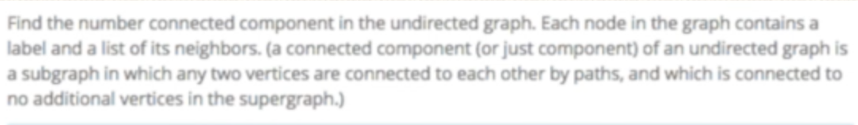
总结：题越做越少



Find the number connected component in the undirected graph. Each node in the graph contains a label and a list of its neighbors. (a connected component (or just component) of an undirected graph is a subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the supergraph.)

并查集

```
class node:
    def __init__(self, val):
        self.val = val
        self.staff = 1
        self.level = 1
        self.boss = self

class QuickUnion:
    def findwWthCompress(self, a):
         while a.boss != a:
             a = a.boss
         bigboss = a
         while a.boss != a:
             next = a.boss
             a.boss = bigboss
             a = next


     def unionWithWeight(self, a, b):
         boss_a = self.find(a)
         boss_b = self.find(b)
         if boss_a != boss_b:
             if boss_a.staff < boss_b.staff:
                 boss_a.boss = boss_b
                 boss_b.staff += boss_a.staff
             else:
                 boss_b.boss = boss_a
                 boss_a.staff += boss_b
     def unionByLevel(self, a, b):
         boss_a = self.find(a)
         boss_b = self.find(b)
         if boss_a != boss_b:
             if boss_a.level <= boss_b.level:
                 boss_a.boss = boss_b
                 if boss_a.level == boss_b.level:
                     boss_b.level += 1
             else:
                 boss_b.boss = boss_a

     def find(self, a):
         while a.boss != a:
             a = a.boss
         return a
     def union(self, a, b):
         boss_a = self.find(a)
         boss_b = self.find(b)
         if boss_a != boss_b:
             boss_b.boss = boss_a
             boss_a.staff += boss_b.staff




```

弱连通块和强连通块

弱连通块：你籍籍无名，认识一个名人，名人不认识你，但是依旧在一个圈子里

求有向图的弱连通块

1. 法一：可以把该图转化成无向图。图如果用矩阵存，很容易。图如果用临接表存，较困难。
2. 并查集，注意自环时间复杂度边的时间复杂度O（m*n）

带路径压缩的查找时间复杂度为O(1)

二维转化一维：id=xm+y

一维转化二维：x=id/m; y=id%m

Find the Connected Component in the Undirected Graph

Find the number connected component in the undirected graph. Each node in the graph contains a label and a list of its neighbors. (a connected component (or just component) of an undirected graph is a subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the supergraph.)

Example

Given graph:

A——B C–E

|

D

Return {A,B,D}, {C,E}. Since there are two connected component which is {A,B,D}, {C,E}

```
class UDGNode:
    def __init__(self, name, neighbors):
        self.name = name
        self.neighbors = neighbors
```

一般不会这样初始化，neighbors很难在它声明之前全部声明了，因为可能有环存在

所以一般的做法是初始化的时候是空，后面在加上邻居。

Find the Weak Connected Component in the Directed Graph

Find the number Weak Connected Component in the directed graph. Each node in the graph contains a label and a list of its neighbors. (a connected set of a directed graph is a subgraph in which any two vertices are connected by direct edge path.)

Number of Islands

1. 首先在使用并查集的时候把矩阵一维化
2. 每个陆地板块只扩张右板块和上板块。所以不需要visited数组。

```
class UnionFind:
    def __init__(self, num):
        self.num = num
        self.rec = [i for i in range(num)]

    def find(self, a):
        assert a < self.num
        while a != self.rec[a]:
            a = self.rec[a]
         return a

     def union(self, a, b):
         assert a < self.num
         assert b < self.num
         fa = self.find(a)
         fb = self.find(b)
         if fa != fb:
             self.rec[fa] = fb

     def countPlate(self):
         c = 0
         for i in range(self.num):
             if self.rec[i] == i:
                 c += 1
         return c


 class Solution:
     """
     @param grid: a boolean 2D matrix
     @return: an integer
     """

     def numIslands(self, grid):
         # write your code here
         if not grid:
             return 0
         n, m = len(grid), len(grid[0])
         uf = UnionFind(n * m)
         water = 0
         for i in range(n):
             for j in range(m):
                 if grid[i][j] == 0:
                     water += 1
                     continue
                 for x, y in [(i + 1, j), (i, j + 1)]:
                     if x >= n or y >= m or grid[x][y] == 0:
                         continue
                     uf.union(i * m + j, x * m + y)
         return uf.countPlate() - water
```

