# sQ811 HASHMAP

Hashmap in python, you can use a dict

Dict = {}

>>> d = {'key':'value'}

>>> print(d)

{'key': 'value'}

>>> d['mynewkey'] = 'mynewvalue'

>>> print(d)

{'mynewkey': 'mynewvalue', 'key': 'value'}

SUBSTRING in python

In general, everything before, or starting from and including the first.

>>> x = "Hello World!"

>>> x[2:]

'llo World!'

>>> x[:2]

'He'

>>> x[:-2]

'Hello Worl'

>>> x[-2:]

'd!'

>>> x[2:-2]

'llo Worl'

# Q763 Partition Labels

String loc and substring, with rfind and rindex, which find the last occurrence of a substring

# Q416 Battleships, DFS

# Q807 Max Increase to keep city sky

Just 2D array iteration

range(stop)

* stop: Number of integers (whole numbers) to generate, starting from zero. eg. range(3) == [0, 1, 2].

range([start], stop[, step])

* start: Starting number of the sequence.
* stop: Generate numbers up to, but not including this number.
* step: Difference between each number in the sequence.
* def maxIncreaseKeepingSkyline(self, grid):
* row, col = map(max, grid), map(max, zip(\*grid))
* return sum(min(i, j) **for** i **in** row **for** j **in** col) - sum(map(sum, grid))

Expression oriented functions of Python provides are:

1. map(aFunction, aSequence)
2. filter(aFunction, aSequence)
3. reduce(aFunction, aSequence)
4. lambda
5. list comprehension

# Q344 Reverse String

# Extended Slices

For example, you can now easily extract the elements of a list that have even 1indexes:

>>> L = range(10)

>>> L[::2]

[0, 2, 4, 6, 8]

Negative values also work to make a copy of the same list in reverse order:

>>> L[::-1]

[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

# Q338 Counting Bits Dynamic programming problem

**Index :** 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21

**num :** **0 1 1 2** 1 2 2 3 1 2 2 3 **2 3 3 4 1 2 2 3 2 3 3 4**

dp[0] = 0;

dp[1] = dp[1-1] + 1;

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

dp[3] = dp[3-2] +1;

dp[4] = dp[4-4] + 1;

dp[5] = dp[5-4] + 1;

dp[6] = dp[6-4] + 1;

dp[7] = dp[7-4] + 1;

dp[8] = dp[8-8] + 1;

# 191 Number of 1 Bits

bin(n).count('1')

# 791 Custom Sort String

Character array with count method,

Run time O(N^2)

You can also use count the number of times a character appears.

# 442 Find All Duplicates in an Array

Using the input array as a hash function, by changing the value to negative to indicate that this spot has been visited.

# 406 Queue Reconstruciton by height

Dynamic programming, find the position for the shortest person first,

Then second shortest.

Second solution,

Hash Map, hash on height,

# Q496 Next Greater Element Stack!!!

Solve by creating a dict for each value, since there is no duplicates, and 1 is subset of 2.

Actually, use stack!!!!

Used array to build a stack structure, and array [-1] is the top of the stack, array has append and pop()

diction, st = {} , []

for i in nums:

if(len(st) == 0 ):

st.append(i)

elif(i < st[-1]):

st.append(i)

else:

while st and st[-1] < i:

diction[st.pop()] = i

st.append(i)

# Q75 Sort Colors Dutch partitioning problem

The basic idea of quick sort

Sort 0 ,1 ,2 counting sort is 2n

Need a n solution.

Use 3 different point to classfy the unknown items to the correct posiotn, using swap

# Q162 find peak element binary search!!!

First define left and right, which is 0 and length -1

Then depending on the condition left= mid +1 or right = mid -1

**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

# Q240 Search a 2D Matrix 2

First solution, use binary search, left, right mid for each solution, then this is n\*n

But, the better solution is to go through column and row at the same time.

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 15since 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.

# Q49 Group Anagrams

The hashmap in python, dict can have tuples as keys, which means, (a,b,c) can be a key

And the tuple() function tuple('abc') returns ('a', 'b', 'c') and tuple([1, 2, 3]) returns (1, 2, 3).

**tuple**([iterable])

\*\*\* you have to sort the strings first !

