peterlamar/python-cp-cheatsheet: Python3 interview prep cheatsheet and examples

Python3 reference for interview coding problems/light competitive programming. Contributions welcome!

How

I built this cheatsheet while teaching myself Python3 for various interviews and leetcoding for fun after not using Python for about a decade. This cheetsheet only contains code that I didn't know but needed to use to solve a specific coding problem. I did this to try to get a smaller high frequency subset of Python vs a comprehensive list of all methods. Additionally, the act of recording the syntax and algorithms helped me store it in memory and as a result I almost never actually referenced this sheet. Hopefully it helps you in your efforts or inspires you to build your own and best of luck!

Why

The rule of least power

I choose Python3 despite being more familiar with Javascript, Java, C++ and Golang for interviews as I felt Python had the combination of the most standard libraries available as well as syntax that resembles psuedo code, therefore being the most expressive. Python and Java both have the most examples but Python wins in this case due to being much more concise. I was able to get myself reasonably prepared with Python syntax in six weeks of practice. After picking up Python I have timed myself solving the same exercises in Golang and Python. Although I prefer Golang, I find that I can complete Python examples in half the time even accounting for +50% more bugs (approximately) that I tend to have in Python vs Go. This is optimizing for solved interview questions under pressure, when performance is considered then Go/C++ does consistently perform 1/10 the time of Python. In some rare cases, algorithms that time out in Python sometimes pass in C++/Go on Leetcode.

Language Mechanics

Literals

```
255, 0b11111111, 0o377, 0xff # Integers (decimal, binary, octal, hex)
123.0, 1.23                  # Float
7 + 5j, 7j                   # Complex
'a', '\141', '\x61'          # Character (literal, octal, hex)
'\n', '\\', '\'', '\"'       # Newline, backslash, single quote, double quote
"string\n"                   # String of characters ending with newline
"hello"+"world"              # Concatenated strings
True, False                  # bool constants, 1 == True, 0 == False
[1, 2, 3, 4, 5]              # List
['meh', 'foo', 5]            # List
(2, 4, 6, 8)                 # Tuple, immutable
{'name': 'a', 'age': 90}     # Dict
{'a', 'e', 'i', 'o', 'u'}    # Set
None                         # Null var
```

Loops

Go through all elements

```
i = 0
while i < len(str):
  i += 1
```

equivalent

```
for i in range(len(message)):
  print(i)
```

Get largest number index from right

```
while i > 0 and nums [i-1] >= nums[i]:
  i -= 1
```

Manually reversing

```
l, r = i, len(nums) - 1
while l < r:
  nums[l], nums[r] = nums[r], nums[l]
  l += 1
  r -= 1
```

Go past the loop if we are clever with our boundry

```
for i in range(len(message) + 1):
  if i == len(message) or message[i] == ' ':
```

Fun with Ranges - range(start, stop, step)

```
for a in range(0,3): # 0,1,2
for a in reversed(range(0,3)) # 2,1,0
for i in range(3,-1,-1) # 3,2,1,0
for i in range(len(A)//2): # A = [0,1,2,3,4,5]
  print(i) # 0,1,2
  print(A[i]) # 0,1,2
  print(~i) # -1,-2,-3
  print(A[~i]) # 5,4,3
```

Strings

```
str1.find('x')          # find first location of char x and return index
str1.rfind('x')         # find first int location of char x from reverse
```

Parse a log on ":"

```
l = "0:start:0"
tokens = l.split(":")
print(tokens) # ['0', 'start', '0']
```

Reverse works with built in split, [::-1] and " ".join()

```
# s = "the sky  is blue"
def reverseWords(self, s: str) -> str:
  wordsWithoutWhitespace = s.split() # ['the', 'sky', 'is', 'blue']
  reversedWords = wordsWithoutWhitespace[::-1] # ['blue', 'is', 'sky', 'the']
  final = " ".join(reversedWords) # blue is sky the
```

Manual split based on isalpha()

```
def splitWords(input_string) -> list:
  words = [] #
  start = length = 0
  for i, c in enumerate(input_string):
    if c.isalpha():
      if length == 0:
        start = i
        length += 1
      else:
        words.append(input_string[start:start+length])
        length = 0
  if length > 0:
    words.append(input_string[start:start+length])
  return words
```

Test type of char

```
def rotationalCipher(input, rotation_factor):
  rtn = []
  for c in input:
    if c.isupper():
      ci = ord(c) - ord('A')
      ci = (ci + rotation_factor) % 26
      rtn.append(chr(ord('A') + ci))
    elif c.islower():
      ci = ord(c) - ord('a')
      ci = (ci + rotation_factor) % 26
      rtn.append(chr(ord('a') + ci))
    elif c.isnumeric():
      ci = ord(c) - ord('0')
      ci = (ci + rotation_factor) % 10
      rtn.append(chr(ord('0') + ci))
    else:
      rtn.append(c)
  return "".join(rtn)
```

AlphaNumberic

Get charactor index

```
print(ord('A')) # 65
print(ord('B')-ord('A')+1) # 2
print(chr(ord('a') + 2)) # c
```

Replace characters or strings

```
def isValid(self, s: str) -> bool:
  while '[]' in s or '()' in s or '{}' in s:
    s = s.replace('[]','').replace('()','').replace('{}','')
  return len(s) == 0
```

Insert values in strings

```
txt3 = "My name is {}, I'm {}".format("John",36) # My name is John, I'm 36
```

Multiply strings/lists with *, even booleans which map to True(1) and False(0)

```
'meh' * 2 # mehmeh
['meh'] * 2 # ['meh', 'meh']
['meh'] * True #['meh']
['meh'] * False #[]
```

Find substring in string

```
txt = "Hello, welcome to my world."
x = txt.find("welcome")  # 7
```

startswith and endswith are very handy

```
str = "this is string example....wow!!!"
str.endswith("!!") # True
str.startswith("this") # True
str.endswith("is", 2, 4) # True
```

Python3 format strings

```
name = "Eric"
profession = "comedian"
affiliation = "Monty Python"
message = (
     f"Hi {name}. "
     f"You are a {profession}. "
     f"You were in {affiliation}."
)
message
'Hi Eric. You are a comedian. You were in Monty Python.'
```

Print string with all chars, useful for debugging

```
print(repr("meh\n"))     # 'meh\n'
```

Slicing

Slicing intro

```
               +---+---+---+---+---+---+
               | P | y | t | h | o | n |
               +---+---+---+---+---+---+
Slice position: 0   1   2   3   4   5   6
Index position:   0   1   2   3   4   5
p = ['P','y','t','h','o','n']
p[0] 'P' # indexing gives items, not lists
alpha[slice(2,4)] # equivalent to p[2:4]
```

```
p[0:1] # ['P'] Slicing gives lists
p[0:5] # ['P','y','t','h','o'] Start at beginning and count 5
p[2:4] = ['t','r'] # Slice assignment  ['P','y','t','r','o','n']
p[2:4] = ['s','p','a','m'] # Slice assignment can be any size['P','y','s','p','a','m','o','n']
p[4:4] = ['x','y'] # insert slice ['P','y','t','h','x','y','o','n']
p[0:5:2] # ['P', 't', 'o'] sliceable[start:stop:step]
p[5:0:-1] # ['n', 'o', 'h', 't', 'y']
```

