1. 电话号码 match 到对应字典中的 string 或者是 substring

```
def num2word(nums,dic):
  mapping = {'2': 'abc', '3': 'def', '4': 'ghi', '5': 'jkl',
             '6': 'mno', '7': 'pqrs', '8': 'tuv', '9': 'wxyz'}
  res = []
  def dfs(nums,dic,mapping,prefix,valid,res):
    if not nums:
      if prefix in dic:
        res.append(prefix)
      if valid != 0 and prefix[valid:] in dic:
        res.append(prefix[:valid]+' '+prefix[valid:])
      return
    if nums[0] == '1' or nums[0] == '0':
      return []
    for letter in mapping[nums[0]]:
      if (len(prefix) == 3 or len(prefix) == 4) and prefix in dic:
        valid = len(prefix)
      dfs(nums[1:],dic,mapping,prefix+letter,valid,res)
  dfs(nums,dic,mapping,'',0,res)
  return res
```

可能能够用到 trie

如果有 access to the dic。 我们可以吧 dic 存在 hashmap 里面这样来寻找 string 是不是 valid。或者是存在 trie 里面。

一个不成熟想法：map 所有 dic 里面的值回 number。这样就能直接 check 输入的 number 了。

2. Word Pattern

String 有空格隔开的情况

```
class Solution(object):
    def wordPattern(self, pattern, str):
        s = str.split(' ')
        if len(pattern) != len(s):
            return False
        dic = dict()
        seen = set()
        for i in range(len(pattern)):
            if pattern[i] not in dic:
                if s[i] in seen:
                    return False
                dic[pattern[i]] = s[i]
                seen.add(s[i])
            else:
                if dic[pattern[i]] != s[i]:
                    return False
        return True
```

没空格隔开：（有可能 pattern 是对称的所以可以分成两半 partition）

```
class Solution(object):
    def wordPatternMatch(self, pattern, str):
        def helper(pattern,str,pat2str):
            if (len(pattern) == 0 or len(str) == 0) and len(pattern) != len(str):
                return False
            if len(pattern) == 0 and len(str) == 0:
                return True
            for stop in range(1,len(str)-len(pattern)+2):
                if pattern[0] in pat2str and pat2str[pattern[0]] == str[:stop]:
                    if helper(pattern[1:],str[stop:],pat2str):
                        return True
                elif pattern[0] not in pat2str and str[:stop] not in pat2str.values():
                    pat2str[pattern[0]] = str[:stop]
                    if helper(pattern[1:],str[stop:],pat2str):
                        return True
                    del pat2str[pattern[0]]
            return False
        return helper(pattern,str,{})
```

3. Design Phone Directory

(1) Bit set O(n)

```
class PhoneDirectory(object):

    def __init__(self, maxNumbers):
        self.maxnum = maxNumbers
        self.record = 0

    def get(self):
        for i in range(self.maxnum):
            if self.record & 1 << i == 0:
                self.record = self.record | 1<<i
                return i
        return -1

    def check(self, number):
        return True if self.record & 1 << (number) == 0 else False

    def release(self, number):
        if self.record & 1 << (number):
            self.record = self.record ^ (1<<(number))
```

(2) Bit set and queue

```
class PhoneDirectory(object):

    def __init__(self, maxNumbers):
        self.maxnum = maxNumbers
        self.record = 0
        self.queue = [i for i in range(maxNumbers)]

    def get(self):
        if not self.queue:
            return -1
        dire = self.queue.pop()
        self.record = self.record | 1<<dire
        return dire

    def check(self, number):
        if number < 0 or number > self.maxnum:
            return False
        return True if self.record & 1 << (number) == 0 else False
```

```
    def release(self, number):
        if self.record & 1 << (number):
            self.queue.append(number)
            self.record = self.record ^ (1<<(number))
```

(3) Using hash set

```
class PhoneDirectory(object):

    def __init__(self, maxNumbers):
        self.maxNumbers = maxNumbers
        self.available = set(range(maxNumbers))

    def get(self):
        return self.available.pop() if self.available else -1

    def check(self, number):
        return number in self.available

    def release(self, number):
        if number < self.maxNumbers and number >= 0:
            self.available.add(number)
```

