Karat

1. Longest Word in Dictionary (LC 720) (easy) (trie, dfs)

Given a list of strings words representing an English Dictionary, find the longest word in words that can be built one character at a time by other words in words. If there is more than one possible answer, return the longest word with the smallest lexicographical order. If there is no answer, return the empty string.

```
Example:
Input:
words = ["w","wo","wor","worl", "world"]
Output: "world"
Explanation:
The word "world" can be built one character at a time by "w", "wo", "wor", and
"worl".
Example 2:
Input:
words = ["a", "banana", "app", "appl", "ap", "apply", "apple"]
Output: "apple"
Explanation:
Both "apply" and "apple" can be built from other words in the dictionary. However,
"apple" is lexicographically smaller than "apply".
```

Idea:

- Build a trie to store all the words.
- Use DFS to traverse the tree to find the longest word.
- Time: `O(N), N~num of characters`, Space: `O(N)`.

```python
class Solution(object):
    def __init__(self):
        # Initialize a global variable to record the longest word.
        self.lst_word = ""

    def longestWord(self, words):
        '''Builds a trie to record all the words.'''
        trie = {"#": "#"}
        for w in words:
            t = trie
            for c in w:
                if c not in t:
                    t[c] = {}
                t = t[c]
            t["#"] = "#"
        self.dfs(trie, "")
        return self.lst_word

    def dfs(self, trie, word):
        '''Uses DFS to search the trie for the longest word.'''
        if trie == "#":
            return
        # Only check the longest word when it reaches the end of a word.
        if "#" in trie:
            # Update the longest word when the new word is longer or same
            # length but has smaller lexicographical order.
            if len(self.lst_word) < len(word) or \
            (len(self.lst_word) == len(word) and self.lst_word > word):
                self.lst_word = word
            # Search in a deeper level.
            for c in trie:
                self.dfs(trie[c], word+c)

sol = Solution()
words = ["a", "banana", "app", "appl", "ap", "apply", "apple"]
sol.longestWord(words)
```

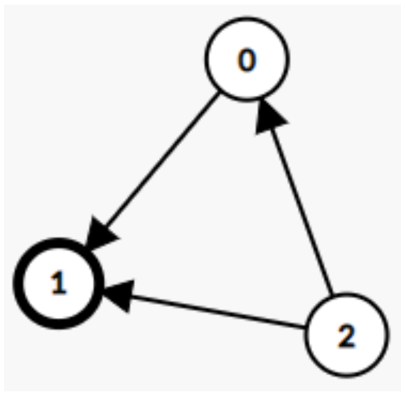
2. Find the Celebrity (LC 277)

Suppose you are at a party with `n` people (labeled from `0` to `n - 1`), and among them, there may exist one celebrity. The definition of a celebrity is that all the other `n - 1` people know him/her, but he/she does not know any of them.

Now you want to find out who the celebrity is or verify that there is not one. The only thing you are allowed to do is to ask questions like: "Hi, A. Do you know B?" to get information about whether A knows B. You need to find out the celebrity (or verify there is not one) by asking as few questions as possible (in the asymptotic sense).

You are given a helper function `bool knows(a, b)` which tells you whether A knows B. Implement a function `int findCelebrity(n)`. There will be exactly one celebrity if he/she is in the party. Return the celebrity's label if there is a celebrity in the party. If there is no celebrity, return `-1`.

Example:



```
Input: graph = [[1,1,0],[0,1,0],[1,1,1]]
Output: 1
Explanation: There are three persons labeled with 0, 1 and 2. graph[i][j] = 1
means person i knows person j, otherwise graph[i][j] = 0 means person i does not
know person j. The celebrity is the person labeled as 1 because both 0 and 2 know
him but 1 does not know anybody.
```

Idea:

- Based on the assupmtion there is at most one celebrity, we can narrow down the searching space to one candidate in `O(N)`.
- Iterate over the people to verify if the candidate is a real celebrity.
- Time: `O(N)`, Space: `O(1)`.

```
class Solution(object):
    def findCelebrity(self, n):
        # Narrow down the search space to one candidate.
        candidate = 0
        for i in range(1, n):
            if knows(candidate, i):
                candidate = i
        # Iterate over the people to verify the candidate.
        for i in range(n):
            if i == candidate: continue
            if not knows(i, candidate) or knows(candidate, i):
                return -1
        return candidate
```

3. Longest Common Substring (dp)

Given two strings ‘X’ and ‘Y’, find the length of the longest common substring.

Examples :

> Input : X = “GeeksforGeeks”, y = “GeeksQuiz”
> Output : 5 or Geeks
> The longest common substring is “Geeks” and is of length 5.
>
> Input : X = “abcdxyz”, y = “xyzabcd”
> Output : 4 or abcd
> The longest common substring is “abcd” and is of length 4.
>
> Input : X = “zxabcdezy”, y = “yzabcdezx”
> Output : 6 or abcdez
> The longest common substring is “abcdez” and is of length 6.

Idea:

- Use dynamic programming to compare each pair of characters from the beginning.
- Update the length of substring according to the previous length.
- Time: `O(|X| * |Y|)`, Space: `O(|X| * |Y|)`.

```
class Solution(object):
    def longestSubstring(self, X, Y):
        maxLength = 0
        lstString = ""
        # Initialize a matrix of size (|X|+1)*(|Y|+1) to store
        # the length of current substring. Expand the sizes of
        # row and column by 1 to smooth the first column
        # computations.
        counter = [[0] * (len(Y) + 1)] * (len(X) + 1)
        for i in range(len(Y)):
            for j in range(len(X)):
                if X[j] == Y[i]:
                    newLength = counter[j][i] + 1
                    if maxLength <= newLength:
                        maxLength = newLength
                        lstString = X[j-newLength+1:j+1]
                    counter[j+1][i+1] = newLength
        return lstString
```

We can optimize the space complexity to `O(min(|X|,|Y|))` by only initialize the count as a vector of size `min(|X|,|Y|)`.

```
class Solution(object):
    def longestHistory(self, X, Y):
        maxLength = 0
        lstString = ""
        # Set the size of counter as the length of the shorter
        # string to save space.
        if len(X) > len(Y):
            X, Y = Y, X
        counter = [0] * (len(X) + 1)
        for i in range(len(Y)):
            # Need a new counter to avoid a direct modification
            # on counter.
            new_counter = [0]
            for j in range(len(X)):
                if X[j] == Y[i]:
                    newLength = counter[j] + 1
                    if maxLength <= newLength:
                        maxLength = newLength
                        lstString = X[j-newLength+1:j+1]
                    new_counter += [newLength]
                else:
                    new_counter += [0]
            counter = new_counter
        return lstString
```

4. Parent and Ancestor (hash, queue, BFS)

```
  1   2  3
 / \ /    \
 4  5      6
            \
             7
```

输入是 `inputlist -> list(list))`, `input[0]`是`input[1]`的parent，比如 `[[1,4], [1,5], [2,5], [3,6], [6,7]]` 会形成上面的图

第一问是只有0个parent和只有1个parent的节点（Time:`O(N)`, Space:`O(N)`)

```
def findParent(inputlist):
    parents = {}
    res = []
    for p, c in inputlist:
        parents[c] = parents.get(c, 0) + 1
        parents[p] = parents.get(p, 0)
    for node, count in parents.items():
        if count <= 1:
            res += [node]
    return res

inputlist = [[1,4], [1,5], [2,5], [3,6], [6,7]]
print(findParent(inputlist))  # [1,2,3,4,6,7]
```

第二问是 两个指定的点有没有公共祖先 (Time: `O(N)`, Space: `O(N)`)

```
def commonAncestor(inputlist, a, b):
    # Build a mapping from child to parent.
    children = {}
    for p, c in inputlist:
        children[c] = children.get(c, []) + [p]

    def ancestors(node):
        # BFS the mapping to retrieve all the parents.
        queue = [node]
        res = set()
        while queue:
            c = queue.pop(0)
            if c not in children: continue
            for p in children[c]:
                # Do not check p if it is already visited.
                if p in res: continue
                queue += [p]
                res.add(p)
        return res
    # Use set intersect to get common ancestors.
    return True if ancestors(a) & ancestors(b) else False

inputlist = [[1,4], [1,5], [2,5], [3,6], [6,7]]
print(commonAncestor(inputlist, 6, 7))  # True
```

第三问是 就一个点的最远祖先，如果有好几个就只需要输出一个就好，举个栗子，这里5的最远祖先可以是1或者2，输出任意一个就可以

(Time: `O(N)`, Space: `O(N)`)

```
def furthestAncestor(inputlist, node):
    children = {}
    for p, c in inputlist:
        children[c] = children.get(c, []) + [p]
    distance = 0
    res = None
    # For each ancestor, record the distance from the original node.
    queue = [(node, 0)]
    visited = set()
    while queue:
        c, d = queue.pop(0)
        if d > distance:
            distance = d
            res = c
        if c not in children: continue
        for p in children[c]:
            if p in visited: continue
            queue += [(p, d+1)]
            visited.add(p)
    return res

inputlist = [[1,4], [1,5], [2,5], [3,6], [6,7]]
print(furthestAncestor(inputlist, 5))
```

5. Domain, History and Ads Conversion

Q1. Subdomain Visit Count (Leetcode 811) (Easy)

A website domain like "discuss.leetcode.com" consists of various subdomains. At the top level, we have "com", at the next level, we have "leetcode.com", and at the lowest level, "discuss.leetcode.com". When we visit a domain like "discuss.leetcode.com", we will also visit the parent domains "leetcode.com" and "com" implicitly.

Now, call a "count-paired domain" to be a count (representing the number of visits this domain received), followed by a space, followed by the address. An example of a count-paired domain might be "9001 discuss.leetcode.com".

We are given a list `cpdomains` of count-paired domains. We would like a list of count-paired domains, (in the same format as the input, and in any order), that explicitly counts the number of visits to each subdomain.

Q2. 给每个user访问历史记录，找出两个user之间longest continuous common history

```
输入入： [
["3234.html", "xys.html", "7hsaa.html"], // user1
["3234.html", ''sdhsfjdsh.html", "xys.html", "7hsaa.html"] // user2
], user1 and user2 （指定两个user求intersect）
输出：["xys.html", "7hsaa.html"]

user0 = [ "/nine.html", "/four.html", "/six.html", "/seven.html", "/one.html" ]
user2 = [ "/nine.html", "/two.html", "/three.html", "/four.html", "/six.html",
"/seven.html" ]
user1 = [ "/one.html", "/two.html", "/three.html", "/four.html", "/six.html" ]
user3 = [ "/three.html", "/eight.html" ].
```

Q3. 3 inputs: a list of user ids + IPs, a list of user ids who have made purchases, a list of advertisement clicks with user IPs.
Each user id has at most 1 IP.
Output: for each ad, output the number of clicks and the number of purchases.

