:

def rob(self, nums):

last, now = 0, 0

for i in nums: last, now = now, max(last + i, now)

return now

# 199,Binary Tree Right Side View\*\*\*:

**Python\_solution:**

5-9 Lines Python, 48+ ms

Solution 1: \*\*Recursive, combine right and left:\*\* 5 lines, 56 ms

Compute the right view of both right and left left subtree, then combine them. For very unbalanced trees, this can be O(n^2), though.

def rightSideView(self, root):

if not root:

return []

right = self.rightSideView(root.right)

left = self.rightSideView(root.left)

return [root.val] + right + left[len(right):]

---

Solution 2: \*\*Recursive, first come first serve:\*\* 9 lines, 48 ms

DFS-traverse the tree right-to-left, add values to the view whenever we first reach a new record depth. This is O(n).

def rightSideView(self, root):

def collect(node, depth):

if node:

if depth == len(view):

view.append(node.val)

collect(node.right, depth+1)

collect(node.left, depth+1)

view = []

collect(root, 0)

return view

---

Solution 3: \*\*Iterative, level-by-level:\*\* 7 lines, 48 ms

Traverse the tree level by level and add the last value of each level to the view. This is O(n).

def rightSideView(self, root):

view = []

if root:

level = [root]

while level:

view += level[-1].val,

level = [kid for node in level for kid in (node.left, node.right) if kid]

return view

# 200,Number of Islands\*\*\*:

**Python\_solution:**

7 lines Python, ~14 lines Java

Sink and count the islands.

---

\*\*Python Solution\*\*

def numIslands(self, grid):

def sink(i, j):

if 0 <= i < len(grid) and 0 <= j < len(grid[i]) and grid[i][j] == '1':

grid[i][j] = '0'

map(sink, (i+1, i-1, i, i), (j, j, j+1, j-1))

return 1

return 0

return sum(sink(i, j) for i in range(len(grid)) for j in range(len(grid[i])))

---

\*\*Java Solution 1\*\*

public class Solution {

char[][] g;

public int numIslands(char[][] grid) {

int islands = 0;

g = grid;

for (int i=0; i<g.length; i++)

for (int j=0; j<g[i].length; j++)

islands += sink(i, j);

return islands;

}

int sink(int i, int j) {

if (i < 0 || i == g.length || j < 0 || j == g[i].length || g[i][j] == '0')

return 0;

g[i][j] = '0';

sink(i+1, j); sink(i-1, j); sink(i, j+1); sink(i, j-1);

return 1;

}

}

---

\*\*Java Solution 2\*\*

public class Solution {

public int numIslands(char[][] grid) {

int islands = 0;

for (int i=0; i<grid.length; i++)

for (int j=0; j<grid[i].length; j++)

islands += sink(grid, i, j);

return islands;

}

int sink(char[][] grid, int i, int j) {

if (i < 0 || i == grid.length || j < 0 || j == grid[i].length || grid[i][j] == '0')

return 0;

grid[i][j] = '0';

for (int k=0; k<4; k++)

sink(grid, i+d[k], j+d[k+1]);

return 1;

}

int[] d = {0, 1, 0, -1, 0};

}

# 201,Bitwise AND of Numbers Range\*\*\*:

**Python\_solution:**

Java/Python easy solution with explanation

First let's think what does bitwise AND do to two numbers, for example ( 0b means base 2)

4 & 7 = 0b100 & 0b111 = 0b100

5 & 7 = 0b101 & 0b111 = 0b101

5 & 6 = 0b101 & 0b110 = 0b100

The operator & is keeping those bits which is set in both number.

For several numbers, the operator & is keeping those bits which is 1 in every number.

In other word, a bit is 0 in any number will result in 0 in the answer's corresponding bit.

Now consider a range

[m = 0bxyz0acd, n=0bxyz1rst]

here xyzpacdrst all are digits in base 2.

We can find two numbers that are special in the range [m, n]

(1) m' = 0bxyz0111

(2) n' = 0bxyz1000

The bitwise AND of all the numbers in range [m, n] is just the bitwise AND of the two special number

rangeBitwiseAnd(m, n) = m' & n' = 0bxyz0000

This tells us, the bitwise and of the range is keeping the common bits of m and n from left to right until the first bit that they are different, padding zeros for the rest.

\*\*Java\*\*

public int rangeBitwiseAnd(int m, int n) {

int i = 0;

for (; m != n; ++i) {

m >>= 1;

n >>= 1;

}

return n << i;

}

// 8266 / 8266 test cases passed.

// Status: Accepted

// Runtime: 8 ms

\*\*Python\*\*

def rangeBitwiseAnd(self, m, n):

i = 0

while m != n:

m >>= 1

n >>= 1

i += 1

return n << i

# 8266 / 8266 test cases passed.

# Status: Accepted

# Runtime: 208 ms

# 202,Happy Number\*\*\*:

**Python\_solution:**

My Python Solution

def isHappy(self, n):

mem = set()

while n != 1:

n = sum([int(i) \*\* 2 for i in str(n)])

if n in mem:

return False

else:

mem.add(n)

else:

return True

# 203,Remove Linked List Elements\*\*\*:

**Python\_solution:**

Python solution

class Solution:

# @param {ListNode} head

# @param {integer} val

# @return {ListNode}

def removeElements(self, head, val):

dummy = ListNode(-1)

dummy.next = head

next = dummy

while next != None and next.next != None:

if next.next.val == val:

next.next = next.next.next

else:

next = next.next

return dummy.next

# 204,Count Primes\*\*\*:

**Python\_solution:**

Fast Python Solution

class Solution:

# @param {integer} n

# @return {integer}

def countPrimes(self, n):

if n < 3:

return 0

primes = [True] \* n

primes[0] = primes[1] = False

for i in range(2, int(n \*\* 0.5) + 1):

if primes[i]:

primes[i \* i: n: i] = [False] \* len(primes[i \* i: n: i])

return sum(primes)

# 205,Isomorphic Strings\*\*\*:

**Python\_solution:**

Python different solutions (dictionary, etc).

def isIsomorphic1(self, s, t):

d1, d2 = {}, {}

for i, val in enumerate(s):

d1[val] = d1.get(val, []) + [i]

for i, val in enumerate(t):

d2[val] = d2.get(val, []) + [i]

return sorted(d1.values()) == sorted(d2.values())

def isIsomorphic2(self, s, t):

d1, d2 = [[] for \_ in xrange(256)], [[] for \_ in xrange(256)]

for i, val in enumerate(s):

d1[ord(val)].append(i)

for i, val in enumerate(t):

d2[ord(val)].append(i)

return sorted(d1) == sorted(d2)

def isIsomorphic3(self, s, t):

return len(set(zip(s, t))) == len(set(s)) == len(set(t))

def isIsomorphic4(self, s, t):

return [s.find(i) for i in s] == [t.find(j) for j in t]

def isIsomorphic5(self, s, t):

return map(s.find, s) == map(t.find, t)

def isIsomorphic(self, s, t):

d1, d2 = [0 for \_ in xrange(256)], [0 for \_ in xrange(256)]

for i in xrange(len(s)):

if d1[ord(s[i])] != d2[ord(t[i])]:

return False

d1[ord(s[i])] = i+1

d2[ord(t[i])] = i+1

return True

# 206,Reverse Linked List\*\*\*:

**Python\_solution:**

Python Iterative and Recursive Solution

class Solution:

# @param {ListNode} head

# @return {ListNode}

def reverseList(self, head):

prev = None

while head:

curr = head

head = head.next

curr.next = prev

prev = curr

return prev

Recursion

class Solution:

# @param {ListNode} head

# @return {ListNode}

def reverseList(self, head):

return self.\_reverse(head)

def \_reverse(self, node, prev=None):

if not node:

return prev

n = node.next

node.next = prev

return self.\_reverse(n, node)

# 207,Course Schedule\*\*\*:

**Python\_solution:**

Python 20 lines DFS solution sharing with explanation

def canFinish(self, numCourses, prerequisites):

graph = [[] for \_ in xrange(numCourses)]

visit = [0 for \_ in xrange(numCourses)]

for x, y in prerequisites:

graph[x].append(y)

def dfs(i):

if visit[i] == -1:

return False

if visit[i] == 1:

return True

visit[i] = -1

for j in graph[i]:

if not dfs(j):

return False

visit[i] = 1

return True

for i in xrange(numCourses):

if not dfs(i):

return False

return True

1. if node `v` has not been visited, then mark it as `0`.

2. if node `v` is being visited, then mark it as `-1`. If we find a vertex marked as `-1` in DFS, then their is a ring.

3. if node `v` has been visited, then mark it as `1`. If a vertex was marked as `1`, then no ring contains `v` or its successors.

\*References: [daoluan.net][1]\*

[1]: http://daoluan.net/blog/map-ring/

# 208,Implement Trie (Prefix Tree)\*\*\*:

**Python\_solution:**

AC Python Solution

class TrieNode:

# Initialize your data structure here.

def \_\_init\_\_(self):

self.children = collections.defaultdict(TrieNode)

self.is\_word = False

class Trie:

def \_\_init\_\_(self):

self.root = TrieNode()

def insert(self, word):

current = self.root

for letter in word:

current = current.children[letter]

current.is\_word = True

def search(self, word):

current = self.root

for letter in word:

current = current.children.get(letter)

if current is None:

return False

return current.is\_word

def startsWith(self, prefix):

current = self.root

for letter in prefix:

current = current.children.get(letter)

if current is None:

return False

return True

# 209,Minimum Size Subarray Sum\*\*\*:

**Python\_solution:**

Python O(n) and O(n log n) solution

class Solution:

def minSubArrayLen(self, s, nums):

total = left = 0

result = len(nums) + 1

for right, n in enumerate(nums):

total += n

while total >= s:

result = min(result, right - left + 1)

total -= nums[left]

left += 1

return result if result <= len(nums) else 0

O(n log n)

class Solution:

def minSubArrayLen(self, target, nums):

result = len(nums) + 1

for idx, n in enumerate(nums[1:], 1):

nums[idx] = nums[idx - 1] + n

left = 0

for right, n in enumerate(nums):

if n >= target:

left = self.find\_left(left, right, nums, target, n)

result = min(result, right - left + 1)

return result if result <= len(nums) else 0

def find\_left(self, left, right, nums, target, n):

while left < right:

mid = (left + right) // 2

if n - nums[mid] >= target:

left = mid + 1

else:

right = mid

return left

# 210,Course Schedule II\*\*\*:

**Python\_solution:**

Python dfs, bfs solutions with comments.

# BFS

def findOrder1(self, numCourses, prerequisites):

dic = {i: set() for i in xrange(numCourses)}

neigh = collections.defaultdict(set)

for i, j in prerequisites:

dic[i].add(j)

neigh[j].add(i)

# queue stores the courses which have no prerequisites

queue = collections.deque([i for i in dic if not dic[i]])

count, res = 0, []

while queue:

node = queue.popleft()

res.append(node)

count += 1

for i in neigh[node]:

dic[i].remove(node)

if not dic[i]:

queue.append(i)

return res if count == numCourses else []

# DFS

def findOrder(self, numCourses, prerequisites):

dic = collections.defaultdict(set)

neigh = collections.defaultdict(set)

for i, j in prerequisites:

dic[i].add(j)

neigh[j].add(i)

stack = [i for i in xrange(numCourses) if not dic[i]]

res = []

while stack:

node = stack.pop()

res.append(node)

for i in neigh[node]:

dic[i].remove(node)

if not dic[i]:

stack.append(i)

dic.pop(node)

return res if not dic else []

# 212,Word Search II\*\*\*:

**Python\_solution:**

Python code use trie and dfs 380ms

class Solution:

# @param {character[][]} board

# @param {string[]} words

# @return {string[]}

def findWords(self, board, words):

#make trie

trie={}

for w in words:

t=trie

for c in w:

if c not in t:

t[c]={}

t=t[c]

t['#']='#'

self.res=set()

self.used=[[False]\*len(board[0]) for \_ in range(len(board))]

for i in range(len(board)):

for j in range(len(board[0])):

self.find(board,i,j,trie,'')

return list(self.res)

def find(self,board,i,j,trie,pre):

if '#' in trie:

self.res.add(pre)

if i<0 or i>=len(board) or j<0 or j>=len(board[0]):

return

if not self.used[i][j] and board[i][j] in trie:

self.used[i][j]=True

self.find(board,i+1,j,trie[board[i][j]],pre+board[i][j])

self.find(board,i,j+1,trie[board[i][j]],pre+board[i][j])

self.find(board,i-1,j,trie[board[i][j]],pre+board[i][j])

self.find(board,i,j-1,trie[board[i][j]],pre+board[i][j])

self.used[i][j]=False

# 213,House Robber II\*\*\*:

**Python\_solution:**

My Python Solution

class Solution(object):

def rob(self, nums):

"""

:type nums: List[int]

:rtype: int

"""

n = len(nums)

if n == 0: return 0

if n < 4: return max(nums)

first, second = 0, 0

for i in nums[:-1]: first, second = second, max(first + i, second)

result = second

first, second = 0, 0

for i in nums[1:]: first, second = second, max(first + i, second)

return max(result, second)

# 214,Shortest Palindrome\*\*\*:

**Python\_solution:**

Python solution(KMP)

class Solution:

# @param {string} s

# @return {string}

def shortestPalindrome(self, s):

A=s+"\*"+s[::-1]

cont=[0]

for i in range(1,len(A)):

index=cont[i-1]

while(index>0 and A[index]!=A[i]):

index=cont[index-1]

cont.append(index+(1 if A[index]==A[i] else 0))

return s[cont[-1]:][::-1]+s

# 215,Kth Largest Element in an Array\*\*\*:

**Python\_solution:**

Python different solutions with comments (bubble sort, selection sort, heap sort and quick sort).