Number of Islands i i

```

    def numIslands2(self, n, m, operators):
        # write your code here

       def find(uf, a):
           if uf[a] == -1:
               return -1
           while a != uf[a]:
               a = uf[a]
            return a
        def union(uf, a, b):
            fa = find(uf, a)
            fb = find(uf, b)
            if fa != fb:
                uf[fa] = fb
                return True
            return False
        ans = 0
        ret = []
        if n==0 or m==0:
            return 0
        water = n*m
        uf = [-1 for _ in range(n*m)]

        for p in operators:
            if uf[p.x*m+p.y] != -1:
                ret.append(ans)
                continue
            uf[p.x*m+p.y] = p.x*m+p.y
            ans += 1
            for x, y in [(p.x+1,p.y),(p.x,p.y+1),(p.x-1,p.y),(p.x,p.y-1)]:
                if x<0  or y<0 or x>=n or y>=m or uf[x*m+y] == -1:
                    continue
                if union(uf, p.x*m+p.y, x*m+y):
                    ans -= 1
            ret.append(ans)
        return ret
```

valid tree

1. 给出点的个数只有一个n==1，肯定是树

2.给出边的个数不是n-1，肯定不是树

3. 并查集看他们是否在一个集合内。

Surrounded Regions

这次我使用递归求解连通区域的办法，可以通过测试

trie树不支持删减

怎么查找后缀在不在。再建一个trie树，倒着把单词插进去

Implement Trie

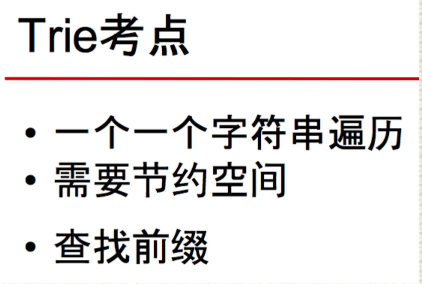
每个单词的结束必须标记一下，不然弄不清楚它到底是一个完整的单词还是只是一个前缀

```

class Node:
    def __init__(self):
        self.isword = False
        self.child = dict()
```

```python
class Trie:

    def __init__(self):
        self.root = Node()

     def insert(self, word):
         node = self.root
         for letter in word:
             child = node.child.get(letter)
             if not child:
                 child = Node()
                 node.child[letter] = child
             node = child
         node.isword = True

     def startsWith(self, word):
         node = self.root
         for letter in word:
             child = node.child.get(letter)
             if not child:
                 return False
             node = child
         return True

     def search(self, word):
         node = self.root
         for letter in word:
             child = node.child.get(letter)
             if not child:
                 return False
             node = child
         if node.isword:
             return True
         return False