解法：过一遍ids + IPs 的list 建一个hashtable （ip->id), 过一遍ads计算clicks和purchases.

Idea:

- Two users: same as question 3: find the longest substring.
    - change `lstString` to `lstHistory`.
- More than two users: merge every two users and find the longest common history -> Time: `O(log N)`.

```
import collections
class Domain(object):

    def domain_count(self, domains):

        res = {}
        for record in domains:
            vec = record.split(" ")
            count = int(vec[0])
            parts = vec[1].split(".")

            for i in range(len(parts)):
                domain = ".".join(parts[i:])
                res[domain] = res.get(domain, 0) + count

        return res


    def LCS(self, A, B):
        maxLength = 0
        lstHisory = []
        status = [0] * (len(A) + 1)

        for i in range(len(B)):
            newStatus = [0]
            for j in range(len(A)):
                if A[j] == B[i]:
                    newLength = status[j] + 1
                    if newLength > maxLength:
                        maxLength = newLength
                        lstHisory = A[j-newLength+1:j+1]
                    newStatus += [newLength]
                else:
                    newStatus += [0]
            status = newStatus
        return lstHisory



    def ad_conversion(self, userIDs, adClicks, userIPs):

        userIDset = set(userIDs)
        adText = collections.defaultdict(list)
        for click in adClicks:
            parsed = click.split(",")
            adText[parsed[2]] += [parsed[0]]

        ipUser = {}
        for ip in userIPs:
            parsed = ip.split(",")
            ipUser[parsed[1]] = parsed[0]

        res = []
        for text, clicks in adText.items():
            buyer = 0
            for ip in clicks:
                if ip in ipUser and ipUser[ip] in userIDset:
                    buyer += 1
            res += ["{} of {} {}".format(buyer, len(clicks), text)]

        return res

sol = Domain()
```

```
print(sol.domain_count(["9001 discuss.leetcode.com"]))
user0 = [ "/nine.html", "/four.html", "/six.html", "/seven.html", "/one.html" ]
user2 = [ "/nine.html", "/two.html", "/three.html", "/four.html", "/six.html",
"/seven.html" ]
print(sol.LCS(user2, user0))

completedId = ["3123122444", "234111110", "8321125440", "99911063"]
adClicks = ["122.121.0.1,2016-11-03 11:41:19,Buy wool coats for your pets",
            "96.3.199.11,2016-10-15 20:18:31,2017 Pet Mittens",
            "122.121.0.250,2016-11-01 06:13:13,The Best Hollywood Coats",
            "82.1.106.8,2016-11-12 23:05:14,Buy wool coats for your pets",
            "92.130.6.144,2017-01-01 03:18:55,Buy wool coats for your pets",
            "92.130.6.145,2017-01-01 03:18:55,2017 Pet Mittens"]
allUser = ["2339985511,122.121.0.155", "234111110,122.121.0.1",
            "3123122444,92.130.6.145",
"39471289472,2001:0db8:ac10:fe01:0000:0000:0000:0000",
            "8321125440,82.1.106.8", "99911063,92.130.6.144"]
print(sol.ad_conversion(completedId, adClicks, allUser))
```

6. Badge and Company

We are working on a security system for a badged-access room in our company's building.

1. Given an ordered list of employees who used their badge to enter or exit the room, write a function that returns two collections:
   a. All employees who didn't use their badge while exiting the room – they recorded an enter without a matching exit.

   b. All employees who didn't use their badge while entering the room – they recorded an exit without a matching enter.

```
exit = -1, enter = 1;
badge_records = [
  ["Martha",   "exit"],
  ["Paul",     "enter"],
  ["Martha",   "enter"],
  ["Martha",   "exit"],
  ["Jennifer", "enter"],
  ["Paul",     "enter"],
  ["Curtis",   "enter"],
  ["Paul",     "exit"],
  ["Martha",   "enter"],
  ["Martha",   "exit"],
  ["Jennifer", "exit"]
]
```

```
find_mismatched_entries(badge_records)
Expected output: ["Paul", "Curtis"], ["Martha"]
```

Idea:

- Hash map to store everyone's status.
- When **exit**, lookup the previous status. Pop it if it is **enter**.

```python
import collections
def find_mismatched_entries(records):
    status = collections.defaultdict(list)
    for r in records:
        if status[r[0]] == [] or r[1] == "enter":
            status[r[0]] += [r[1]]
        elif r[1] == "exit" and status[r[0]][-1] == "enter":
            status[r[0]].pop()
    exitNoBadge, enterNoBadge = [], []
    for e in status:
        if "enter" in status[e]:
            exitNoBadge += [e]
        if "exit" in status[e]:
            enterNoBadge += [e]
    return exitNoBadge, enterNoBadge
```

2. We want to find employees who badged into our secured room unusually often. We have an unordered list of names and access times over a single day. Access times are given as three or four-digit numbers using 24-hour time, such as "800" or "2250".
   Write a function that finds anyone who badged into the room 3 or more times in a 1-hour period, and returns each time that they badged in during that period. (If there are multiple 1-hour periods where this was true, just return the first one.)

```
badge_records = [
  ["Paul", 1315],
  ["Jennifer", 1910],
  ["John", 830],
  ["Paul", 1355],
  ["John", 835],
  ["Paul", 1405],
  ["Paul", 1630],
  ["John", 855],
  ["John", 915],
  ["John", 930],
  ["Jennifer", 1335],
  ["Jennifer", 730],
  ["John", 1630],
]
Expected output:
{"John": [830, 835, 855, 915, 930]
 "Paul": [1315, 1355, 1405]}
```

Idea:

- Iteratively check the difference beween `i+2` th time and `i` th time. If it's within 60 min, check `i+3` th time ...
- Time: `O(N)`, Space: `O(N)`.

```python
def find_unusual_entries(records):
    # Build a mapping from each employee to their badge usages.
    entries = collections.defaultdict(list)
    for r in records:
        entries[r[0]] += [r[1]]
    unusual = {}
    for e, t in entries.items():
        if len(t) < 3: continue
        for i in range(len(t)-2):
            j = i + 2
            # Stop until the period is more than 1 hour.
            while 0 <= t[j] - t[i] <= 100 and j < len(t):
                j += 1
                unusual[e] = t[i:j]
            # Return the first 1-hour period.
            if e in unusual:
                break
    return unusual
```

7. Matrix and Rectangle

给一个矩阵，矩阵里的每个元素是1，但是其中分布着一些长方形区域， 这些长方形区域中的元素为0. 要求输出每个长方形的位置（用长方形的左上角元素坐标和右下角元素坐标表示）。

```
EXAMPLE:
input:
[
[1,1,1,1,1,1],
[0,0,1,0,1,1],
[0,0,1,0,1,0],
[1,1,1,0,1,0],
[1,0,0,1,1,1]
]
output:
```

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

Followup1 如果 Matrix 中有多个由0组成的长方体，请返回多套值（前提每两个长方体之间是不会连接的).
就是两层for循环遍历一一遍2D matrix，然后把已经搜到的长长方方形记录下来.

```
def find_rectangles(grid):
    def locate_rectangle(i, j):
        # Start from (i, j) and find the right down corner.
        initial_i, initial_j = i, j
        while i < len(grid) and grid[i][j] == 0:
            i += 1
        while j < len(grid[0]) and grid[initial_i][j] == 0:
            j += 1
        # Mark every position in the rectangle to avoid repetitive search.
        for x in range(initial_i, i):
            for y in range(initial_j, j):
                grid[x][y] = -1
        # The right down corner is (i-1, j-1).
        return [initial_i, initial_j, i-1, j-1]
    res = []
    for i in range(len(grid)):
        for j in range(len(grid[0])):
            if grid[i][j] == 0:
                res.append(locate_rectangle(i, j))
    return res
```

Followup2 connected components.
 第三问 基本上就是leetcode connected components,只不过是返回一个list of list，每个list是一个component的所有点坐标
 那个图是1,01,0组成的矩阵，0组成的就是各种图形。跟前面关系的确不大
 如果矩阵里有多个不规则的形状，返回这些形状。这里需要自己思考并定义何谓“返回这些形状”.

Leetcode 200. Number of Islands (Medium, DFS)

Given an `m x n` 2d `grid` map of `'1'`s (land) and `'0'`s (water), return *the number of islands*.

An **island** is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Example:

```
Input: grid = [
  ["1","1","0","0","0"],
  ["1","1","0","0","0"],
  ["0","0","1","0","0"],
  ["0","0","0","1","1"]
]
Output: 3
```

```
class Solution(object):
    def numIslands(self, grid):
        def dfs(i, j):
            if i < 0 or i >= len(grid) or j < 0 or j >= len(grid[0]):
                return
            if grid[i][j] == "1":
                grid[i][j] = "0"
                # self.area.add((i, j))
                dfs(i-1, j); dfs(i+1, j); dfs(i, j-1); dfs(i, j+1)
            return
        res = 0
        # areas = []
        for i in range(len(grid)):
            for j in range(len(grid[0])):
                if grid[i][j] == "1":
                    res += 1
                    # self.area = set()
                    dfs(i, j)
                    # areas += [self.area]
        # print(areas)
        return res
```

8. Calculator

Q1: 给输入为string，例如"2+3-999"，之包含+-操作，返回计算结果。

Q2: 加上parenthesis， 例如"2+((8+2)+(3-999))"，返回计算结果。

Q3: 不光有数字和operator，还有一些变量，这些变量有些可以表示为一个数值，需要从给定的map里去get这个变量的value。

```
class Solution(object):
    def calculator_1(self, s):

        if not s:
            return 0

        sign = 1
        res = 0
        digits = ""
        for ch in s:
            if ch.isdigit():
                digits += ch
            else:
                if digits:
                    res += sign * int(digits)
                    digits = ""
                if ch == "+": sign = 1
                elif ch == "-": sign = -1
        if digits:
            res += sign * int(digits)
        return res


    def calculator_2(self, s):

        if not s:
            return 0

        sign = 1
        res = 0
        stack = []
        digits = ""
        for ch in s:
            if ch.isdigit():
                digits += ch
            else:
                if digits:
                    res += sign * int(digits)
                    digits = ""
                if ch == "+": sign = 1
                elif ch == "-": sign = -1
                elif ch == "(":
                    stack += [res, sign]
                    res, sign = 0, 1
                elif ch == ")":
                    res = res * stack.pop() + stack.pop()
        if digits:
            res += sign * int(digits)
        return res


    def calculator_3(self, s, d):
        if not s:
            return 0

        sign = 1
        res = 0
        stack = []
        alphas = []
        digits = ""
        for ch in s:
            if ch.isalpha():
                if ch in d:
                    res += sign * d[ch]
                else:
                    if sign == 1:
                        alphas += ["+"+ch]
                    else:
                        alphas += ["-"+ch]
            elif ch.isdigit():
                digits += ch
            else:
                if digits:
                    res += sign * int(digits)
                    digits = ""
                if ch == "+": sign = 1
                elif ch == "-": sign = -1
                elif ch == "(":
                    stack += [res, sign]
                    res, sign = 0, 1
                elif ch == ")":
                    res = res * stack.pop() + stack.pop()
        if digits:
            res += sign * int(digits)
        for al in alphas:
            res = "{}{}".format(str(res), al)
        return res

sol = Solution()
print(sol.calculator_1("7+13-4")) #16
print(sol.calculator_2("-(1+2+3)+5")) #-1
print(sol.calculator_2("-(1-(2+3)+2)+5")) #7
print(sol.calculator_3("-(1-(2+3)+2)+a", {"a":5})) #7
print(sol.calculator_3("-(1-(a+3)+a)+5", {"a":2})) #7
print(sol.calculator_3("c-(1-(a+3)+a)-b", {"a":2})) #7
```

Basic Calculator (Leetcode 227) (Medium) (Stack)

Given a string `s` which represents an expression, *evaluate this expression and return its value*.

