# 【LeetCode】代码模板，刷题必会



## 目录



本文的目的是收集一些典型的题目，记住其写法，理解其思想，即可做到一通百通。欢迎大家提出宝贵意见！

# 二分查找

最明显的题目就是34. Find First and Last Position of Element in Sorted Array

花花酱的二分查找专题视频：https://www.youtube.com/watch?v=v57INF2mb_s

模板：

区间定义： `[l, r)` 左闭右开

其中f(m)函数代表找到了满足条件的情况，有这个条件的判断就返回对应的位置，如果没有这个条件的判断就是lowwer_bound和higher_bound.

```
def binary_search(l, r):
    while l < r:
        m = l + (r - l) // 2
        if f(m):    # 判断找了没有，optional
            return m
        if g(m):
            r = m   # new range [l, m)
        else:
            l = m + 1 # new range [m+1, r)
    return l    # or not found
```

**lower bound**: find index of i, such that `A[i] >= x`

```
def lowwer_bound(self, nums, target):
    # find in range [left, right)
    left, right = 0, len(nums)
    while left < right:
        mid = left + (right - left) // 2
        if nums[mid] < target:
            left = mid + 1
        else:
            right = mid
    return left
```

**upper bound**: find index of i, such that `A[i] > x`

```
def higher_bound(self, nums, target):
    # find in range [left, right)
    left, right = 0, len(nums)
    while left < right:
        mid = left + (right - left) // 2
        if nums[mid] <= target:
            left = mid + 1
        else:
            right = mid
    return left
```

比如，题目69. Sqrt(x)。

```
class Solution(object):
    def mySqrt(self, x):
        """
        :type x: int
        :rtype: int
        """
        left, right = 0, x + 1
        # [left, right)
        while left < right:
            mid = left + (right - left) // 2
            if mid ** 2 == x:
                return mid
            if mid ** 2 < x:
                left = mid + 1
            else:
                right = mid
        return left - 1
```

# 排序的写法

C++的排序方法，使用sort并且重写comparator，如果需要使用外部变量，需要在中括号中放入&。

题目451. Sort Characters By Frequency。

```
class Solution {
public:
    string frequencySort(string s) {
        unordered_map<char, int> m;
        for (char c : s) ++m[c];
        sort(s.begin(), s.end(), [&](char& a, char& b){
            return m[a] > m[b] || (m[a] == m[b] && a < b);
        });
        return s;
    }
};
```

# BFS的写法

下面的这个写法是在一个邻接矩阵中找出离某一个点距离是k的点。

来自文章：【LeetCode】863. All Nodes Distance K in Binary Tree 解题报告（Python）

```
# BFS
bfs = [target.val]
visited = set([target.val])
for k in range(K):
    bfs = [y for x in bfs for y in conn[x] if y not in visited]
    visited |= set(bfs)
return bfs
```

127. Word Ladder

在BFS中保存已走过的步，并把已经走的合法路径删除掉。

```
class Solution(object):
    def ladderLength(self, beginWord, endWord, wordList):
        """
        :type beginWord: str
        :type endWord: str
        :type wordList: List[str]
        :rtype: int
        """
        wordset = set(wordList)
        bfs = collections.deque()
        bfs.append((beginWord, 1))
        while bfs:
            word, length = bfs.popleft()
            if word == endWord:
                return length
            for i in range(len(word)):
                for c in "abcdefghijklmnopqrstuvwxyz":
                    newWord = word[:i] + c + word[i + 1:]
                    if newWord in wordset and newWord != word:
                        wordset.remove(newWord)
                        bfs.append((newWord, length + 1))
        return 0