Go through num and get combinations missing a member

```
numList = [1,2,3,4]
for i in range(len(numList)):
    newList = numList[0:i] + numList[i+1:len(numList)]
    print(newList) # [2, 3, 4], [1, 3, 4], [1, 2, 4], [1, 2, 3]
```

Tuple

Collection that is ordered and unchangable

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[1]) # banana
```

Can be used with Dicts

```
def groupAnagrams(self, strs: List[str]) -> List[List[str]]:
    d = defaultdict(list)
    for w in strs:
        key = tuple(sorted(w))
        d[key].append(w)
    return d.values()
```

Sort

sorted(iterable, key=key, reverse=reverse)

Sort sorts alphabectically, from smallest to largest

```
print(sorted(['Ford', 'BMW', 'Volvo'])) # ['BMW', 'Ford', 'Volvo']
nums = [-4,-1,0,3,10]
print(sorted(n*n for n in nums)) # [0,1,9,16,100]
```

```
cars = ['Ford', 'BMW', 'Volvo']
cars.sort() # returns None type
cars.sort(key=lambda x: len(x) ) # ['BMW', 'Ford', 'Volvo']
print(sorted(cars, key=lambda x:len(x))) # ['BMW', 'Ford', 'Volvo']
```

Sort key by value, even when value is a list

```
meh = {'a':3,'b':0,'c':2,'d':-1}
print(sorted(meh, key=lambda x:meh[x])) # ['d', 'b', 'c', 'a']
meh = {'a':[0,3,'a'],'b':[-2,-3,'b'],'c':[2,3,'c'],'d':[-2,-2,'d']}
print(sorted(meh, key=lambda x:meh[x])) # ['b', 'd', 'a', 'c']
```

```
def merge_sorted_lists(arr1, arr2): # built in sorted does Timsort optimized for subsection sorted lists
    return sorted(arr1 + arr2)
```

Sort an array but keep the original indexes

```
self.idx, self.vals = zip(*sorted([(i,v) for i,v in enumerate(nums)], key=lambda x:x[1]))
```

Sort by tuple, 2nd element then 1st ascending

```
a = [(5,10), (2,20), (2,3), (0,100)]
test = sorted(a, key = lambda x: (x[1],x[0]))
print(test) # [(2, 3), (5, 10), (2, 20), (0, 100)]
test = sorted(a, key = lambda x: (-x[1],x[0]))
print(test) # [(0, 100), (2, 20), (5, 10), (2, 3)]
```

Sort and print dict values by key

```
ans = {-1: [(10, 1), (3, 3)], 0: [(0, 0), (2, 2), (7, 4)], -3: [(8, 5)]}
for key, value in sorted(ans.items()): print(value)
# [(8, 5)]
# [(10, 1), (3, 3)]
# [(0, 0), (2, 2), (7, 4)]

# sorted transforms dicts to lists
sorted(ans) # [-3, -1, 0]
sorted(ans.values()) # [[(0, 0), (2, 2), (7, 4)], [(8, 5)], [(10, 1), (3, 3)]]
sorted(ans.items()) # [(-3, [(8, 5)]), (-1, [(10, 1), (3, 3)]), (0, [(0, 0), (2, 2), (7, 4)])]
# Or just sort the dict directly
[ans[i] for i in sorted(ans)]
# [[(8, 5)], [(10, 1), (3, 3)], [(0, 0), (2, 2), (7, 4)]]
```

Hash

```
for c in s1: # Adds counter for c
  ht[c] = ht.get(c, 0) + 1 # ht[a] = 1, ht[a]=2, etc
```

Set

```
a = 3
st = set()
st.add(a) # Add to st
st.remove(a) # Remove from st
st.discard(a) # Removes from set with no error
st.add(a) # Add to st
next(iter(s)) # return 3 without removal
st.pop() # returns 3
```

```
s = set('abc') # {'c', 'a', 'b'}
s |= set('cdf') # {'f', 'a', 'b', 'd', 'c'} set s with elements from new set
s &= set('bd') # {'d', 'b'} only elements from new set
s -= set('b') # {'d'} remove elements from new set
s ^= set('abd') # {'a', 'b'} elements from s or new but not both
```

List

Stacks are implemented with Lists. Stacks are good for parsing and graph traversal

```
test = [0] * 100 # initialize list with 100 0's
```

2D

```
rtn.append([])
rtn[0].append(1) # [[1]]
```

List Comprehension

```
number_list = [ x for x in range(20) if x % 2 == 0]
print(number_list) # [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Reverse a list

```
ss = [1,2,3]
ss.reverse()
print(ss) #3,2,1
```

Join list

```
list1 = ["a", "b" , "c"]
list2 = [1, 2, 3]
list3 = list1 + list2 # ['a', 'b', 'c', 1, 2, 3]
```

Dict

Hashtables are implemented with dictionaries

```
d = {'key': 'value'}         # Declare dict{'key': 'value'}
d['key'] = 'value'           # Add Key and Value
{x:0 for x in {'a', 'b'}}    # {'a': 0, 'b': 0} declare through comprehension
d['key'])                    # Access value
d.items()                    # Items as tuple list dict_items([('key', 'value')])
if 'key' in d: print("meh")  # Check if value exists
par = {}
par.setdefault(1,1)          # returns 1, makes par = { 1 : 1 }
par = {0:True, 1:False}
par.pop(0)                   # Remove key 0, Returns True, par now {1: False}
for k in d: print(k)         # Iterate through keys
```

Create Dict of Lists that match length of list to count votes

```
votes = ["ABC","CBD","BCA"]
rnk = {v:[0] * len(votes[0]) for v in votes[0]}
```

```
print(rnk) # {'A': [0, 0, 0], 'B': [0, 0, 0], 'C': [0, 0, 0]}
```

Tree

1. A tree is an undirected graph in which any two vertices are connected by exactly one path.
2. Any connected graph who has n nodes with n-1 edges is a tree.
3. The degree of a vertex is the number of edges connected to the vertex.
4. A leaf is a vertex of degree 1. An internal vertex is a vertex of degree at least 2.
5. A path graph is a tree with two or more vertices with no branches, degree of 2 except for leaves which have degree of 1
6. Any two vertices in G can be connected by a unique simple path.
7. G is acyclic, and a simple cycle is formed if any edge is added to G.
8. G is connected and has no cycles.
9. G is connected but would become disconnected if any single edge is removed from G.