# O(nlgn) time

def findKthLargest1(self, nums, k):

return sorted(nums, reverse=True)[k-1]

# O(nk) time, bubble sort idea, TLE

def findKthLargest2(self, nums, k):

for i in xrange(k):

for j in xrange(len(nums)-i-1):

if nums[j] > nums[j+1]:

# exchange elements, time consuming

nums[j], nums[j+1] = nums[j+1], nums[j]

return nums[len(nums)-k]

# O(nk) time, selection sort idea

def findKthLargest3(self, nums, k):

for i in xrange(len(nums), len(nums)-k, -1):

tmp = 0

for j in xrange(i):

if nums[j] > nums[tmp]:

tmp = j

nums[tmp], nums[i-1] = nums[i-1], nums[tmp]

return nums[len(nums)-k]

# O(k+(n-k)lgk) time, min-heap

def findKthLargest4(self, nums, k):

heap = []

for num in nums:

heapq.heappush(heap, num)

for \_ in xrange(len(nums)-k):

heapq.heappop(heap)

return heapq.heappop(heap)

# O(k+(n-k)lgk) time, min-heap

def findKthLargest5(self, nums, k):

return heapq.nlargest(k, nums)[k-1]

# O(n) time, quick selection

def findKthLargest(self, nums, k):

# convert the kth largest to smallest

return self.findKthSmallest(nums, len(nums)+1-k)

def findKthSmallest(self, nums, k):

if nums:

pos = self.partition(nums, 0, len(nums)-1)

if k > pos+1:

return self.findKthSmallest(nums[pos+1:], k-pos-1)

elif k < pos+1:

return self.findKthSmallest(nums[:pos], k)

else:

return nums[pos]

# choose the right-most element as pivot

def partition(self, nums, l, r):

low = l

while l < r:

if nums[l] < nums[r]:

nums[l], nums[low] = nums[low], nums[l]

low += 1

l += 1

nums[low], nums[r] = nums[r], nums[low]

return low

# 216,Combination Sum III\*\*\*:

**Python\_solution:**

Concise python solution using DFS

class Solution:

# @param {integer} k

# @param {integer} n

# @return {integer[][]}

def combinationSum3(self, k, n):

if n > sum([i for i in range(1, 11)]):

return []

res = []

self.sum\_help(k, n, 1, [], res)

return res

def sum\_help(self, k, n, curr, arr, res):

if len(arr) == k:

if sum(arr) == n:

res.append(list(arr))

return

if len(arr) > k or curr > 9:

return

for i in range(curr, 10):

arr.append(i)

self.sum\_help(k, n, i + 1, arr, res)

arr.pop()

# 217,Contains Duplicate\*\*\*:

**Python\_solution:**

One line solution in python

class Solution(object):

def containsDuplicate(self, nums):

"""

:type nums: List[int]

:rtype: bool

"""

return len(nums) != len(set(nums))

# 218,The Skyline Problem\*\*\*:

**Python\_solution:**

14 line python code, straightforward & easy to understand

class Solution(object):

def getSkyline(self, buildings):

"""

:type buildings: List[List[int]]

:rtype: List[List[int]]

"""

def addsky(pos, hei):

if sky[-1][1] != hei:

sky.append([pos, hei])

sky = [[-1,0]]

# possible corner positions

position = set([b[0] for b in buildings] + [b[1] for b in buildings])

# live buildings

live = []

i = 0

for t in sorted(position):

# add the new buildings whose left side is lefter than position t

while i < len(buildings) and buildings[i][0] <= t:

heappush(live, (-buildings[i][2], buildings[i][1]))

i += 1

# remove the past buildings whose right side is lefter than position t

while live and live[0][1] <= t:

heappop(live)

# pick the highest existing building at this moment

h = -live[0][0] if live else 0

addsky(t, h)

return sky[1:]

# 219,Contains Duplicate II\*\*\*:

**Python\_solution:**

Python concise solution with dictionary.

def containsNearbyDuplicate(self, nums, k):

dic = {}

for i, v in enumerate(nums):

if v in dic and i - dic[v] <= k:

return True

dic[v] = i

return False

# 220,Contains Duplicate III\*\*\*:

**Python\_solution:**

Java/Python one pass solution, O(n) time O(n) space using buckets

The idea is like the bucket sort algorithm. Suppose we have consecutive buckets covering the range of nums with each bucket a width of (t+1). If there are two item with difference <= t, one of the two will happen:

(1) the two in the same bucket

(2) the two in neighbor buckets

For detailed explanation see my blog [here](http://algobox.org/contains-duplicate-iii/)

\*\*Python\*\*

def containsNearbyAlmostDuplicate(self, nums, k, t):

if t < 0: return False

n = len(nums)

d = {}

w = t + 1

for i in xrange(n):

m = nums[i] / w

if m in d:

return True

if m - 1 in d and abs(nums[i] - d[m - 1]) < w:

return True

if m + 1 in d and abs(nums[i] - d[m + 1]) < w:

return True

d[m] = nums[i]

if i >= k: del d[nums[i - k] / w]

return False

# 30 / 30 test cases passed.

# Status: Accepted

# Runtime: 56 ms

# 93.81%

\*\*Java\*\*

private long getID(long i, long w) {

return i < 0 ? (i + 1) / w - 1 : i / w;

}

public boolean containsNearbyAlmostDuplicate(int[] nums, int k, int t) {

if (t < 0) return false;

Map<Long, Long> d = new HashMap<>();

long w = (long)t + 1;

for (int i = 0; i < nums.length; ++i) {

long m = getID(nums[i], w);

if (d.containsKey(m))

return true;

if (d.containsKey(m - 1) && Math.abs(nums[i] - d.get(m - 1)) < w)

return true;

if (d.containsKey(m + 1) && Math.abs(nums[i] - d.get(m + 1)) < w)

return true;

d.put(m, (long)nums[i]);

if (i >= k) d.remove(getID(nums[i - k], w));

}

return false;

}

# 221,Maximal Square\*\*\*:

**Python\_solution:**

Share my concise python solution

class Solution:

# @param {character[][]} matrix

# @return {integer}

def maximalSquare(self, matrix):

if not matrix: return 0

m , n = len(matrix),len(matrix[0])

dp = [[0 if matrix[i][j]=='0' else 1for j in xrange(n)]for i in xrange(m)]

for i in xrange(1,m):

for j in xrange(1,n):

if matrix[i][j] =='1': dp[i][j] = min(dp[i-1][j],dp[i][j-1],dp[i-1][j-1])+1

else: dp[i][j] = 0

ans = max([max(i) for i in dp])

return ans \*\* 2

# 222,Count Complete Tree Nodes\*\*\*:

**Python\_solution:**

My python solution in O(lgn \* lgn) time

compare the depth between left sub tree and right sub tree.

A, If it is equal, it means the left sub tree is a full binary tree

B, It it is not , it means the right sub tree is a full binary tree

class Solution:

# @param {TreeNode} root

# @return {integer}

def countNodes(self, root):

if not root:

return 0

leftDepth = self.getDepth(root.left)

rightDepth = self.getDepth(root.right)

if leftDepth == rightDepth:

return pow(2, leftDepth) + self.countNodes(root.right)

else:

return pow(2, rightDepth) + self.countNodes(root.left)

def getDepth(self, root):

if not root:

return 0

return 1 + self.getDepth(root.left)

# 223,Rectangle Area\*\*\*:

**Python\_solution:**

Python concise solution.

def computeArea(self, A, B, C, D, E, F, G, H):

overlap = max(min(C,G)-max(A,E), 0)\*max(min(D,H)-max(B,F), 0)

return (A-C)\*(B-D) + (E-G)\*(F-H) - overlap

# 224,Basic Calculator\*\*\*:

**Python\_solution:**

Easy 18 lines C++, 16 lines Python

Keep a global running total and a stack of signs (+1 or -1), one for each open scope. The "global" outermost sign is +1.

- Each number consumes a sign.

- Each `+` and `-` causes a new sign.

- Each `(` duplicates the current sign so it can be used for the first term inside the new scope. That's also why I start with `[1, 1]` - the global sign 1 and a duplicate to be used for the first term, since expressions start like `3...` or `(...`, not like `+3...` or `+(...`.

- Each `)` closes the current scope and thus drops the current sign.

Also see the example trace below my programs.

\*\*C++:\*\*

int calculate(string s) {

int total = 0;

vector<int> signs(2, 1);

for (int i=0; i<s.size(); i++) {

char c = s[i];

if (c >= '0') {

int number = 0;

while (i < s.size() && s[i] >= '0')

number = 10 \* number + s[i++] - '0';

total += signs.back() \* number;

signs.pop\_back();

i--;

}

else if (c == ')')

signs.pop\_back();

else if (c != ' ')

signs.push\_back(signs.back() \* (c == '-' ? -1 : 1));

}

return total;

}

\*\*Python:\*\*

def calculate(self, s):

total = 0

i, signs = 0, [1, 1]

while i < len(s):

c = s[i]

if c.isdigit():

start = i

while i < len(s) and s[i].isdigit():

i += 1

total += signs.pop() \* int(s[start:i])

continue

if c in '+-(':

signs += signs[-1] \* (1, -1)[c == '-'],

elif c == ')':

signs.pop()

i += 1

return total

\*\*Example trace:\*\*

Here's an example trace for input `3-(2+(9-4))`.

remaining sign stack total

3-(2+(9-4)) [1, 1] 0

-(2+(9-4)) [1] 3

(2+(9-4)) [1, -1] 3

2+(9-4)) [1, -1, -1] 3

+(9-4)) [1, -1] 1

(9-4)) [1, -1, -1] 1

9-4)) [1, -1, -1, -1] 1

-4)) [1, -1, -1] -8

4)) [1, -1, -1, 1] -8

)) [1, -1, -1] -4

) [1, -1] -4

[1] -4

If you want to see traces for other examples, you can add this at the start inside the loop and after the loop (that's for the Python solution, where it's all easier):

print '%11s %-16s %2d' % (s[i:], signs, total)

# 225,Implement Stack using Queues\*\*\*:

**Python\_solution:**

Concise 1 Queue - Java, C++, Python

\*\*Explanation:\*\*

Just use a queue where you \*"push to front"\* by pushing to back and then rotating the queue until the new element is at the front (i.e., size-1 times move the front element to the back).

---

\*\*C++:\*\* 0 ms

class Stack {

queue<int> q;

public:

void push(int x) {

q.push(x);

for (int i=1; i<q.size(); i++) {

q.push(q.front());

q.pop();

}

}

void pop() {

q.pop();

}

int top() {

return q.front();

}

bool empty() {

return q.empty();

}

};

---

\*\*Java:\*\* 140 ms

class MyStack {

private Queue<Integer> queue = new LinkedList<>();

public void push(int x) {

queue.add(x);

for (int i=1; i<queue.size(); i++)

queue.add(queue.remove());

}

public void pop() {

queue.remove();

}

public int top() {

return queue.peek();

}

public boolean empty() {

return queue.isEmpty();

}

}

---

\*\*Python:\*\* 36 ms

class Stack:

def \_\_init\_\_(self):

self.\_queue = collections.deque()

def push(self, x):

q = self.\_queue

q.append(x)

for \_ in range(len(q) - 1):

q.append(q.popleft())

def pop(self):

return self.\_queue.popleft()

def top(self):

return self.\_queue[0]

def empty(self):

return not len(self.\_queue)

# 226,Invert Binary Tree\*\*\*:

**Python\_solution:**

3-4 lines Python

def invertTree(self, root):

if root:

root.left, root.right = self.invertTree(root.right), self.invertTree(root.left)

return root

Maybe make it four lines for better readability:

def invertTree(self, root):

if root:

invert = self.invertTree

root.left, root.right = invert(root.right), invert(root.left)

return root

---

And an iterative version using my own stack:

def invertTree(self, root):

stack = [root]

while stack:

node = stack.pop()

if node:

node.left, node.right = node.right, node.left

stack += node.left, node.right

return root

# 227,Basic Calculator II\*\*\*:

**Python\_solution:**

Python short solution with stack.

def calculate(self, s):

if not s:

return "0"

stack, num, sign = [], 0, "+"

for i in xrange(len(s)):

if s[i].isdigit():

num = num\*10+ord(s[i])-ord("0")

if (not s[i].isdigit() and not s[i].isspace()) or i == len(s)-1:

if sign == "-":

stack.append(-num)

elif sign == "+":

stack.append(num)

elif sign == "\*":

stack.append(stack.pop()\*num)

else:

tmp = stack.pop()

if tmp//num < 0 and tmp%num != 0:

stack.append(tmp//num+1)

else:

stack.append(tmp//num)

sign = s[i]

num = 0

return sum(stack)

# 228,Summary Ranges\*\*\*:

**Python\_solution:**

6 lines in Python

Three versions of the same algorithm, all take O(n) time.

---

\*\*Solution 1\*\*

Just collect the ranges, then format and return them.

def summaryRanges(self, nums):

ranges = []

for n in nums:

if not ranges or n > ranges[-1][-1] + 1:

ranges += [],

ranges[-1][1:] = n,

return ['->'.join(map(str, r)) for r in ranges]

---

\*\*Solution 2\*\*

A variation of solution 1, holding the current range in an extra variable `r` to make things easier. Note that `r` contains at most two elements, so the `in`-check takes constant time.

def summaryRanges(self, nums):

ranges, r = [], []

for n in nums:

if n-1 not in r:

r = []

ranges += r,

r[1:] = n,

return ['->'.join(map(str, r)) for r in ranges]

---

\*\*Solution 3\*\*

A tricky short version.

def summaryRanges(self, nums):

ranges = r = []

for n in nums:

if `n-1` not in r:

r = []

ranges += r,

r[1:] = `n`,

return map('->'.join, ranges)

---

\*\*About the commas :-)\*\*

Three people asked about them in the comments, so I'll also explain it here as well. I have these two basic cases:

ranges += [],

r[1:] = n,

Why the trailing commas? Because it turns the right hand side into a tuple and I get the same effects as these more common alternatives:

ranges += [[]]

or

ranges.append([])

r[1:] = [n]

Without the comma, ...

- `ranges += []` wouldn't add `[]` itself but only its elements, i.e., nothing.

- `r[1:] = n` wouldn't work, because my `n` is not an iterable.

Why do it this way instead of the more common alternatives I showed above? Because it's shorter and faster (according to tests I did a while back).