```

778. Swim in Rising Water

使用优先级队列来优先走比较矮的路，最后保存最高的那个格子的高度。

```python
class Solution(object):
    def swimInWater(self, grid):
        """
        :type grid: List[List[int]]
        :rtype: int
        """
        n = len(grid)
        visited, pq = set((0, 0)), [(grid[0][0], 0, 0)]
        res = 0
        while pq:
            T, i, j = heapq.heappop(pq)
            res = max(res, T)
            directions = [(0, 1), (0, -1), (-1, 0), (1, 0)]
            if i == j == n - 1:
                break
            for dir in directions:
                x, y = i + dir[0], j + dir[1]
                if x < 0 or x >= n or y < 0 or y >= n or (x, y) in visited:
                    continue
                heapq.heappush(pq, (grid[x][y], x, y))
                visited.add((x, y))
        return res
```

### 847. Shortest Path Visiting All Nodes

需要找出某顶点到其他顶点的最短路径。出发顶点不是确定的，每个顶点有可能访问多次。使用N位bit代表访问过的顶点的状态。如果到达了最终状态，那么现在步数就是所求。这个题把所有的节点都放入了起始队列中，相当于每次都是所有的顶点向前走一步。

```python
class Solution(object):
    def shortestPathLength(self, graph):
        """
        :type graph: List[List[int]]
        :rtype: int
        """
        N = len(graph)
        que = collections.deque()
        step = 0
        goal = (1 << N) - 1
        visited = [[0 for j in range(1 << N)] for i in range(N)]
        for i in range(N):
            que.append((i, 1 << i))
        while que:
            s = len(que)
            for i in range(s):
                node, state = que.popleft()
                if state == goal:
                    return step
                if visited[node][state]:
                    continue
                visited[node][state] = 1
                for nextNode in graph[node]:
                    que.append((nextNode, state | (1 << nextNode)))
            step += 1
        return step
```

### 429. N-ary Tree Level Order Traversal多叉树的层次遍历，这个BFS写法我觉得很经典。适合记忆。

```python
"""
# Definition for a Node.
class Node(object):
    def __init__(self, val, children):
        self.val = val
        self.children = children
```

```
"""
class Solution(object):
    def levelOrder(self, root):
        """
        :type root: Node
        :rtype: List[List[int]]
        """
        res = []
        que = collections.deque()
        que.append(root)
        while que:
            level = []
            size = len(que)
            for _ in range(size):
                node = que.popleft()
                if not node:
                    continue
                level.append(node.val)
                for child in node.children:
                    que.append(child)
            if level:
                res.append(level)
        return res
```

# DFS的写法

329. Longest Increasing Path in a Matrix

417. Pacific Atlantic Water Flow

778. Swim in Rising Water

二分查找+DFS

```
class Solution(object):
    def swimInWater(self, grid):
        """
        :type grid: List[List[int]]
        :rtype: int
        """
        n = len(grid)
        left, right = 0, n * n - 1
        while left <= right:
            mid = left + (right - left) / 2
            if self.dfs([[False] * n for _ in range(n)], grid, mid, n, 0, 0):
                right = mid - 1
            else:
                left = mid + 1
        return left

    def dfs(self, visited, grid, mid, n, i, j):
        visited[i][j] = True
        if i == n - 1 and j == n - 1:
            return True
        directions = [(0, 1), (0, -1), (-1, 0), (1, 0)]
        for dir in directions:
            x, y = i + dir[0], j + dir[1]
            if x < 0 or x >= n or y < 0 or y >= n or visited[x][y] or max(mid, grid[i][j]) != max(mid, gri
                continue
            if self.dfs(visited, grid, mid, n, x, y):
                return True
        return False
```

# 回溯法

下面这个题使用了回溯法，但是写的不够简单干练，遇到更好的解法的时候，要把这个题进行更新。

这个回溯思想，先去添加一个新的状态，看在这个状态的基础上，能不能找结果，如果找不到结果的话，那么就回退，即把这个结果和访问的记录给去掉。这个题使用了return True的方法让我们知道已经找出了结果，所以不用再递归了。