BinaryTree

DFS Pre, In Order, and Post order Traversal

- Preorder
  - encounters roots before leaves
  - Create copy
- Inorder
  - flatten tree back to original sequence
  - Get values in non-decreasing order in BST
- Post order
  - encounter leaves before roots
  - Helpful for deleting

Recursive

```
"""
     1
    / \
   2   3
  / \
 4   5
"""
# PostOrder 4 5 2 3 1  (Left-Right-Root)
def postOrder(node):
  if node is None:
    return
  postorder(node.left)
  postorder(node.right)
  print(node.value, end=' ')
```

Iterative PreOrder

```
# PreOrder  1 2 4 5 3 (Root-Left-Right)
def preOrder(tree_root):
  stack = [(tree_root, False)]
  while stack:
    node, visited = stack.pop()
    if node:
      if visited:
        print(node.value, end=' ')
      else:
        stack.append((node.right, False))
        stack.append((node.left, False))
        stack.append((node, True))
```

Iterative InOrder

```
# InOrder   4 2 5 1 3 (Left-Root-Right)
def inOrder(tree_root):
  stack = [(tree_root, False)]
  while stack:
    node, visited = stack.pop()
    if node:
      if visited:
        print(node.value, end=' ')
      else:
        stack.append((node.right, False))
        stack.append((node, True))
        stack.append((node.left, False))
```

Iterative PostOrder

```
# PostOrder 4 5 2 3 1  (Left-Right-Root)
def postOrder(tree_root):
  stack = [(tree_root, False)]
  while stack:
    node, visited = stack.pop()
    if node:
      if visited:
        print(node.value, end=' ')
      else:
        stack.append((node, True))
        stack.append((node.right, False))
        stack.append((node.left, False))
```

Iterative BFS(LevelOrder)

```
from collections import deque

#BFS levelOrder 1 2 3 4 5
def levelOrder(tree_root):
  queue = deque([tree_root])
  while queue:
    node = queue.popleft()
    if node:
        print(node.value, end=' ')
        queue.append(node.left)
        queue.append(node.right)

def levelOrderStack(tree_root):
    stk = [(tree_root, 0)]
    rtn = []
    while stk:
        node, depth = stk.pop()
        if node:
            if len(rtn) < depth + 1:
                rtn.append([])
            rtn[depth].append(node.value)
            stk.append((node.right, depth+1))
            stk.append((node.left, depth+1))
    print(rtn)
    return True

def levelOrderStackRec(tree_root):
    rtn = []

    def helper(node, depth):
        if len(rtn) == depth:
            rtn.append([])
        rtn[depth].append(node.value)
        if node.left:
            helper(node.left, depth + 1)
        if node.right:
            helper(node.right, depth + 1)

    helper(tree_root, 0)
    print(rtn)
    return rtn
```

Traversing data types as a graph, for example BFS

```
def removeInvalidParentheses(self, s: str) -> List[str]:
    rtn = []
    v = set()
    v.add(s)
    if len(s) == 0: return [""]
    while True:
        for n in v:
            if self.isValid(n):
                rtn.append(n)
        if len(rtn) > 0: break
```

```
        level = set()
        for n in v:
            for i, c in enumerate(n):
                if c == '(' or c == ')':
                    sub = n[0:i] + n[i + 1:len(n)]
                    level.add(sub)
        v = level
    return rtn
```

Reconstructing binary trees

1. Binary tree could be constructed from preorder and inorder traversal
2. Inorder traversal of BST is an array sorted in the ascending order

Convert tree to array and then to balanced tree

```
def balanceBST(self, root: TreeNode) -> TreeNode:
    self.inorder = []

    def getOrder(node):
        if node is None:
            return
        getOrder(node.left)
        self.inorder.append(node.val)
        getOrder(node.right)

    # Get inorder treenode ["1,2,3,4"]
    getOrder(root)

    # Convert to Tree
    #        2
    #       1 3
    #          4
    def bst(listTree):
        if not listTree:
            return None
        mid = len(listTree) // 2
        root = TreeNode(listTree[mid])
        root.left = bst(listTree[:mid])
        root.right = bst(listTree[mid+1:])
        return root

    return bst(self.inorder)
```

Graph

Build an adjecency graph from edges list

```
# N = 6, edges = [[0,1],[0,2],[2,3],[2,4],[2,5]]
graph = [[] for _ in range(N)]
for u,v in edges:
    graph[u].append(v)
    graph[v].append(u)
# [[1, 2], [0], [0, 3, 4, 5], [2], [2], [2]]
```

Build adjecency graph from traditional tree

```
adj = collections.defaultdict(list)
def dfs(node):
    if node.left:
        adj[node].append(node.left)
        adj[node.left].append(node)
        dfs(node.left)
    if node.right:
        adj[node].append(node.right)
        adj[node.right].append(node)
        dfs(node.right)
dfs(root)
```

Traverse Tree in graph notation

```
# [[1, 2], [0], [0, 3, 4, 5], [2], [2], [2]]
def dfs(node, par=-1):
    for nei in graph[node]:
        if nei != par:
            res = dfs(nei, node)
dfs(0) # 1->2->3->4->5
```

Heapq

```
     1
    / \
   2   3
  / \ / \
 5  6 8  7
```

Priority Queue

1. Implemented as complete binary tree, which has all levels as full excepted deepest
2. In a heap tree the node is smaller than its children

```
def maximumProduct(self, nums: List[int]) -> int:
  l = heapq.nlargest(3, nums)
  s = heapq.nsmallest(3, nums)
  return max(l[0]*l[1]*l[2],s[0]*s[1]*l[0])
```

Heap elements can be tuples, heappop() frees the smallest element (flip sign to pop largest)

```
def kClosest(self, points: List[List[int]], K: int) -> List[List[int]]:
    heap = []
    for p in points:
        distance = sqrt(p[0]* p[0] + p[1]*p[1])
        heapq.heappush(heap,(-distance, p))
        if len(heap) > K:
            heapq.heappop(heap)
    return ([h[1] for h in heap])
```

nsmallest can take a lambda argument

```
def kClosest(self, points: List[List[int]], K: int) -> List[List[int]]:
    return heapq.nsmallest(K, points, lambda x: x[0]*x[0] + x[1]*x[1])
```

The key can be a function as well in nsmallest/nlargest

```
def topKFrequent(self, nums: List[int], k: int) -> List[int]:
    count = Counter(nums)
    return heapq.nlargest(k, count, count.get)
```

Tuple sort, 1st/2nd element. increasing frequency then decreasing order

```
def topKFrequent(self, words: List[str], k: int) -> List[str]:
    freq = Counter(words)
    return heapq.nsmallest(k, freq.keys(), lambda x:(-freq[x], x))
```

Lambda

Can be used with (list).sort(), sorted(), min(), max(), (heapq).nlargest,nsmallest(), map()

```
# a=3,b=8,target=10
min((b,a), key=lambda x: abs(target - x)) # 8
```

```
>>> ids = ['id1', 'id2', 'id30', 'id3', 'id22', 'id100']
>>> print(sorted(ids)) # Lexicographic sort
['id1', 'id100', 'id2', 'id22', 'id3', 'id30']
>>> sorted_ids = sorted(ids, key=lambda x: int(x[2:])) # Integer sort
>>> print(sorted_ids)
['id1', 'id2', 'id3', 'id22', 'id30', 'id100']
```

```
trans = lambda x: list(al[i] for i in x) # apple, a->0..
print(trans(words[0])) # [0, 15, 15, 11, 4]
```

Lambda can sort by 1st, 2nd element in tuple

```
sorted([('abc', 121),('bbb',23),('abc', 148),('bbb', 24)], key=lambda x: (x[0],x[1]))
# [('abc', 121), ('abc', 148), ('bbb', 23), ('bbb', 24)]
```

Zip

Combine two dicts or lists

```
s1 = {2, 3, 1}
s2 = {'b', 'a', 'c'}
list(zip(s1, s2)) # [(1, 'a'), (2, 'c'), (3, 'b')]
```

Traverse in Parallel

```
letters = ['a', 'b', 'c']
numbers = [0, 1, 2]
for l, n in zip(letters, numbers):
  print(f'Letter: {l}') # a,b,c
  print(f'Number: {n}') # 0,1,2
```