# 229,Majority Element II\*\*\*:

**Python\_solution:**

Clear O(n) solution in python, no data structure or sort.

class Solution:

# @param {integer[]} nums

# @return {integer[]}

def majorityElement(self, nums):

a, b, ca, cb = 0, 1, 0, 0

for num in nums:

if a == num:

ca += 1

elif b == num:

cb += 1

elif ca == 0:

a, ca = num, 1

elif cb == 0:

b, cb = num, 1

else:

ca -= 1

cb -= 1

ca = len([0 for num in nums if num == a])

cb = len([0 for num in nums if num == b])

res = []

if ca > len(nums) / 3:

res.append(a)

if cb > len(nums) / 3:

res.append(b)

return res

# 230,Kth Smallest Element in a BST\*\*\*:

**Python\_solution:**

Pythonic approach with generator

With generator in python, one very straightforward solution might be:

class Solution:

# @param {TreeNode} root

# @param {integer} k

# @return {integer}

def kthSmallest(self, root, k):

for val in self.inorder(root):

if k == 1:

return val

else:

k -= 1

def inorder(self, root):

if root is not None:

for val in self.inorder(root.left):

yield val

yield root.val

for val in self.inorder(root.right):

yield val

# 231,Power of Two\*\*\*:

**Python\_solution:**

Python one line solution

class Solution(object):

def isPowerOfTwo(self, n):

"""

:type n: int

:rtype: bool

"""

return n > 0 and not (n & n-1)

# 232,Implement Queue using Stacks\*\*\*:

**Python\_solution:**

Share my python solution (32ms)

The idea is to simulate a queue using two stacks (same as previous posts). I use python list as the underlying data structure for stack and add a "move()" method to simplify code: it moves all elements of the "inStack" to the "outStack" when the "outStack" is empty. Here is the code

class Queue(object):

def \_\_init\_\_(self):

"""

initialize your data structure here.

"""

self.inStack, self.outStack = [], []

def push(self, x):

"""

:type x: int

:rtype: nothing

"""

self.inStack.append(x)

def pop(self):

"""

:rtype: nothing

"""

self.move()

self.outStack.pop()

def peek(self):

"""

:rtype: int

"""

self.move()

return self.outStack[-1]

def empty(self):

"""

:rtype: bool

"""

return (not self.inStack) and (not self.outStack)

def move(self):

"""

:rtype nothing

"""

if not self.outStack:

while self.inStack:

self.outStack.append(self.inStack.pop())

# 233,Number of Digit One\*\*\*:

**Python\_solution:**

4+ lines, O(log n), C++/Java/Python

Go through the digit positions one at a time, find out how often a "1" appears at each position, and sum those up.

\*\*C++ solution\*\*

int countDigitOne(int n) {

int ones = 0;

for (long long m = 1; m <= n; m \*= 10)

ones += (n/m + 8) / 10 \* m + (n/m % 10 == 1) \* (n%m + 1);

return ones;

}

\*\*Explanation\*\*

Let me use variables `a` and `b` to make the explanation a bit nicer.

int countDigitOne(int n) {

int ones = 0;

for (long long m = 1; m <= n; m \*= 10) {

int a = n/m, b = n%m;

ones += (a + 8) / 10 \* m + (a % 10 == 1) \* (b + 1);

}

return ones;

}

Go through the digit positions by using position multiplier `m` with values 1, 10, 100, 1000, etc.

For each position, split the decimal representation into two parts, for example split n=3141592 into a=31415 and b=92 when we're at m=100 for analyzing the hundreds-digit. And then we know that the hundreds-digit of n is 1 for prefixes "" to "3141", i.e., 3142 times. Each of those times is a streak, though. Because it's the hundreds-digit, each streak is 100 long. So `(a / 10 + 1) \* 100` times, the hundreds-digit is 1.

Consider the thousands-digit, i.e., when m=1000. Then a=3141 and b=592. The thousands-digit is 1 for prefixes "" to "314", so 315 times. And each time is a streak of 1000 numbers. However, since the thousands-digit is a 1, the very last streak isn't 1000 numbers but only 593 numbers, for the suffixes "000" to "592". So `(a / 10 \* 1000) + (b + 1)` times, the thousands-digit is 1.

The case distincton between the current digit/position being 0, 1 and >=2 can easily be done in one expression. With `(a + 8) / 10` you get the number of full streaks, and `a % 10 == 1` tells you whether to add a partial streak.

\*\*Java version\*\*

public int countDigitOne(int n) {

int ones = 0;

for (long m = 1; m <= n; m \*= 10)

ones += (n/m + 8) / 10 \* m + (n/m % 10 == 1 ? n%m + 1 : 0);

return ones;

}

\*\*Python version\*\*

def countDigitOne(self, n):

ones, m = 0, 1

while m <= n:

ones += (n/m + 8) / 10 \* m + (n/m % 10 == 1) \* (n%m + 1)

m \*= 10

return ones

Using `sum` or recursion it can also be a [one-liner](https://leetcode.com/discuss/44302/1-liners-in-python).

---

Old solution

---

Go through the digit positions from back to front. I found it ugly to explain, so I made up that above new solution instead. The `n` here is the new solution's `a`, and the `r` here is the new solution's `b+1`.

\*\*Python\*\*

def countDigitOne(self, n):

ones = 0

m = r = 1

while n > 0:

ones += (n + 8) / 10 \* m + (n % 10 == 1) \* r

r += n % 10 \* m

m \*= 10

n /= 10

return ones

\*\*Java\*\*

public int countDigitOne(int n) {

int ones = 0, m = 1, r = 1;

while (n > 0) {

ones += (n + 8) / 10 \* m + (n % 10 == 1 ? r : 0);

r += n % 10 \* m;

m \*= 10;

n /= 10;

}

return ones;

}

\*\*C++\*\*

int countDigitOne(int n) {

int ones = 0, m = 1, r = 1;

while (n > 0) {

ones += (n + 8) / 10 \* m + (n % 10 == 1) \* r;

r += n % 10 \* m;

m \*= 10;

n /= 10;

}

return ones;

}

# 234,Palindrome Linked List\*\*\*:

**Python\_solution:**

Python easy to understand solution with comments (operate nodes directly).

def isPalindrome(self, head):

fast = slow = head

# find the mid node

while fast and fast.next:

fast = fast.next.next

slow = slow.next

# reverse the second half

node = None

while slow:

nxt = slow.next

slow.next = node

node = slow

slow = nxt

# compare the first and second half nodes

while node: # while node and head:

if node.val != head.val:

return False

node = node.next

head = head.next

return True

# 235,Lowest Common Ancestor of a Binary Search Tree\*\*\*:

**Python\_solution:**

Python Iterative Solution

class Solution:

def lowestCommonAncestor(self, root, p, q):

while root:

if root.val > p.val and root.val > q.val:

root = root.left

elif root.val < p.val and root.val < q.val:

root = root.right

else:

return root

# 236,Lowest Common Ancestor of a Binary Tree\*\*\*:

**Python\_solution:**

4 lines C++/Java/Python/Ruby

Same solution in several languages. It's recursive and expands the meaning of the function. If the current (sub)tree contains both p and q, then the function result is their LCA. If only one of them is in that subtree, then the result is that one of them. If neither are in that subtree, the result is null/None/nil.

Update: I also wrote [two iterative solutions](https://leetcode.com/discuss/45603/iterative-solution) now, one of them being a version of the solution here. They're more complicated than this simple recursive solution, but I do find them interesting.

---

\*\*C++\*\*

TreeNode\* lowestCommonAncestor(TreeNode\* root, TreeNode\* p, TreeNode\* q) {

if (!root || root == p || root == q) return root;

TreeNode\* left = lowestCommonAncestor(root->left, p, q);

TreeNode\* right = lowestCommonAncestor(root->right, p, q);

return !left ? right : !right ? left : root;

}

---

\*\*Python\*\*

def lowestCommonAncestor(self, root, p, q):

if root in (None, p, q): return root

left, right = (self.lowestCommonAncestor(kid, p, q)

for kid in (root.left, root.right))

return root if left and right else left or right

Or using that `None` is considered smaller than any node:

def lowestCommonAncestor(self, root, p, q):

if root in (None, p, q): return root

subs = [self.lowestCommonAncestor(kid, p, q)

for kid in (root.left, root.right)]

return root if all(subs) else max(subs)

---

\*\*Ruby\*\*

def lowest\_common\_ancestor(root, p, q)

return root if [nil, p, q].index root

left = lowest\_common\_ancestor(root.left, p, q)

right = lowest\_common\_ancestor(root.right, p, q)

left && right ? root : left || right

end

---

\*\*Java\*\*

public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {

if (root == null || root == p || root == q) return root;

TreeNode left = lowestCommonAncestor(root.left, p, q);

TreeNode right = lowestCommonAncestor(root.right, p, q);

return left == null ? right : right == null ? left : root;

}

# 237,Delete Node in a Linked List\*\*\*:

**Python\_solution:**

1-3 lines, C++/Java/Python/C/C#/JavaScript/Ruby

We can't really delete the node, but we can kinda achieve the same effect by instead removing the \*\*next\*\* node after copying its data into the node that we were asked to delete.

\*\*C++\*\*

void deleteNode(ListNode\* node) {

\*node = \*node->next;

}

But better properly delete the next node:

void deleteNode(ListNode\* node) {

auto next = node->next;

\*node = \*next;

delete next;

}

\*\*Java and C#\*\*

public void deleteNode(ListNode node) {

node.val = node.next.val;

node.next = node.next.next;

}

\*\*Python\*\*

def deleteNode(self, node):

node.val = node.next.val

node.next = node.next.next

\*\*C\*\*

void deleteNode(struct ListNode\* node) {

\*node = \*node->next;

}

But better properly free the next node's memory:

void deleteNode(struct ListNode\* node) {

struct ListNode\* next = node->next;

\*node = \*next;

free(next);

}

\*\*JavaScript\*\*

var deleteNode = function(node) {

node.val = node.next.val;

node.next = node.next.next;

};

\*\*Ruby\*\*

def delete\_node(node)

node.val = node.next.val

node.next = node.next.next

nil

end

# 238,Product of Array Except Self\*\*\*:

**Python\_solution:**

Python solution (Accepted), O(n) time, O(1) space

class Solution:

# @param {integer[]} nums

# @return {integer[]}

def productExceptSelf(self, nums):

p = 1

n = len(nums)

output = []

for i in range(0,n):

output.append(p)

p = p \* nums[i]

p = 1

for i in range(n-1,-1,-1):

output[i] = output[i] \* p

p = p \* nums[i]

return output

# 239,Sliding Window Maximum\*\*\*:

**Python\_solution:**

9 lines Ruby, 11 lines Python, O(n)

Keep indexes of good candidates in deque `d`. The indexes in `d` are from the current window, they're increasing, and their corresponding `nums` are decreasing. Then the first deque element is the index of the largest window value.

For each index `i`:

1. Pop (from the end) indexes of smaller elements (they'll be useless).

2. Append the current index.

3. Pop (from the front) the index `i - k`, if it's still in the deque (it falls out of the window).

4. If our window has reached size `k`, append the current window maximum to the output.

---

\*\*Ruby\*\*

Apparently Ruby doesn't have a deque, so I simulate one with an array, where `s` tells the start index of the queue in the array.

def max\_sliding\_window(nums, k)

d, s = [], 0

out = []

nums.each\_index{ |i|

d.pop while d[s] && nums[d[-1]] < nums[i]

d << i

s += 1 if d[s] == i - k

out << nums[d[s]] if i >= k - 1

}

out

end

---

\*\*Python\*\*

def maxSlidingWindow(self, nums, k):

d = collections.deque()

out = []

for i, n in enumerate(nums):

while d and nums[d[-1]] < n:

d.pop()

d += i,

if d[0] == i - k:

d.popleft()

if i >= k - 1:

out += nums[d[0]],

return out

Last three lines could be this, but for relatively large k it would waste space:

out += nums[d[0]],

return out[k-1:]

# 240,Search a 2D Matrix II\*\*\*:

**Python\_solution:**

6-9 lines C++/Python Solutions with Explanations

Well, the idea is to search from the \*\*top-right\*\* element and then reduce the range for further searching by comparisons between `target` and the current element.

Let's take the matrix in the problem statement as an example.

[

[1, 4, 7, 11, 15],

[2, 5, 8, 12, 19],

[3, 6, 9, 16, 22],

[10, 13, 14, 17, 24],

[18, 21, 23, 26, 30]

]

Suppose we want to search for `12`. We first initialize `r = 0` and `c = 4`. We compare `12` with `matrix[r][c] = matrix[0][4] = 15` and `12 < 15`, so `12` cannot appear in the column of `15` since all elements below `15` are not less than `15`. Thus, we decrease `c` by `1` and reduce the search range by a column. Now we compare `12` with `matrix[r][c] = matrix[0][3] = 11` and `12 > 11`, so `12` cannot appear in the row of `11` since all elements left to `11` are not greater than `11`. Thus, we increase `r` by `1` and reduce the search range by a row. Then we reach `matrix[1][3] = 12 = target` and we are done (return `true`). If we have moved beyond the matrix and have not found the `target`, return `false`.

Putting these together, we will have the following short codes.

----------

\*\*C++\*\*

class Solution {

public:

bool searchMatrix(vector<vector<int>>& matrix, int target) {

int m = matrix.size(), n = matrix[0].size(), r = 0, c = n - 1;

while (r < m && c >= 0) {

if (matrix[r][c] == target) return true;

if (matrix[r][c] > target) c--;

else r++;

}

return false;

}

};

----------

\*\*Python\*\*

class Solution:

# @param {integer[][]} matrix

# @param {integer} target

# @return {boolean}

def searchMatrix(self, matrix, target):

m, n, r, c = len(matrix), len(matrix[0]), 0, n - 1

while r < m and c >= 0:

if matrix[r][c] == target:

return True

if matrix[r][c] > target:

c -= 1

else:

r += 1

return False

# 241,Different Ways to Add Parentheses\*\*\*:

**Python\_solution:**

1-11 lines Python, 9 lines C++

Just doing it...