The integer division should truncate toward zero.

Examples:

```
Input: s = "3+2*2" Output: 7
Input: s = " 3/2 " Output: 1
Input: s = " 3+5 / 2 " Output: 5
```

Idea:

- Convert infix to postfix with stack because postfix is not ambiguous.
  - Pop all the previous operators that have higher precedence to the postfix.
  - Push current operator to the stack.
  - Be careful of numbers with more than one digits.
- Evaluarte postfix with stack.
  - More straightforward.
  - Push numbers from the postfix to a stack.
  - Pop out two numbers when see a operator in the postfix.
- Time: `O(N)`, Space: `O(N)`.

```
class Solution(object):
    def calculate(self, s):
        # Save all possible operators and their precedences.
        self.operator = {"+":0, "-":0, "*":1, "/":1}
        postfix = []
        stack = []
        # Need a buffer to record numbers with more than one digit.
        digits = ""
        for ch in s:
            if ch == " ":
                continue
            elif "0" <= ch <= "9":
                digits += ch
            else:
                if digits:
                    postfix += [int(digits)]
                    digits = ""
                while stack and self.operator[ch] <= self.operator[stack[-1]]:
                    # Pop every operate with higher or equal precedence than
current one.
                    postfix += [stack.pop()]
                stack += [ch]
        # Add the remaining number.
        if digits != "":
            postfix += [int(digits)]
        # Add the remaining operators.
        while stack:
            postfix += [stack.pop()]
        return self.evaluate(postfix)

    def evaluate(self, postfix):
        def compute(op1, op2, ch):
            '''Defines basic operations.'''
            if ch == "+":
                return op1 + op2
            elif ch == "-":
                return op1 - op2
            elif ch == "*":
                return op1 * op2
            else:
                return op1 / op2

        stack = []
        for ch in postfix:
            # Push numbers to the stack.
            if ch not in self.operator:
                stack += [ch]
            # Operations are solved with two operands.
            else:
                op2 = stack.pop()
                op1 = stack.pop()
                stack += [compute(op1, op2, ch)]
        return stack[0]
```

Basic Calculator III with Parenthesis and Negation. (Leetcode 224, 772) (Hard) (Stack)

The expression string contains only non-negative integers, `+`, `-`, `*`, `/` operators , open `(` and closing parentheses `)` and empty spaces ``. The integer division should truncate toward zero.

Idea:

- Deal with left and right patenthesis separately.
    - Simply push left parenthesis to the stack.
    - Pop out everything before the left parenthesis when seeing a right parenthesis.
- Deal with negation carefully in the both parsing and evaluating.

- Time: `O(N)`, Space: `O(N)`.

Examples:

```
Input: s = "1 + 1" Output: 2
Input: s = "(1+(4+5+2)-3)+(6+8)" Output: 23
Input: s = "-1 + (-(1+2)*3) - 5" Output: -15
```

```
class Solution(object):
    def calculate(self, s):
        # Save all possible operators and their precedences.
        self.operator = {"+":0, "-":0, "*":1, "/":1, "(":-1, ")":-1, "~": 2}
        postfix = []
        stack = []
        # Need a buffer to record numbers with more than one digit.
        digits = ""
        # Remove all the empty sapces.
        s = list(filter(lambda x: x != " ", s))

        for i, ch in enumerate(s):
            if "0" <= ch <= "9":
                digits += ch
            else:
                if digits:
                    postfix += [int(digits)]
                    digits = ""
                # A special case of "-". E.g. 3-(-(2+3)) is 323+~- instead of
323+--.
                if ch == "-" and (i == 0 or s[i-1] == "("):
                    stack += ["~"]
                elif ch == "(":
                    stack += [ch]
                elif ch == ")":
                    # Pop every operator in the stack to postfix until "(".
                    while stack and stack[-1] != "(":
                        postfix += [stack.pop()]
                    stack.pop()
                else:
                    while stack and self.operator[ch] <= self.operator[stack[-1]]:
                        # Pop every operate with higher or equal precedence than
current one.
                        postfix += [stack.pop()]
                    stack += [ch]
        # Add the remaining number.
        if digits != "":
            postfix += [int(digits)]
        # Add the remaining operators.
        while stack:
            postfix += [stack.pop()]
        return self.evaluate(postfix)

    def evaluate(self, postfix):
        def compute(op1, op2, ch):
            '''Defines basic operations.'''
            if ch == "+":
                return op1 + op2
            elif ch == "-":
                return op1 - op2
            elif ch == "*":
                return op1 * op2
            else:
                return op1 / op2

        stack = []
        for ch in postfix:
            # Push numbers to the stack.
            if ch not in self.operator:
                stack += [ch]
            # Special case of negative.
            elif ch == "~":
                stack[-1] = -stack[-1]
            # Other operations are solved with two operands.
            else:
                op2 = stack.pop()
                op1 = stack.pop()
                stack += [compute(op1, op2, ch)]
        return stack[0]
```

Basic Calculator IV with Variables (Leetcode 770) (Hard) (Stack)

Given an `expression` such as `expression = "e + 8 - a + 5"` and an evaluation map such as `{"e": 1}` (given in terms of `evalvars = ["e"]` and `evalints = [1]`), return a list of tokens representing the simplified expression, such as `["-1*a","14"]`

Examples:

```
Input: expression = "e + 8 - a + 5", evalvars = ["e"], evalints = [1]
Output: ["-1*a","14"]

Input: expression = "e - 8 + temperature - pressure",
evalvars = ["e", "temperature"], evalints = [1, 12]
Output: ["-1*pressure","5"]

Input: expression = "(e + 8) * (e - 8)", evalvars = [], evalints = []
Output: ["1*e*e","-64"]

Input: expression = "7 - 7", evalvars = [], evalints = []
Output: []

Input: expression = "a * b * c + b * a * c * 4", evalvars = [], evalints = []
Output: ["5*a*b*c"]

Input: expression = "((a - b) * (b - c) + (c - a)) * ((a - b) + (b - c) * (c -
a))",
evalvars = [], evalints = []
Output:
["-1*a*a*b*b","2*a*a*b*c","-1*a*a*c*c","1*a*b*b*b","-1*a*b*b*c","-1*a*b*c*c","1*a*
c*c*c","-1*b*b*b*c","2*b*b*c*c","-1*b*c*c*c","2*a*a*b","-2*a*a*c","-2*a*b*b","2*a*
c*c","1*b*b*b","-1*b*b*c","1*b*c*c","-1*c*c*c","-1*a*a","1*a*b","1*a*c","-1*b*c"]
```

Idea:

- Create a buff for storing the variable with more than one alphabets.
- Check the value from the map
  - If it's in the map, replace it with a number
  - If no, regard it as a number to parse to postfix.
- Good Luck this question is asked.

10. Meeting Room

第一题：类似meeting rooms，输入是一个int[][] meetings, int start, int end, 每个数都是时间，13：00 =》 1300， 9：30 =》 18930， 看新的meeting 能不能安排到meetings
ex: {[1300, 1500], [930, 1200],[830, 845]}, 新的meeting[820, 830], return true; [1450, 1500] return false;

第二题：类似merge interval，唯一的区别是输出，输出空闲的时间段，merge完后，再把两两个之间的空的输出就好，注意要加上0 - 第一个的start time.

```
from heapq import heappush, heappop, heapify
class Solution(object):

    def canSchedule(self, meetings, start, end):

        for s, e in meetings:
            if start < e and end > s:
                return False
        return True

    def spareTime(self, meetings):

        merged = []
        meetings.sort(key=lambda x: x[0])
        for i in range(len(meetings)-1):
            [s1, e1], [s2, e2] = meetings[i], meetings[i+1]
            if s2 < e1:
                meetings[i+1] = [s1, max(e1, e2)]
            else:
                merged += [[s1, e1]]
        merged += [meetings[-1]]

        res = []
        if merged[0][0] > 0:
            res += [[0, merged[0][0]]]
        for i in range(len(merged)-1):
            [_, e1], [s2, _] = merged[i], merged[i+1]
            res += [[e1, s2]]
        if merged[-1][-1] < 2400:
            res += [[merged[-1][-1], 2400]]

        return res


    def minimum_meeting_rooms(self, meetings):

        if not meetings:
            return 0

        meetings.sort(key=lambda x:x[0])

        meeting_heap = [meetings[0][1]]
        heapify(meeting_heap)

        for s, e in meetings[1:]:
            if s >= meeting_heap[0]:
                heappop(meeting_heap)
            heappush(meeting_heap, e)

        return len(meeting_heap)


sol = Solution()
```

```
meetings = [[1300, 1500], [930, 1200],[830, 845]]
print(sol.canSchedule(meetings, 820, 830))
print(sol.canSchedule(meetings, 1450, 1500))

print(sol.spareTime(meetings))

meetings = [[1300, 1500], [930, 1200],[830, 845], [1200, 1400], [1100, 1400]]
print(sol.minimum_meeting_rooms(meetings))

Q3:
    // some rough idea:
    // Mapping: MeetingRoom - Capcity, using for Heap1
    // Mapping: start time - meeting
    // Mapping: start time - numPeople
    // Mapping: start time - endtime
    // Mapping: UsedMeetingRoom - end time, using for Heap2
    // Heap1: all the meeting room, sort base on the capacity
    // Heap2: all the Used meeting room, sort base on the endTime
    // sort the start time
    // sort the end time
    // two pointers: start, end;
    //
    // loop the start: 0 - N:
    // if (start >= end):
    // pop the room in heap2 back
    // end++;
    //
    // loop the Heap1 to find one whose capacity is enough
    // if no such MeetingRoom: return impossible
    // else: pop the room, add to heap2, mapping put(room, end time of this start
    // time), add to result
```

Meeting Rooms (Leetcode 252) (Easy) (Sort)

Given an array of meeting time `intervals` where `intervals[i] = [starti, endi]`, determine if a person could attend all meetings.

```
class Solution(object):
    def canAttendMeetings(self, intervals):

        intervals = sorted(intervals, key=lambda x: x[0])
        for i in range(len(intervals)-1):
            if intervals[i][1] > intervals[i+1][0]:
                return False
        return True
```

Meeting Rooms II (Leetcode 253) (Medium) (Sort, Heap)

Given an array of meeting time intervals `intervals` where `intervals[i] = [starti, endi]`, return *the minimum number of conference rooms required*.