Following is the syntax for **get()** method −

dict.get(key, default = None)

## **Parameters**

* **key** − This is the Key to be searched in the dictionary.
* **default** − This is the Value to be returned in case key does not exist.

# 300 Longest increasing subsequences DP and binary search

Using dp, for each new added in value, check all the ones before it, and if this value is larger than anyone before it ,find the max LIS then +1

# 136 single number

dic = {}

**for** num **in** nums:

dic[num] = dic.get(num, 0)+1

**for** key, val **in** dic.items():

**if** val == 1:

**return** key

Hash map, iterate with key and vals

Or you can use xor and the only one number with one copy will be the result.

# 104 Maximum Depth of Binary Tree

When you are doing recursion, you need to use self.

if(root.left != None):

dpl += self.maxDepth(root.left)

if(root.right != None):

dpr += self.maxDepth(root.right)

# Google interview requirements

* Construct / traverse data structures
* Implement system routines
* Distill large data sets to single values
* Transform one data set to another

## Sorting and hashing

Handling obscenely large amounts of data

quicksort and merge sort

Merge sort can be highly useful in situations where quicksort is impractical

HeapSort

**Data Structures**

We recommend you know about the most famous classes of NP-complete problems, such as traveling salesman and the knapsack problem.

Trees, basic tree construction, traversal and manipulation algorithms, hash tables, stacks, arrays, linked lists, priority queues.

**Trees**

We recommend you know about basic tree construction, traversal and manipulation algorithms. You should be familiar with binary trees, n-ary trees, and trie-trees at the very least. You should be familiar with at least one flavor of balanced binary tree, whether it's a red/black tree, a splay tree or an AVL tree. You’ll want to know how it's implemented. You should know about tree traversal algorithms: BFS and DFS, and know the difference between inorder, postorder and preorder.

**Min/Max Heaps**

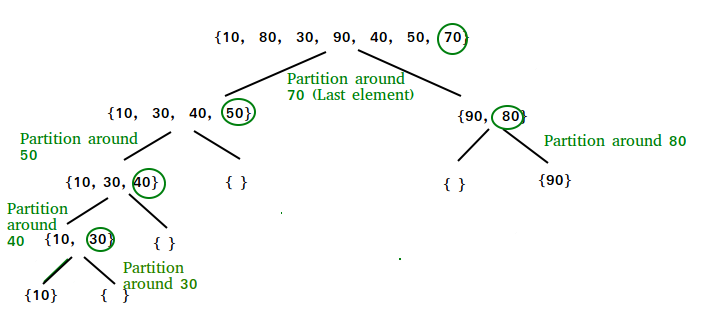
Heaps are incredibly useful. Understand their application and O() characteristics. We probably won’t ask you to implement one during an interview, but you should know when it makes sense to use one.

# Q88 merge sorted array

def mergeSort(alist):  
 if len(alist) > 1:  
 mid = len(alist)/2  
 lefthalf = alist[:mid]  
 righthalf = alist[mid:]  
  
 #recursion  
 mergeSort(lefthalf)  
 mergeSort(righthalf)  
  
 i=0  
 j=0  
 k=0  
  
 while i < len(lefthalf) and j < len(righthalf):  
 if lefthalf[i] < righthalf[j]:  
 alist[k] = lefthalf[i]  
 i = i +1  
 else:  
 alist[k] = righthalf[j]  
 j=j+1  
 k=k+1  
  
 while i < len(lefthalf):  
 alist[k] = lefthalf[i]  
 i=i+1  
 k=k+1  
 while j < len(righthalf):  
 alist[k] = righthalf[j]  
 j = j+1  
 k = k+1  
 print("Merging ", alist)

# **QuickSort**

Like [Merge Sort](http://quiz.geeksforgeeks.org/merge-sort/), QuickSort is a Divide and Conquer algorithm.



def quickSort(alist,sIndex,eIndex):  
  
 if(sIndex < eIndex):  
 pi = partition(alist,sIndex,eIndex)  
  
 quickSort(alist,sIndex,pi-1)  
 quickSort(alist,pi+1,eIndex)  
  
def partition(alist,sIndex,EIndex):  
 pivotV = alist[EIndex] # always choosing the last value in the list be the pivot  
 i = sIndex -1 # Index of smaller element  
  
 for j in range(sIndex,EIndex):  
 if(alist[j] <= pivotV):  
 i += 1  
 temp = alist[i]  
 alist[i] = alist[j]  
 alist[j] = temp  
  
 temp = alist[i+1]  
 alist[i+1] = alist[EIndex]  
 alist[EIndex] = temp  
 return i+1  
  
alist = [54,26,93,17,77,31,44,55,20]#[17, 20, 26, 31, 44, 54, 55, 77, 93]  
quickSort(alist,0,8)  
print(alist)