(4) Using page table

```
class PhoneDirectory(object):

    def __init__(self, maxNumbers):
        self.maxNumbers = maxNumbers
        self.N = 64
        self.pages = [0 for _ in range(maxNumbers/self.N + 1)]
        self.capa = 2^self.N-1
        self.lastpagemax = maxNumbers%self.N

    def get(self):
        for ind in range(len(self.pages)):
            if self.pages[ind] == self.capa:
                continue
            if ind == len(self.pages)-1:
                N = self.lastpagemax
            else:
                N = self.N
            for i in range(N):
                if self.pages[ind] & 1<<i == 0:
                    self.pages[ind] = self.pages[ind] | 1<<i
                    return self.N*ind+i
        return -1

    def check(self, number):
        return True if self.pages[number/self.N] & 1 << (number%self.N) == 0 else False

    def release(self, number):
        if number < self.maxNumbers and number >= 0:
            if self.pages[number/self.N] & 1 << (number%self.N):
                self.pages[number/self.N] = self.pages[number/self.N] ^ 1 << (number%self.N)
```

4. Word break

```
        def helper(self, s, wordDict, memo):
            if s in memo: return memo[s]
            if not s: return []
            res = []
            for word in wordDict:
                if word == s:
                    res.append(word)
                elif s[:len(word)] == word:
                    resultOfTheRest = self.helper(s[len(word):], wordDict, memo)
                    for item in resultOfTheRest:
                        item = word + ' ' + item
                        res.append(item)
            memo[s] = res
            return res
```

```
def wordBreak(self, s, wordDict):
    """
    :type s: str
    :type wordDict: List[str]
    :rtype: List[str]
    """
    if not s or not wordDict:
        return []
    dpmatrix = [[]for j in range(len(s)+1)]
    dpmatrix[0] = ['']
    for row in range(1,len(dpmatrix)):
        for ele in wordDict:
            if row - len(ele) > -1 and dpmatrix[row-len(ele)] != [] and ele == s[row-len(ele):row]:
                dpmatrix[row] += [pre +' '+  ele if pre != '' else ele for pre in dpmatrix[row-len(ele)]]
    return dpmatrix[-1]
```

5. Game of life

class Solution(object): 简单（3 象征上次是 1 这次是 0，2 象征上次是 1 这次是 0）

```
    def gameOfLife(self, board):
        def statusCheck(board,row,col):
            alive = 0
            for i in range(row-1,row+2):
                for j in range(col-1,col+2):
                    if row==i and col == j:
                        continue
                    if 0<=i<len(board) and 0<=j<len(board[0]):
                        if board[i][j] == 1 or board[i][j] == 3:
                            alive += 1
            if board[row][col] == 0:
                if alive == 3:
                    board[row][col] = 2
            else:
                if alive != 2 and alive != 3:
                    board[row][col] = 3

        for row in range(len(board)):
            for col in range(len(board[0])):
                statusCheck(board,row,col)
        for row in range(len(board)):
            for col in range(len(board[0])):
                if board[row][col] == 2:
                    board[row][col] = 1
                elif board[row][col] == 3:
                    board[row][col] = 0
```

```
class Solution(object):  用了位操作
    def gameOfLife(self, board):
        """
        :type board: List[List[int]]
        :rtype: void Do not return anything, modify board in-place instead.
        """
        if not board or not board[0]:
            return
        def statusUpdate(board,row,col):
            alive = 0
            for i in range(max(row-1,0),min(len(board),row+2)):
                for j in range(max(col-1,0),min(len(board[0]),col+2)):
                        alive += board[i][j] & 1
            alive -= board[row][col] & 1
            if alive == 3 or board[row][col]+alive == 3:
                board[row][col] |= 2

        for i in range(len(board)):
            for j in range(len(board[0])):
                statusUpdate(board,i,j)
        for i in range(len(board)):
            for j in range(len(board[0])):
                board[i][j] >>= 1
```

6. Num of islands(1)

class Solution(object): (简单版)

```
def numIslands(self, grid):
    if not grid or not grid[0]:
        return 0
    def checkArround(grid,row,col):
        if row <0 or row >=len(grid) or col <0 or col >= len(grid[0]) or grid[row][col] == '0':
            return
        grid[row][col] = '0'
        checkArround(grid,row+1,col)
        checkArround(grid,row-1,col)
        checkArround(grid,row,col+1)
        checkArround(grid,row,col-1)
    count = 0
    for i in range(len(grid)):
        for j in range(len(grid[0])):
            if grid[i][j] == '1':
                count += 1
                checkArround(grid,i,j)
    return count
```

class Solution(object): （逐渐向里面增加可能出现的陆地）

```
    """
    :type m: int
    :type n: int
    :type positions: List[List[int]]
    :rtype: List[int]
    """
  def numIslands2(self, m, n, positions):
    parent = {}
    def find(x):
      if parent[x] != x:
        parent[x] = find(parent[x])
      return parent[x]

    def union(x, y):
      x, y = find(x), find(y)
      if x == y:
        return 0
      parent[y] = x
      return 1

    counts, count = [], 0
    for i, j in positions:
      x = parent[x] = i, j
      count += 1
      for y in (i+1, j), (i-1, j), (i, j+1), (i, j-1):
        if y in parent:
          count -= union(x, y)
      counts.append(count)
    return counts
```