### 753. Cracking the Safe

```
class Solution(object):
    def crackSafe(self, n, k):
        """
        :type n: int
        :type k: int
        :rtype: str
        """
        res = ["0"] * n
        size = k ** n
        visited = set()
        visited.add("".join(res))
        if self.dfs(res, visited, size, n, k):
            return "".join(res)
        return ""

    def dfs(self, res, visited, size, n, k):
        if len(visited) == size:
            return True
        node = "".join(res[len(res) - n + 1:])
        for i in range(k):
            node = node + str(i)
            if node not in visited:
                res.append(str(i))
                visited.add(node)
                if self.dfs(res, visited, size, n, k):
                    return True
                res.pop()
                visited.remove(node)
            node = node[:-1]
```

### 312. Burst Balloons

```
class Solution(object):
    def maxCoins(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        n = len(nums)
        nums.insert(0, 1)
        nums.append(1)
        c = [[0] * (n + 2) for _ in range(n + 2)]
        return self.dfs(nums, c, 1, n)

    def dfs(self, nums, c, i, j):
        if i > j: return 0
        if c[i][j] > 0: return c[i][j]
        if i == j: return nums[i - 1] * nums[i] * nums[i + 1]
        res = 0
        for k in range(i, j + 1):
            res = max(res, self.dfs(nums, c, i, k - 1) + nums[i - 1] * nums[k] * nums[j + 1] + self.dfs(nu
        c[i][j] = res
        return c[i][j]
```

```
class Solution {
public:
    int countArrangement(int N) {
        int res = 0;
        vector<int> visited(N + 1, 0);
        helper(N, visited, 1, res);
        return res;
    }
private:
    void helper(int N, vector<int>& visited, int pos, int& res) {
        if (pos > N) {
            res++;
            return;
        }
        for (int i = 1; i <= N; i++) {
            if (visited[i] == 0 && (i % pos == 0 || pos % i == 0)) {
                visited[i] = 1;
                helper(N, visited, pos + 1, res);
                visited[i] = 0;
            }
        }
    }
};
```

如果需要保存路径的回溯法：

```
class Solution {
public:
    vector<vector<int>> permute(vector<int>& nums) {
        const int N = nums.size();
        vector<vector<int>> res;
        vector<int> path;
        vector<int> visited(N, 0);
        dfs(nums, 0, visited, res, path);
        return res;
    }
private:
    void dfs(vector<int>& nums, int pos, vector<int>& visited, vector<vector<int>>& res, vector<int>& path
        const int N = nums.size();
        if (pos == N) {
            res.push_back(path);
            return;
        }
        for (int i = 0; i < N; i++) {
            if (!visited[i]) {
                visited[i] = 1;
                path.push_back(nums[i]);
                dfs(nums, pos + 1, visited, res, path);
                path.pop_back();
                visited[i] = 0;
            }
        }
    }
};
```

# 树

## 递归

617. Merge Two Binary Trees把两个树重叠，重叠部分求和，不重叠部分是两个树不空的节点。

```
class Solution:
    def mergeTrees(self, t1, t2):
        if not t2:
            return t1
        if not t1:
            return t2
        newT = TreeNode(t1.val + t2.val)
        newT.left = self.mergeTrees(t1.left, t2.left)
        newT.right = self.mergeTrees(t1.right, t2.right)
        return newT
```

## 迭代

### 226. Invert Binary Tree

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def invertTree(self, root):
        """
        :type root: TreeNode
        :rtype: TreeNode
        """
        stack = []
        stack.append(root)
        while stack:
            node = stack.pop()
            if not node:
                continue
            node.left, node.right = node.right, node.left
            stack.append(node.left)
            stack.append(node.right)
        return root
```

## 前序遍历

### 144. Binary Tree Preorder Traversal

迭代写法：

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def preorderTraversal(self, root):
        """
        :type root: TreeNode
        :rtype: List[int]
        """
        if not root: return []
        res = []
        stack = []
        stack.append(root)
        while stack:
```

```
            node = stack.pop()
            if not node:
                continue
            res.append(node.val)
            stack.append(node.right)
            stack.append(node.left)
        return res
```