# Google interview questions

# 280. Wiggle Sort

if nums[i - 1] > nums[i], then we swap element at i -1 and i. Due to previous wiggled elements (nums[i - 2] >= nums[i - 1]), we know after swap the sequence is ensured to be nums[i - 2] > nums[i - 1] < nums[i], which is wiggled.

Finding out the condition of the wiggle is important.

# 4. Median of Two Sorted Arrays

Merge sort

# 20. Valid Parentheses stack!!

stack = []

dic = {'}':'{',']':'[',')':'('}

for i in s:

if i in dic.values():

stack.append(i)

elif i in dic.keys():

if(stack == [] or stack.pop() != dic[i]):

return False

else:

return False

return stack == []

# 66. Plus One

Used runtime O(n), and modify the array in place with extraspace of O(1)

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

if(digits[-1\*i] == 9):

digits[-1\*i] = 0

if(i==len(digits)):

return [1] + digits

else:

digits[-1\*i] += 1

break

return digits

# 535. Encode and Decode TinyURL Hashmap with class

Def \_\_init\_\_(self):

Self.dic ={}

# 155 min stack

Using hashmap to store both the value and the minvalue, so if you remove TOS, the next element still have the minvalue.

# 23 Merge k sorted lists

Does merge sort ‘s merge function multiple time.,or use a minheap

# 42 Trapping Rain water

Since the limit of the water is restricted by either the leftmax or the right max, so you can scan from both side, but only case you have to consider is that if two block are together and there is no space in between, but there is still a difference, then it will not work from only on side, so you should scan from both side and meet there.

# 238 products of array except self

Can the array from two different two, first from 0 to the number, and keep the value of all the multiple to that number, then do everything in reverse order.

# Google Interview

## **Array and Strings**

String manipulation problems are in the same category as array, because string is represented as an array of characters internally. Array problems usually does not require knowledge of advanced data structures, so just basic data structures such as Hash Table and basic techniques like Two Pointers should suffice.

Google likes to test your ability to think at large scale by asking variation of problems represented in data stream model. For example, instead of giving you an integer array, you are given a stream of integers and all integers are too large to be fit in memory.

# Q Plus one Array iteration

Goal is to plus one to the number in the array, and the solution is O(N) and start from the end of the array by using s[-1]

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

if(digits[-1\*i] == 9):

digits[-1\*i] = 0

if(i==len(digits)):

return [1] + digits

else:

digits[-1\*i] += 1

break

return digits

# Q Trapping Rain Water 2 pointer questions

Start from both the front and the end of the array, and while loop till the two index meet, a<b

# Q  Longest Substring with At Most K Distinct Characters

When I first see this questions, the brute force solution is check each substring that don’t pass limit k for each character, so this is O(n) for each character, anotherO(n), total O(n^2)

However, it pass the time limit, so this is a datastream problem, I should probably use Dynamic programming.

Basicly, I want an O(n) solution, so from N^2 to N , I will need to use a sliding window with size of K , where the sliding window is a dic with char value as key, and value as the last appear of key. We will also want to remove the lowest last appear, to keep the longest substring.

dic = {}

rightmost, result = 0,0

for i , v in enumerate(s):

dic[v] = i

if(k < len(dic)):

rightmost = min(dic.values())

del dic[s[rightmost]]

rightmost+=1

result = max(i-rightmost+1,result)

return result

# Q Add Bold Tag in String

First of all the length of values in the array are not constant, thus I need loop through the dict, instead of the string. KMP alogirthm + boolean.

So, for <b>abc</b><b>123</b> = <b>abc123</b> , "<b>aaa</b>bbc</b>c""<b>aaabbc</b>c"

Two different solution,

you find **the** index **of** **each** **string** **in** dict, conver to **an** interval, you will get

[[0, 3], [1, 4], [4, 6]]

aaa aab bc

**then** combine these intervals

Ater merged, we got [0,6], so we know "aaabbc" needs to be surrounded **by** tag.