7. Duplicate files in system

```
class Solution(object):
    def findDuplicate(self, paths):
        """
        :type paths: List[str]
        :rtype: List[List[str]]
        """
        filecont = {}
        for path in paths:
            path = path.split()
            for i in range(1,len(path)):
                name,cont = path[i].split('(')
                if cont[:-1] in filecont:
                    filecont[cont[:-1]].append(path[0]+'/'+name)
                else:
                    filecont[cont[:-1]] = [path[0]+'/'+name]
        temp = filecont.values()
        res = []
        for ele in temp:
            if len(ele) > 1:
                res.append(ele)
        return res
```

1. Imagine you are given a real file system, how will you search files? DFS or BFS ?
In general, BFS will use more memory then DFS. However BFS can take advantage of the locality of files in inside directories, and therefore will probably be faster

2. If the file content is very large (GB level), how will you modify your solution?
In a real life solution we will not hash the entire file content, since it's not practical. Instead we will first map all the files according to size. Files with different sizes are guaranteed to be different. We will than hash a small part of the files with equal sizes (using MD5 for example). Only if the md5 is the same, we will compare the files byte by byte

3. If you can only read the file by 1kb each time, how will you modify your solution?
This won't change the solution. We can create the hash from the 1kb chunks, and then read the entire file if a full byte by byte comparison is required.

What is the time complexity of your modified solution? What is the most time consuming part and memory consuming part of it? How to optimize?
Time complexity is O(n^2 * k) since in worse case we might need to compare every file to all others. k is the file size

How to make sure the duplicated files you find are not false positive?
We will use several filters to compare: File size, Hash and byte by byte comparisons.

8. Sharpness value in path

```
def find_highest_minimum_sharpness(board):
    rows, cols = len(board), len(board[0])
    for c in xrange(1, cols):
        for r in xrange(rows):
            left_max = board[r][c - 1]
            if r - 1 >= 0:
                left_max = max(left_max, board[r - 1][c - 1])
            if r + 1 < rows:
                left_max = max(left_max, board[r + 1][c - 1])
            board[r][c] = min(board[r][c], left_max)
    res = board[0][-1]
    for r in xrange(1, rows):
        res = max(res, board[r][-1])
    return res
```

followup:如果矩阵很大不能够一次读进来怎么办？
可以每次读一列。
但是文件读取的时候每一列读取速度很慢
那我们 transpose 这个矩阵然后把它保存在另一个 file 里面。一小块儿一小块儿的读然后翻转。

9. 水淹路径

```
def findPath1(board):
    for col in xrange(1,len(board[0])):
        for row in xrange(len(board)):
            if board[row][col] < col:
                continue
            leftmin = []
            if board[row][col-1] >= col-1:
                leftmin.append(board[row][col-1])
            if row > 0 and board[row-1][col-1] >= col-1:
                leftmin.append(board[row-1][col-1])
            if row < len(board)-1 and board[row+1][col-1] >= col-1:
                leftmin.append(board[row+1][col-1])
            board[row][col] = max(board[row][col],min(leftmin))
    res = []
    for i in xrange(len(board)):
        if board[i][-1] >= len(board[0])-1:
            res.append(board[i][-1])
    return min(res)
```

10. Soda combination:

DFS：

```
def combinationSum(self, candidates, target):
    """
    :type candidates: List[int]
    :type target: int
    :rtype: List[List[int]]
    """
    candidates = sorted(candidates,reverse = True)
```

```
    level = len(candidates)
    def sub(candidates,remain,prefix):
      if remain == 0:
        return [prefix]
      if not candidates or remain < 0:
        return []
      units = remain/candidates[0]
      res = []
      for i in range(units+1):
       nextsub = sub(candidates[1:],remain-i*candidates[0],prefix + [candidates[0] for j in range(i)])
        res = res+nextsub
      return res
    return sub(candidates,target,[])
```

The fact that we are doing brute force gives us the answer of complexity. If you think, we are essentially selecting all possible subsets of of set.