Example:

```
Input: intervals = [[0,30],[5,10],[15,20]]
Output: 2
```

Idea:

- Initialize a new `min-heap` and add the first meeting's ending time to the heap. We simply need to keep track of the ending times as that tells us when a meeting room will get free.
- The final size of the heap will tell us the number of rooms allocated.
- Time: `O(NlogN)`, Space: `O(K)`, `K~min rooms`. Worst case, `K=N`.

```
class Solution(object):
    def minMeetingRooms(self, intervals):
        if intervals == []:
            return 0
        empty_room = []
        # Sort the intervals to guarantee the next meeting
        # starts after the current one.
        intervals.sort(key=lambda x: x[0])
        # Use heap to keep track of all the necessary rooms.
        heapq.heappush(empty_room, intervals[0][1])
        for (s, e) in intervals[1:]:
            # Free the room that ends a meeting.
            if s >= empty_room[0]:
                heapq.heappop(empty_room)
            # Push end to the heap to arrange a room.
            heapq.heappush(empty_room, e)
        return len(empty_room)
```

Merge Intervals (Leetcode 56) (Medium) (sort)

Given an array of `intervals` where `intervals[i] = [starti, endi]`, merge all overlapping intervals, and return *an array of the non-overlapping intervals that cover all the intervals in the input*.

Example:

```
Input: intervals = [[1,3],[2,6],[8,10],[15,18]]
Output: [[1,6],[8,10],[15,18]]
Explanation: Since intervals [1,3] and [2,6] overlaps, merge them into [1,6].
```

```
class Solution(object):
    def merge(self, intervals):

        if len(intervals) <= 1:
            return intervals
        intervals = sorted(intervals, key=lambda x: x[0])
        result = []
        for i in range(len(intervals)-1):
            if intervals[i][1] < intervals[i+1][0]:
                result.append(intervals[i])
            else:
                intervals[i+1] = [min(intervals[i][0], intervals[i+1][0]),\
                                  max(intervals[i][1], intervals[i+1][1])]
        result.append(intervals[-1])
        return result
```

11. Sparse Vector (Leetcode 1570) (easy) (hash table)

Q1: 設計一個 sparse vector class. Throw an error if the index is larger than the size.

```
sparseVector v = new sparseVector(100); //size constructor; size is 100.
v.set(0, 1.0);
v.set(3, 2.0);
v.set(80,-4.5);
```

Q2: Add these operations to your library: Addition, dot product, and cosine. Formular for each are provided below; we're more interested in you writing the code than whether you've memorized the formula. For each operation, your code should **throw an error if the two input vectors are not equal length.**

```
Sample input/output:.
 //Note: This is pseudocode. Your actual syntax will vary by language.
 v1 = new vector(5)
 v1[0] = 4.0
 v1[1] = 5.0
 v2 = new vector(5)
 v2[1] = 2.0
 v2[3] = 3.0

 v3 = new vector(2)
 print v1.add(v2) //should print [4.0, 7.0, 0.0, 3.0, 0.0]
 print v1.add(v3) //error -- vector lengths don't match

 print v1.dot(v2) //should print 10
 print v1.dot(v3) //error -- vector lengths don’t match

 print v1.cos(v2) //should print 0.433
 print v1.cos(v3) //error -- vector lengths don’t match

 Formular:
 Addition
 a.add(b) = [a[0]+b[0], a[1]+b[1], a[2]+b[2], ...]
 Dot product
 a.dot(b) = a[0]*b[0] + a[1]*b[1] + a[2]*b[2] + ...
 Cosine
 a.cos(b) = a.dot(b) / (norm(a) * norm(b))
 //norm(a) = sqrt(a[0]^2 + a[1]^2 + a[2]^2 + ...).
```

Idea:

- Use hashmap to store non-zero numbers.
- Remember to throw errors.

```python
import collections
import math

class SparseVector:
    def __init__(self, dim):

        self.dim = dim
        self.hash = collections.defaultdict(int)

    def set(self, value, index):

        if index >= self.dim:
            raise ValueError("Out of range!")
        self.hash[index] = value

    # Return the dotProduct of two sparse vectors
    def dotProduct(self, vec):
        if vec.dim != self.dim:
            raise ValueError("The dimensions of two vectors mismatch!")
        res = 0
        # Iterate over the vector with fewer non-zero numbers.
        if len(vec.hash) >= len(self.hash):
            for i, num in self.hash.items():
                res += num * vec.hash[i]
        else:
            for i, num in vec.hash.items():
                res += num * self.hash[i]
        return res

    def addition(self, vec):
        if vec.dim != self.dim:
            raise ValueError("The dimensions of two vectors mismatch!")
        res = [0] * self.dim
        for i, num in self.hash.items():
            res[i] += num
        for i, num in vec.hash.items():
            res[i] += num
        return res

    def cosine(self, vec):
        def norm(v):
            res = 0
            for num in v.values():
                res += num * num
            return math.sqrt(res)

        res = self.dotProduct(vec)
        res /= (norm(vec.hash) * norm(self.hash))
        return res


sv = SparseVector(100)
sv2 = SparseVector(100)
sv.set(5, 10)
sv.set(10, 11)
sv2.set(5, 10)
sv2.set(10, 12)
```

12. Friend Circle

1st Question: 输出所有的employee的friendlist -> 就是用一个map存起来然后打印就好了（这个是无向图，e.g: 1和2是朋友，2的列表里也要有1）
2nd Question: 输出每个department里有多少人的朋友是其他部门的 ->也就是遍历一遍就好了.
3rd Question: 输出是否所有employee都在一个社交圈 -> 我当时想的就是随便找一个点，用DFS遍历一遍，如果所有点都被遍历到就return true，不然就是false.

```
有一个employList
  employee_records = [
      "1,Richard,Engineering",
      "2,Erlich,HR",
      "3,Monica,Business",
      "4,Dinesh,Engineering",
       "6,Carla,Engineering",
       "9,Laurie,Directors"
       ]
一个friendshipList, friend关系是双向的
  friendshipsInput = [
         "1,2",
         "1,3",
         "1,6",
         "2,4"
       ]
```

1. 写一个函数返回每个人的friend的adjacency list.
   比如这个例子里返回

```
  1: 2 3 6
  2: 1 4
  3: 1
  4: 2
  6: 1
  9:
```

2. 按照department分类，统计每一个department人数，和这个department有多少人有其他department的朋友
   比如这个例子里
   engineering: 3(人数) 2(有其他部门朋友的员工数)

   ```
   engineering: 3 2
   hr: 1 1
   business: 1 1
   directors: 1 0
   ```

这里我理解错了，理解成了这个部门有多少在部门外的朋友，结果最后输出总是不和需要的相符，面试官似乎也没有发现我理解错了，最后没改完，面试结束发现我理解错了。

```
import collections
class Solution(object):
    def find_friends(self, edges, records):
        self.friends = {}
        for r in records:
            self.friends[r.split(",")[0]] = []

        for e in edges:
            f1, f2 = e.split(",")
```

```
            self.friends[f1] += [f2]
            self.friends[f2] += [f1]
        return self.friends

    def count_by_department(self, records):
        e2dp = {}
        dp2e = collections.defaultdict(list)
        res = {}
        for r in records:
            eid, _, dp = r.split(",")
            e2dp[eid] = dp
            dp2e[dp] += [eid]
        for dp, employees in dp2e.items():
            e_has_outer = 0
            for e in employees:
                for f in self.friends[e]:
                    if e2dp[f] != dp:
                        e_has_outer += 1
                        break
            res[dp] = (len(employees), e_has_outer)
        return res

    def same_friend_circle(self):
        visited = set()
        def dfs(eid):
            visited.add(eid)
            for friends in self.friends[eid]:
                for fid in friends:
                    if fid in visited:
                        continue
                    dfs(fid)
        dfs(list(self.friends.keys())[0])
        return len(visited) == len(self.friends)

employee_records = [
    "1,Richard,Engineering",
    "2,Erlich,HR",
    "3,Monica,Business",
    "4,Dinesh,Engineering",
    "6,Carla,Engineering",
    "9,Laurie,Directors"
    ]
friendshipsInput = [
      "1,2",
      "1,3",
      "1,6",
        "2,4"
      ]
sol = Solution()
print(sol.find_friends(friendshipsInput, employee_records))
print(sol.count_by_department(employee_records))
print(sol.same_friend_circle())
```

13. Immutable Map

One implementation using Python.

`super()` allows you to build classes that easily extend the functionality of previously built classes without implementing their functionality again.

```
class FrozenDict(dict):
    def __init__(self, *args, **kwargs):
        self._hash = None
        super(FrozenDict, self).__init__(*args, **kwargs)

    def __hash__(self):
        if self._hash is None:
            self._hash = hash(tuple(sorted(self.items())))  # iteritems() on py2
        return self._hash

    def _immutable(self, *args, **kws):
        raise TypeError('cannot change object - object is immutable')

    __setitem__ = _immutable
    __delitem__ = _immutable
    pop = _immutable
    popitem = _immutable
    clear = _immutable
    update = _immutable
    setdefault = _immutable
```

14. Stale Tasks

# Now let's switch over to the back-end of our social network. We have some automated batch jobs that we use to handle expensive tasks that run periodically throughout the day, like updating statistics for the most popular posts. We've been given some input that shows the dependencies between each of these batch jobs.

# For example, in this input, "clean" must be executed before "mapper" can execute.

# Given the last execution time for each step of the workflow, we want to find the set of all steps that are "stale" -- steps that have not executed since the last time one of their precursor steps executed. For example, in this case, "update" is in the output because "mapper" must occur before "update", but "update" has not been executed since the last time "mapper" was executed. If a task is stale, all tasks after it are stale too -- so "statistics" is stale because of "mapper".

```
#               /--> reducer
#     /---> mapper --|
# clean --|          \--> update --\
#     \                    --> statistics
#      \---> metadata --------------/
#               \
#               \--> timestamp
```

```
\# Sample input:
\# precursor_steps = [
\#  ["clean", "mapper"],
\#  ["metadata", "statistics"],
\#  ["mapper", "update"],
\#  ["update", "statistics"],
\#  ["clean", "metadata"],
\#  ["mapper", "reducer"],
\#  ["metadata", "timestamp"],
\# last_execution_times = [
\#  ["clean", "20170302-1129"],
\#  ["mapper", "20170302-1155"],
\#  ["update", "20170302-1150"],
\#  ["statistics", "20170302-1153"],
\#  ["metadata", "20170302-1130"],
\#  ["reducer", "20170302-1540"],

\# Sample output (in any order):
\# find_stale_steps(precursor_steps, last_execution_times) =
\#  ["update", "statistics", "timestamp"]
```

15. Find First and Last Position of Element in Sorted Array (Leetcode 34) (Medium) (Binary Search)

Given an array of integers `nums` sorted in ascending order, find the starting and ending position of a given `target` value.