Empty in one list is ignored

```
letters = ['a', 'b', 'c']
numbers = []
for l, n in zip(letters, numbers):
  print(f'Letter: {l}') #
  print(f'Number: {n}') #
```

Compare characters of alternating words

```
for a, b in zip(words, words[1:]):
    for c1, c2 in zip(a,b):
        print("c1 ", c1, end=" ")
        print("c2 ", c2, end=" ")
```

Passing in * unpacks a list or other iterable, making each of its elements a separate argument.

```
a = [[1,2],[3,4]]
test = zip(*a)
print(test) # (1, 3) (2, 4)
matrix = [[1,2,3],[4,5,6],[7,8,9]]
test = zip(*matrix)
print(*test) # (1, 4, 7) (2, 5, 8) (3, 6, 9)
```

Useful when rotating a matrix

```
# matrix = [[1,2,3],[4,5,6],[7,8,9]]
matrix[:] = zip(*matrix[::-1]) # [[7,4,1],[8,5,2],[9,6,3]]
```

Iterate through chars in a list of strs

```
strs = ["cir","car","caa"]
for i, l in enumerate(zip(*strs)):
    print(l)
    # ('c', 'c', 'c')
    # ('i', 'a', 'a')
    # ('r', 'r', 'a')
```

Diagonals can be traversed with the help of a list

```
"""
[[1,2,3],
 [4,5,6],
 [7,8,9],
 [10,11,12]]
"""
def printDiagonalMatrix(self, matrix: List[List[int]]) -> bool:
    R = len(matrix)
    C = len(matrix[0])

    tmp = [[] for _ in range(R+C-1)]

    for r in range(R):
        for c in range(C):
            tmp[r+c].append(matrix[r][c])

    for t in tmp:
        for n in t:
            print(n, end=' ')
        print("")
"""
 1,
 2,4
 3,5,7
 6,8,10
 9,11
 12
"""
```

Random

```
for i, l in enumerate(shuffle):
  r = random.randrange(0+i, len(shuffle))
  shuffle[i], shuffle[r] = shuffle[r], shuffle[i]
return shuffle
```

Other random generators

```
import random
ints = [0,1,2]
random.choice(ints) # 0,1,2
random.choices([1,2,3],[1,1,10]) # 3, heavily weighted
random.randint(0,2) # 0,1, 2
random.randint(0,0) # 0
random.randrange(0,0) # error
random.randrange(0,2) # 0,1
```

Constants

```
max = float('-inf')
min = float('inf')
```

Ternary

a if condition else b

```
test = stk.pop() if stk else '#'
```

Bitwise Operators

```
'0b{:04b}'.format(0b1100 & 0b1010) # '0b1000' and
'0b{:04b}'.format(0b1100 | 0b1010) # '0b1110' or
'0b{:04b}'.format(0b1100 ^ 0b1010) # '0b0110' exclusive or
'0b{:04b}'.format(0b1100 >> 2)     # '0b0011' shift right
'0b{:04b}'.format(0b0011 << 2)     # '0b1100' shift left
```

For Else

Else condition on for loops if break is not called

```
for w1, w2 in zip(words, words[1:]): #abc, ab
    for c1, c2 in zip(w1, w2):
        if c1 != c2:
            adj[c1].append(c2)
            degrees[c2] += 1
            break
    else: # nobreak
        if len(w1) > len(w2):
            return ""   # Triggers since ab should be before abc, not after
```

Modulo

```
for n in range(-8,8):
    print n, n//4, n%4

 -8 -2 0
 -7 -2 1
 -6 -2 2
 -5 -2 3

 -4 -1 0
 -3 -1 1
 -2 -1 2
 -1 -1 3

  0  0 0
  1  0 1
  2  0 2
  3  0 3

  4  1 0
  5  1 1
  6  1 2
  7  1 3
```

Any

if any element of the iterable is True

```
def any(iterable):
    for element in iterable:
        if element:
            return True
    return False
```

All

```
def all(iterable):
    for element in iterable:
        if not element:
            return False
    return True
```

Bisect

- bisect.bisect_left returns the leftmost place in the sorted list to insert the given element
- bisect.bisect_right returns the rightmost place in the sorted list to insert the given element

```
import bisect
bisect.bisect_left([1,2,3,4,5], 2)  # 1
bisect.bisect_right([1,2,3,4,5], 2) # 2
bisect.bisect_left([1,2,3,4,5], 7)  # 5
bisect.bisect_right([1,2,3,4,5], 7) # 5
```

Insert x in a in sorted order. This is equivalent to a.insert(bisect.bisect_left(a, x, lo, hi), x) assuming that a is already sorted. Search is binary search O(logn) and insert is O(n)

```
import bisect
l = [1, 3, 7, 5, 6, 4, 9, 8, 2]
result = []
for e in l:
    bisect.insort(result, e)
print(result) # [1, 2, 3, 4, 5, 6, 7, 8, 9]

li1 = [1, 3, 4, 4, 4, 6, 7] # [1, 3, 4, 4, 4, 5, 6, 7]
bisect.insort(li1, 5) #
```

Bisect can give two ends of a range, if the array is sorted of course

```
s = bisect.bisect_left(nums, target)
e = bisect.bisect(nums, target) -1
if s <= e:
    return [s,e]
else:
    return [-1,-1]
```

Math

Calulate power

```
# (a ^ b) % p.
d = pow(a, b, p)
```

Division with remainder

```
divmod(8, 3) # (2, 2)
divmod(3, 8) #  (0, 3)
```

eval

Evaluates an expression

```
x = 1
print(eval('x + 1'))
```

Iter

Creates iterator from container object such as list, tuple, dictionary and set

```
mytuple = ("apple", "banana", "cherry")
myit = iter(mytuple)
print(next(myit)) # apple
print(next(myit)) # banana
```

Map

map(func, *iterables)

```
my_pets = ['alfred', 'tabitha', 'william', 'arla']
uppered_pets = list(map(str.upper, my_pets)) # ['ALFRED', 'TABITHA', 'WILLIAM', 'ARLA']
my_strings = ['a', 'b', 'c', 'd', 'e']
my_numbers = [1,2,3,4,5]
results = list(map(lambda x, y: (x, y), my_strings, my_numbers)) # [('a', 1), ('b', 2), ('c', 3), ('d', 4), ('e', 5)]

A1 = [1, 4, 9]
''.join(map(str, A1))
```

Filter

filter(func, iterable)

```
scores = [66, 90, 68, 59, 76, 60, 88, 74, 81, 65]
over_75 = list(filter(lambda x: x>75, scores)) # [90, 76, 88, 81]

scores = [66, 90, 68, 59, 76, 60, 88, 74, 81, 65]
def is_A_student(score):
    return score > 75
over_75 = list(filter(is_A_student, scores)) # [90, 76, 88, 81]

dromes = ("demigod", "rewire", "madam", "freer", "anutforajaroftuna", "kiosk")
palindromes = list(filter(lambda word: word == word[::-1], dromes)) # ['madam', 'anutforajaroftuna']
```

Get degrees == 0 from list

```
stk = list(filter(lambda x: degree[x]==0, degree.keys()))
```