{1,2,3} -> {1} {2} {3} {1,2} {1,3} {2,3} {1,2,3}

There are **2 ^n** such elements and hence the time complexity is **O(2^n)**

DP：

```
def sodaComb(candidates,total):
  dpmatrix = [[[] for i in xrange(len(candidates))] for _ in xrange(total+1)]
  dpmatrix[0] = [[[]] for _ in xrange(len(candidates))]
  for row in xrange(1,len(dpmatrix)):
    for col in xrange(len(dpmatrix[0])):
      if row - candidates[col] > -1 and dpmatrix[row - candidates[col]][col] != []:
        dpmatrix[row][col].extend([ele+[candidates[col]] for ele in dpmatrix[row - candidates[col]][col]])
      if col-1>-1 and dpmatrix[row][col-1] != []:
        dpmatrix[row][col].extend(dpmatrix[row][col-1])
  return dpmatrix[-1][-1]
```

11. Bit torrent

Bit map 用一个 bit 代表一个 byte 看是不是都覆盖了

Merge interval

```
 # Definition for an interval.
# class Interval(object):
#     def __init__(self, s=0, e=0):
#         self.start = s
#         self.end = e

class Solution(object):
  def merge(self, intervals):
    """
    :type intervals: List[Interval]
    :rtype: List[Interval]
    """
```

```
if not intervals:
    return []
intervals = sorted(intervals,key=lambda x: x.start)
stack = [intervals[0]]
for i in range(1,len(intervals)):
    if intervals[i].start > stack[-1].end:
        stack.append(intervals[i])
    else:
        stack[-1].end = max(intervals[i].end,stack[-1].end)
return stack
```

https://leetcode.com/problems/merge-intervals/discuss/21452/share-my-interval-tree-solution-no-sorting

12. Hit counter

用一个 array 实现里面存的是 timestamp：

```
class HitCounter(object):

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.clicks = []

    def hit(self, timestamp):
        """
        Record a hit.
        @param timestamp - The current timestamp (in seconds granularity).
        :type timestamp: int
        :rtype: void
        """
        self.clicks.append(timestamp)
        self.update(timestamp)


    def getHits(self, timestamp):
        """
        Return the number of hits in the past 5 minutes.
        @param timestamp - The current timestamp (in seconds granularity).
        :type timestamp: int
        :rtype: int
        """
        self.update(timestamp)
        return len(self.clicks)


    def update(self,timestamp):
```

```
            for i in range(len(self.clicks)-1,-1,-1):
                if self.clicks[i] <= timestamp - 300:
                    self.clicks = self.clicks[i+1:]
                    break
```

```
Class ListNode(object):
  def __init__(time):
      self.val = time
      self.next = None

class HitCounter(object):  (链表 node 里面存 timestamp)

    def __init__(self):
        self.head = ListNode(0)
        self.c = self.head
        self.count = 0

    def hit(self, timestamp):
        self.c.next = ListNode(timestamp)
        self.c = self.c.next
        self.count += 1
        self.clean(timestamp)

    def getHits(self, timestamp):
        self.clean(timestamp)
        return self.count

    def clean(self, timestamp):
        bound = timestamp - 300
        h = self.head
        while h.next and h.next.val <= bound:
            h = h.next
            self.count -= 1
        self.head = h
```

用一个 list 存 node,node 里面有时间和此时间 hit 的次数

```
class TimeNode(object):
    def __init__(self,t):
        self.time = t
        self.hit = 1
class HitCounter(object):

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.clicks = []
        self.size = 300
        self.count = 0

    def hit(self, timestamp):
        """
        Record a hit.
        @param timestamp - The current timestamp (in seconds granularity).
        :type timestamp: int
        :rtype: void
        """
        if self.clicks and self.clicks[-1].time == timestamp:
            self.clicks[-1].hit += 1
        else:
            self.clicks.append(TimeNode(timestamp))
        self.count += 1
        self.update(timestamp)


    def getHits(self, timestamp):
        self.update(timestamp)
```

```
        return self.count


    def update(self,timestamp):
        for i in range(len(self.clicks)):
            if self.clicks[i].time <= timestamp - self.size:
                self.count -= self.clicks[i].hit
            else:
                self.clicks = self.clicks[i:]
                return
        self.clicks = []
```

调用 update when necessary

```
    class TimeNode(object):
        def __init__(self,t):
            self.time = t
            self.hit = 1
    class HitCounter(object):

        def __init__(self):
            self.clicks = []
            self.size = 300
            self.count = 0
            self.earlist = 0

        def hit(self, timestamp):
            if self.clicks and self.clicks[-1].time == timestamp:
                self.clicks[-1].hit += 1
            else:
                self.clicks.append(TimeNode(timestamp))
            self.count += 1
            if self.earlist <= timestamp - self.size:
                self.update(timestamp)


        def getHits(self, timestamp):
            """
            Return the number of hits in the past 5 minutes.
            @param timestamp - The current timestamp (in seconds granularity).
            :type timestamp: int
            :rtype: int
            """
            if self.earlist <= timestamp - self.size:
                self.update(timestamp)
            return self.count
```

```
def update(self,timestamp):
    for i in range(len(self.clicks)):
        if self.clicks[i].time <= timestamp - self.size:
            self.count -= self.clicks[i].hit
        else:
            self.clicks = self.clicks[i:]
            self.earlist = self.clicks[0].time
            return
    self.clicks = []
```