---

\*\*Solution 1\*\* ... 48 ms

def diffWaysToCompute(self, input):

tokens = re.split('(\D)', input)

nums = map(int, tokens[::2])

ops = map({'+': operator.add, '-': operator.sub, '\*': operator.mul}.get, tokens[1::2])

def build(lo, hi):

if lo == hi:

return [nums[lo]]

return [ops[i](a, b)

for i in xrange(lo, hi)

for a in build(lo, i)

for b in build(i + 1, hi)]

return build(0, len(nums) - 1)

---

\*\*Solution 2\*\* ... 168 ms

One-liner inspired by [Soba](https://leetcode.com/discuss/48410/python-solution-52ms-with-simple-interpretation?show=48432#a48432).

def diffWaysToCompute(self, input):

return [eval(`a`+c+`b`)

for i, c in enumerate(input) if c in '+-\*'

for a in self.diffWaysToCompute(input[:i])

for b in self.diffWaysToCompute(input[i+1:])] or [int(input)]

---

\*\*Solution 3\*\* ... 64 ms

Faster version of solution 2.

def diffWaysToCompute(self, input):

return [a+b if c == '+' else a-b if c == '-' else a\*b

for i, c in enumerate(input) if c in '+-\*'

for a in self.diffWaysToCompute(input[:i])

for b in self.diffWaysToCompute(input[i+1:])] or [int(input)]

---

\*\*Solution 4\*\* ... 188 ms

A code golf version of solution 2.

diffWaysToCompute=d=lambda s,t:[eval(`a`+c+`b`)for i,c in enumerate(t)if

c<'0'for a in s.d(t[:i])for b in s.d(t[i+1:])]or[int(t)]

---

\*\*C++\*\* ... 8 ms

C++ version of solution 3.

vector<int> diffWaysToCompute(string input) {

vector<int> output;

for (int i=0; i<input.size(); i++) {

char c = input[i];

if (ispunct(c))

for (int a : diffWaysToCompute(input.substr(0, i)))

for (int b : diffWaysToCompute(input.substr(i+1)))

output.push\_back(c=='+' ? a+b : c=='-' ? a-b : a\*b);

}

return output.size() ? output : vector<int>{stoi(input)};

}

# 242,Valid Anagram\*\*\*:

**Python\_solution:**

Python solutions (sort and dictionary).

def isAnagram1(self, s, t):

dic1, dic2 = {}, {}

for item in s:

dic1[item] = dic1.get(item, 0) + 1

for item in t:

dic2[item] = dic2.get(item, 0) + 1

return dic1 == dic2

def isAnagram2(self, s, t):

dic1, dic2 = [0]\*26, [0]\*26

for item in s:

dic1[ord(item)-ord('a')] += 1

for item in t:

dic2[ord(item)-ord('a')] += 1

return dic1 == dic2

def isAnagram3(self, s, t):

return sorted(s) == sorted(t)

# 257,Binary Tree Paths\*\*\*:

**Python\_solution:**

Python solutions (dfs+stack, bfs+queue, dfs recursively).

# dfs + stack

def binaryTreePaths1(self, root):

if not root:

return []

res, stack = [], [(root, "")]

while stack:

node, ls = stack.pop()

if not node.left and not node.right:

res.append(ls+str(node.val))

if node.right:

stack.append((node.right, ls+str(node.val)+"->"))

if node.left:

stack.append((node.left, ls+str(node.val)+"->"))

return res

# bfs + queue

def binaryTreePaths2(self, root):

if not root:

return []

res, queue = [], collections.deque([(root, "")])

while queue:

node, ls = queue.popleft()

if not node.left and not node.right:

res.append(ls+str(node.val))

if node.left:

queue.append((node.left, ls+str(node.val)+"->"))

if node.right:

queue.append((node.right, ls+str(node.val)+"->"))

return res

# dfs recursively

def binaryTreePaths(self, root):

if not root:

return []

res = []

self.dfs(root, "", res)

return res

def dfs(self, root, ls, res):

if not root.left and not root.right:

res.append(ls+str(root.val))

if root.left:

self.dfs(root.left, ls+str(root.val)+"->", res)

if root.right:

self.dfs(root.right, ls+str(root.val)+"->", res)

# 258,Add Digits\*\*\*:

**Python\_solution:**

3 methods for python with explains

2. Iteration method

class Solution(object):

def addDigits(self, num):

"""

:type num: int

:rtype: int

"""

while(num >= 10):

temp = 0

while(num > 0):

temp += num % 10

num /= 10

num = temp

return num

1. Digital Root

this method depends on the truth:

N=(a[0] \* 1 + a[1] \* 10 + ...a[n] \* 10 ^n),and a[0]...a[n] are all between [0,9]

we set M = a[0] + a[1] + ..a[n]

and another truth is that:

1 % 9 = 1

10 % 9 = 1

100 % 9 = 1

so N % 9 = a[0] + a[1] + ..a[n]

means N % 9 = M

so N = M (% 9)

as 9 % 9 = 0,so we can make (n - 1) % 9 + 1 to help us solve the problem when n is 9.as N is 9, ( 9 - 1) % 9 + 1 = 9

class Solution(object):

def addDigits(self, num):

"""

:type num: int

:rtype: int

"""

if num == 0 : return 0

else:return (num - 1) % 9 + 1

# 260,Single Number III\*\*\*:

**Python\_solution:**

Easy Python O(n) - O(1) solution

class Solution(object):

def singleNumber(self, nums):

"""

:type nums: List[int]

:rtype: List[int]

"""

xor = 0

a = 0

b = 0

for num in nums:

xor ^= num

mask = 1

while(xor&mask == 0):

mask = mask << 1

for num in nums:

if num&mask:

a ^= num

else:

b ^= num

return [a, b]

# 262,Trips and Users\*\*\*:

**Best\_solution:**

Sharing my solution,

select

t.Request\_at Day,

round(sum(case when t.Status like 'cancelled\_%' then 1 else 0 end)/count(\*),2) Rate

from Trips t

inner join Users u

on t.Client\_Id = u.Users\_Id and u.Banned='No'

where t.Request\_at between '2013-10-01' and '2013-10-03'

group by t.Request\_at

# 263,Ugly Number\*\*\*:

**Python\_solution:**

My python solution

def isUgly(self, num):

"""

:type num: int

:rtype: bool

"""

if num <= 0:

return False

for x in [2, 3, 5]:

while num % x == 0:

num = num / x

return num == 1

# 264,Ugly Number II\*\*\*:

**Python\_solution:**

My expressive Python solution

def nthUglyNumber(self, n):

ugly = [1]

i2, i3, i5 = 0, 0, 0

while n > 1:

u2, u3, u5 = 2 \* ugly[i2], 3 \* ugly[i3], 5 \* ugly[i5]

umin = min((u2, u3, u5))

if umin == u2:

i2 += 1

if umin == u3:

i3 += 1

if umin == u5:

i5 += 1

ugly.append(umin)

n -= 1

return ugly[-1]

# 268,Missing Number\*\*\*:

**Python\_solution:**

1+ lines Ruby, Python, Java, C++

Several different solutions, some with O(1) extra space, some with O(n).

---

\*\*Sum of 0..n minus sum of the given numbers is the missing one.\*\*

These only use O(1) extra space.

Ruby

def missing\_number(nums)

(n = nums.size) \* (n+1) / 2 - nums.reduce(:+)

end

Python

def missingNumber(self, nums):

n = len(nums)

return n \* (n+1) / 2 - sum(nums)

Java

public int missingNumber(int[] nums) {

long n = nums.length;

return (int) (n \* (n+1) / 2 - IntStream.of(nums).sum());

}

C++

int missingNumber(vector<int>& nums) {

long n = nums.size();

return n \* (n+1) / 2 - accumulate(begin(nums), end(nums), 0);

}

Using `long` for Java and C++ to prevent overflow (the n\*(n+1) overflows ints already for n=46341, and then the /2 causes an actual wrong result).

---

\*\*Xor-ing the given numbers and 0..n.\*\*

These use O(n) extra space, but I like them anyway.

Ruby

def missing\_number(nums)

nums.zip(1.step).flatten.reduce(:^)

end

Python

def missingNumber(self, nums):

return reduce(operator.xor, nums + range(len(nums)+1))

---

\*\*Xor-ing with O(1) space\*\*

Saw this from ts before. Xoring 0..n results in [n, 1, n+1, 0][n % 4]. You can also spot the pattern by looking at xors of such ranges, and it's easy to explain as well.

Ruby

def missing\_number(nums)

n = nums.size

nums.reduce(:^) ^ [n, 1, n+1, 0][n % 4]

end

Python

def missingNumber(self, nums):

n = len(nums)

return reduce(operator.xor, nums) ^ [n, 1, n+1, 0][n % 4]

---

\*\*Sum, without formula.\*\*

Java and C++:

int miss = 0, i = 0;

for (int num : nums)

miss += ++i - num;

return miss;

In Java I believe this is safe, overflow might happen but not cause a wrong result (because another overflow will fix it). In C++ I believe it's \*probably safe\* in the same way, except that that behavior isn't defined in the standard(s) but is a de-facto standard anyway. In any case, I could just use 64-bit ints again to be safe.

---

\*\*Set/array difference\*\*

Don't know about Ruby's runtime, might not be linear. Python's sets are hash sets and the difference is linear time on average. Don't know about its worst case, and apparently neither does the [TimeComplexity page](https://wiki.python.org/moin/TimeComplexity).

Ruby

def missing\_number(nums)

((0..nums.size).to\_a - nums)[0]

end

Python

def missingNumber(self, nums):

return (set(range(len(nums)+1)) - set(nums)).pop()

# 273,Integer to English Words\*\*\*:

**Python\_solution:**

Recursive Python

def numberToWords(self, num):

to19 = 'One Two Three Four Five Six Seven Eight Nine Ten Eleven Twelve ' \

'Thirteen Fourteen Fifteen Sixteen Seventeen Eighteen Nineteen'.split()

tens = 'Twenty Thirty Forty Fifty Sixty Seventy Eighty Ninety'.split()

def words(n):

if n < 20:

return to19[n-1:n]

if n < 100:

return [tens[n/10-2]] + words(n%10)

if n < 1000:

return [to19[n/100-1]] + ['Hundred'] + words(n%100)

for p, w in enumerate(('Thousand', 'Million', 'Billion'), 1):

if n < 1000\*\*(p+1):

return words(n/1000\*\*p) + [w] + words(n%1000\*\*p)

return ' '.join(words(num)) or 'Zero'

# 274,H-Index\*\*\*:

**Python\_solution:**

Python O(n lgn) time with sort, O(n) time with O(n) space

Sort

def hIndex(self, citations):

citations.sort()

n = len(citations)

for i in xrange(n):

if citations[i] >= (n-i):

return n-i

return 0

O(n) space, O(n) time

def hIndex(self, citations):

n = len(citations)

citeCount = [0] \* (n+1)

for c in citations:

if c >= n:

citeCount[n] += 1

else:

citeCount[c] += 1

i = n-1

while i >= 0:

citeCount[i] += citeCount[i+1]

if citeCount[i+1] >= i+1:

return i+1

i -= 1

return 0

# 275,H-Index II\*\*\*:

**Python\_solution:**

O(logN)-time O(1)-space Easy Solution with Detailed Explanations (C++/Java/Python)

The basic idea of this solution is to use \*\*binary search\*\* to find the minimum `index` such that

citations[index] >= length(citations) - index

After finding this `index`, the answer is `length(citations) - index`.

This logic is very similar to the C++ function `lower\_bound` or `upper\_bound`.

----------

Complexities:

- Time: O(log \*n\*)

- Space: O(1)

----------

\*\*C++:\*\*

class Solution {

public:

int hIndex(vector<int>& citations) {

int size = citations.size();

int first = 0;

int mid;

int count = size;

int step;

while (count > 0) {

step = count / 2;

mid = first + step;

if (citations[mid] < size - mid) {

first = mid + 1;

count -= (step + 1);

}

else {

count = step;

}

}

return size - first;

}

};

\*\*Java:\*\*

public class Solution {

public int hIndex(int[] citations) {

int len = citations.length;

int first = 0;

int mid;

int count = len;

int step;

while (count > 0) {

step = count / 2;

mid = first + step;

if (citations[mid] < len - mid) {

first = mid + 1;

count -= (step + 1);

}

else {

count = step;

}

}

return len - first;

}

}

\*\*Python:\*\*

class Solution(object):

def hIndex(self, citations):

"""

:type citations: List[int]

:rtype: int

"""

length = len(citations)

first = 0

count = length

while count > 0:

step = count / 2

mid = first + step

if citations[mid] < length - mid:

first = mid + 1

count -= (step + 1)

else:

count = step

return length - first

------------------

\*\*@daviantan1890 @ruichang\*\* Thank you for your comments and discussions.

I am very sure that two-branch binary search is more efficient than three branch binary search.

and (low + high) is not good idea since it may rely on the overflow behavior.

In fact, using `count` `step` `first` `mid` is the standard implement way of C++, so I do not think there are better ways to implement the binary search.