Reduce

reduce(func, iterable[, initial]) where initial is optional

```
numbers = [3, 4, 6, 9, 34, 12]
result = reduce(lambda x, y: x+y, numbers) # 68
result = reduce(lambda x, y: x+y, numbers, 10) #78
```

itertools

itertools.accumulate(iterable[, func]) -> accumulate object

```
import itertools
data = [3, 4, 6, 2, 1, 9, 0, 7, 5, 8]
list(itertools.accumulate(data)) # [3, 7, 13, 15, 16, 25, 25, 32, 37, 45]
list(accumulate(data, max))  # [3, 4, 6, 6, 6, 9, 9, 9, 9, 9]
cashflows = [1000, -90, -90, -90, -90]  # Amortize a 5% loan of 1000 with 4 annual payments of 90
list(itertools.accumulate(cashflows, lambda bal, pmt: bal*1.05 + pmt)) [1000, 960.0, 918.0, 873.9000000000001, 827.5950000000001]
for k,v in groupby("aabbbc")    # group by common letter
    print(k)                    # a,b,c
    print(list(v))              # [a,a],[b,b,b],[c,c]
```

Regular Expression

RE module allows regular expressions in python

```
def removeVowels(self, S: str) -> str:
    return re.sub('a|e|i|o|u', '', S)
```

Types

from typing import List, Set, Dict, Tuple, Optional cheat sheet

Grids

Useful helpful function

```
R = len(grid)
C = len(grid[0])

def neighbors(r, c):
    for nr, nc in ((r,c-1), (r,c+1), (r-1, c), (r+1,c)):
        if 0<=nr<R and 0<=nc<C:
            yield nr, nc

def dfs(r,c, index):
    area = 0
    grid[r][c] = index
    for x,y in neighbors(r,c):
        if grid[x][y] == 1:
            area += dfs(x,y, index)
    return area + 1
```

Collections

Stack with appendleft() and popleft()

Deque

```
from collections import deque
deq = deque([1, 2, 3])
deq.appendleft(5)
deq.append(6)
deq
deque([5, 1, 2, 3, 6])
deq.popleft()
5
deq.pop()
6
deq
deque([1, 2, 3])
```

Counter

```
from collections import Counter
count = Counter("hello") # Counter({'h': 1, 'e': 1, 'l': 2, 'o': 1})
count['l'] # 2
count['l'] += 1
count['l'] # 3
```

Get counter k most common in list of tuples

```
# [1,1,1,2,2,3]
# Counter  [(1, 3), (2, 2)]
def topKFrequent(self, nums: List[int], k: int) -> List[int]:
    if len(nums) == k:
        return nums
    return [n[0] for n in Counter(nums).most_common(k)] # [1,2]
```

elements() lets you walk through each number in the Counter

```
def intersect(self, nums1: List[int], nums2: List[int]) -> List[int]:
    c1 = collections.Counter(nums1) # [1,2,2,1]
    c2 = collections.Counter(nums2) # [2,2]
    dif = c1 & c2                   # {2:2}
    return list(dif.elements())     # [2,2]
```

operators work on Counter

```
c = Counter(a=3, b=1)
d = Counter(a=1, b=2)
c + d # {'a': 4, 'b': 3}
c - d # {'a': 2}
c & d # {'a': 1, 'b': 1}
c | d # {'a': 3, 'b': 2}
c = Counter(a=2, b=-4)
+c # {'a': 2}
-c # {'b': 4}
```

Default Dict

```
d={}
print(d['Grapes'])# This gives Key Error

from collections import defaultdict
d = defaultdict(int) # set default
print(d['Grapes']) # 0, no key error
d = collections.defaultdict(lambda: 1)
print(d['Grapes']) # 1, no key error
```

```
from collections import defaultdict
dd = defaultdict(list)
dd['key'].append(1) # defaultdict(<class 'list'>, {'key': [1]})
dd['key'].append(2) # defaultdict(<class 'list'>, {'key': [1, 2]})
```

Algorithms

General Tips

- Get all info
- Debug example, is it a special case?
- Brute Force
    - Get to brute-force solution as soon as possible. State runtime and then optimize, don't code yet
- Optimize
    - Look for unused info
    - Solve it manually on example, then reverse engineer thought process
    - Space vs time, hashing
    - BUDS (Bottlenecks, Unnecessary work, Duplication)
- Walk through approach
- Code
- Test
    - Start small
    - Hit edge cases

Binary Search

```
def firstBadVersion(self, n):
    l, r = 0, n
    while l < r:
        m = l + (r-l) // 2
        if isBadVersion(m):
            r = m
        else:
            l = m + 1
    return l
```

```
"""
12345678
FFTTTTTT
"""
def mySqrt(self, x: int) -> int:
  def condition(value, x) -> bool:
    return value * value > x

  if x == 1:
    return 1

  left, right = 1, x
  while left < right:
    mid = left + (right-left) // 2
    if condition(mid, x):
      right = mid
    else:
      left = mid + 1

  return left - 1
```

binary search

Binary Search Tree

Use values to detect if number is missing

```
def isCompleteTree(self, root: TreeNode) -> bool:
    self.total = 0
    self.mx = float('-inf')
    def dfs(node, cnt):
```

```
        if node:
            self.total += 1
            self.mx = max(self.mx, cnt)
            dfs(node.left, (cnt*2))
            dfs(node.right, (cnt*2)+1)
    dfs(root, 1)
    return self.total == self.mx
```

Get a range sum of values

```
def rangeSumBST(self, root: TreeNode, L: int, R: int) -> int:
    self.total = 0
    def helper(node):
        if node is None:
            return 0
        if L <= node.val <= R:
            self.total += node.val
        if node.val > L:
            left = helper(node.left)
        if node.val < R:
            right = helper(node.right)
    helper(root)
    return self.total
```

Check if valid

```
def isValidBST(self, root: TreeNode) -> bool:
    if not root:
        return True
    stk = [(root, float(-inf), float(inf))]
    while stk:
        node, floor, ceil = stk.pop()
        if node:
            if node.val >= ceil or node.val <= floor:
                return False
            stk.append((node.right, node.val, ceil))
            stk.append((node.left, floor, node.val))
    return True
```

Topological Sort

Kahn's algorithm, detects cycles through degrees and needs all the nodes represented to work

1. Initialize vertices as unvisited
2. Pick vertex with zero indegree, append to result, decrease indegree of neighbors
3. Now repeat for neighbors, resulting list is sorted by source -> dest

If cycle, then degree of nodes in cycle will not be 0 since there is no origin

```
def canFinish(self, numCourses: int, prerequisites: List[List[int]]) -> bool:
    # Kahns algorithm, topological sort
    adj = collections.defaultdict(list)
    degree = collections.Counter()

    for dest, orig in prerequisites:
        adj[orig].append(dest)
        degree[dest] += 1

    bfs = [c for c in range(numCourses) if degree[c] == 0]

    for o in bfs:
        for d in adj[o]:
            degree[d] -= 1
            if degree[d] == 0:
                bfs.append(d)

    return len(bfs) == numCourses
```

```
def alienOrder(self, words: List[str]) -> str:
    nodes = set("".join(words))
    adj = collections.defaultdict(list)
    degree = collections.Counter(nodes)

    for w1, w2 in zip(words, words[1:]):
        for c1, c2 in zip(w1, w2):
            if c1 != c2:
                adj[c1].append(c2)
                degree[c2] += 1
                break
        else:
            if len(w1) > len(w2):
                return ""

    stk = list(filter(lambda x: degree[x]==1, degree.keys()))

    ans = []
    while stk:
        node = stk.pop()
        ans.append(node)
        for nei in adj[node]:
            degree[nei] -= 1
            if degree[nei] == 1:
                stk.append(nei)

    return "".join(ans) * (set(ans) == nodes)
```

Sliding Window

1. Have a counter or hash-map to count specific array input and keep on increasing the window toward right using outer loop.
2. Have a while loop inside to reduce the window side by sliding toward right. Movement will be based on constraints of problem.
3. Store the current maximum window size or minimum window size or number of windows based on problem requirement.