If there are multiple threads accessing the structs, then there are two solutions. One is to have a lock on top of it. The second is to have thread-local version of the struct, so when updating, update the thread-local one, and when querying, get all data and sum them together. Both version requires locks, however, the first solution, every hit may wait on the lock while the second solution, only when calling getcount() may block the lock. In the real life, usually hit() is called much higher frequency than getcount(), so the second solution could perform better. But the second solution also harder to implemented, so there is another trade-off need to be made, depends on the real requirement.

13. Top k frequent elements

智障解法：

```
class Solution(object):
    def topKFrequent(self, nums, k):
        """
        :type nums: List[int]
        :type k: int
        :rtype: List[int]
        """
        dic = {}
        for ele in nums:
            dic[ele] = dic.get(ele,0)+1
        pairs = dic.items()
        pairs = sorted(pairs,key = lambda pair:pair[1],reverse = True)
        return [pairs[i][0] for i in range(k)]
```

用 heap:

```
import heapq
class Solution(object):
    def topKFrequent(self, nums, k):
        """
        :type nums: List[int]
        :type k: int
        :rtype: List[int]
        """
        freq = {}
        freq_list=[]
        for num in nums:
            if num in freq:
                freq[num] = freq[num] + 1
            else:
                freq[num] = 1
        for key in freq.keys():
            freq_list.append((-freq[key], key))
        heapq.heapify(freq_list)
        topk = []
        for i in range(0,k):
            topk.append(heapq.heappop(freq_list)[1])
        return topk
```

14. 给定一个字典还有还有 string 的数组，看能拼出来的最长的 string 是啥。

（1）.求出所有可能的拼法。

（2）.维护一个 trie。然后比较的时候吧 trie 的节点也带进去比较。这样可以避免很多不必要的递归。

```
def findLongest(dic,wordList):
    Trie = {}
    for word in dic:
        node = Trie
        for letter in word:
            if letter not in node:
                node[letter] = {}
            node = node[letter]
        node['*'] = 1
    def dfs(node,wordList):
        if not wordList:
            return ''
        res = []
        for i in range(len(wordList)):
            passed = True
            newnode = node
            for letter in wordList[i]:
                if letter not in newnode:
                    passed = False
                    break
                else:
                    newnode = newnode[letter]
            if passed:
                if '*' in newnode:
                    res.append(wordList[i])
                later = dfs(newnode,wordList[:i]+wordList[i+1:])
                if later:
                    res.append(wordList[i]+later)
        ret = ''
        for ele in res:
            if len(ele)>len(ret):
                ret = ele
        return ret
    return dfs(Trie,wordList)
```

15. String in file:

弱智解法，每一次读一个 buffer，然后把 buffer 最后的 len(pattern)-1 挪到前面然后继续读。挨个比较 string 是不是和 buffer 里面的 substring 重合

```
def byteinfile(path,name,pattern):

    N = 1000
    # fullfill the buf if enough content left in the file
    # else return a shorter buf with remain content in file
    buf = readfile(N,path,name)
    while buf:
        if len(buf)<len(patter):
            break
        for i in range(len(buf)-len(pattern)+1):
            if buf[i:i+len(pattern)] == pattern:
                return True
            buf= buf[len(buf)-len(pattern)+1:]+readfile(len(buf)-len(pattern)+1,path,name)
    return False
```

Rolling-hash:

```
def byteinfile(path,name,pattern):
    N = 1000
    base = 7
    buf = readfile(N,path,name)
    aimhashv = 0
    curhashv = 0
    for i in range(len(pattern)):
        curhashv += ord(buf[i]*base**i)
        aimhashv += ord(pattern[i])*base**i
    sub = buf[:len(pattern)]
    if sub == pattern:
        return True
    buf = buf[len(pattern):]
    while buf:
        for i in range(len(buf)):
            curhashv = (curhashv - ord(sub[0]))/base+buf[i]*base**(len(pattern)-1)
            sub = sub[1:]+buf[i]
            if curhashv == aimhashv:
                if sub == pattern:
                    return True
        buf = readfile(N,path,name)
    return False
```