```

trie和hash比较

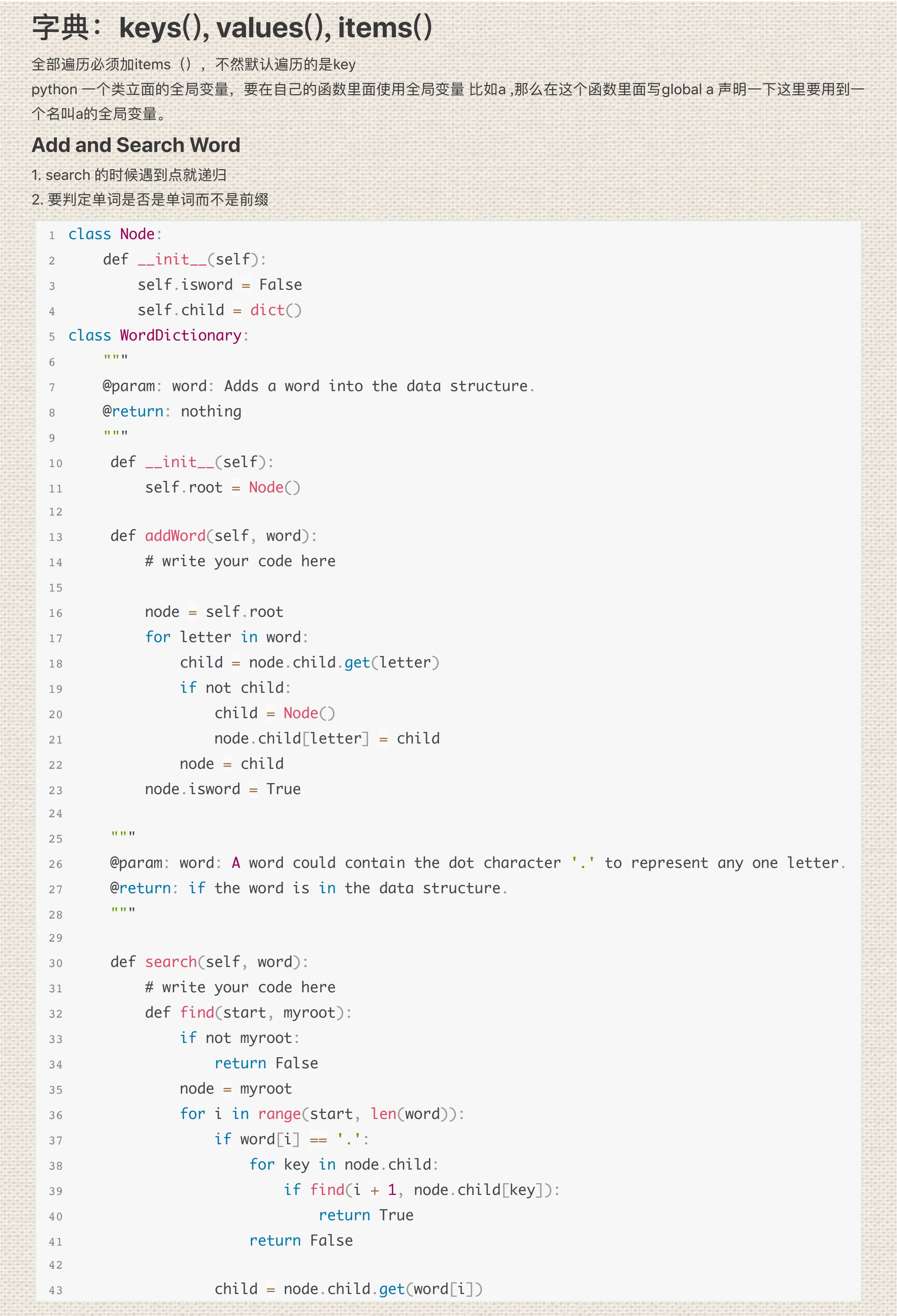
1. trie和hash.时间复杂度一样，空间复杂度trie好一些。



Trie考点

- 一个一个字符串遍历
- 需要节约空间
- 查找前缀

set指定元素删除，remove（）



字典： keys(), values(), items()