Typical Problem Clues:

1. Get min/max/number of satisfied sub arrays
2. Return length of the subarray with max sum/product
3. Return max/min length/number of subarrays whose sum/product equals K

Can require 2 or 3 pointers to solve

```
    def slidingWindowTemplate(self, s: str):
        #init a collection or int value to save the result according the question.
        rtn = []

        # create a hashmap to save the Characters of the target substring.
        # (K, V) = (Character, Frequence of the Characters)
        hm = {}

        # maintain a counter to check whether match the target string as needed
        cnt = collections.Counter(s)

        # Two Pointers: begin - left pointer of the window; end - right pointer of the window if needed
        l = r = 0

        # loop at the begining of the source string
        for r, c in enumerate(s):

            if c in hm:
                l = max(hm[c]+1, l) # +/- 1 or set l to index, max = never move l left

            # update hm
            hm[c] = r

            # increase l pointer to make it invalid/valid again
            while cnt == 0: # counter condition
                cnt[c] += 1  # modify counter if needed

            # Save result / update min/max after loop is valid
            rtn = max(rtn, r-l+1)

        return rtn
```

```
def fruits_into_baskets(fruits):
  maxCount, j = 0, 0
  ht = {}

  for i, c in enumerate(fruits):
    if c in ht:
      ht[c] += 1
    else:
      ht[c] = 1

    if len(ht) <= 2:
      maxCount = max(maxCount, i-j+1)
    else:
```

```
      jc = fruits[j]
      ht[jc] -= 1
      if ht[jc] <= 0:
        del ht[jc]
      j += 1

  return maxCount
```

Greedy

Make the optimal choice at each step.

Increasing Triplet Subsequence, true if i < j < k

```
def increasingTriplet(self, nums: List[int]) -> bool:
    l = m = float('inf')

    for n in nums:
        if n <= l:
            l = n
        elif n <= m:
            m = n
        else:
            return True

    return False
```

Tree Tricks

Bottom up solution with arguments for min, max

```
def maxAncestorDiff(self, root: TreeNode) -> int:
    if not root:
        return 0
    self.ans = 0
    def dfs(node, minval, maxval):
        if not node:
            self.ans = max(self.ans, abs(maxval - minval))
            return
        dfs(node.left, min(node.val, minval), max(node.val, maxval))
        dfs(node.right, min(node.val, minval), max(node.val, maxval))
    dfs(root, float('inf'), float('-inf'))
    return self.ans
```

Building a path through a tree

```
def binaryTreePaths(self, root: TreeNode) -> List[str]:
    rtn = []
    if root is None: return []
    stk = [(root, str(root.val))]
    while stk:
        node, path = stk.pop()
        if node.left is None and node.right is None:
            rtn.append(path)
        if node.left:
            stk.append((node.left, path + "->" + str(node.left.val)))
        if node.right:
            stk.append((node.right, path + "->" + str(node.right.val)))
    return rtn
```

Using return value to sum

```
def diameterOfBinaryTree(self, root: TreeNode) -> int:
    self.mx = 0
    def dfs(node):
        if node:
            l = dfs(node.left)
            r = dfs(node.right)
            total = l + r
            self.mx = max(self.mx, total)
            return max(l, r) + 1
        else:
            return 0
    dfs(root)
    return self.mx
```

Change Tree to Graph

```
def distanceK(self, root: TreeNode, target: TreeNode, K: int) -> List[int]:
    adj = collections.defaultdict(list)

    def dfsa(node):
        if node.left:
            adj[node].append(node.left)
            adj[node.left].append(node)
            dfsa(node.left)
        if node.right:
            adj[node].append(node.right)
            adj[node.right].append(node)
            dfsa(node.right)

    dfsa(root)

    def dfs(node, prev, d):
        if node:
            if d == K:
                rtn.append(node.val)
            else:
                for nei in adj[node]:
                    if nei != prev:
                        dfs(nei, node, d+1)

    rtn = []
    dfs(target, None, 0)
    return rtn
```

Anagrams

Subsection of sliding window, solve with Counter Dict

i.e. abc = bca != eba 111 111 111

```
def isAnagram(self, s: str, t: str) -> bool:
    sc = collections.Counter(s)
    st = collections.Counter(t)
    if sc != st:
        return False
    return True
```

Sliding Window version (substring)

```
def findAnagrams(self, s: str, p: str) -> List[int]:
    cntP = collections.Counter(p)
    cntS = collections.Counter()
    P = len(p)
    S = len(s)
    if P > S:
        return []
    ans = []
    for i, c in enumerate(s):
        cntS[c] += 1
        if i >= P:
            if cntS[s[i-P]] > 1:
                cntS[s[i-P]] -= 1
            else:
                del cntS[s[i-P]]
        if cntS == cntP:
            ans.append(i-(P-1))
    return ans
```

Dynamic Programming

1. dynamic programming

```
def coinChange(self, coins: List[int], amount: int) -> int:
  MAX = float('inf')
  dp =  [MAX] * (amount + 1)
  dp[0] = 0
  for c in coins:
    for a in range(c, amount+1):
      dp[a] =  min(dp[a], dp[a-c]+1)
  return dp[amount] if dp[amount] != MAX else -1
```

Classic DP grid, longest common subsequence

```
def longestCommonSubsequence(self, text1: str, text2: str) -> int:
    Y = len(text2)+1
```

```
    X = len(text1)+1
    dp = [[0] * Y for _ in range(X)]
    # [[0,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0]]
    for i, c in enumerate(text1):
        for j, d in enumerate(text2):
            if c == d:
                dp[i + 1][j + 1] = 1 + dp[i][j]
            else:
                dp[i + 1][j + 1] = max(dp[i][j + 1], dp[i + 1][j])
    return dp[-1][-1]
# [[0,0,0,0],[0,1,1,1],[0,1,1,1],[0,1,2,2],[0,1,2,2],[0,1,2,3]]
# abcde
# "ace"
```

Cyclic Sort

1. Useful algo when sorting in place

```
# if my number is equal to my index, i+1
# if my number is equal to this other number, i+1 (dups)
# else swap
def cyclic_sort(nums):
  i = 0
  while i < len(nums):
    j = nums[i] - 1
    if nums[i] != nums[j]:
      nums[i], nums[j] = nums[j], nums[i]
    else:
      i += 1
  return nums
```

Quick Sort

1. Can be modified for divide in conquer problems

```
def quickSort(array):
        def sort(arr, l, r):
                if l < r:
                        p = part(arr, l, r)
                        sort(arr, l, p-1)
                        sort(arr, p+1, r)

        def part(arr, l, r):
                pivot = arr[r]
                a = l
                for i in range(l,r):
                        if arr[i] < pivot:
                                arr[i], arr[a] = arr[a], arr[i]
                                a += 1
                arr[r], arr[a] = arr[a], arr[r]
                return a

        sort(array, 0, len(array)-1)
        return array
```