Or you can use a boolean array to check if a character need to be colored.

N^2

KMP algorithm.

def preprocessKMP(self,p):

result = [0]\*len(p)

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

length = result[i-1]

while((length > 0 ) and (p[i] != p[length])):

length = result[length -1]

if(p[i] == p[length]):

result[i] = length +1

return result

for w in dict:

kmp = self.preprocessKMP(w)

idx = 0

last = -1

for i in range(len(s)):

if(w[idx] == s[i]):

idx += 1

else:

while(idx > 0 and s[i] != w[idx]):

idx = kmp[idx-1]

if(w[idx] == s[i]):

idx += 1

if(idx == len(kmp)):

start = max(last+1,i-len(kmp)+1)

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

bol[z] = True

last = i

idx = kmp[idx -1 ]

**Blueprint**

**lambda** argument: manipulate(argument)

**Example**

add = **lambda** x, y: x + y

print(add(3, 5))

*# Output: 8*

**List sorting**

a = [(1, 2), (4, 1), (9, 10), (13, -3)]

a.sort(key=**lambda** x: x[1])

x = [2, 3, 4, 5, 6]

y = map(**lambda** v : v \* 5, x)

[v \* 5 **for** v **in** x] --> map(**lambda** **for** v: v \* 5, **in** x) --> map(**lambda** v : v \* 5, x)

# Q Game of Life

Given a *board* with *m* by *n* cells, each cell has an initial state *live* (1) or *dead* (0).

For alive, 2 or 3 neighbour alive, else dead

For dead, 3 alive neigbour to alive, else dead

Solution 1: using to 2bit to store the current state and next state, so you can solve it in place

# Q Spiral Matrix

Solution 1: basic, with RowStart, RowEnd,ColStart, ColEnd.

Solution 2: using recursion, and remove the top of the matrix and rotate.

# Linked List

# Q Merge K sorted List

First solution, solved by using the merge idea from merge sort, time is O(n\*k)

Second solution, using priority queue, using minHeap, put all values into heap,then pop

class Solution(object):

def mergeKLists(self, lists):

from heapq import heappush, heappop, heapreplace, heapify

dummy = node = ListNode(0)

h = [(n.val, n) for n in lists if n]

heapify(h)

while h:

v, n = h[0]

if n.next is None:

heappop(h) #only change heap size when necessary

else:

heapreplace(h, (n.next.val, n.next))

node.next = n

node = node.next return dummy.next

# Q insert into a cyclic sorted list

# Tree and graphs

# A function used by DFS

def DFSUtil(self,v,visited):

# Mark the current node as visited and print it

visited[v]= True

print v,

# Recur for all the vertices adjacent to this vertex

for i in self.graph[v]:

if visited[i] == False:

self.DFSUtil(i, visited)

# The function to do DFS traversal. It uses

# recursive DFSUtil()

def DFS(self,v):

# Mark all the vertices as not visited

visited = [False]\*(len(self.graph))

# Call the recursive helper function to print

# DFS traversal

self.DFSUtil(v,visited)

# Q Inorder Successor in BST

**Inorder Traversal:**

Algorithm Inorder(tree)

1. Traverse the left subtree, i.e., call Inorder(left-subtree)

2. Visit the root.

3. Traverse the right subtree, i.e., call Inorder(right-subtree)

class Solution(object):

def inorderTraversal(self, root):

res = []

if root:

res = self.inorderTraversal(root.left)

res.append(root.val)

res = res + self.inorderTraversal(root.right)

return res

**Preorder Traversal:**

Algorithm Preorder(tree)

1. Visit the root.

2. Traverse the left subtree, i.e., call Preorder(left-subtree)

3. Traverse the right subtree, i.e., call Preorder(right-subtree)

**Postorder Traversal:**

Algorithm Postorder(tree)

1. Traverse the left subtree, i.e., call Postorder(left-subtree)

2. Traverse the right subtree, i.e., call Postorder(right-subtree)

3. Visit the root.

For this questions, The inorder traversal of a BST is the nodes in ascending order. To find a successor, you just need to find the smallest one that is larger than the given value since there are no duplicate values in a BST. It just like the binary search in a sorted list. The time complexity should be O(h) where h is the depth of the result node. succ is a pointer that keeps the possible successor. Only in a balanced BST O(h) = O(log n). In the worst case h can be as large as n.