If `target` is not found in the array, return `[-1, -1]`.

Follow up: Could you write an algorithm with `O(log n)` runtime complexity?

Idea:

- Set up two binary search to locate the first and last indexes separately.
  - In `lbs()`, we keeping moving `r` even when the `mid` equals to the target. In this way, `r` will finally stop at the position where we can find the largest number that is smaller than the target. Meanwhile, we stop moving `r` when `r < l`, so we return `l` as the start of the target.
  - Similarly, we return `r` in `rbs()` as the end of the target.
- Time: `O(logN)`.

```
class Solution(object):
    def searchRange(self, nums, target):
    '''Searches the first and last indexes of target num in the list.'''

        # Set up two binary search to locate the first and last indexes
separately.
        def lbs(nums, target):
            '''Locates the first index.'''
            l, r = 0, len(nums)-1
            while l <= r:
                mid = (l+r)//2
                if nums[mid] < target:
                    l = mid + 1
                # Keep moving r to look for other targets even when the mid equals
to target.
                else:
                    r = mid - 1
            return l

        def rbs(nums, target):
            '''Locates the last index.'''
            l, r = 0, len(nums)-1
            while l <= r:
                mid = (l+r)//2
                if nums[mid] > target:
                    r = mid - 1
                else:
                    l = mid + 1
            return r

        l, r = lbs(nums, target), rbs(nums, target)
        return [l, r] if l <= r else [-1, -1]
```

16. Russian Doll (Leetcode 354, 300) (Hard) (DP, Binary Search)

You have a number of envelopes with widths and heights given as a pair of integers `(w, h)`. One envelope can fit into another if and only if both the width and height of one envelope is greater than the width and height of the other envelope.

What is the maximum number of envelopes can you Russian doll? (put one inside other)

Example:

```
Input: [[5,4],[6,4],[6,7],[2,3]]
Output: 3
Explanation: The maximum number of envelopes you can Russian doll is 3 ([2,3] =>
[5,4] => [6,7]).
```

Idea:

- Sort + Longest Increasing Subsequence
    - After we sort our envelopes, we can simply find the length of the longest increasing subsequence on the second dimension (`h`).
    - We don't just sort increasing in the first dimension - we also sort *decreasing* on the second dimension, so two envelopes that are equal in the first dimension can never be in the same increasing subsequence.
- In the `longest_increasing_subseqence` function, we use dynamic programming.
    - Keep updating/inserting numbers in the `seq` to extend the longest subsequence.
    - `seq` in the `longest_increasing_subseqence` is not the real longest increasing subsequence but has the same length.
- Time: `O(NlogN)`, Space: `O(N)`.

```
from bisect import bisect_left

class Solution(object):
    def maxEnvelopes(self, envelopes):
        """
        :type envelopes: List[List[int]]
        :rtype: int
        """
        if not envelopes: return 0
        envelopes.sort(key=lambda x: (x[0], -x[1]))

        def longest_increasing_subseqence(nums):
            # Initialize a sequence where numbers will be in a increasing order.
            seq = []
            for num in nums:
                # Locate the position to replace or insert.
                index = bisect_left(seq, num)
                if index == len(seq):
                    # Extend the sequence by one more number because a bigger
number
                    # is seen.
                    seq += [num]
                else:
                    # We replace every number in the sequence if possible in order
to
                    # to make the last number updatable and the sequence
extendable.
                    seq[index] = num
            return len(seq)

        return longest_increasing_subseqence([x[1] for x in envelopes])
```

17. Clone Graph (Leetcode 133) (Medium) (DFS, Graph)

Given a reference of a node in a connected undirected graph, return a deep copy (clone) of the graph. Each node in the graph contains a val (int) and a list (List[Node]) of its neighbors.

Idea:

- Traverse the graph with DFS algorithm.
- Create new nodes for deep copy.

```
class Solution:
    def cloneGraph(self, node):
        '''Deep clones a graph by DFSing the graph. '''
        # define a global variable res to store the results.
        # key: original node; value: deep copy node.
        res = {}

        def dfs(node):
            if not node: return node
            # Process nodes not in res.
            if node not in res:
                # Create deep copy for current node.
                res[node] = Node(node.val, [])
                for n in node.neighbors:
                    # Update neighbor list for current node.
                    res[node].neighbors.append(dfs(n))
            return res[node]

        return dfs(node)
```

18. Alien Dictionary (Leetcode 269) (Hard) (Graph, DFS, Hashmap)

There is a new alien language that uses the English alphabet. However, the order among letters are unknown to you.

You are given a list of strings `words` from the dictionary, where `words` are **sorted lexicographically** by the rules of this new language.

Derive the order of letters in this language, and return it. If the given input is invalid, return `""`. If there are multiple valid solutions, return **any of them**.

Example:

```
Input: words = ["wrt","wrf","er","ett","rftt"]
Output: "wertf"
```

Idea:

- Traverse the strings and build mappings from "larger" character to "smaller" character by comparing each adjacent strings.
    - We made a reverse adjacency list instead of a forward one, the output order would be correct.
- DFS the mappings and detect if there is a circle using **graph coloring**.
    - All nodes start as white, and then once they're first visited they become grey, and then once all their outgoing nodes have been fully explored, they become black. We know there is a cycle if we *enter* a node that is currently grey.
- Time: `O(C + U + min(U^2, N)) ~ O(C)`, where `C ~ num of characters`,
    - Building the adjacency list has a time complexity of `O(C)`.
    - DFS the graph takes `O(|V| + |E|)`.
        - `O(|V|) ~ O(U)`, where `U ~ num of unique characters`.
        - `O(|E|) ~ O(min(U^2, N))`, where `N ~ num of strings`.
            - one upper bound of edges is `N-1`.

        - One upper bound is `U(U-1)`.
    - We know `U < C and N < C`, so `min(U^2, N) < C`.
- Space: `O(U + min(U^2, N))`.

```python
class Solution(object):
    def alienOrder(self, words):
        if len(words) == 0:
            return ""
        if len(words) == 1:
            return words[0]

        # Build a reverse adjacency list instead of a forward one.
        hashmap = {ch: [] for word in words for ch in word}
        for i in range(len(words)-1):
            first, second = words[i], words[i+1]
            for j in range(len(first)):
                if j == len(second):
                    return ""
                if first[j] != second[j]:
                    hashmap[second[j]].append(first[j])
                    break

        # DFS the hashmap to derive the correct order.
        # Map visited nodes to their status. Grey (False) means visited in DFS.
        # Finally turned to white (True) if no cycle is detected.
        visited = {}
        res = []
        def dfs(node):
            # To avoid duplicated DFS on white nodes. Grey nodes are revisited
            # and turned to black.
            if node in visited:
                return visited[node]
            visited[node] = False
            for next_node in hashmap[node]:
                if not dfs(next_node):
                    return False
            # After DFS the current node, set its status as white (True).
            visited[node] = True
            res.append(node)
            return True

        for node in hashmap:
            if not dfs(node):
                return ""
        return "".join(res)
```

19. Substring with Concatenation of All Words (Leetcode 30) (Hard) (Two Pointer)

You are given a string `s` and an array of strings `words` of **the same length**. Return all starting indices of substring(s) in `s` that is a concatenation of each word in `words` **exactly once**, **in any order**, and **without any intervening characters**.

You can return the answer in **any order**.

Example:

```
Input: s = "barfoothefoobarman", words = ["foo","bar"]
Output: [0,9]
Explanation: Substrings starting at index 0 and 9 are "barfoo" and "foobar"
respectively.
The output order does not matter, returning [9,0] is fine too.
```

Idea:

- Two Pointer:
  - Slow pointer `l` to track the beginning of the sequence.
  - Fast pointer `r` to continuously find new targets in the word list.
- Whenever a target is found, move `r` len(each word) steps forward and update the hashmap (a counter).
- Whenever a sequence is found, move `l` one step forward and let `r` equal to `l`.
- Set a `isSeq` variable to mark whether some of the targets have been found. We don't have to update the hashmap if `False`.
- Time: `O(N * wlen)`, Space: `O(|words|)`.

```
class Solution(object):
    def findSubstring(self, s, words):
        counter = collections.Counter(words)
        # Deep copy a counter to track the  words satisfied.
        hashmap = dict(counter)
        res = []
        wlen = len(words[0])
        # Initialize a slow and fast pointer. l is used to track the beginning
        # of the sequence. r is used to find new words.
        l, r = 0, 0
        inSeq = False
        while r <= len(s)-wlen:
            word = s[r:r+wlen]
            # Only move l and r one step forward if the current search fails.
            if word not in hashmap:
                r = l = l + 1
                # Reset the counter only if we found some of the targets before.
                if inSeq:
                    hashmap = dict(counter)
                    inSeq = False
            else:
                # Update the hashmap.
                hashmap[word] -= 1
                if hashmap[word] == 0:
                    # Remove the key if count equals to 0.
                    del hashmap[word]
                    # Reset everthing when one result is found.
                    if not hashmap:
                        res += [l]
                        hashmap = dict(counter)
                        r = l = l + 1
                        inseq = False
                        continue
                # Let r jump wlen step to find the next word. Do not move l.
                inSeq = True
                r += wlen
        return res
```

20. Treasure (DFS) (Hard)

Now we also have treasures, denoted by 1. Given a board and start and end positions for the player, write a function to return the shortest simple path from start to end that includes all the treasures, if one exists. A simple path is one that does not revisit any location.

```
board1 = [
    [ 0, -1, 0, 0 ],
    [ 0, 0, -1, 0 ],
    [ 0, 0, 0, -1 ]
]
board3 = [
    [  1,  0,  0, 0, 0 ],
    [  0, -1, -1, 0, 0 ],
    [  0, -1,  0, 1, 0 ],
    [ -1,  0,  0, 0, 0 ],
    [  0,  1, -1, 0, 0 ],
    [  0,  0,  0, 0, 0 ],
]
```

```
treasure(board3, (5, 0), (0, 4)) -> None
treasure(board3, (5, 1), (2, 0)) ->  [(5, 1), (4, 1), (3, 1), (3, 2), (2, 2), (2,
3), (1, 3), (0, 3), (0, 2), (0, 1), (0, 0), (1, 0), (2, 0)]
```

1. 给一个cell=0的position，返回上下左右四个方向都是0的neighbors。
2. 给一个cell=0的end position，判断这个board的其他所有0是否能reach它。
3. board中1是treasures。给一个start和end position。要找到一条shortest path from start to end 并且能经过所有的，不能走重读的格子。