Merge Sort

```
from collections import deque
def mergeSort(array):
    def sortArray(nums):
        if len(nums) > 1:
            mid = len(nums)//2
            l1 = sortArray(nums[:mid])
            l2 = sortArray(nums[mid:])
            nums = sort(l1,l2)
        return nums

    def sort(l1,l2):
        result = []
        l1 = deque(l1)
        l2 = deque(l2)
        while l1 and l2:
            if l1[0] <= l2[0]:
                result.append(l1.popleft())
            else:
                result.append(l2.popleft())
        result.extend(l1 or l2)
        return result
        return sortArray(array)
```

Merge Arrays

Merge K sorted Arrays with a heap

```
def mergeSortedArrays(self, arrays):
    return list(heapq.merge(*arrays))
```

Or manually with heappush/heappop.

```
class Solution:
def mergeSortedArrays(self, arrays):
    pq = []
    for i, arr in enumerate(arrays):
        pq.append((arr[0], i, 0))
    heapify(pq)

    res = []
    while pq:
        num, i, j = heappop(pq)
        res.append(num)
        if j + 1 < len(arrays[i]):
            heappush(pq, (arrays[i][j + 1], i, j + 1))
    return res
```

Merging K Sorted Lists

```
def mergeKLists(self, lists: List[ListNode]) -> ListNode:
    prehead = ListNode()
    heap = []
    for i in range(len(lists)):
        node = lists[i]
        while node:
            heapq.heappush(heap, node.val)
            node = node.next
    node = prehead
    while len(heap) > 0:
        val = heapq.heappop(heap)
        node.next = ListNode()
        node = node.next
        node.val = val
    return prehead.next
```

Linked List

1. Solutions typically require 3 pointers: current, previous and next
2. Solutions are usually made simplier with a prehead or dummy head node you create and then add to. Then return dummy.next

Reverse:

```
def reverseLinkedList(head):
    prev, node  = None, head
    while node:
        node.next, prev, node = prev, node, node.next
    return prev
```

Reversing is easier if you can modify the values of the list

```
def reverse(head):
  node = head
  stk = []
  while node:
    if node.data % 2 == 0:
      stk.append(node)
    if node.data % 2 == 1 or node.next is None:
      while len(stk) > 1:
        stk[-1].data, stk[0].data = stk[0].data, stk[-1].data
        stk.pop(0)
        stk.pop(-1)
      stk.clear()
    node = node.next
  return head
```

Merge:

```
def mergeTwoLists(self, l1: ListNode, l2: ListNode) -> ListNode:
```

```
    dummy = ListNode(-1)

    prev = dummy

    while l1 and l2:
        if l1.val < l2.val:
            prev.next = l1
            l1 = l1.next
        else:
            prev.next = l2
            l2 = l2.next
        prev = prev.next

    prev.next = l1 if l1 is not None else l2

    return dummy.next
```

Convert Base

1. Typically two steps. A digit modulo step and a integer division step by the next base then reverse the result or use a deque()

Base 10 to 16, or any base by changing '16' and index

```
def toHex(self, num: int) -> str:
  rtn = []
  index = "0123456789abcdef"
  if num == 0: return '0'
  if num < 0: num += 2 ** 32
  while num > 0:
    digit = num % 16
    rtn.append(index[digit])
    num = num // 16
  return "".join(rtn[::-1])
```

Parenthesis

1. Count can be used if simple case, otherwise stack. Basic Calculator is an extension of this algo

```
def isValid(self, s) -> bool:
  cnt = 0
  for c in s:
    if c == '(':
      cnt += 1
    elif c == ')':
      cnt -= 1
      if cnt < 0:
        return False
  return cnt == 0
```

Stack can be used if more complex

```
def isValid(self, s: str) -> bool:
  stk = []
  mp = {")":"(", "}":"{", "]":"["}
    for c in s:
      if c in mp.values():
        stk.append(c)
      elif c in mp.keys():
        test = stk.pop() if stk else '#'
        if mp[c] != test:
          return False
  return len(stk) == 0
```

Or must store parenthesis index for further modification

```
def minRemoveToMakeValid(self, s: str) -> str:
  rtn = list(s)
  stk = []
  for i, c in enumerate(s):
    if c == '(':
      stk.append(i)
    elif c == ')':
      if len(stk) > 0:
        stk.pop()
      else:
        rtn[i] = ''
  while stk:
    rtn[stk.pop()] = ''
  return "".join(rtn)
```

Max Profit Stock

Infinite Transactions, base formula

```
def maxProfit(self, prices: List[int]) -> int:
    t0, t1 = 0, float('-inf')
    for p in prices:
        t0old = t0
        t0 = max(t0, t1 + p)
        t1 = max(t1, t0old - p)
    return t0
```

Single Transaction, t0 (k-1) = 0

```
def maxProfit(self, prices: List[int]) -> int:
    t0, t1 = 0, float('-inf')
    for p in prices:
        t0 = max(t0, t1 + p)
        t1 = max(t1, - p)
    return t0
```

K Transactions

```
t0 = [0] * (k+1)
t1 = [float(-inf)] * (k+1)
for p in prices:
    for i in range(k, 0, -1):
        t0[i] = max(t0[i], t1[i] + p)
        t1[i] = max(t1[i], t0[i-1] - p)
return t0[k]
```

Shift Array Right

Arrays can be shifted right by reversing the whole string, and then reversing 0,k-1 and k,len(str)

```
def rotate(self, nums: List[int], k: int) -> None:
    def reverse(l, r, nums):
        while l < r:
            nums[l], nums[r] = nums[r], nums[l]
            l += 1
            r -= 1
    if len(nums) <= 1: return
    k = k % len(nums)
    reverse(0, len(nums)-1, nums)
    reverse(0, k-1, nums)
    reverse(k, len(nums)-1, nums)
```

Continuous Subarrays with Sum k

The total number of continuous subarrays with sum k can be found by hashing the continuous sum per value and adding the count of continuous sum - k

```
def subarraySum(self, nums: List[int], k: int) -> int:
    mp = {0: 1}
    rtn, total = 0, 0
    for n in nums:
        total += n
        rtn += mp.get(total - k, 0)
        mp[total] = mp.get(total, 0) + 1
    return rtn
```

Events

Events pattern can be applied when to many interval problems such as 'Find employee free time between meetings' and 'find peak population' when individual start/ends are irrelavent and sum start/end times are more important

```
def employeeFreeTime(self, schedule: '[[Interval]]') -> '[Interval]':
    events = []
    for e in schedule:
        for m in e:
            events.append((m.start, 1))
            events.append((m.end, -1))
```

```
    events.sort()
    itv = []
    prev = None
    bal = 0
    for t, c in events:
        if bal == 0 and prev is not None and t != prev:
            itv.append(Interval(prev, t))
        bal += c
        prev = t
    return itv
```

Merge Meetings

Merging a new meeting into a list

```
def insert(self, intervals: List[List[int]], newInterval: List[int]) -> List[List[int]]:
    bisect.insort(intervals, newInterval)
    merged = [intervals[0]]
    for i in intervals:
        ms, me = merged[-1]
        s, e = i
        if me >= s:
            merged[-1] = (ms, max(me, e))
        else:
            merged.append(i)
    return merged
```