# 278,First Bad Version\*\*\*:

**Python\_solution:**

1-liner in Ruby / Python

\*\*Ruby\*\*

def first\_bad\_version(n)

(1..n).bsearch { |i| is\_bad\_version(i) }

end

Or:

def first\_bad\_version(n)

(1..n).bsearch(&method(:is\_bad\_version))

end

---

\*\*Python\*\*

In Python I was only able to do it with a rather ugly wrapper:

def firstBadVersion(self, n):

return bisect.bisect(type('', (), {'\_\_getitem\_\_': lambda self, i: isBadVersion(i)})(), False, 0, n)

Nicer, more readable version:

def firstBadVersion(self, n):

class Wrap:

def \_\_getitem\_\_(self, i):

return isBadVersion(i)

return bisect.bisect(Wrap(), False, 0, n)

# 279,Perfect Squares\*\*\*:

**Python\_solution:**

Short Python solution using BFS

def numSquares(self, n):

if n < 2:

return n

lst = []

i = 1

while i \* i <= n:

lst.append( i \* i )

i += 1

cnt = 0

toCheck = {n}

while toCheck:

cnt += 1

temp = set()

for x in toCheck:

for y in lst:

if x == y:

return cnt

if x < y:

break

temp.add(x-y)

toCheck = temp

return cnt

The basic idea of this solution is a BSF search for shortest path, take 12 as an example, as shown below, the shortest path is 12-8-4-0:

![exapmle][1]

[1]:http://i.imgur.com/XCoQwiN.png

# 282,Expression Add Operators\*\*\*:

**Python\_solution:**

Clean Python DFS with comments

dfs() parameters:

num: remaining num string

temp: temporally string with operators added

cur: current result of "temp" string

last: last multiply-level number in "temp". if next operator is "multiply", "cur" and "last" will be updated

res: result to return

def addOperators(self, num, target):

res, self.target = [], target

for i in range(1,len(num)+1):

if i == 1 or (i > 1 and num[0] != "0"): # prevent "00\*" as a number

self.dfs(num[i:], num[:i], int(num[:i]), int(num[:i]), res) # this step put first number in the string

return res

def dfs(self, num, temp, cur, last, res):

if not num:

if cur == self.target:

res.append(temp)

return

for i in range(1, len(num)+1):

val = num[:i]

if i == 1 or (i > 1 and num[0] != "0"): # prevent "00\*" as a number

self.dfs(num[i:], temp + "+" + val, cur+int(val), int(val), res)

self.dfs(num[i:], temp + "-" + val, cur-int(val), -int(val), res)

self.dfs(num[i:], temp + "\*" + val, cur-last+last\*int(val), last\*int(val), res)

# 283,Move Zeroes\*\*\*:

**Python\_solution:**

Share my one line python solution

nums.sort(key= lambda x: 1 if x == 0 else 0)

# 284,Peeking Iterator\*\*\*:

**Python\_solution:**

Simple Python Solution

Store the next value outside the iterator. When next is called return the stored value and populate with next value from iterator.

class PeekingIterator(object):

def \_\_init\_\_(self, iterator):

self.iter = iterator

self.temp = self.iter.next() if self.iter.hasNext() else None

def peek(self):

return self.temp

def next(self):

ret = self.temp

self.temp = self.iter.next() if self.iter.hasNext() else None

return ret

def hasNext(self):

return self.temp is not None

# 287,Find the Duplicate Number\*\*\*:

**Python\_solution:**

Python Solution with O(1) space and O(nlogn) time

class Solution(object):

def findDuplicate(self, nums):

low = 0

high = len(nums) - 1

mid = (high + low) / 2

while high - low > 1:

count = 0

for k in nums:

if mid < k <= high:

count += 1

if count > high - mid:

low = mid

else:

high = mid

mid = (high + low) / 2

return high

The difficulty in this problem lies in O(1) space, and many solution using O(n) space can also be accepted by OJ.

The solution is applying bi-search in the range[1, n] by counting the element which falls in sub range(n/2, n].

If the number is bigger than capacity of that sub range, it means the duplicated integer falls in the sub-range.

Otherwise the duplicated integer falls in the other half sub range.

# 289,Game of Life\*\*\*:

**Python\_solution:**

Python solution, easy to understand..

0,2 are "dead", and "dead->live"

1,3 are "live", and "live->dead"

def gameOfLife(self, board):

m,n = len(board), len(board[0])

for i in range(m):

for j in range(n):

if board[i][j] == 0 or board[i][j] == 2:

if self.nnb(board,i,j) == 3:

board[i][j] = 2

else:

if self.nnb(board,i,j) < 2 or self.nnb(board,i,j) >3:

board[i][j] = 3

for i in range(m):

for j in range(n):

if board[i][j] == 2: board[i][j] = 1

if board[i][j] == 3: board[i][j] = 0

def nnb(self, board, i, j):

m,n = len(board), len(board[0])

count = 0

if i-1 >= 0 and j-1 >= 0: count += board[i-1][j-1]%2

if i-1 >= 0: count += board[i-1][j]%2

if i-1 >= 0 and j+1 < n: count += board[i-1][j+1]%2

if j-1 >= 0: count += board[i][j-1]%2

if j+1 < n: count += board[i][j+1]%2

if i+1 < m and j-1 >= 0: count += board[i+1][j-1]%2

if i+1 < m: count += board[i+1][j]%2

if i+1 < m and j+1 < n: count += board[i+1][j+1]%2

return count

# 290,Word Pattern\*\*\*:

**Python\_solution:**

Short in Python

This problem is pretty much equivalent to [Isomorphic Strings](https://leetcode.com/problems/isomorphic-strings/). Let me reuse two old solutions.

From [here](https://leetcode.com/discuss/36438/1-liner-in-python?show=39070#c39070):

def wordPattern(self, pattern, str):

s = pattern

t = str.split()

return map(s.find, s) == map(t.index, t)

Improved version also from there:

def wordPattern(self, pattern, str):

f = lambda s: map({}.setdefault, s, range(len(s)))

return f(pattern) == f(str.split())

From [here](https://leetcode.com/discuss/41379/1-line-in-python?show=41382#a41382):

def wordPattern(self, pattern, str):

s = pattern

t = str.split()

return len(set(zip(s, t))) == len(set(s)) == len(set(t)) and len(s) == len(t)

Thanks to zhang38 for pointing out the need to check len(s) == len(t) here.

# 292,Nim Game\*\*\*:

**Python\_solution:**

2 methods for python

class Solution(object):

def canWinNim(self, n):

"""

:type n: int

:rtype: bool

"""

return bool(n % 4)

class Solution(object):

def canWinNim(self, n):

"""

:type n: int

:rtype: bool

"""

return bool(n & 3)

# 295,Find Median from Data Stream\*\*\*:

**Python\_solution:**

Short simple Java/C++/Python, O(log n) + O(1)

I keep two heaps (or priority queues):

- Max-heap `small` has the smaller half of the numbers.

- Min-heap `large` has the larger half of the numbers.

This gives me direct access to the one or two middle values (they're the tops of the heaps), so getting the median takes O(1) time. And adding a number takes O(log n) time.

Supporting both min- and max-heap is more or less cumbersome, depending on the language, so I simply negate the numbers in the heap in which I want the reverse of the default order. To prevent this from causing a bug with -2<sup>31<\/sup> (which negated is itself, when using 32-bit ints), I use integer types larger than 32 bits.

Using larger integer types also prevents an overflow error when taking the mean of the two middle numbers. I think almost all solutions posted previously have that bug.

\*\*Update:\*\* These are pretty short already, but by now I wrote [even shorter ones](https://leetcode.com/discuss/64910/very-short-o-log-n-o-1).

---

\*\*Java\*\*

class MedianFinder {

private Queue<Long> small = new PriorityQueue(),

large = new PriorityQueue();

public void addNum(int num) {

large.add((long) num);

small.add(-large.poll());

if (large.size() < small.size())

large.add(-small.poll());

}

public double findMedian() {

return large.size() > small.size()

? large.peek()

: (large.peek() - small.peek()) / 2.0;

}

};

Props to [larrywang2014's solution](https://leetcode.com/discuss/64842/32ms-easy-to-understand-java-solution) for making me aware that I can use Queue in the declaration instead of PriorityQueue (that's all I got from him, though (just saying because I just saw he changed his previously longer addNum and it's now equivalent to mine)).

---

\*\*C++\*\*

class MedianFinder {

priority\_queue<long> small, large;

public:

void addNum(int num) {

small.push(num);

large.push(-small.top());

small.pop();

if (small.size() < large.size()) {

small.push(-large.top());

large.pop();

}

}

double findMedian() {

return small.size() > large.size()

? small.top()

: (small.top() - large.top()) / 2.0;

}

};

Big thanks to jianchao.li.fighter for telling me that C++'s priority\_queue is a max-queue (see comments below).

---

\*\*Python\*\*

from heapq import \*

class MedianFinder:

def \_\_init\_\_(self):

self.heaps = [], []

def addNum(self, num):

small, large = self.heaps

heappush(small, -heappushpop(large, num))

if len(large) < len(small):

heappush(large, -heappop(small))

def findMedian(self):

small, large = self.heaps

if len(large) > len(small):

return float(large[0])

return (large[0] - small[0]) / 2.0

# 297,Serialize and Deserialize Binary Tree\*\*\*:

**Python\_solution:**

Recursive preorder, Python and C++, O(n)

\*\*Python\*\*

class Codec:

def serialize(self, root):

def doit(node):

if node:

vals.append(str(node.val))

doit(node.left)

doit(node.right)

else:

vals.append('#')

vals = []

doit(root)

return ' '.join(vals)

def deserialize(self, data):

def doit():

val = next(vals)

if val == '#':

return None

node = TreeNode(int(val))

node.left = doit()

node.right = doit()

return node

vals = iter(data.split())

return doit()

---

\*\*C++\*\*

class Codec {

public:

string serialize(TreeNode\* root) {

ostringstream out;

serialize(root, out);

return out.str();

}

TreeNode\* deserialize(string data) {

istringstream in(data);

return deserialize(in);

}

private:

void serialize(TreeNode\* root, ostringstream& out) {

if (root) {

out << root->val << ' ';

serialize(root->left, out);

serialize(root->right, out);

} else {

out << "# ";

}

}

TreeNode\* deserialize(istringstream& in) {

string val;

in >> val;

if (val == "#")

return nullptr;

TreeNode\* root = new TreeNode(stoi(val));

root->left = deserialize(in);

root->right = deserialize(in);

return root;

}

};

# 299,Bulls and Cows\*\*\*:

**Python\_solution:**

Python 3 lines solution

use `Counter` to count `guess` and `secret` and sum their overlap. Then use `zip` to count `A`.

s, g = Counter(secret), Counter(guess)

a = sum(i == j for i, j in zip(secret, guess))

return '%sA%sB' % (a, sum((s & g).values()) - a)

# 300,Longest Increasing Subsequence\*\*\*:

**Python\_solution:**

Java/Python Binary search O(nlogn) time with explanation

`tails` is an array storing the smallest tail of all increasing subsequences with length `i+1` in `tails[i]`.

For example, say we have `nums = [4,5,6,3]`, then all the available increasing subsequences are:

len = 1 : [4], [5], [6], [3] => tails[0] = 3

len = 2 : [4, 5], [5, 6] => tails[1] = 5

len = 3 : [4, 5, 6] => tails[2] = 6

We can easily prove that tails is a increasing array. Therefore it is possible to do a binary search in tails array to find the one needs update.

Each time we only do one of the two:

(1) if x is larger than all tails, append it, increase the size by 1

(2) if tails[i-1] < x <= tails[i], update tails[i]

Doing so will maintain the tails invariant. The the final answer is just the size.

\*\*Java\*\*

public int lengthOfLIS(int[] nums) {

int[] tails = new int[nums.length];

int size = 0;

for (int x : nums) {

int i = 0, j = size;

while (i != j) {

int m = (i + j) / 2;

if (tails[m] < x)

i = m + 1;

else

j = m;

}

tails[i] = x;

if (i == size) ++size;

}

return size;

}

// Runtime: 2 ms

\*\*Python\*\*

def lengthOfLIS(self, nums):

tails = [0] \* len(nums)

size = 0

for x in nums:

i, j = 0, size

while i != j:

m = (i + j) / 2

if tails[m] < x:

i = m + 1

else:

j = m

tails[i] = x

size = max(i + 1, size)

return size

# Runtime: 48 ms

# 301,Remove Invalid Parentheses\*\*\*:

**Python\_solution:**

Short Python BFS

\*\*Solution 1\*\*

Being lazy and using `eval` for checking:

def removeInvalidParentheses(self, s):

level = {s}

while True:

valid = []

for s in level:

try:

eval('0,' + filter('()'.count, s).replace(')', '),'))

valid.append(s)

except:

pass

if valid:

return valid

level = {s[:i] + s[i+1:] for s in level for i in range(len(s))}

---

Update: Meh, ok, checking it myself isn't that much longer, and it's three times as fast:

\*\*Solution 2\*\*

def removeInvalidParentheses(self, s):

def isvalid(s):

ctr = 0

for c in s:

if c == '(':

ctr += 1

elif c == ')':

ctr -= 1

if ctr < 0:

return False

return ctr == 0

level = {s}

while True:

valid = filter(isvalid, level)

if valid:

return valid

level = {s[:i] + s[i+1:] for s in level for i in range(len(s))}

---

\*\*Solution 3\*\*

Just a mix of the above two.

def removeInvalidParentheses(self, s):

def isvalid(s):

try:

eval('0,' + filter('()'.count, s).replace(')', '),'))

return True

except:

pass

level = {s}

while True:

valid = filter(isvalid, level)

if valid:

return valid

level = {s[:i] + s[i+1:] for s in level for i in range(len(s))}

---

\*\*Solution 4\*\*

Yet another way to do `isvalid`.

def removeInvalidParentheses(self, s):

def isvalid(s):

s = filter('()'.count, s)

while '()' in s:

s = s.replace('()', '')

return not s

level = {s}

while True:

valid = filter(isvalid, level)

if valid:

return valid

level = {s[:i] + s[i+1:] for s in level for i in range(len(s))}

# 306,Additive Number\*\*\*:

**Python\_solution:**

Python solution

Just trying all possibilities for the first two numbers and checking whether the rest fits.

def isAdditiveNumber(self, num):

n = len(num)

for i, j in itertools.combinations(range(1, n), 2):

a, b = num[:i], num[i:j]

if b != str(int(b)):

continue

while j < n:

c = str(int(a) + int(b))

if not num.startswith(c, j):

break

j += len(c)

a, b = b, c

if j == n:

return True

return False

# 309,Best Time to Buy and Sell Stock with Cooldown\*\*\*:

**Python\_solution:**

4-line Python solution, 52 ms

The key is 3 states and 5 edges for state transition. 3 states are `notHold (stock)`, `hold (stock)`, and `notHold\_cooldown`. The initial values of the latter two are negative infinity since they are meaningless, i.e. you won't hold stocks at first and there's no cooldown at first. The 5 edges:

`hold` -----do nothing----->`hold`

`hold` -----sell----->`notHold\_cooldown`

`notHold` -----do nothing -----> `notHold`

`notHold` -----buy-----> `hold`

`notHold\_cooldown` -----do nothing----->`notHold`

def maxProfit(self, prices):

notHold, notHold\_cooldown, hold = 0, float('-inf'), float('-inf')

for p in prices:

hold, notHold, notHold\_cooldown = max(hold, notHold - p), max(notHold, notHold\_cooldown), hold + p

return max(notHold, notHold\_cooldown)

# 310,Minimum Height Trees\*\*\*:

**Python\_solution:**

Share my Accepted BFS Python Code with O(n) Time

The obvious method is to BFS for each node with the complexity of O(n^2) (and will get TLE).