Idea:

- DFS the four directions.
- Use min_path to keep track of the length of shortest path.
    - Float('inf') if not found
- Time: ??

```
class Solution:

    def check_neighbors(self, board, pos, visited):

        res = []

        def valid(i, j):
            return 0 <= i < len(board) and 0 <= j < len(board[0]) and \
            board[i][j] != -1 and (i,j) not in visited

        [i, j] = pos
        if valid(i, j+1): res += [[i, j+1]]
        if valid(i+1, j): res += [[i+1, j]]
        if valid(i, j-1): res += [[i, j-1]]
        if valid(i-1, j): res += [[i-1, j]]

        return res

    def check_all_zeros(self, board, pos):

        visited = set()
        queue = [pos]
        while queue:
            i, j = queue.pop(0)
            visited.add((i, j))
            queue.extend(self.check_neighbors(board, [i, j], visited))
        count = 0
        for i in range(len(board)):
            for j in range(len(board[0])):
                if board[i][j] != -1:
                    count += 1
        return count == len(visited)


    def find_shortest_path(self, board, s, e):
        # Find all the treasures first.
        treasures = set()
        for i in range(len(board)):
            for j in range(len(board[0])):
                if board[i][j] == 1:
                    treasures.add((i,j))

        def valid(i, j, visited):
            return 0 <= i < len(board) and 0 <= j < len(board[0]) and \
            visited[i][j] == 0 and board[i][j] != -1

        def dfs(i, j, visited, trs, min_path, path):
            if [i,j] == e and trs == treasures:
                return min(min_path, path), e
            # Update the visited nodes and treasure status.
            visited[i][j] = 1
            if board[i][j] == 1:
                trs.add((i,j))
            paths = None
            # DFS the four directions.
            if valid(i, j+1, visited):
                min_path, paths = dfs(i, j+1, visited, trs, min_path, path+1)
            if valid(i, j-1, visited):
                min_path, paths = dfs(i, j-1, visited, trs, min_path, path+1)
            if valid(i+1, j, visited):
                min_path, paths = dfs(i+1, j, visited, trs, min_path, path+1)
            if valid(i-1, j, visited):
                min_path, paths = dfs(i-1, j, visited, trs, min_path, path+1)
            # Backtrack: recover visited and treasure status.
            visited[i][j] = 0
            if board[i][j] == 1:
                trs.discard((i,j))
            # Add the current node to the path if end reached and treasures found.

            if paths:
                return min_path, [[i, j]] + paths
            else:
                return min_path, None
        # Initialize the visited matrix.
        visited = [[0 for _ in range(len(board[0]))] for _ in range(len(board))]
        _, res_path = dfs(s[0], s[1], visited, set(), float("inf"), 0)
        return res_path



board1 = [
    [ 0, -1, 0, 0 ],
    [ 0, 0, -1, 0 ],
    [ 0, 0, 0, -1 ]
]
board3 = [
    [  1,  0,  0, 0, 0 ],
    [  0, -1, -1, 0, 0 ],
    [  0, -1,  0, 1, 0 ],
    [ -1,  0,  0, 0, 0 ],
    [  0,  1, -1, 0, 0 ],
    [  0,  0,  0, 0, 0 ],
]
sol = Solution()
print(sol.check_neighbors(board1, [2, 1]))
print(sol.check_neighbors(board3, [3, 3]))

print(sol.check_all_zeros(board1, [0, 0]))
print(sol.check_all_zeros(board1, [1, 3]))
print(sol.check_all_zeros(board3, [5, 0]))
print(sol.check_all_zeros(board3, [1, 3]))

print(sol.find_shortest_path(board3, [5,0], [0,4]))
print(sol.find_shortest_path(board3, [5,1], [2,0]))
```

Use BFS to look for the shortest path, but neglecting the requirements of treasures. I don't think we can use BFS to solve this question.

```
    def find_shortest_path_bfs(self, board, s, e):
        visited = set()
        def isSafe(i, j):
            return 0 <= i < len(board) and 0 <= j < len(board[0]) and \
            (i,j) not in visited and board[i][j] != -1

        queue = [(s[0], s[1], 0)]
        while queue:
            i, j, step = queue.pop(0)
            visited.add((i, j))
            if [i, j] == e:
                return step
            if isSafe(i, j+1): queue += [(i, j+1, step+1)]
            if isSafe(i, j-1): queue += [(i, j-1, step+1)]
            if isSafe(i+1, j): queue += [(i+1, j, step+1)]
            if isSafe(i-1, j): queue += [(i-1, j, step+1)]

        return None
```

21. K maximum sum combinations from two arrays (Hard) (GG) (Heap)

Given two equally sized arrays (A, B) and N (size of both arrays).
A **sum combination** is made by adding one element from array A and another element of array B.
Display the **maximum K valid sum combinations** from all the possible sum combinations.

Examples:

```
Input :  A[] : {3, 2}
         B[] : {1, 4}
         K : 2 [Number of maximum sum
               combinations to be printed]
Output : 7    // (A : 3) + (B : 4)
         6    // (A : 2) + (B : 4)

Input :  A[] : {4, 2, 5, 1}
         B[] : {8, 0, 3, 5}
         K : 3
Output : 13   // (A : 5) + (B : 8)
         12   // (A : 4) + (B :  8)
         10   // (A : 2) + (B : 8)
```

Idea:

- Sort both arrays array A and array B so that we can start from the beginning of the two arrays to push sums to a heap.
- Create a max heap to store the sum combinations along with the indices of elements from both arrays A and B which make up the sum. Heap is ordered by the sum.
- Pop the largest (i, j) and push two candidates (i+1, j) and (i, j+1) until K == 0.
- Time: `O(NlogN)`, assuming `K <= N`, Space: `O(K)`.

```
import heapq
def KMaxCombinations(A, B, K):

    A.sort(reverse=True)
    B.sort(reverse=True)
    max_heap = [(-A[0]-B[0], 0, 0)]
    heapq.heapify(max_heap)
    # Avoid using max_heap = heapq.heapify([(-A[0]-B[0], 0, 0)]) because heapify
returns None.
    res = []
    while max_heap and K > 0:
        val, i, j = heapq.heappop(max_heap)
        res += [-val]
        K -= 1
        if j < len(B):
            heapq.heappush(max_heap, (-A[i]-B[j+1], i, j+1))
        if i < len(A):
            heapq.heappush(max_heap, (-A[i+1]-B[j], i+1, j))

    return res

A = [5,4,7,0,10]
B = [3,6,12,8,9]
K = 10
print(KMaxCombinations(A, B, K))
```

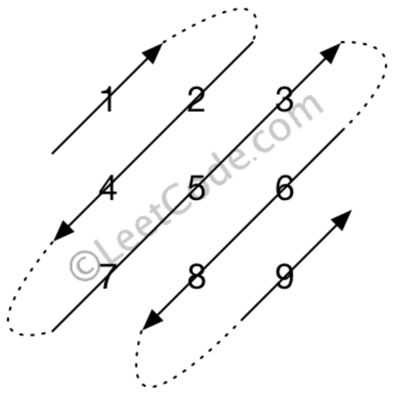
22. Diagonal Traverse (Leetcode 498) (Medium) (Array)

Given a matrix of M x N elements (M rows, N columns), return all elements of the matrix in diagonal order as shown in the below image.

Example:

```
Input:
[
 [ 1, 2, 3 ],
 [ 4, 5, 6 ],
 [ 7, 8, 9 ]
]

Output:  [1,2,4,7,5,3,6,8,9]
```



Idea:

- Traverse the diagonals and reverse it when oven.
- Time: `O(R*C)`, Space: `O(R)`.

```
class Solution(object):
    def findDiagonalOrder(self, matrix):
        if matrix == [] or matrix[0] == []:
            return []
        res = []
        R, C = len(matrix), len(matrix[0])
        # Num of diagonals equals R + C - 1
        for d in range(R + C - 1):
            column = []
            # Find the start position in the matrix for each diagonal.
            i = 0 if d < C else d - C + 1
            j = d if d < C else C - 1
            while i < R and j > -1:
                column.append(matrix[i][j])
                # Increase i and decrease j for next element.
                j -= 1
                i += 1
            # Use d to determine whether to reverse or not.
            if d % 2 == 1:
                res += column
            else:
                res += column[::-1]
        return res
```

23. Read4

Read N Characters Given Read4 (Easy) (Leetcode 157)

Given a file and assume that you can only read the file using a given method `read4`, implement a method to read *n* characters.

Method read4:

The API `read4` reads 4 consecutive characters from the file, then writes those characters into the buffer array `buf4`.

The return value is the number of actual characters read.

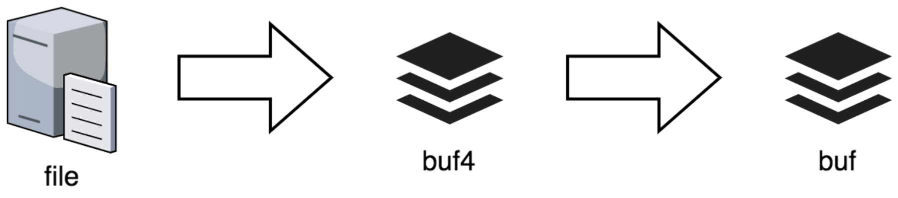
Note that `read4()` has its own file pointer, much like `FILE *fp` in C.