全部遍历必须加items（），不然默认遍历的是key

python 一个类立面的全局变量，要在自己的函数里面使用全局变量 比如a ,那么在这个函数里面写global a 声明一下这里要用到一个名叫a的全局变量。

Add and Search Word

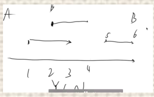
1. search 的时候遇到点就递归
2. 要判定单词是否是单词而不是前缀

```
class Node:
    def __init__(self):
        self.isword = False
        self.child = dict()
class WordDictionary:
    """
    @param: word: Adds a word into the data structure.
    @return: nothing
    """
     def __init__(self):
         self.root = Node()

     def addWord(self, word):
         # write your code here

         node = self.root
         for letter in word:
             child = node.child.get(letter)
             if not child:
                 child = Node()
                 node.child[letter] = child
             node = child
         node.isword = True

     """
     @param: word: A word could contain the dot character '.' to represent any one letter.
     @return: if the word is in the data structure.
     """

     def search(self, word):
         # write your code here
         def find(start, myroot):
             if not myroot:
                 return False
             node = myroot
             for i in range(start, len(word)):
                 if word[i] == '.':
                     for key in node.child:
                         if find(i + 1, node.child[key]):
                             return True
                     return False

                 child = node.child.get(word[i])
```

```
                if not child:
                    return False
                node = child
            if node.isword:
                return True
            return False


        return find(0, self.root)

```

sweep line

把起点和终点拆分开并且标记好起点和终点。把这些所有点排个序
从前到后扫描这些点，是起点就加1，是终点就减一。



1. 开会，需要会议室数目
2. 需要多少铁轨让这些火车不想撞

python多条件排序

temp = sorted(temp, key = lambda x: (x[0], x[1]), reverse=False)

扫描线类问题最重要的是交界处，也就是同一时刻有起有落，那么这个时刻要一起处理。

事情是这样子的。
起初对这道题的想法是按照时间线排个序，扫描每个时间点。
如果是起始时间点，就把此刻新高度放到堆里面。如果该高度比堆顶还大，那就记录一下新的区间（）
如果是结束时间点，就把此刻高度从堆里面弹出来。如果该高度是唯一的堆顶，那就记录一下新区间。
未考虑全面的因素：

1. 没有考虑拿到根本没高度的区间例如：(2,4,4),(5,6,2).这个结果会变成：(2,4,4),(4,6,2).

   解决方法：在hashheap里面预先存放一个0，表示默认值是0,每次记录新区间的时候坚持堆顶是0就跳过并更新sweep
2. 没有考虑你起始点重复区间例如：(1,2,3),(1,2,4).这个结果会变成 (1,1,3),(1,2,4)

   解决方法：每次记录新区间的时候判断区间两端点是否相等
3. 没有考虑起点和终点同时，而且高度相等的情况例如：(1,2,3),(2,3,3),不能正确输出（1,3,3)

   没有解决方法，推翻重来

预处理的原则，不要动数据基本结构，也就是你要删除一个起点，必须删除一个终点.