Here is one insight for this problem: the root of MHT is the middle point of the longest path in the tree; hence there are at most two MHT roots.

How to find them? We can BFS from the bottom (leaves) to the top until the last level with <=2 nodes. To build the current level from the previous level, we can monitor the degree of each node. If the node has degree of one, it will be added to the current level. Since it only check the edges once, the complexity is O(n).

def findMinHeightTrees(self, n, edges):

"""

:type n: int

:type edges: List[List[int]]

:rtype: List[int]

"""

if n == 1: return [0]

neighbors = collections.defaultdict(list)

degrees = collections.defaultdict(int)

for u, v in edges:

neighbors[u].append(v)

neighbors[v].append(u)

degrees[u] += 1

degrees[v] += 1

# First find the leaves

preLevel, unvisited = [], set(range(n))

for i in range(n):

if degrees[i] == 1: preLevel.append(i)

while len(unvisited) > 2:

thisLevel = []

for u in preLevel:

unvisited.remove(u)

for v in neighbors[u]:

if v in unvisited:

degrees[v] -= 1

if degrees[v] == 1: thisLevel += [v]

preLevel = thisLevel

return preLevel

# 312,Burst Balloons\*\*\*:

**Python\_solution:**

Python DP N^3 Solutions

Analysis:

We need to find a way to divide the problems. If we start from the first balloon, we can't determine the left/right for the number in each sub-problem, If we start from the last balloon, we can.

We can see the transformation equation is very similar to the one for matrix multiplication.

dp[i][j] = max(dp[i][j], nums[i] \* nums[k] \* nums[j] + dp[i][k] + dp[k][j]) # i < k < j

This is a typical interval DP problem. Because the order of the number extracted matters, we need to do a O(n^3) DP. If we only need to expand the interval to the left or right, we only need to do a O(n^2) DP.

Top-down:

class Solution(object):

def maxCoins(self, nums):

"""

:type nums: List[int]

:rtype: int

"""

nums = [1] + nums + [1]

n = len(nums)

dp = [[0] \* n for \_ in xrange(n)]

def calculate(i, j):

if dp[i][j] or j == i + 1: # in memory or gap < 2

return dp[i][j]

coins = 0

for k in xrange(i+1, j): # find the last balloon

coins = max(coins, nums[i] \* nums[k] \* nums[j] + calculate(i, k) + calculate(k, j))

dp[i][j] = coins

return coins

return calculate(0, n-1)

Bottom-up:

class Solution(object):

def maxCoins(self, nums):

"""

:type nums: List[int]

:rtype: int

"""

nums = [1] + nums + [1] # build the complete array

n = len(nums)

dp = [[0] \* n for \_ in xrange(n)]

for gap in xrange(2, n):

for i in xrange(n-gap):

j = i + gap

for k in xrange(i+1, j):

dp[i][j] = max(dp[i][j], nums[i] \* nums[k] \* nums[j] + dp[i][k] + dp[k][j])

return dp[0][n-1]

# 313,Super Ugly Number\*\*\*:

**Python\_solution:**

Python, generators on a heap

\*\*Solution 1\*\* ... ~1570 ms

Using generators and `heapq.merge`. Too bad there's no `itertools.unique`.

def nthSuperUglyNumber(self, n, primes):

uglies = [1]

def gen(prime):

for ugly in uglies:

yield ugly \* prime

merged = heapq.merge(\*map(gen, primes))

while len(uglies) < n:

ugly = next(merged)

if ugly != uglies[-1]:

uglies.append(ugly)

return uglies[-1]

---

\*\*Solution 2\*\* ... ~1400 ms

Same thing done differently and it's a bit faster.

def nthSuperUglyNumber(self, n, primes):

uglies = [1]

merged = heapq.merge(\*map(lambda p: (u\*p for u in uglies), primes))

uniqed = (u for u, \_ in itertools.groupby(merged))

map(uglies.append, itertools.islice(uniqed, n-1))

return uglies[-1]

# 315,Count of Smaller Numbers After Self\*\*\*:

**Python\_solution:**

3 ways (Segment Tree, Binary Indexed Tree, Binary Search Tree) clean python code

\*\*Segment Tree\*\*

class SegmentTreeNode(object):

def \_\_init\_\_(self, val, start, end):

self.val = val

self.start = start

self.end = end

self.children = []

class SegmentTree(object):

def \_\_init\_\_(self, n):

self.root = self.build(0, n - 1)

def build(self, start, end):

if start > end:

return

root = SegmentTreeNode(0, start, end)

if start == end:

return root

mid = start + end >> 1

root.children = filter(None, [

self.build(start, end)

for start, end in ((start, mid), (mid + 1, end))])

return root

def update(self, i, val, root=None):

root = root or self.root

if i < root.start or i > root.end:

return root.val

if i == root.start == root.end:

root.val += val

return root.val

root.val = sum([self.update(i, val, c) for c in root.children])

return root.val

def sum(self, start, end, root=None):

root = root or self.root

if end < root.start or start > root.end:

return 0

if start <= root.start and end >= root.end:

return root.val

return sum([self.sum(start, end, c) for c in root.children])

class Solution(object):

def countSmaller(self, nums):

hashTable = {v: i for i, v in enumerate(sorted(set(nums)))}

tree, r = SegmentTree(len(hashTable)), []

for i in xrange(len(nums) - 1, -1, -1):

r.append(tree.sum(0, hashTable[nums[i]] - 1))

tree.update(hashTable[nums[i]], 1)

return r[::-1]

\*\*Binary Indexed Tree\*\*

class BinaryIndexedTree(object):

def \_\_init\_\_(self, n):

self.sums = [0] \* (n + 1)

def update(self, i, val):

while i < len(self.sums):

self.sums[i] += 1

i += i & -i

def sum(self, i):

r = 0

while i > 0:

r += self.sums[i]

i -= i & -i

return r

class Solution(object):

def countSmaller(self, nums):

hashTable = {v: i for i, v in enumerate(sorted(set(nums)))}

tree, r = BinaryIndexedTree(len(hashTable)), []

for i in xrange(len(nums) - 1, -1, -1):

r.append(tree.sum(hashTable[nums[i]]))

tree.update(hashTable[nums[i]] + 1, 1)

return r[::-1]

\*\*Binary Search Tree\*\*

class BinarySearchTreeNode(object):

def \_\_init\_\_(self, val):

self.val = val

self.left = None

self.right = None

self.count = 1

self.leftTreeSize = 0

class BinarySearchTree(object):

def \_\_init\_\_(self):

self.root = None

def insert(self, val, root):

if not root:

self.root = BinarySearchTreeNode(val)

return 0

if val == root.val:

root.count += 1

return root.leftTreeSize

if val < root.val:

root.leftTreeSize += 1

if not root.left:

root.left = BinarySearchTreeNode(val)

return 0

return self.insert(val, root.left)

if not root.right:

root.right = BinarySearchTreeNode(val)

return root.count + root.leftTreeSize

return root.count + root.leftTreeSize + self.insert(

val, root.right)

class Solution(object):

def countSmaller(self, nums):

tree = BinarySearchTree()

return [

tree.insert(nums[i], tree.root)

for i in xrange(len(nums) - 1, -1, -1)

][::-1]

# 316,Remove Duplicate Letters\*\*\*:

**Python\_solution:**

Some Python solutions

Solutions inspired by those of others. Simpler but less efficient (all still get accepted, of course, in about 50 to 100 ms, normal for Python).

---

\*\*Solution 1\*\*

Inspired by [lixx2100's explanation](https://leetcode.com/discuss/73761/a-short-o-n-recursive-greedy-solution).

def removeDuplicateLetters(self, s):

for c in sorted(set(s)):

suffix = s[s.index(c):]

if set(suffix) == set(s):

return c + self.removeDuplicateLetters(suffix.replace(c, ''))

return ''

---

\*\*Solution 2\*\*

Inspired by [WHJ425's explanation](https://leetcode.com/discuss/73777/easy-to-understand-iterative-java-solution).

def removeDuplicateLetters(self, s):

result = ''

while s:

i = min(map(s.rindex, set(s)))

c = min(s[:i+1])

result += c

s = s[s.index(c):].replace(c, '')

return result

---

\*\*Solution 3\*\*

Inspired by [halibut735's solution](https://leetcode.com/discuss/73824/short-16ms-solution-using-stack-which-can-optimized-down-4ms).

def removeDuplicateLetters(self, s):

rindex = {c: i for i, c in enumerate(s)}

result = ''

for i, c in enumerate(s):

if c not in result:

while c < result[-1:] and i < rindex[result[-1]]:

result = result[:-1]

result += c

return result

# 318,Maximum Product of Word Lengths\*\*\*:

**Python\_solution:**

Python solution, beats 99.67%

class Solution(object):

def maxProduct(self, words):

d = {}

for w in words:

mask = 0

for c in set(w):

mask |= (1 << (ord(c) - 97))

d[mask] = max(d.get(mask, 0), len(w))

return max([d[x] \* d[y] for x in d for y in d if not x & y] or [0])

# 319,Bulb Switcher\*\*\*:

**Python\_solution:**

One line Python Solution of O(1)

class Solution(object):

def bulbSwitch(self, n):

return int(n\*\*(0.5))

# 321,Create Maximum Number\*\*\*:

**Python\_solution:**

Short Python / Ruby / C++

\*\*Python\*\*

def maxNumber(self, nums1, nums2, k):

def prep(nums, k):

drop = len(nums) - k

out = []

for num in nums:

while drop and out and out[-1] < num:

out.pop()

drop -= 1

out.append(num)

return out[:k]

def merge(a, b):

return [max(a, b).pop(0) for \_ in a+b]

return max(merge(prep(nums1, i), prep(nums2, k-i))

for i in range(k+1)

if i <= len(nums1) and k-i <= len(nums2))

Solved it on my own but now I see others already posted this idea. Oh well, at least it's short, particularly my `merge` function.

The last two lines can be combined, but I find it rather ugly and not worth it:

`for i in range(max(k-len(nums2), 0), min(k, len(nums1))+1))`

---

\*\*Ruby\*\*

def prep(nums, k)

drop = nums.size - k

out = [9]

nums.each do |num|

while drop > 0 && out[-1] < num

out.pop

drop -= 1

end

out << num

end

out[1..k]

end

def max\_number(nums1, nums2, k)

([k-nums2.size, 0].max .. [nums1.size, k].min).map { |k1|

parts = [prep(nums1, k1), prep(nums2, k-k1)]

(1..k).map { parts.max.shift }

}.max

end

---

\*\*C++\*\*

Translated it to C++ as well now. Not as short anymore, but still decent. And C++ allows different functions with the same name, so I chose to do that here to show how nicely the `maxNumber(nums1, nums2, k)` problem can be based on the problems `maxNumber(nums, k)` and `maxNumber(nums1, nums2)`, which would make fine problems on their own.

vector<int> maxNumber(vector<int>& nums1, vector<int>& nums2, int k) {

int n1 = nums1.size(), n2 = nums2.size();

vector<int> best;

for (int k1=max(k-n2, 0); k1<=min(k, n1); ++k1)

best = max(best, maxNumber(maxNumber(nums1, k1),

maxNumber(nums2, k-k1)));

return best;

}

vector<int> maxNumber(vector<int> nums, int k) {

int drop = nums.size() - k;

vector<int> out;

for (int num : nums) {

while (drop && out.size() && out.back() < num) {

out.pop\_back();

drop--;

}

out.push\_back(num);

}

out.resize(k);

return out;

}

vector<int> maxNumber(vector<int> nums1, vector<int> nums2) {

vector<int> out;

while (nums1.size() + nums2.size()) {

vector<int>& now = nums1 > nums2 ? nums1 : nums2;

out.push\_back(now[0]);

now.erase(now.begin());

}

return out;

}

An alternative for `maxNumber(nums1, nums2)`:

vector<int> maxNumber(vector<int> nums1, vector<int> nums2) {

vector<int> out;

auto i1 = nums1.begin(), end1 = nums1.end();

auto i2 = nums2.begin(), end2 = nums2.end();

while (i1 != end1 || i2 != end2)

out.push\_back(lexicographical\_compare(i1, end1, i2, end2) ? \*i2++ : \*i1++);

return out;

}

# 322,Coin Change\*\*\*:

**Python\_solution:**

Fast Python BFS Solution

This solution is inspired by the BFS solution for problem [Perfect Square][1]. Since it is to find the least coin solution (like a shortest path from 0 to amount), using BFS gives results much faster than DP.

class Solution(object):

def coinChange(self, coins, amount):

"""

:type coins: List[int]

:type amount: int

:rtype: int

"""

if amount == 0:

return 0

value1 = [0]

value2 = []

nc = 0

visited = [False]\*(amount+1)

visited[0] = True

while value1:

nc += 1

for v in value1:

for coin in coins:

newval = v + coin

if newval == amount:

return nc

elif newval > amount:

continue

elif not visited[newval]:

visited[newval] = True

value2.append(newval)

value1, value2 = value2, []

return -1

[1]: https://leetcode.com/discuss/62229/short-python-solution-using-bfs

# 324,Wiggle Sort II\*\*\*:

**Python\_solution:**

3 lines Python, with Explanation / Proof

Solution

---

Roughly speaking I put the smaller half of the numbers on the even indexes and the larger half on the odd indexes.

def wiggleSort(self, nums):

nums.sort()

half = len(nums[::2])

nums[::2], nums[1::2] = nums[:half][::-1], nums[half:][::-1]

Alternative, maybe nicer, maybe not:

def wiggleSort(self, nums):

nums.sort()

half = len(nums[::2]) - 1

nums[::2], nums[1::2] = nums[half::-1], nums[:half:-1]

---

\*\*Explanation / Proof\*\*

---

I put the smaller half of the numbers on the even indexes and the larger half on the odd indexes, both from right to left:

Example nums = [1,2,...,7] Example nums = [1,2,...,8]

Small half: 4 . 3 . 2 . 1 Small half: 4 . 3 . 2 . 1 .