Definition of read4:

```
    Parameter:  char[] buf4
    Returns:    int

Note: buf4[] is destination not source, the results from read4 will be copied to
buf4[]
```

Idea:

- use an internal buffer of 4 characters to solve this problem: File -> Internal Buffer of 4 Characters -> Buffer of N Characters.
- 

- Time: `O(N)`, Space: `O(1)`.

```
"""
The read4 API is already defined for you.

    @param buf4, a list of characters
    @return an integer
    def read4(buf4):

# Below is an example of how the read4 API can be called.
file = File("abcdefghijk") # File is "abcdefghijk", initially file pointer (fp)
points to 'a'
```

```
buf4 = [' '] * 4 # Create buffer with enough space to store characters
read4(buf4) # read4 returns 4. Now buf = ['a','b','c','d'], fp points to 'e'
read4(buf4) # read4 returns 4. Now buf = ['e','f','g','h'], fp points to 'i'
read4(buf4) # read4 returns 3. Now buf = ['i','j','k',...], fp points to end of
file
"""


class Solution(object):
    def read(self, buf, n):
        """
        :type buf: Destination buffer (List[str])
        :type n: Number of characters to read (int)
        :rtype: The number of actual characters read (int)
        """
        # Initialize an index for storing char to the destination buf.
        index = 0
        while n > 0:
            # read file to buf4
            buf4 = [""] * 4
            chars_read = read4(buf4)

            # Return when no more chars in the file.
            if not chars_read:
                return index

            # Write buf4 into buf directly.
            for i in range(min(chars_read, n)):
                buf[index] = buf4[i]
                index += 1
                n -= 1

        return index
```

Read N Characters Given Read4 II - Call multiple times (Leetcode 158) (Hard) (Deque)

Given a file and assume that you can only read the file using a given method `read4`, implement a method `read` to read *n* characters. **Your method `read` may be called multiple times.**

```
File file("abc");
Solution sol;
// Assume buf is allocated and guaranteed to have enough space for storing all
characters from the file.
sol.read(buf, 1); // After calling your read method, buf should contain "a". We
read a total of 1 character from the file, so return 1.
sol.read(buf, 2); // Now buf should contain "bc". We read a total of 2 characters
from the file, so return 2.
sol.read(buf, 1); // We have reached the end of file, no more characters can be
read. So return 0.
```

Idea:

- Create a global `extra_buf` to deal with remaining characters in the previous read.
- `extra_buf` is better to use **deque**, whose `popleft()` takes `O(1)`.
- Time: `O(N)`, Space: `O(1)`.

```python
# The read4 API is already defined for you.
# @param buf4, List[str]
# @return an integer
# def read4(buf4):

class Solution(object):
    def __init__(self):
        self.extra_buf = deque([])

    def read(self, buf, n):
        # Initialize an index for storing char to the destination buf.
        index = 0

        # Process chars in the extra buf from previous read first.
        while self.extra_buf and n:
            buf[index] = self.extra_buf.popleft()
            index += 1
            n -= 1

        # Call read4 again if more chars need to be read.
        while n:
            # Read file to buf4.
            buf4 = [""] * 4
            chars_read = read4(buf4)

            # Return when no more chars in the file.
            if not chars_read:
                return index
            # Save extra chars to extra_buf.
            if chars_read > n:
                self.extra_buf += buf4[n: chars_read]

            # Write buf4 into buf directly.
            for i in range(min(chars_read, n)):
                buf[index] = buf4[i]
                index += 1
                n -= 1

        return index
```

24. Subdomain Visit Count (Leetcode 811) (Easy)

A website domain like "discuss.leetcode.com" consists of various subdomains. At the top level, we have "com", at the next level, we have "leetcode.com", and at the lowest level, "discuss.leetcode.com". When we visit a domain like "discuss.leetcode.com", we will also visit the parent domains "leetcode.com" and "com" implicitly.

Now, call a "count-paired domain" to be a count (representing the number of visits this domain received), followed by a space, followed by the address. An example of a count-paired domain might be "9001 discuss.leetcode.com".

We are given a list `cpdomains` of count-paired domains. We would like a list of count-paired domains, (in the same format as the input, and in any order), that explicitly counts the number of visits to each subdomain.

```
Example 1:
Input:
["9001 discuss.leetcode.com"]
Output:
["9001 discuss.leetcode.com", "9001 leetcode.com", "9001 com"]
Explanation:
We only have one website domain: "discuss.leetcode.com". As discussed above, the
subdomain "leetcode.com" and "com" will also be visited. So they will all be
visited 9001 times.
```

```
def subdomainVisits(self, cpdomains):
    """
    :type cpdomains: List[str]
    :rtype: List[str]
    """
    res = {}
    for record in cpdomains:
        count, domain = record.split(" ")
        count = int(count)
        parts = domain.split(".")
        for i in range(len(parts)):
            subdomain = ".".join(parts[i:])
            res[subdomain] = res.get(subdomain, 0) + count
    return ["{} {}".format(v, k) for k,v in res.items()]
```

- Time: `O(N)`, Space: `O(N)`

25. Word Ladder (Leetcode 127)

26. Is Graph Bipartite? (Leetcode 785) (Medium) (Graph Coloring, DFS)

Given an undirected `graph`, return `true` if and only if it is bipartite.

Recall that a graph is *bipartite* if we can split its set of nodes into two independent subsets A and B, such that every edge in the graph has one node in A and another node in B.

The graph is given in the following form: `graph[i]` is a list of indexes `j` for which the edge between nodes `i` and `j` exists. Each node is an integer between `0` and `graph.length - 1`. There are no self edges or parallel edges: `graph[i]` does not contain `i`, and it doesn't contain any element twice.

Idea:

- DFS from each node and color them differently according to the order.
  - Neighbors are colored different.
- If the new starting node is not visited in the previous DFS, color it as 0 since this node is definitely disconnected with other nodes. Nodes connected can be reached via DFS.
- Time: `O(|V| + |E|)`, Space:`O(|V|)`, where `|V| ~ |graph|`.

```
class Solution(object):
    def isBipartite(self, graph):
        color = {}

        def dfs(pos):
            for p in graph[pos]:
                if p in color:
                    # Neighboring nodes in bipartite graph cannot have the same
color.
                    if color[p] == color[pos]:
                        return False
                    # Skip the node if already visited and have the correct color
-> DFS end condition.
```

```
                else:
                    # Color the new node and DFS its other neighbors.
                    color[p] = 1 - color[pos]
                    if not dfs(p):
                        return False
            return True

        # Start from every node to guarantee the complete coverage.
        for p in range(len(graph)):
            # Skip the node visited.
            if p in color:
                continue
            # Because the edges are undirected, so if the node is not visited, it
is not connected to
            # the previous cluster. 0 can be colored to this new node.
            color[p] = 0
            if not dfs(p):
                return False

        return True
```

27. 3 Sum & 4 Sum

28. Tree Diameter

Diameter of Binary Tree (Leetcode 543) (Easy)

Given a binary tree, you need to compute the length of the diameter of the tree. The diameter of a binary tree is the length of the **longest** path between any two nodes in a tree. This path may or may not pass through the root.

Example:
Given a binary tree

```
          1
         / \
        2   3
       / \
      4   5
```

Return **3**, which is the length of the path [4,2,1,3] or [5,2,1,3].

Idea:

- DFS the tree and diameter equals left depth + right depth.
- Time: `O(N)`, Space: `O(N)`.

```
class Solution(object):
    def diameterOfBinaryTree(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        self.diameter = 0
        def dfs(root):
            if not root:
                return 0

            L = dfs(root.left)
            R = dfs(root.right)
            self.diameter = max(L + R, self.diameter)
            return max(L, R) + 1

        dfs(root)
        return self.diameter
```

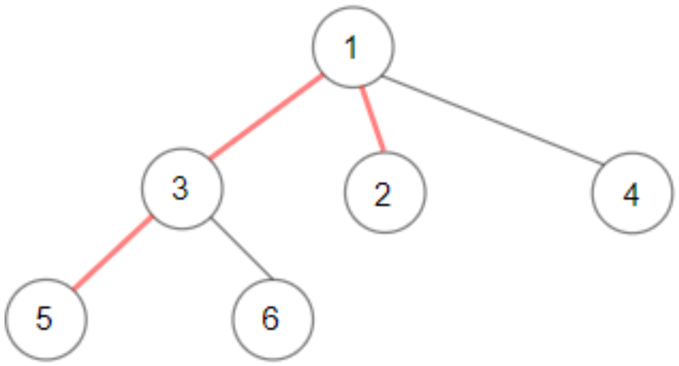
Diameter of N-Ary Tree (Leetcode 1522) (Medium) (DFS)

Given a `root` of an N-ary tree, you need to compute the length of the diameter of the tree.

The diameter of an N-ary tree is the length of the **longest** path between any two nodes in the tree. This path may or may not pass through the root.

(*Nary-Tree input serialization is represented in their level order traversal, each group of children is separated by the null value.)*

Example 1:



```
Input: root = [1,null,3,2,4,null,5,6]
Output: 3
Explanation: Diameter is shown in red color.
```

Idea:

- Change looking for depth of left and right to every child and find the biggest two (initialized as 0).
- Time: `O(N)`, Space: `O(N)`.

```python
class Solution(object):
    def diameter(self, root):
        self.diameter = 0
        def dfs(root):
            if not root: return 0
            d1, d2 = 0, 0
            for child in root.children:
                d = dfs(child)
                if d >= d1:
                    d1, d2 = d, d1
                elif d >= d2:
                    d2 = d

            self.diameter = max(self.diameter, d1 + d2)
            return d1 + 1

        dfs(root)
        return self.diameter
```

29. Binary Tree Longest Consecutive Sequence

I (Leetcode 298)

Given a binary tree, find the length of the longest consecutive sequence path.

The path refers to any sequence of nodes from some starting node to any node in the tree along the parent-child connections. The longest consecutive path need to be from parent to child (cannot be the reverse).

Example 1:

```
Input:

   1
    \
     3
    / \
   2   4
        \
         5

Output: 3

Explanation: Longest consecutive sequence path is 3-4-5, so return 3.
```

Idea:

- Three conditions:
    - Null nodes: return 0
    - Non-null nodes: 与一个child差为1，length += 1
    - Non-null nodes: 与一个child差不为1，length = 1，重新开始计数。
- `max_length = max(max_length, length)`.
- Tine: `O(N)`, Space: `O(N)`.

```python
class Solution(object):
    def longestConsecutive(self, root):
        self.max_length = 0
        def dfs(root):
            if not root:
                return 0
            # Default the sequence is expanding.
            L = dfs(root.left) + 1
            # Same procedure for the left and right branches.
            R = dfs(root.right) + 1
            # Condition not satisfied, reset the length to 1.
            if root.left and root.val + 1 != root.left.val:
                L = 1
            if root.right and root.val + 1 != root.right.val:
                R = 1
            length = max(L, R)
            self.max_length = max(self.max_length, length)
            return length

        dfs(root)
        return self.max_length
```

II (Leetcode)

Given a binary tree, you need to find the length of Longest Consecutive Path in Binary Tree.

Especially, this path can be either increasing or decreasing. For example, [1,2,3,4] and [4,3,2,1] are both considered valid, but the path [1,2,4,3] is not valid. On the other hand, the path can be in the child-Parent-child order, where not necessarily be parent-child order.