别人的解法：

感悟，遇到区间边界重叠的情况可以记录一个x+Δx的情况

```
class HashHeap:

    def __init__(self, desc=False):
        self.hash = dict()
        self.heap = []
        self.desc = desc

    @property
```

```python
    def size(self):
        return len(self.heap)

    def push(self, item):
        self.heap.append(item)
        self.hash[item] = self.size - 1
        self._sift_up(self.size - 1)

    def pop(self):
        item = self.heap[0]
        self.remove(item)
        return item

    def top(self):
        return self.heap[0]

    def remove(self, item):
        if item not in self.hash:
            return

        index = self.hash[item]
        self._swap(index, self.size - 1)

        del self.hash[item]
        self.heap.pop()

        # in case of the removed item is the last item
        if index < self.size:
            self._sift_up(index)
            self._sift_down(index)

    def _smaller(self, left, right):
        return right < left if self.desc else left < right

    def _sift_up(self, index):
        while index != 0:
            parent = (index - 1) // 2
            if self._smaller(self.heap[parent], self.heap[index]):
                break
            self._swap(parent, index)
            index = parent

    def _sift_down(self, index):
        if index is None:
            return
        while index * 2 + 1 < self.size:
            smallest = index
            left = index * 2 + 1
            right = index * 2 + 2

            if self._smaller(self.heap[left], self.heap[smallest]):
                smallest = left

            if right < self.size and self._smaller(self.heap[right], self.heap[smallest]):
                smallest = right

            if smallest == index:
                break

            self._swap(index, smallest)
            index = smallest

    def _swap(self, i, j):
        elem1 = self.heap[i]
        elem2 = self.heap[j]
        self.heap[i] = elem2
        self.heap[j] = elem1
        self.hash[elem1] = j
        self.hash[elem2] = i


class Solution:
    """
    @param buildings: A list of lists of integers
    @return: Find the outline of those buildings
    """
    def buildingOutline(self, buildings):
        points = []
        for index, (start, end, height) in enumerate(buildings):
            points.append((start, height, index, True))
            points.append((end, height, index, False))
        points = sorted(points)

        maxheap = HashHeap(desc=True)
        intervals = []
        last_position = None
        for position, height, index, is_start in points:
            max_height = maxheap.top()[0] if maxheap.size else 0
            self.merge_to(intervals, last_position, position, max_height)
            if is_start:
                maxheap.push((height, index))
            else:
                maxheap.remove((height, index))
            last_position = position

        return intervals

    def merge_to(self, intervals, start, end, height):
        if start is None or height == 0 or start == end:
            return

        if not intervals:
            intervals.append([start, end, height])
            return

        _, prev_end, prev_height = intervals[-1]
        if prev_height == height and prev_end == start:
            intervals[-1][1] = end
            return

        intervals.append([start, end, height])
```

```
class HashHeap:

    def __init__(self):
        self.heap = [0]
        self.hash = {}

    def add(self, key, value):
        self.heap.append((key, value))
        self.hash[key] = self.heap[0] + 1
        self.heap[0] += 1
        self._siftup(self.heap[0])

    def remove(self, key):
        index = self.hash[key]
        self._swap(index, self.heap[0])
        del self.hash[self.heap[self.heap[0]][0]]
        self.heap.pop()
        self.heap[0] -= 1
        if index <= self.heap[0]:
            index = self._siftup(index)
            self._siftdown(index)

    def hasKey(self, key):
        return key in self.hash

    def max(self):
        return 0 if self.heap[0] == 0 else self.heap[1][1]

    def _swap(self, a, b):
        self.heap[a], self.heap[b] = self.heap[b], self.heap[a]
        self.hash[self.heap[a][0]] = a
        self.hash[self.heap[b][0]] = b

    def _siftup(self, index):
        while index != 1:
            if self.heap[index][1] <= self.heap[index / 2][1]:
                break
            self._swap(index, index / 2)
            index = index / 2
        return index

    def _siftdown(self, index):
        size = self.heap[0]
        while index < size:
            t = index
            if index * 2 <= size and self.heap[t][1] < self.heap[index * 2][1]:
                t = index * 2
            if index * 2 + 1 <= size and self.heap[t][1] < self.heap[index * 2 + 1][1]:
                t = index * 2 + 1
            if t == index:
                break
            self._swap(index, t)
            index = t
        return index

class Solution:
    # @param buildings: A list of lists of integers
    # @return: A list of lists of integers
    def buildingOutline(self, buildings):
        if len(buildings) == 0:
            return []

        begins = [(b[0], b[2], index) for index, b in enumerate(buildings)]
        ends = [(b[1], b[2], index) for index, b in enumerate(buildings)]
        heights = sorted(begins + ends, key=lambda x: x[0])

        hashheap = HashHeap()
        y = {}
        for x, height, index in heights:
            if hashheap.hasKey(index):
                hashheap.remove(index)
            else:
                hashheap.add(index, height)
            y[x] = hashheap.max()

        temp = []
        lastX, lastY = None, None
        for x in sorted(y.keys()):
            if lastX is not None and lastY != 0:
                temp.append((lastX, x, lastY))
            lastX, lastY = x, y[x]

        results = []
        lastInterval = temp[0]
        for start, end, height in temp[1:]:
            if start == lastInterval[1] and height == lastInterval[2]:
                lastInterval = lastInterval[0], end, height
            else:
                results.append(lastInterval)
                lastInterval = (start, end, height)
        results.append(lastInterval)
        return results
```