Large half: . 7 . 6 . 5 . Large half: . 8 . 7 . 6 . 5

-------------------------- --------------------------

Together: 4 7 3 6 2 5 1 Together: 4 8 3 7 2 6 1 5

I want:

- Odd-index numbers are larger than their neighbors.

Since I put the larger numbers on the odd indexes, clearly I already have:

- Odd-index numbers are larger than \*\*or equal to\*\* their neighbors.

Could they be "equal to"? That would require some number M to appear both in the smaller and the larger half. It would be the largest in the smaller half and the smallest in the larger half. Examples again, where S means some number smaller than M and L means some number larger than M.

Small half: M . S . S . S Small half: M . S . S . S .

Large half: . L . L . M . Large half: . L . L . L . M

-------------------------- --------------------------

Together: M L S L S M S Together: M L S L S L S M

You can see the two M are quite far apart. Of course M could appear more than just twice, for example:

Small half: M . M . S . S Small half: M . S . S . S .

Large half: . L . L . M . Large half: . L . M . M . M

-------------------------- --------------------------

Together: M L M L S M S Together: M L S M S M S M

You can see that with seven numbers, three M are no problem. And with eight numbers, four M are no problem. Should be easy to see that in general, with n numbers, floor(n/2) times M is no problem. Now, if there were more M than that, then my method would fail. But... it would also be impossible:

- If n is even, then having more than n/2 times the same number clearly is unsolvable, because you'd have to put two of them next to each other, no matter how you arrange them.

- If n is odd, then the only way to successfully arrange a number appearing more than floor(n/2) times is if it appears exactly floor(n/2)+1 times and you put them on all the even indexes. And to have the wiggle-property, all the other numbers would have to be larger. But then we wouldn't have an M in both the smaller and the larger half.

So if the input has a valid answer at all, then my code will find one.

# 326,Power of Three\*\*\*:

**Python\_solution:**

Python O(1) Solution 96.6%

class Solution(object):

def isPowerOfThree(self, n):

return n > 0 and 1162261467 % n == 0

# 327,Count of Range Sum\*\*\*:

**Python\_solution:**

O(NlogN) Python solution, binary indexed tree, 268 ms

`Sum[k]` is the sum of first k numbers. O(N^2) solution is

for j in range(n + 1):

for i in range(j):

if lower <= Sum[j] - Sum[i] <= upper: res += 1

This is equal to:

collection = empty

for sum\_j in Sum:

sum\_i\_count = how many sum\_i in this collection that sum\_j - upper <= sum\_i <= sum\_j - lower

res += sum\_i\_count

put sum\_j into this collection

With [Binary indexed tree][1], counting `sum\_i` number is O(logN), putting `sum\_i` into tree is also O(logN). Here we store the index of `sortSum` in the tree. Since index of `BITree` starts from 1, we need `bisect.bisect\_left(sortSum, sum\_j) + 1` for `update()`.

def countRangeSum(self, nums, lower, upper):

n = len(nums)

Sum, BITree = [0] \* (n + 1), [0] \* (n + 2)

def count(x):

s = 0

while x:

s += BITree[x]

x -= (x & -x)

return s

def update(x):

while x <= n + 1:

BITree[x] += 1

x += (x & -x)

for i in range(n):

Sum[i + 1] = Sum[i] + nums[i]

sortSum, res = sorted(Sum), 0

for sum\_j in Sum:

sum\_i\_count = count(bisect.bisect\_right(sortSum, sum\_j - lower)) - count(bisect.bisect\_left(sortSum, sum\_j - upper))

res += sum\_i\_count

update(bisect.bisect\_left(sortSum, sum\_j) + 1)

return res

[1]: https://en.wikipedia.org/wiki/Fenwick\_tree

# 328,Odd Even Linked List\*\*\*:

**Python\_solution:**

Clear Python Solution

def oddEvenList(self, head):

dummy1 = odd = ListNode(0)

dummy2 = even = ListNode(0)

while head:

odd.next = head

even.next = head.next

odd = odd.next

even = even.next

head = head.next.next if even else None

odd.next = dummy2.next

return dummy1.next

# 329,Longest Increasing Path in a Matrix\*\*\*:

**Python\_solution:**

Python solution, memoization dp, 288ms

We can find longest decreasing path instead, the result will be the same. Use `dp` to record previous results and choose the max `dp` value of smaller neighbors.

def longestIncreasingPath(self, matrix):

def dfs(i, j):

if not dp[i][j]:

val = matrix[i][j]

dp[i][j] = 1 + max(

dfs(i - 1, j) if i and val > matrix[i - 1][j] else 0,

dfs(i + 1, j) if i < M - 1 and val > matrix[i + 1][j] else 0,

dfs(i, j - 1) if j and val > matrix[i][j - 1] else 0,

dfs(i, j + 1) if j < N - 1 and val > matrix[i][j + 1] else 0)

return dp[i][j]

if not matrix or not matrix[0]: return 0

M, N = len(matrix), len(matrix[0])

dp = [[0] \* N for i in range(M)]

return max(dfs(x, y) for x in range(M) for y in range(N))

# 330,Patching Array\*\*\*:

**Python\_solution:**

Simple 9-line Python Solution

class Solution(object):

def minPatches(self, nums, n):

"""

:type nums: List[int]

:type n: int

:rtype: int

"""

miss, i, added = 1, 0, 0

while miss <= n:

if i < len(nums) and nums[i] <= miss:

miss += nums[i]

i += 1

else:

miss += miss

added += 1

return added

# 331,Verify Preorder Serialization of a Binary Tree\*\*\*:

**Python\_solution:**

The simplest python solution with explanation (no stack, no recursion)

We just need to remember how many empty slots we have during the process.

Initially we have one ( for the root ).

for each node we check if we still have empty slots to put it in.

- a null node occupies one slot.

- a non-null node occupies one slot before he creates two more. the net gain is one.

----------

class Solution(object):

def isValidSerialization(self, preorder):

"""

:type preorder: str

:rtype: bool

"""

# remember how many empty slots we have

# non-null nodes occupy one slot but create two new slots

# null nodes occupy one slot

p = preorder.split(',')

#initially we have one empty slot to put the root in it

slot = 1

for node in p:

# no empty slot to put the current node

if slot == 0:

return False

# a null node?

if node == '#':

# ocuppy slot

slot -= 1

else:

# create new slot

slot += 1

#we don't allow empty slots at the end

return slot==0

# 332,Reconstruct Itinerary\*\*\*:

**Python\_solution:**

Short Ruby / Python / Java / C++

Just Eulerian path. Greedy DFS, building the route backwards when retreating.

More explanation and example under the codes.

Iterative versions inspired by [fangyang](https://leetcode.com/discuss/84706/share-solution-java-greedy-stack-15ms-with-explanation) (I had only thought of recursion, d'oh).

---

\*\*Ruby\*\*

def find\_itinerary(tickets)

tickets = tickets.sort.reverse.group\_by(&:first)

route = []

visit = -> airport {

visit[tickets[airport].pop()[1]] while (tickets[airport] || []).any?

route << airport

}

visit["JFK"]

route.reverse

end

Iterative version:

def find\_itinerary(tickets)

tickets = tickets.sort.reverse.group\_by(&:first)

route, stack = [], ["JFK"]

while stack.any?

stack << tickets[stack[-1]].pop()[1] while (tickets[stack[-1]] || []).any?

route << stack.pop()

end

route.reverse

end

---

\*\*Python\*\*

def findItinerary(self, tickets):

targets = collections.defaultdict(list)

for a, b in sorted(tickets)[::-1]:

targets[a] += b,

route = []

def visit(airport):

while targets[airport]:

visit(targets[airport].pop())

route.append(airport)

visit('JFK')

return route[::-1]

Iterative version:

def findItinerary(self, tickets):

targets = collections.defaultdict(list)

for a, b in sorted(tickets)[::-1]:

targets[a] += b,

route, stack = [], ['JFK']

while stack:

while targets[stack[-1]]:

stack += targets[stack[-1]].pop(),

route += stack.pop(),

return route[::-1]

---

\*\*Java\*\*

public List<String> findItinerary(String[][] tickets) {

for (String[] ticket : tickets)

targets.computeIfAbsent(ticket[0], k -> new PriorityQueue()).add(ticket[1]);

visit("JFK");

return route;

}

Map<String, PriorityQueue<String>> targets = new HashMap<>();

List<String> route = new LinkedList();

void visit(String airport) {

while(targets.containsKey(airport) && !targets.get(airport).isEmpty())

visit(targets.get(airport).poll());

route.add(0, airport);

}

Iterative version:

public List<String> findItinerary(String[][] tickets) {

Map<String, PriorityQueue<String>> targets = new HashMap<>();

for (String[] ticket : tickets)

targets.computeIfAbsent(ticket[0], k -> new PriorityQueue()).add(ticket[1]);

List<String> route = new LinkedList();

Stack<String> stack = new Stack<>();

stack.push("JFK");

while (!stack.empty()) {

while (targets.containsKey(stack.peek()) && !targets.get(stack.peek()).isEmpty())

stack.push(targets.get(stack.peek()).poll());

route.add(0, stack.pop());

}

return route;

}

---

\*\*C++\*\*

vector<string> findItinerary(vector<pair<string, string>> tickets) {

for (auto ticket : tickets)

targets[ticket.first].insert(ticket.second);

visit("JFK");

return vector<string>(route.rbegin(), route.rend());

}

map<string, multiset<string>> targets;

vector<string> route;

void visit(string airport) {

while (targets[airport].size()) {

string next = \*targets[airport].begin();

targets[airport].erase(targets[airport].begin());

visit(next);

}

route.push\_back(airport);

}

---

\*\*Explanation\*\*

First keep going forward until you get stuck. That's a good main path already. Remaining tickets form cycles which are found on the way back and get merged into that main path. By writing down the path backwards when retreating from recursion, merging the cycles into the main path is easy - the end part of the path has already been written, the start part of the path hasn't been written yet, so just write down the cycle now and then keep backwards-writing the path.

Example:

![enter image description here][1]

From JFK we first visit JFK -> A -> C -> D -> A. There we're stuck, so we write down A as the end of the route and retreat back to D. There we see the unused ticket to B and follow it: D -> B -> C -> JFK -> D. Then we're stuck again, retreat and write down the airports while doing so: Write down D before B, then JFK before D, etc. When we're back from our cycle at D, the written route is D -> B -> C -> JFK -> D -> A. Then we retreat further along the original path, prepending C, A and finally JFK to the route, ending up with the route JFK -> A -> C -> D -> B -> C -> JFK -> D -> A.

[1]: http://www.stefan-pochmann.info/misc/reconstruct-itinerary.png

# 334,Increasing Triplet Subsequence\*\*\*:

**Python\_solution:**

Python Easy O(n) Solution

Start with the maximum numbers for the first and second element. Then:

(1) Find the first smallest number in the 3 subsequence

(2) Find the second one greater than the first element, reset the first one if it's smaller

def increasingTriplet(nums):

first = second = float('inf')

for n in nums:

if n <= first:

first = n

elif n <= second:

second = n

else:

return True

return False

# 335,Self Crossing\*\*\*:

**Python\_solution:**

Another python...

Checking out every six pack.

\*\*Solution 1\*\*

def isSelfCrossing(self, x):

return any(d >= b > 0 and (a >= c or a >= c-e >= 0 and f >= d-b)

for a, b, c, d, e, f in ((x[i:i+6] + [0] \* 6)[:6]

for i in xrange(len(x))))

\*\*Solution 2\*\*

def isSelfCrossing(self, x):

b = c = d = e = 0

for a in x:

if d >= b > 0 and (a >= c or a >= c-e >= 0 and f >= d-b):

return True

b, c, d, e, f = a, b, c, d, e

return False

\*\*Explanation\*\*

b b

+----------------+ +----------------+

| | | |

| | | | a

c | | c | |

| | a | | f

+-----------> | | | <----+

d | | | | e

| | |

+-----------------------+

d

Draw a line of length `a`. Then draw further lines of lengths `b`, `c`, etc. How does the `a`-line get crossed? From the left by the `d`-line or from the right by the `f`-line, see the above picture. I just encoded the criteria for actually crossing it.

Two details:

- In both cases, `d` needs to be at least `b`. In the first case to cross the `a`-line directly, and in the second case to get behind it so that the `f`-line can cross it. So I factored out `d >= b`.

- The "special case" of the `e`-line stabbing the `a`-line from below is covered because in that case, the `f`-line "crosses" it (note that even if there is no actual `f`-line, my code uses `f = 0` and thus still finds that "crossing").

# 336,Palindrome Pairs\*\*\*:

**Python\_solution:**

Python solution~

wordict = {}

res = []

for i in range(len(words)):

wordict[words[i]] = i

for i in range(len(words)):

for j in range(len(words[i])+1):

tmp1 = words[i][:j]

tmp2 = words[i][j:]

if tmp1[::-1] in wordict and wordict[tmp1[::-1]]!=i and tmp2 == tmp2[::-1]:

res.append([i,wordict[tmp1[::-1]]])

if j!=0 and tmp2[::-1] in wordict and wordict[tmp2[::-1]]!=i and tmp1 == tmp1[::-1]:

res.append([wordict[tmp2[::-1]],i])

return res

# 337,House Robber III\*\*\*:

**Python\_solution:**

6-line Python solution, return (subtree max money if not rob this node, subtree max money)

def rob(self, root):

def dfs(node):

# return (subtree max money if not rob this node, subtree max money)

if not node: return 0, 0

max\_l\_ignore, max\_l = dfs(node.left)

max\_r\_ignore, max\_r = dfs(node.right)

return max\_l + max\_r, max(max\_l + max\_r, node.val + max\_l\_ignore + max\_r\_ignore)

return dfs(root)[1]

# 338,Counting Bits\*\*\*:

**Python\_solution:**

Simple Python Solution

Code works by constantly extending a list with itself but with the values incremented by 1.

def countBits(self, num):

"""

:type num: int

:rtype: List[int]

"""

iniArr = [0]

if num > 0:

amountToAdd = 1

while len(iniArr) < num + 1:

iniArr.extend([x+1 for x in iniArr])

return iniArr[0:num+1]

Simple python solution that runs in O(n) time. Let me know if there are any ways to improve it.