def inorderSuccessor(self, root, p):

result = None

while root:

if(p.val < root.val):

result = root

root = root.left

else:

root= root.right

return result

# Q Evaluate Division

Floyd–Warshall algorithm is an algorithm for finding shortest paths in a weighted graph with positive or negative edge weights (but with no negative cycles)

The difference is that a defaultdict will "default" a value if that key has not been set yet. If you didn't use a defaultdict you'd have to check to see if that key exists, and if it doesn't, set it to what you want

**zip**([*iterable*, *...*])

This function returns a list of tuples, where the *i*-th tuple contains the *i*-th element from each of the argument sequences or iterables.

[**zip()**](https://docs.python.org/2/library/functions.html#zip) in conjunction with the \* operator can be used to unzip a list:

**>>>** x = [1, 2, 3]

**>>>** y = [4, 5, 6]

**>>>** zipped = zip(x, y)

**>>>** zipped

[(1, 4), (2, 5), (3, 6)]

**>>>** x2, y2 = zip(\*zipped)

**>>>** x == list(x2) **and** y == list(y2)

True

class Solution(object):

def calcEquation(self, equations, values, queries):

def dfs(start, end, path, paths):

if start == end and start in G:

paths[0] = path

return

if start in vis:

return

vis.add(start)

for node in G[start]:

dfs(node, end, path \* W[start, node], paths)

G, W = collections.defaultdict(set), collections.defaultdict(float)

for (A, B), V in zip(equations, values):

G[A], G[B] = G[A] | {B}, G[B] | {A}

W[A, B], W[B, A] = V, 1.0 / V

res = []

for X, Y in queries:

paths, vis = [-1.0], set()

dfs(X, Y, 1.0, paths)

res += paths[0],

return res

The [**sets**](https://docs.python.org/2/library/sets.html#module-sets) module provides classes for constructing and manipulating unordered collections of unique elements. Common uses include membership testing, removing duplicates from a sequence, and computing standard math operations on sets such as intersection, union, difference, and symmetric difference.

**>>> from** **sets** **import** Set

**>>>** engineers = Set(['John', 'Jane', 'Jack', 'Janice'])

**>>>** programmers = Set(['Jack', 'Sam', 'Susan', 'Janice'])

**>>>** managers = Set(['Jane', 'Jack', 'Susan', 'Zack'])

**>>>** employees = engineers | programmers | managers *# union*

**>>>** engineering\_management = engineers & managers *# intersection*

**>>>** fulltime\_management = managers - engineers - programmers *# difference*

**>>>** engineers.add('Marvin') *# add element*

**>>>** print engineers

Set(['Jane', 'Marvin', 'Janice', 'John', 'Jack'])

**>>>** employees.issuperset(engineers) *# superset test*

False

**>>>** employees.update(engineers) *# update from another set*

**>>>** employees.issuperset(engineers)

True

**>>> for** group **in** [engineers, programmers, managers, employees]:

**...**  group.discard('Susan') *# unconditionally remove element*

**...**  print group

**...**

Set(['Jane', 'Marvin', 'Janice', 'John', 'Jack'])

Set(['Janice', 'Jack', 'Sam'])

Set(['Jane', 'Zack', 'Jack'])

Set(['Jack', 'Sam', 'Jane', 'Marvin', 'Janice', 'John', 'Zack'])

# Recursion

Recursion usually involve some kind of backtracking to enumerate all possibilities.

# Q Word Squares

Solution 1, for each word, try to place it at different position, and check if there is a possible solution, first create a map of all the prefix of all the words, so easy to find the matching words,

Then, recusively find the next possible word, and if no more, append to result

class Solution(object):

def wordSquares(self, words):

n = len(words[0])

allp = collections.defaultdict(list)

for word in words:

for i in range(n):

allp[word[:i]].append(word)

#print(allp)

def build(square):

if len(square) == n:

squares.append(square)

return

#print(zip(\*square)[len(square)]) the len determines which char is current at

for word in allp[''.join(zip(\*square)[len(square)])]:

build(square + [word])

squares = []

for word in words:

build([word])

return squares

# Q Moving average from data stream

Solution 1, using a array and check limit

Solution 2 using a queue.

