Python Input-output

```
# input the array
arr = [int(x) for x in input().split()]

# Unknown number of input data, and ask you to read in per group
while True:
   try:
       a, b = map(int, input().strip().split())
       print(a+b)
   except EOFError:
      break

# 输入一个整数，告诉我们接下来有多少组数据，然后在输入每组数据的具体值。
# input an integer that tells number of subsequent data, and read in the exact value of each data
tcase=int(input().strip())
for case in range(tcase):
  a, b = map(int, input().strip().split())


#这次的输入实现输入一个整数，告诉我们有多少行，在输入每一行。
#对于每一行的输入，有划分为第一个数和其他的数，第一个数代表那一组数据一共有多少输入。
tcase = int(input().strip())
for case in range(tcase):
   data = map(int, input().strip().split())
   n, array = data[0], data[1:]
   sum = 0
   for i in range(n):
       sum += array[i]
```

Union Find

```
# Account Merge
class Solution:
    def accountsMerge(self, accounts):
        """
        :type accounts: List[List[str]]
        :rtype: List[List[str]]
        """
        # inverted index.. from email to users
        self.initialize(accounts)
        email_to_id = {}
        for i, account in enumerate(accounts):
            for email in account[1:]:
                email_to_id[email] = email_to_id.get(email, [])
                for user_id in email_to_id[email]:
                    self.union(i, user_id)
                email_to_id[email].append(i)
        merged_account = []
        for user_id, emails in self.id_to_email.items():
            if self.father[user_id] != user_id:
                continue
            merged_account.append([accounts[user_id][0], *sorted(emails)])
        return merged_account

    # union find
    def initialize(self,accounts):
        self.id_to_email = {}
        self.father = {}
        for user_id, emails in enumerate(accounts):
            self.father[user_id] = user_id
            self.id_to_email[user_id] = set(emails[1:])
    def find(self,user_id):
        path = []
        while user_id != self.father[user_id]:
            path.append(user_id)
            user_id = self.father[user_id]
        for n in path:
            self.father[n] = user_id
        return user_id

    def union(self,d1, d2):
        root_id1 = self.find(d1)
        root_id2 = self.find(d2)
        if root_id1!= root_id2:
            self.father[root_id1] = root_id2
            self.id_to_email[root_id2] =
self.id_to_email[root_id1].union(self.id_to_email[root_id2])
```

Trie

```
class TrieNode:

    def __init__(self):
        self.children = {}
        self.is_word = False


class Trie:

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

    """
    @param: word: a word
    @return: nothing
    """
    def insert(self, word):
        node = self.root
        for c in word:
            if c not in node.children:
                node.children[c] = TrieNode()
            node = node.children[c]

        node.is_word = True

    """
    return the node in the trie if exists
    """
    def find(self, word):
        node = self.root
        for c in word:
            node = node.children.get(c)
            if node is None:
                return None
        return node

    """
    @param: word: A string
    @return: if the word is in the trie.
    """
    def search(self, word):
        node = self.find(word)
        return node is not None and node.is_word

    """
    @param: prefix: A string
    @return: if there is any word in the trie that starts with the given
prefix.
    """
    def startsWith(self, prefix):
        return self.find(prefix) is not None
```

```
class TrieNode {
public:
    // Initialize your data structure here.
    TrieNode() {
        for (int i = 0; i < 26; i++)
            next[i] = NULL;
        isString = false;
    }
    TrieNode *next[26];
    bool isString;
};

class Trie {
public:
    Trie() {
        root = new TrieNode();
    }

    // Inserts a word into the trie.
    void insert(string word) {
        TrieNode *p = root;
        for (int i = 0; i < word.size(); i++) {
            if (p->next[word[i]-'a'] == NULL) {
                p->next[word[i]-'a'] = new TrieNode();
            }
            p = p->next[word[i]-'a'];
        }
        p->isString = true;
    }

    // Returns if the word is in the trie.
    bool search(string word) {
        TrieNode *p = root;
        for (int i = 0; i < word.size(); i++) {
            if (p == NULL) return false;
            p = p->next[word[i]-'a'];
        }
        if (p == NULL || p->isString == false)
            return false;
        return true;

    }

    // Returns if there is any word in the trie
    // that starts with the given prefix.
    bool startsWith(string prefix) {
        TrieNode *p = root;
        for (int i = 0; i < prefix.size(); i++) {
            p = p->next[prefix[i]-'a'];
            if (p == NULL) return false;
        }
        return true;
    }

private:
    TrieNode* root;
};
```

BFS

```
def bfs(graph, s):
   queue = []
   path = []
   seen = set()
   queue.append(s)
   seen.add(s)
   while len(queue) != 0:
       vertex = queue.pop(0)  # take from start

       for w in graph[vertex]:
           if w not in seen:
               queue.append(w)
               seen.add(w)
       path.append(vertex)

   return path
print("bfs", bfs(graph, input("starting vertex:")))
```