Trie

Good for autocomplete, spell checker, IP routing (match longest prefix), predictive text, solving word games

```
class Trie:
    def __init__(self):
        self.root = {}

    def addWord(self, s: str):
        tmp = self.root
        for c in s:
            if c not in tmp:
                tmp[c] = {}
            tmp = tmp[c]
        tmp['#'] = s # Store full word at '#' to simplify

    def matchPrefix(self, s: str, tmp=None):
        if not tmp: tmp = self.root
        for c in s:
            if c not in tmp:
                return []
            tmp = tmp[c]

        rtn = []

        for k in tmp:
            if k == '#':
                rtn.append(tmp[k])
            else:
                rtn += self.matchPrefix('', tmp[k])
        return rtn

    def hasWord(self, s: str):
        tmp = self.root
        for c in s:
            if c in tmp:
                tmp = tmp[c]
            else:
                return False
        return True
```

Search example with . for wildcards

```
def search(self, word: str) -> bool:
    def searchNode(word, node):
        for i,c in enumerate(word):
            if c in node:
                node = node[c]
            elif c == '.':
                return any(searchNode(word[i+1:], node[cn]) for cn in node if cn != '$' )
            else:
                return False
        return '$' in node
    return searchNode(word, self.trie)
```

Kadane

local_maxiumum[i] = max(A[i], A[i] + local_maximum[i-1]) Explanation Determine max subarray sum

```
# input: [-2,1,-3,4,-1,2,1,-5,4]
def maxSubArray(self, nums: List[int]) -> int:
    for i in range(1, len(nums)):
        if nums[i-1] > 0:
            nums[i] += nums[i-1]
    return max(nums) # max([-2,1,-2,4,3,5,6,1,5]) = 6
```

Union Find

Union Find is a useful algorithm for graph

DSU for integers

```
class DSU:
    def __init__(self, N):
        self.par = list(range(N))

    def find(self, x): # Find Parent
        if self.par[x] != x:
            self.par[x] = self.find(self.par[x])
        return self.par[x]

    def union(self, x, y):
        xr, yr = self.find(x), self.find(y)
        if xr == yr: # If parents are equal, return False
            return False
        self.par[yr] = xr # Give y node parent of x
        return True # return True if union occured
```

DSU for strings

```
class DSU:
    def __init__(self):
        self.par = {}

    def find(self, x):
        if x != self.par.setdefault(x, x):
            self.par[x] = self.find(self.par[x])
        return self.par[x]

    def union(self, x, y):
        xr, yr = self.find(x), self.find(y)
        if xr == yr: return
        self.par[yr] = xr
```

DSU with union by rank

```
class DSU:
    def __init__(self, N):
        self.par = list(range(N))
        self.sz = [1] * N

    def find(self, x):
        if self.par[x] != x:
            self.par[x] = self.find(self.par[x])
        return self.par[x]

    def union(self, x, y):
        xr, yr = self.find(x), self.find(y)
        if xr == yr:
            return False
        if self.sz[xr] < self.sz[yr]:
            xr, yr = yr, xr
        self.par[yr] = xr
        self.sz[xr] += self.sz[yr]
        return True
```

Fast Power

Fast Power, or Exponential by squaring allows calculating squares in logn time (x^n)2 = x^(2n)

```
def myPow(self, x: float, n: int) -> float:
    if n < 0:
        n *= -1
        x = 1/x
    ans = 1
    while n > 0:
        if n % 2 == 1:
            ans = ans * x
        x *= x
        n = n // 2
    return ans
```

Fibonacci Golden

Fibonacci can be calulated with Golden Ratio

```
def fib(self, N: int) -> int:
    golden_ratio = (1 + 5 ** 0.5) / 2
    return int((golden_ratio ** N + 1) / 5 ** 0.5)
```

Basic Calculator

A calculator can be simulated with stack

```
class Solution:
    def calculate(self, s: str) -> int:
        s += '$'
        def helper(stk, i):
            sign = '+'
            num = 0
            while i < len(s):
                c = s[i]
                if c == " ":
                    i += 1
                    continue
                elif c.isdigit():
                    num = num * 10 + int(c)
                    i += 1
                elif c == '(':
                    num, i = helper([], i+1)
                else:
                    if sign == '+':
                        stk.append(num)
                    if sign == '-':
                        stk.append(-num)
                    if sign == '*':
                        stk.append(stk.pop() * num)
                    if sign == '/':
                        stk.append(int(stk.pop() / num))
                    i += 1
                    num = 0
                    if c == ')':
                        return sum(stk), i
                    sign = c
            return sum(stk)
        return helper([],0)
```

Reverse Polish

```
class Solution:
    def evalRPN(self, tokens: List[str]) -> int:
        stk = []
        while tokens:
            c = tokens.pop(0)
            if c not in '+-/*':
                stk.append(int(c))
            else:
                a = stk.pop()
                b = stk.pop()
                if c == '+':
                    stk.append(a + b)
                if c == '-':
                    stk.append(b-a)
                if c == '*':
                    stk.append(a * b)
                if c == '/':
                    stk.append(int(b / a))
        return stk[0]
```

Resevior Sampling

Used to sample large unknown populations. Each new item added has a 1/count chance of being selected

```
def __init__(self, nums):
    self.nums = nums
def pick(self, target):
    res = None
    count = 0
    for i, x in enumerate(self.nums):
        if x == target:
            count += 1
            chance = random.randint(1, count)
            if chance == 1:
                res = i
    return res
```

String Subsequence

Can find the min number of subsequences of strings in some source through binary search and a dict of the indexes of the source array

```
def shortestWay(self, source: str, target: str) -> int:
    ref = collections.defaultdict(list)
    for i,c in enumerate(source):
        ref[c].append(i)

    ans = 1
    i = -1
    for c in target:
        if c not in ref:
            return -1
        offset = ref[c]
        j = bisect.bisect_left(offset, i)
        if j == len(offset):
            ans += 1
            i = offset[0] + 1
        else:
            i = offset[j] + 1

    return ans
```

Candy Crush

Removing adjacent duplicates is much more effective with a stack

```
def removeDuplicates(self, s: str, k: int) -> str:
    stk = []
    for c in s:
        if stk and stk[-1][0] == c:
            stk[-1][1] += 1
            if stk[-1][1] >= k:
                stk.pop()
        else:
            stk.append([c, 1])
    ans = []
    for c in stk:
        ans.extend([c[0]] * c[1])
    return "".join(ans)
```

Dutch Flag

Dutch National Flag Problem proposed by Edsger W. Dijkstra

```
def sortColors(self, nums: List[int]) -> None:
    """
    Do not return anything, modify nums in-place instead.
    """
    # for all idx < p0 : nums[idx < p0] = 0
    # curr is an index of element under consideration
    p0 = curr = 0
    # for all idx > p2 : nums[idx > p2] = 2
    p2 = len(nums) - 1

    while curr <= p2:
        if nums[curr] == 0:
```

```
        nums[p0], nums[curr] = nums[curr], nums[p0]
        p0 += 1
        curr += 1
    elif nums[curr] == 2:
        nums[curr], nums[p2] = nums[p2], nums[curr]
        p2 -= 1
    else:
        curr += 1
```