# Q longest subsequence

Use hash map to map all the value first, and check if it is a starting point by see if there is arr[i]-1, then count from theres

# Q LRU Cache

Solved using double link list to keep the value and key, and use hash map to keep track of the key and node.

class Node:

def \_\_init\_\_(self,key,val):

self.val = val

self.key = key

self.next = None

self.prev = None

class LRUCache(object):

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

"""

:type capacity: int

"""

self.capacity = capacity

self.dic = {}

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:

node = self.dic[key]

self.\_remove(node)

self.\_add(node)

return node.val

return -1

def put(self, key, value):

if key in self.dic:

node = self.dic[key]

self.\_remove(node)

n = Node(key,value)

self.\_add(n)

self.dic[n.key] = n

if(self.capacity < len(self.dic)):

del self.dic[self.head.next.key]

self.\_remove(self.head.next)

def \_add(self,node):

p = self.tail.prev

p.next = node

node.prev = p

node.next = self.tail

self.tail.prev = node

def \_remove(self,node):

p = node.prev

n = node.next

p.next = n

n.prev = p

# Q Number of Island

Use dfs to mark the traveled 1s to #

class Solution(object):

def numIslands(self, grid):

if not grid:

return 0

count = 0

for i in range(len(grid)):

for j in range(len(grid[0])):

if(grid[i][j] == '1'):

self.dfs(grid,i,j)

count +=1

return count

def dfs(self,grid,i,j):

if i<0 or j<0 or len(grid) <= i or len(grid[0]) <= j or grid[i][j] != '1':

return

grid[i][j] = '#'

self.dfs(grid,i-1,j)

self.dfs(grid,i+1,j)

self.dfs(grid,i,j-1)

self.dfs(grid,i,j+1)

# Q finding the k largest element, minheap

class Solution(object):

def findKthLargest(self, nums, k):

heapq.heapify(nums)

for i in range(len(nums)-k):

heapq.heappop(nums)

return heapq.heappop(nums)

2.

/\*\*

\* Create a key-value map

\* Expected behavior example:

\*

\* 12:00:00 - put(10, 25, 5000)   <-- 5000 milliseconds == 5 seconds

\* 12:00:04 - get(10) -> 25

\* 12:00:06 - get(10) -> None

\* Assume you have a getTime() function available.

\*/

def put(key,val,time):

   pass

def get(key):

   pass

# Yelp

# Q Two Sum

class Solution(object):

def twoSum(self, nums, target):

dic = collections.defaultdict(list)

for i,v in enumerate(nums):

dic[v] = dic[v]+ [i]

for n in nums:

dif = target - n

if(dif == n):

if(len(dic[n]) > 1):

return dic[n][:2]

elif(dif in dic):

return dic[n] + dic[dif]

use collections.defaultdict(list)

and use enumerate.

# Q three sum

def generateSubset(nums):

result = [[]]

for num in nums:

result += [i + [num] for i in result]

return result

first sort the list,

then for each value, check if there is a possible solution, with

class Solution(object):

def threeSum(self, nums):

result = []

nums = sorted(nums)

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

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

j=i+1

k=len(nums)-1

while(j<k):

if((nums[k]+nums[i]+nums[j]) == 0):

result.append([nums[k],nums[i],nums[j]])

if( 0 > nums[i]+ nums[k]+nums[j]):

current = j

while(nums[j] == nums[current] and j<k):

j+=1

else:

current = k

while(nums[k] == nums[current] and j<k):

k -=1

return result

# Q5 longest palindrome of a string

Two solutions:

DP:

Start from mid: and check each char at the string,

class Solution(object):

def longestPalindrome(self, s):

result = ""

def findP(s,left,right,result):

while(0 <= left and right < len(s) and s[left] == s[right]):

left-=1

right+=1

cur = s[left+1:right]

if(len(cur) > len(result)):

result = cur

return result

if(s == None or len(s) ==0):

return s

for i in range(len(s)):

result=findP(s,i,i,result)

result = findP(s,i,i+1,result)

return result

# Q Merge Interval

扫描线的算法：

# Definition for an interval.

# class Interval(object):

# def \_\_init\_\_(self, s=0, e=0):

# self.start = s

# self.end = e

#Sta END

# | | End

# | |

class Solution(object):

def merge(self, intervals):

"""

:type intervals: List[Interval]

:rtype: List[Interval]

"""

if(intervals == None or len(intervals) <= 1):

return intervals

result = []

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

start = intervals[0].start

end = intervals[0].end

for i in intervals:

if(i.start <= end):

end = max(end,i.end)

else:

result.append(Interval(start,end))

start = i.start

end = i.end

result.append(Interval(start,end))

return result

# 599 Minimum Index Sum of Two Lists

Create hashmap on one list ,and enumerate on the other.