Topological Sorting

```
from collections import deque
class Solution:
    """
    @param: numCourses: a total of n courses
    @param: prerequisites: a list of prerequisite pairs
    @return: true if can finish all courses or false
    """
    def canFinish(self, numCourses, prerequisites):
        # write your code here
        edges = {i: [] for i in range(numCourses)}
        degrees = {i: 0 for i in range(numCourses)}
        for i, j in prerequisites:
            edges[j].append(i)
            degrees[i]+=1


        queue, count = deque([]), 0
        for i in range(numCourses):
            if degrees[i] == 0:
                queue.append(i)

        while queue:
            node = queue.popleft()
            count+=1
            for x in edges[node]:
                degrees[x]-=1
                if degrees[x] == 0:
                    queue.append(x)

        return count == numCourses

from heapq import heappush, heappop, heapify
class Solution:
    """
    @param words: a list of words
    @return: a string which is correct order
    """
    def alienOrder(self, words):
        # Write your code here
        graph = self.buildGraph(words)
        return self.topologicalSorting(graph)



    def buildGraph(self, words):
        graph = {}
        for word in words:
            for c in word:
                if c not in graph:

        n = len(words)
        for i in range(n-1):
            for j in range(min(len(words[i]), len(words[i+1]))):
                if words[i][j] != words[i+1][j]:
                    graph[words[i][j]].add(words[i+1][j])
                    break
        return graph

    def topologicalSorting(self, graph):
        indegree = {
            node: 0
            for node in graph
        }

        for node in graph:
            for neighbor in graph[node]:
                indegree[neighbor]+=1

        queue = [node for node in graph if indegree[node] == 0]
        heapify(queue)

        order = ""
        while queue:
            node = heappop(queue)
            order+=node
            for neighbor in graph[node]:
                indegree[neighbor]-=1
                if indegree[neighbor] == 0:
                    heappush(queue, neighbor)
        if len(order) == len(graph):
            return order
        return ""
```

DFS

```
def dfs(graph, s):
   stack = []
   path = []
   seen = set()
   stack.append(s)
   seen.add(s)
   while len(stack) != 0:
       vertex = stack.pop()
       for w in graph[vertex]:
           if w not in seen:
               seen.add(w)
               stack.append(w)
```

```python
        path.append(vertex)
    return path
print("dfs iterative", dfs(graph, input("starting vertex:")))
```

Combination

```python
def subsets( nums):
    nums = sorted(nums)
    combinations = []
    dfs(nums, 0, [], combinations)
    return combinations

def dfs(nums, start_index, combination, combinations):
    combinations.append(list(combination))

    for i in range(start_index, len(nums)):
        combination.append(nums[i])
        dfs(nums, i + 1, combination, combinations)
        combination.pop()
```

```python
class Solution:
    """
    @param: s: A string
    @param: wordDict: A set of words.
    @return: All possible sentences.
    """
    def wordBreak(self, s, wordDict):
        return self.dfs(s, wordDict, {})

    # 找到 s 的所有切割方案并 return
    def dfs(self, s, wordDict, memo):
        if s in memo:
            return memo[s]

        if len(s) == 0:
            return []

        partitions = []

        for i in range(1, len(s)):
            prefix = s[:i]
            if prefix not in wordDict:
                continue

            sub_partitions = self.dfs(s[i:], wordDict, memo)
            for partition in sub_partitions:
                partitions.append(prefix + " " + partition)

        if s in wordDict:
            partitions.append(s)

        memo[s] = partitions
        return partitions
```

```
class Solution:
    def isMatch(self, s: str, p: str) -> bool:
        memo = {}
        return self._isMatch(s, 0, p, 0, memo)

    def _isMatch(self, string, i, pattern, j, memo):
        if (i, j) in memo:
            return memo[(i, j)]
        if i == len(string):
            return self.isEmpty(pattern[j:])
        if j == len(pattern):
            return False

        if j + 1 < len(pattern) and pattern[j + 1] == '*':
            matched = self._isMatch(string, i, pattern, j + 2, memo)
or self.isMatchedChar(string[i], pattern[j]) and
self._isMatch(string, i + 1, pattern, j , memo)
        else:
            matched = self.isMatchedChar(string[i], pattern[j]) and
self._isMatch(string, i + 1, pattern, j + 1, memo)
        memo[(i, j)] = matched
        return matched
    def isEmpty(self, pattern):
        if len(pattern) % 2 == 1:
            return False
        for i in range(len(pattern)//2):
            if pattern[i*2 + 1] != '*':
                return False
        return True
    def isMatchedChar(self, s, p):
        return s == p or p == '.'
```

```
class Solution {
public:
    /*
    题意： 正则表达式匹配，'.'可以匹配任意字符，'*'可以匹配任意个(可以为0)'*'之前的字符
    不考虑'*'的话，题目变成简单的匹配。考虑'*'，可能产生的情况有匹配0、1、2...个字符
    因此可以使用递归或dp或其他方法解决
    */
    bool isMatch(string s, string p) {
        if (s.length() == 0){
            // s串匹配完合法的情况只有p为空，或是 "X*X*"的形式
            if (p.length() & 1) return false;  //不满足"X*X*"的形式
            else {
                for (int i = 1; i < p.length(); i += 2) {
                    if (p[i] != '*') return false;   //如果不是'*',不满足"X*X*"的形式
                }
            }
            return true;
        }
        if (p.length() == 0) return false;
        if (p.length() > 1 && p[1] == '*') {
            if (p[0] == '.' || s[0] == p[0]) {   //第0位匹配成功
                return isMatch(s.substr(1), p) || isMatch(s, p.substr(2)); //对于'*'，存在两种选择，'*'不进行匹配，或者匹配s的下一个
            } else return isMatch(s, p.substr(2));   //s没有匹配成功，继续拿p的下一位匹配
        } else {
            if (p[0] == '.' || s[0] == p[0]) {                //如果第0位置匹配成功
                return isMatch(s.substr(1), p.substr(1));            //继续匹配下一位
            } else return false;
        }
    }
};
```