# 341,Flatten Nested List Iterator\*\*\*:

**Python\_solution:**

Real iterator in Python, Java, C++

An iterator shouldn't copy the entire data but just iterate over the original data structure.

I keep the current progress in a stack. My `hasNext` tries to find an integer. My `next` returns it and moves on. I call `hasNext` in `next` because `hasNext` is optional. Some user of the iterator might call only `next` and never `hasNext`, e.g., if they know how many integers are in the structure or if they want to handle the ending with exception handling.

---

\*\*Python\*\*

Using a stack of [list, index] pairs.

class NestedIterator(object):

def \_\_init\_\_(self, nestedList):

self.stack = [[nestedList, 0]]

def next(self):

self.hasNext()

nestedList, i = self.stack[-1]

self.stack[-1][1] += 1

return nestedList[i].getInteger()

def hasNext(self):

s = self.stack

while s:

nestedList, i = s[-1]

if i == len(nestedList):

s.pop()

else:

x = nestedList[i]

if x.isInteger():

return True

s[-1][1] += 1

s.append([x.getList(), 0])

return False

---

\*\*Java\*\*

Using a stack of ListIterators.

public class NestedIterator implements Iterator<Integer> {

public NestedIterator(List<NestedInteger> nestedList) {

lists = new Stack<>();

lists.push(nestedList.listIterator());

}

public Integer next() {

hasNext();

return lists.peek().next().getInteger();

}

public boolean hasNext() {

while (!lists.empty()) {

if (!lists.peek().hasNext()) {

lists.pop();

} else {

NestedInteger x = lists.peek().next();

if (x.isInteger())

return lists.peek().previous() == x;

lists.push(x.getList().listIterator());

}

}

return false;

}

private Stack<ListIterator<NestedInteger>> lists;

}

---

\*\*C++\*\*

Using stacks of begin and end iterators.

class NestedIterator {

public:

NestedIterator(vector<NestedInteger> &nestedList) {

begins.push(nestedList.begin());

ends.push(nestedList.end());

}

int next() {

hasNext();

return (begins.top()++)->getInteger();

}

bool hasNext() {

while (begins.size()) {

if (begins.top() == ends.top()) {

begins.pop();

ends.pop();

} else {

auto x = begins.top();

if (x->isInteger())

return true;

begins.top()++;

begins.push(x->getList().begin());

ends.push(x->getList().end());

}

}

return false;

}

private:

stack<vector<NestedInteger>::iterator> begins, ends;

};

# 342,Power of Four\*\*\*:

**Python\_solution:**

Python one line solution with explanations

def isPowerOfFour(self, num):

return num != 0 and num &(num-1) == 0 and num & 1431655765== num

Consider the valid numbers within 32 bit, and turn them into binary form, they are:

1

100

10000

1000000

100000000

10000000000

1000000000000

100000000000000

10000000000000000

1000000000000000000

100000000000000000000

10000000000000000000000

1000000000000000000000000

100000000000000000000000000

10000000000000000000000000000

1000000000000000000000000000000

Any other number not it the list should be considered as invalid.

So if you XOR them altogether, you will get a mask value, which is:

1010101010101010101010101010101 (1431655765)

Any number which is power of 4, it should be power of 2, I use num &(num-1) == 0 to make sure of that.

Obviously 0 is not power of 4, I have to check it.

and finally I need to check that if the number 'AND' the mask value is itself, to make sure it's in the list above.

here comes the final code:

return num != 0 and num &(num-1) == 0 and num & 1431655765== num

# 343,Integer Break\*\*\*:

**Best\_solution:**

Why factor 2 or 3? The math behind this problem.

I saw many solutions were referring to factors of 2 and 3. But why these two magic numbers? Why other factors do not work?

Let's study the math behind it.

For convenience, say \*\*n\*\* is sufficiently large and can be broken into any smaller real positive numbers. We now try to calculate which real number generates the largest product.

Assume we break \*\*n\*\* into \*\*(n / x)\*\* \*\*x\*\*'s, then the product will be \*\*x<sup>n/x<\/sup>\*\*, and we want to maximize it.

Taking its derivative gives us \*\*n \* x<sup>n/x-2<\/sup> \* (1 - ln(x))\*\*.

The derivative is positive when \*\*0 < x < e\*\*, and equal to \*\*0\*\* when \*\*x = e\*\*, then becomes negative when \*\*x > e\*\*,

which indicates that the product increases as \*\*x\*\* increases, then reaches its maximum when \*\*x = e\*\*, then starts dropping.

This reveals the fact that if \*\*n\*\* is sufficiently large and we are allowed to break \*\*n\*\* into real numbers,

the best idea is to break it into nearly all \*\*e\*\*'s.

On the other hand, if \*\*n\*\* is sufficiently large and we can only break \*\*n\*\* into integers, we should choose integers that are closer to \*\*e\*\*.

The only potential candidates are \*\*2\*\* and \*\*3\*\* since \*\*2 < e < 3\*\*, but we will generally prefer \*\*3\*\* to \*\*2\*\*. Why?

Of course, one can prove it based on the formula above, but there is a more natural way shown as follows.

\*\*6 = 2 + 2 + 2 = 3 + 3\*\*. But \*\*2 \* 2 \* 2 < 3 \* 3\*\*.

Therefore, if there are three \*\*2\*\*'s in the decomposition, we can replace them by two \*\*3\*\*'s to gain a larger product.

All the analysis above assumes \*\*n\*\* is significantly large. When \*\*n\*\* is small (say \*\*n <= 10\*\*), it may contain flaws.

For instance, when \*\*n = 4\*\*, we have \*\*2 \* 2 > 3 \* 1\*\*.

To fix it, we keep breaking \*\*n\*\* into \*\*3\*\*'s until \*\*n\*\* gets smaller than \*\*10\*\*, then solve the problem by brute-force.

# 344,Reverse String\*\*\*:

**Python\_solution:**

Python solution

Python:

class Solution(object):

def reverseString(self, s):

"""

:type s: str

:rtype: str

"""

return s[::-1]

# 345,Reverse Vowels of a String\*\*\*:

**Python\_solution:**

1-2 lines Python/Ruby

\*\*Ruby\*\*

def reverse\_vowels(s)

vowels = s.scan(/[aeiou]/i)

s.gsub(/[aeiou]/i) { vowels.pop }

end

---

\*\*Python\*\*

def reverseVowels(self, s):

vowels = re.findall('(?i)[aeiou]', s)

return re.sub('(?i)[aeiou]', lambda m: vowels.pop(), s)

---

It's possible in one line, but I don't really like it:

def reverseVowels(self, s):

return re.sub('(?i)[aeiou]', lambda m, v=re.findall('(?i)[aeiou]', s): v.pop(), s)

---

Another version, finding replacement vowels on the fly instead of collecting all in advance:

def reverseVowels(self, s):

vowels = (c for c in reversed(s) if c in 'aeiouAEIOU')

return re.sub('(?i)[aeiou]', lambda m: next(vowels), s)

# 347,Top K Frequent Elements\*\*\*:

**Python\_solution:**

1-line Python Solution using Counter with explanation

import collections

class Solution(object):

def topKFrequent(self, nums, k):

"""

:type nums: List[int]

:type k: int

:rtype: List[int]

"""

# Use Counter to extract the top k frequent elements

# most\_common(k) return a list of tuples, where the first item of the tuple is the element,

# and the second item of the tuple is the count

# Thus, the built-in zip function could be used to extract the first item from the tuples

return zip(\*collections.Counter(nums).most\_common(k))[0]

# 349,Intersection of Two Arrays\*\*\*:

**Python\_solution:**

Python code, 3 lines using set

class Solution(object):

def intersection(self, nums1, nums2):

"""

:type nums1: List[int]

:type nums2: List[int]

:rtype: List[int]

"""

nums1=set(nums1)

nums2=set(nums2)

return list(nums1&nums2)

# 350,Intersection of Two Arrays II\*\*\*:

**Python\_solution:**

2 lines in Python

from collections import Counter

class Solution(object):

def intersect(self, nums1, nums2):

c1, c2 = Counter(nums1), Counter(nums2)

return sum([[num] \* min(c1[num], c2[num]) for num in c1 & c2], [])

# 352,Data Stream as Disjoint Intervals\*\*\*:

**Python\_solution:**

Share my python solution using heap

Since there is no standard TreeMap library for python, I am implementing this structure with a min heap.

The idea is straight froward:

Append interval to heap when addNum called

Merge intervals when getIntervals called

class SummaryRanges(object):

def \_\_init\_\_(self):

self.intervals = []

def addNum(self, val):

heapq.heappush(self.intervals, (val, Interval(val, val)))

def getIntervals(self):

stack = []

while self.intervals:

idx, cur = heapq.heappop(self.intervals)

if not stack:

stack.append((idx, cur))

else:

\_, prev = stack[-1]

if prev.end + 1 >= cur.start:

prev.end = max(prev.end, cur.end)

else:

stack.append((idx, cur))

self.intervals = stack

return list(map(lambda x: x[1], stack))

# 354,Russian Doll Envelopes\*\*\*:

**Python\_solution:**

Python O(nlogn) O(n) solution, beats 97%, with explanation

class Solution(object):

def maxEnvelopes(self, envs):

def liss(envs):

def lmip(envs, tails, k):

b, e = 0, len(tails) - 1

while b <= e:

m = (b + e) >> 1

if envs[tails[m]][1] >= k[1]:

e = m - 1

else:

b = m + 1

return b

tails = []

for i, env in enumerate(envs):

idx = lmip(envs, tails, env)

if idx >= len(tails):

tails.append(i)

else:

tails[idx] = i

return len(tails)

def f(x, y):

return -1 if (x[0] < y[0] or x[0] == y[0] and x[1] > y[1]) else 1

envs.sort(cmp=f)

return liss(envs)

# Runtime: 100ms

The idea is to order the envelopes and then calculate the longest increasing subsequence (LISS). We first sort the envelopes by width, and we also make sure that when the width is the same, the envelope with greater height comes first. Why? This makes sure that when we calculate the LISS, we don't have a case such as [3, 4] [3, 5] (we could increase the LISS but this would be wrong as the width is the same. It can't happen when [3, 5] comes first in the ordering).

We could calculate the LISS using the standard DP algorithm (quadratic runtime), but we can just use the tails array method with a twist: we store the index of the tail, and we do leftmost insertion point as usual to find the right index in `nlogn` time. Why not rightmost? Think about the case [1, 1], [1, 1], [1, 1].

# 357,Count Numbers with Unique Digits\*\*\*:

**Python\_solution:**

O(1) Python solution with cheat sheet :P

cheat\_sheet = [1, 10, 91, 739, 5275, 32491, 168571, 712891, 2345851, 5611771]

class Solution(object):

def countNumbersWithUniqueDigits(self, n):

return cheat\_sheet[n] if n<11 else cheat\_sheet[10]

# 365,Water and Jug Problem\*\*\*:

**Python\_solution:**

Python solution gcd

class Solution(object):

def canMeasureWater(self, x, y, z):

"""

:type x: int

:type y: int

:type z: int

:rtype: bool

"""

a,b=x,y

while y:

r=x%y

x=y

y=r

return bool(not z or (x and z<=a+b and not z%x))

# 367,Valid Perfect Square\*\*\*:

**Python\_solution:**

Python solution using Newton's method

class Solution(object):

def isPerfectSquare(self, num):

"""

:type num: int

:rtype: bool

"""

if num < 0: return False

if num <= 1: return True

n = num/2 # start guessing using n = num/2

while n\*n!= num:

inc = (num-n\*n)/(2\*n)

n += inc

if -1 <= inc <= 1: break

if n\*n < num: n+=1

if n\*n > num: n-=1

return n\*n == num

f(x) = x^2 (find x that f(x) = num)

f'(x) = 2\*x

start process with x = n (any positive number)

if f(x) != num, update x = x + (num - f(x))/f'(x) = x + (num - n^2)/(2n)

# 368,Largest Divisible Subset\*\*\*:

**Python\_solution:**

4 lines in Python

def largestDivisibleSubset(self, nums):

S = {-1: set()}

for x in sorted(nums):

S[x] = max((S[d] for d in S if x % d == 0), key=len) | {x}

return list(max(S.values(), key=len))

My `S[x]` is the largest subset with `x` as the largest element, i.e., the subset of all divisors of `x` in the input. With `S[-1] = emptyset` as useful base case. Since divisibility is transitive, a multiple `x` of some divisor `d` is also a multiple of all elements in `S[d]`, so it's not necessary to explicitly test divisibility of `x` by all elements in `S[d]`. Testing `x % d` suffices.

While storing entire subsets isn't super efficient, it's also not that bad. To extend a subset, the new element must be divisible by all elements in it, meaning it must be at least twice as large as the largest element in it. So with the 31-bit integers we have here, the largest possible set has size 31 (containing all powers of 2).

# 371,Sum of Two Integers\*\*\*:

**Python\_solution:**

Python code as follows:

class Solution(object):

def getSum(self, a, b):

"""

:type a: int

:type b: int

:rtype: int

"""

MAX\_INT = 0x7FFFFFFF

MIN\_INT = 0x80000000

MASK = 0x100000000

while b:

a, b = (a ^ b) % MASK, ((a & b) << 1) % MASK

return a if a <= MAX\_INT else (a % MIN\_INT) - MIN\_INT