# Q Group Anagrams

List the strings that are anagrams from a set of strings?

class Solution:

def groupAnagrams(self, strs):

"""

:type strs: List[str]

:rtype: List[List[str]]

"""

dic = collections.defaultdict(list)

for i in strs:

key = str(sorted(i))

dic[key].append(i)

result = []

for key,value in dic.items():

result.append(value)

return result

# Q Top K Frequent Words

class Solution(object):

def topKFrequent(self, words, k):

"""

:type words: List[str]

:type k: int

:rtype: List[str]

"""

dic = collections.Counter(words)

result = sorted(dic, key = lambda x: (-dic[x],x))

return result[:k]

Python priority queue

Sample B - tuple

The priority queue can store objects such as **tuples**:

try:

import Queue as Q # ver. < 3.0

except ImportError:

import queue as Q

q = Q.PriorityQueue()

q.put((10,'ten'))

q.put((1,'one'))

q.put((5,'five'))

while not q.empty():

print q.get(),

Output:

(1, 'one') (5, 'five') (10, 'ten')

Given a list of stock prices brought over a period of time. Find the max loss that could be possible among the data. Ex. [9,2,10] --> max loss is when brought for 9 and sold for 2.

What machine learning algorithm you'll use to avoid spam comments.

One basic technique is [Naive Bayes](http://en.wikipedia.org/wiki/Naive_Bayes_classifier), which is often used for [spam filtering](http://en.wikipedia.org/wiki/Bayesian_spam_filtering). Essentially, you look at the frequencies of words appearing in sentences that you have already judged to be spam, and also at the frequencies of those words appearing in sentences you have already judged to be not spam, and then use those frequencies to make a judgement about new sentences.

In [machine learning](https://en.wikipedia.org/wiki/Machine_learning), **naive Bayes classifiers** are a family of simple "[probabilistic classifiers](https://en.wikipedia.org/wiki/Probabilistic_classifier)" based on applying [Bayes' theorem](https://en.wikipedia.org/wiki/Bayes%27_theorem) with strong (naive) [independence](https://en.wikipedia.org/wiki/Statistical_independence) assumptions between the features. Basicly, find the probability for each word being a spam, then combine all the word’s probability together to determine whether

https://hackernoon.com/how-to-build-a-simple-spam-detecting-machine-learning-classifier-4471fe6b816e

**Naive Bayes Classifier**

This is a simple (naive) classification method based on Bayes rule. It relies on a very simple representation of the document (called the **bag of words** representation)

Why yelp? how to detect spam? features you can come up with? Build a model using these features? loss function?

A s*pam classifier* will classify all messages you receive. There will be 4 possibilities :

* correct classification :
  + **True positives** : true spams which will be classified as true spams;
  + **True negatives** : true legitimate messages which will be classified as legitimate messages.
* wrong classification :
  + **False negatives** : true spam which will be classified as being legitimate messages;
  + **False positives** : legitimate messages which will be classified as being spams.

That said, a good classifier will minimize the two “**False**” categories. But, either way, it’s better to receive some false negative errors (spam being put in your inbox) than some false positives (legitimate messages being lost).

All this is controlled at some place by some sort of t*hreshold*. So, IMHO, a good classifier should :

* minimize errors, both false negatives and false positives;
* as much as possible, eliminate f**alse positives** : on’t loose legitimate messages.

Either way, there isn’t a perfect classifier ! It’s important to understand this. Some people selling spam filters say that their error is 0 % : it’s never true.

a **loss function** or **cost function** is a function that maps an [event](https://en.wikipedia.org/wiki/Event_(probability_theory)) or values of one or more variables onto a [real number](https://en.wikipedia.org/wiki/Real_number) intuitively representing some "cost" associated with the event. An [optimization problem](https://en.wikipedia.org/wiki/Optimization_problem) seeks to minimize a loss function.