Permutation

```
def permute(nums):
    if nums is None:
        return []
    if nums is []:
        return [[]]
    result1 = []
    result2 = [] # visited = {i: False for i in range(len(nums))} # visited = [0 for i in range(len(nums))]
    print("dfs1")
    print(dfs1(result1, [], sorted(nums)))

    print("dfs2")
    print(dfs2(result2, [], sorted(nums)))

    return [[]]


# unique value
def dfs1(result, curr, nums):
    if nums == []:
        result += [curr]
    else:
        for i in range(len(nums)):
            dfs1(result, curr + [nums[i]], nums[:i] + nums[i + 1:])


# with [1,1,2]
def dfs2(result, curr, nums):
    if nums == []:
        result += [curr]
    else:
        for i in range(len(nums)):
            if i > 0 and nums[i] == nums[i - 1]:
                continue
            dfs2(result, curr + [nums[i]], nums[:i] + nums[i + 1:])


class Solution:
    def wordPatternMatch(self, pattern: str, str: str) -> bool:
        return self._isMatch(pattern, str, {}, set())
    def _isMatch(self, pattern, string, char_to_word, used):
        if len(pattern) == 0:
            return len(string) == 0
        char = pattern[0]
        if char in char_to_word:
            word = char_to_word[char]
            if not string.startswith(word):
                return False
            return self._isMatch(pattern[1:], string[len(word):], 
char_to_word, used)

        for i in range(len(string)):
            word = string[:i+1]
            if word in used:
                continue
            used.add(word)
            char_to_word[char] = word

            if self._isMatch(pattern[1:], string[len(word):], 
char_to_word, used):
                return True
            used.remove(word)
            del char_to_word[char]
        return False
```

DP: BackPack Question

With Bag Size m, Size array A, Value V, and O(m) space

```
    public int backPackII(int m, int[] A, int V[]) {
        // write your code here
        int[] f = new int[m + 1];

        for(int i = 0; i < n; i++){
            for(int j = m; j >= A[i]; j--){
                if(f[j] < f[j - A[i]] + V[i]){
                    f[j] = f[j - A[i]] + V[i];
                }
            }
        }
```

Binary Search

```
#class SVNRepo:
#    @classmethod
#    def isBadVersion(cls, id)
#        # Run unit tests to check whether verison `id` is a bad version
#        # return true if unit tests passed else false.
# You can use SVNRepo.isBadVersion(10) to check whether version 10 is a 
# bad version.
class Solution:
    """
    @param n: An integer
    @return: An integer which is the first bad version.
    """
    def findFirstBadVersion(self, n):
        # write your code here
        start, end = 0, n
        while start + 1 < end:
            mid = (start + end)//2
            if SVNRepo.isBadVersion(mid):
                end = mid
            else:
                start = mid
        if SVNRepo.isBadVersion(start):
            return start
        else:
            return end

# Search in a rotated sorted array
class Solution:
    """
    @param A: an integer rotated sorted array
    @param target: an integer to be searched
    @return: an integer
    """
    def search(self, A, target):
        # write your code here
        if not A:
            return -1
        start, end = 0, len(A) - 1
        while start + 1 < end:
            mid = (start + end)//2
            if A[mid] >= A[start]:
                if A[start] <= target <= A[mid]:
                    end = mid
                else:
                    start = mid
            else:
                if A[mid] <= target <= A[end]:
                    start = mid
                else:
                    end = mid
        if A[start] == target:
            return start
        if A[end] == target:
            return end
        return -1
```