Example 1:

```
Input:
        1
       / \
      2   3
Output: 2
Explanation: The longest consecutive path is [1, 2] or [2, 1].
```

Idea:

- Difference between I and II is:
    - `child -> parent -> child` sequence is permitted.
    - decrease and increase are both permitted.
- Return `[increase_length, decrease_length]` for each DFS to represent the length of the longest incrementing branch and decrementing branch below the current node including itself.
- Use `increase_length + decrease_length - 1` as the consecutive path length.
- Time: `O(N)`, Space: `O(N)`.

```
class Solution(object):
    def longestConsecutive(self, root):
        self.max_length = 0

        def dfs(root):
            if not root:
                return [0, 0]

            L = dfs(root.left)
            R = dfs(root.right)

            inc_L = L[0] + 1
            inc_R = R[0] + 1
            if root.left and root.val + 1 != root.left.val:
                inc_L = 1
            if root.right and root.val + 1 != root.right.val:
                inc_R = 1
            inc_length = max(inc_L, inc_R)
            dec_L = L[1] + 1
            dec_R = R[1] + 1
            if root.left and root.val - 1 != root.left.val:
                dec_L = 1
            if root.right and root.val - 1 != root.right.val:
                dec_R = 1
            dec_length = max(dec_L, dec_R)

            self.max_length = max(self.max_length, inc_length + dec_length - 1)
            return [inc_length, dec_length]

        dfs(root)
        return self.max_length
```

30. 24 Game

31. Employee Free Time (Leetcode 759) (Hard)

32. Text Justification (Leetcode 68) (Hard)

33. Course Schedule

Q1:

```
You are a developer for a university. Your current project is to develop a system
for students to find courses they share with friends. The university has a system
for querying courses students are enrolled in, returned as a list of (ID, course)
pairs.
Write a function that takes in a list of (student ID number, course name) pairs
and returns, for every pair of students, a list of all courses they share.
```

Q2:

```
Students may decide to take different "tracks" or sequences of courses in the
Computer Science curriculum. There may be more than one track that includes the
same course, but each student follows a single linear track from a "root" node to
a "leaf" node. In the graph below, their path always moves left to right.

Write a function that takes a list of (source, destination) pairs, and returns the
name of all of the courses that the students could be taking when they are halfway
through their track of courses.
```

```
import collections
class Solution(object):

    def find_pairs(self, records):
        students = set()
        courseStudent = collections.defaultdict(list)
        for record in records:

            courseStudent[record[1]] += [record[0]]
            students.add(record[0])

        sharedCourse = {}
        students = list(students)
        for i, s1 in enumerate(students):
            for s2 in students[i+1:]:
                sharedCourse[(s1, s2)] = []

        for c, s in courseStudent.items():
            for i, s1 in enumerate(s):
                for s2 in s[i+1:]:
                    if (s1, s2) in sharedCourse:
                        sharedCourse[(s1,s2)] += [c]
                    else:
                        sharedCourse[(s2,s1)] += [c]
        return sharedCourse


    def find_midway(self, courses):

        firstCourses = set()
        secondCourses = set()

        hashmap = collections.defaultdict(list)

        for c1, c2 in courses:
            firstCourses.add(c1)
            secondCourses.add(c2)
            hashmap[c1] += [c2]

        starters = firstCourses - secondCourses
        self.paths = []
        res = set()

        def dfs(c, path):
            if c not in hashmap:
                self.paths += [path+[c]]
                return
            for next_c in hashmap[c]:
                dfs(next_c, path+[c])
            return

        for s in list(starters):
            dfs(s, [])

        for p in self.paths:
            res.add(p[(len(p)-1)//2])

        return list(res)


sol = Solution()
student_course_pairs_1 = [
  ["58", "Software Design"],
  ["58", "Linear Algebra"],
  ["94", "Art History"],
  ["94", "Operating Systems"],
  ["17", "Software Design"],
  ["58", "Mechanics"],
  ["58", "Economics"],
  ["17", "Linear Algebra"],
  ["17", "Political Science"],
  ["94", "Economics"],
  ["25", "Economics"],
]
print(sol.find_pairs(student_course_pairs_1))
# find_pairs(student_course_pairs_1) =>
# {
#   [58, 17]: ["Software Design", "Linear Algebra"]
#   [58, 94]: ["Economics"]
#   [58, 25]: ["Economics"]
#   [94, 25]: ["Economics"]
#   [17, 94]: []
#   [17, 25]: []
# }

student_course_pairs_2 = [
  ["42", "Software Design"],
  ["0", "Advanced Mechanics"],
  ["9", "Art History"],
]
```

```
# find_pairs(student_course_pairs_2) =>
# {
#   [0, 42]: []
#   [0, 9]: []
#   [9, 42]: []
# }

print(sol.find_pairs(student_course_pairs_2))

all_courses = [
    ["Logic", "COBOL"],
    ["Data Structures", "Algorithms"],
    ["Creative Writing", "Data Structures"],
    ["Algorithms", "COBOL"],
    ["Intro to Computer Science", "Data Structures"],
    ["Logic", "Compilers"],
    ["Data Structures", "Logic"],
    ["Creative Writing", "System Administration"],
    ["Databases", "System Administration"],
    ["Creative Writing", "Databases"],
    ["Intro to Computer Science", "Graphics"],
]
print(sol.find_midway(all_courses))

"""
Visual representation:

                    ____________
                    |          |
                    | Graphics |
               ---->|__________|
               |                          ______________
____________   |                          |            |
|          |   |    ______________     -->| Algorithms |--\     _____________
| Intro to |   |    |            |    /   |____________|   \    |           |
| C.S.     |---+    | Data       |   /                      >-->| COBOL     |
|__________|    \   | Structures |--+     ______________   /    |___________|
                 >->|____________|   \    |            |  /
____________    /                     \-->| Logic      |-+      _____________
|          |   /    ______________        |____________|  \     |           |
| Creative |  /     |            |                         \--->| Compilers |
| Writing  |-+----->| Databases  |                              |___________|
|__________|  \     |____________|-\     _________________________
               \                    \    |                       |
                \--------------------+-->| System Administration |
                                         |_______________________|
"""

# Sample output (in any order):
#            ["Data Structures", "Creative Writing", "Databases", "Intro to
Computer Science"]
```

34. Valid Matrix

Q1: 给一个N*N的矩阵，判定是否是有效的矩阵。有效矩阵的定义是每一行或者每一列的数字都必须正好是1到N的数。输出一个bool。

Q2: nonogram

```
"""
A nonogram is a logic puzzle, similar to a crossword, in which the player is given
a blank grid and has to color it according to some instructions. Specifically,
each cell can be either black or white, which we will represent as 0 for black and
1 for white.

+------------+
| 1  1  1  1 |
| 0  1  1  1 |
| 0  1  0  0 |
| 1  1  0  1 |
| 0  0  1  1 |
+------------+

For each row and column, the instructions give the lengths of contiguous runs of
black (0) cells. For example, the instructions for one row of [ 2, 1 ] indicate
that there must be a run of two black cells, followed later by another run of one
black cell, and the rest of the row filled with white cells.

These are valid solutions: [ 1, 0, 0, 1, 0 ] and [ 0, 0, 1, 1, 0 ] and also [ 0,
0, 1, 0, 1 ]
This is not valid: [ 1, 0, 1, 0, 0 ] since the runs are not in the correct order.
This is not valid: [ 1, 0, 0, 0, 1 ] since the two runs of 0s are not separated by
1s.

Your job is to write a function to validate a possible solution against a set of
instructions. Given a 2D matrix representing a player's solution; and instructions
for each row along with additional instructions for each column; return True or
False according to whether both sets of instructions match.

Example instructions #1
```

```
matrix1 = [[1,1,1,1],
           [0,1,1,1],
           [0,1,0,0],
           [1,1,0,1],
           [0,0,1,1]]
rows1_1    =  [], [1], [1,2], [1], [2]
columns1_1 =  [2,1], [1], [2], [1]
validateNonogram(matrix1, rows1_1, columns1_1) => True

Example solution matrix:
matrix1 ->
                                   row
                +------------+     instructions
                | 1  1  1  1 | <-- []
                | 0  1  1  1 | <-- [1]
                | 0  1  0  0 | <-- [1,2]
                | 1  1  0  1 | <-- [1]
                | 0  0  1  1 | <-- [2]
                +------------+
                  ^  ^  ^  ^
                  |  |  |  |
  column       [2,1] | [2] |
  instructions      [1]   [1]


Example instructions #2

(same matrix as above)
rows1_2    =  [], [], [1], [1], [1,1]
columns1_2 =  [2], [1], [2], [1]
validateNonogram(matrix1, rows1_2, columns1_2) => False
"""
```

```python
import numpy as np
class Solution(object):

    def valid_matrix(self, matrix):
        def check(m):
            standard = set(list(range(1, len(m)+1)))
            for line in m:
                if set(line) != standard:
                    return False
            return True
        return check(matrix) and check(np.transpose(matrix))
    def valid_nonogram(self, matrix, rows, cols):

        if not matrix or len(matrix) != len(rows) or len(matrix[0]) != len(cols):
            return False

        def check(m, r):
            for i in range(len(m)):
                str_row = [str(x) for x in m[i]]
                zeros = ''.join(str_row).split('1')
                zeros = list(filter(lambda x: x!= "", zeros))
                num_zeros = list(map(len, zeros))
                if num_zeros != r[i]:
                    return False
            return True

        return check(matrix, rows) and check(np.transpose(matrix), cols)

sol = Solution()
matrix1 = [[1, 1, 3 ], [ 2, 3, 1 ], [ 3, 1, 2]]
matrix2 = [[1, 2, 3 ], [ 2, 3, 1 ], [ 3, 1, 2]]
print(sol.valid_matrix(matrix1))
print(sol.valid_matrix(matrix2))

matrix1 = [[1,1,1,1],
           [0,1,1,1],
           [0,1,0,0],
           [1,1,0,1],
           [0,0,1,1]]
rows1_1    =  [[], [1], [1,2], [1], [2]]
columns1_1 =  [[2,1], [1], [2], [1]]
print(sol.valid_nonogram(matrix1, rows1_1, columns1_1))

rows1_2    =  [[], [], [1], [1], [1,1]]
columns1_2 =  [[2], [1], [2], [1]]
print(sol.valid_nonogram(matrix1, rows1_2, columns1_2))

matrix2 = [
[ 1, 1 ],
[ 0, 0 ],
[ 0, 0 ],
[ 1, 0 ]
]
rows2_1    = [[], [2], [2], [1]]
columns2_1 = [[1, 1], [3]]
print(sol.valid_nonogram(matrix2, rows2_1, columns2_1))
```

Q3

```
"""

Suppose you have a one-dimensional board of two colors of tiles. Red tiles can only move to the right, black tiles can only move to the left. A tile can move 1 space at a time. Either they move to an adjacent empty space, or they can jump over a single tile of the other color to an empty space.

Eg:
red = R
black = B
empty = _

R _ B _ => _ R B _ or
         R B _ _

R B _ _ => _ B R _

Given a start and end configuration represented as a list of strings, return a list of valid moves to get from start to end (doesn't need to be shortest), or None if none exist. Include the start and end states in the list of moves.

Example:
start = ['R', '_', 'B', 'B']
end = ['B', '_', 'B', 'R']
->
moves = [
  ['R', '_', 'B', 'B'],
  ['_', 'R', 'B', 'B'],
  ['B', 'R', '_', 'B'],
  ['B', 'R', 'B', '_'],
  ['B', '_', 'B', 'R']
]

n: number of rows in the board
m: number of columns in the board

"""

# TODO --- Write your function

start = ["R","_","B","B"]
finish = ["B","_","B","R"]

# TODO --- Call your function with the test cases from above
```