## 中序遍历

### 94. Binary Tree Inorder Traversal

迭代写法：

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def inorderTraversal(self, root):
        """
        :type root: TreeNode
        :rtype: List[int]
        """
        stack = []
        answer = []
        while True:
            while root:
                stack.append(root)
                root = root.left
            if not stack:
                return answer
            root = stack.pop()
            answer.append(root.val)
            root = root.right
```

## 后序遍历

### 145. Binary Tree Postorder Traversal

迭代写法如下：

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    vector<int> postorderTraversal(TreeNode* root) {
        vector<int> res;
        if (!root) return res;
        stack<TreeNode*> st;
        st.push(root);
        while (!st.empty()) {
            TreeNode* node = st.top(); st.pop();
            if (!node) continue;
```

```
            res.push_back(node->val);
            st.push(node->left);
            st.push(node->right);
        }
        reverse(res.begin(), res.end());
        return res;
    }
};
```

# 构建完全二叉树

完全二叉树是每一层都满的，因此找出要插入节点的父亲节点是很简单的。如果用数组tree保存着所有节点的层次遍历，那么新节点的父亲节点就是tree[(N -1)/2]，N是未插入该节点前的树的元素个数。
构建树的时候使用层次遍历，也就是BFS把所有的节点放入到tree里。插入的时候直接计算出新节点的父亲节点。获取root就是数组中的第0个节点。

### 919. Complete Binary Tree Inserter

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class CBTInserter(object):

    def __init__(self, root):
        """
        :type root: TreeNode
        """
        self.tree = list()
        queue = collections.deque()
        queue.append(root)
        while queue:
            node = queue.popleft()
            self.tree.append(node)
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)

    def insert(self, v):
        """
        :type v: int
        :rtype: int
        """
        _len = len(self.tree)
        father = self.tree[(_len - 1) / 2]
        node = TreeNode(v)
        if not father.left:
            father.left = node
        else:
            father.right = node
        self.tree.append(node)
        return father.val


    def get_root(self):
        """
        :rtype: TreeNode
        """
        return self.tree[0]

```

```
46
47 # Your CBTInserter object will be instantiated and called as such:
48 # obj = CBTInserter(root)
49 # param_1 = obj.insert(v)
50 # param_2 = obj.get_root()
51
```

# 并查集

不包含rank的话，代码很简短，应该背会。

721. Accounts Merge
https://leetcode.com/articles/accounts-merge/

```
1  class DSU:
2      def __init__(self):
3          self.par = range(10001)
4
5      def find(self, x):
6          if x != self.par[x]:
7              self.par[x] = self.find(self.par[x])
8          return self.par[x]
9
10     def union(self, x, y):
11         self.par[self.find(x)] = self.find(y)
12
13     def same(self, x, y):
14         return self.find(x) == self.find(y)
```

C++版本如下：

```
1  vector<int> map_; //i的parent，默认是i
2  int f(int a) {
3      if (map_[a] == a)
4          return a;
5      return f(map_[a]);
6  }
7  void u(int a, int b) {
8      int pa = f(a);
9      int pb = f(b);
10     if (pa == pb)
11         return;
12     map_[pa] = pb;
13 }
```

包含rank的，这里的rank表示树的高度：

684. Redundant Connection

```
1  class DSU(object):
2      def __init__(self):
3          self.par = range(1001)
4          self.rnk = [0] * 1001
5
6      def find(self, x):
7          if self.par[x] != x:
8              self.par[x] = self.find(self.par[x])
9          return self.par[x]
10
11     def union(self, x, y):
12         xr, yr = self.find(x), self.find(y)
13         if xr == yr:
```

```
            return False
        elif self.rnk[xr] < self.rnk[yr]:
            self.par[xr] = yr
        elif self.rnk[xr] > self.rnk[yr]:
            self.par[yr] = xr
        else:
            self.par[yr] = xr
            self.rnk[xr] += 1
        return True
```

另外一种rank方法是，保存树中节点的个数。

547. Friend Circles，代码如下：

```
class Solution(object):
    def findCircleNum(self, M):
        """
        :type M: List[List[int]]
        :rtype: int
        """
        dsu = DSU()
        N = len(M)
        for i in range(N):
            for j in range(i, N):
                if M[i][j]:
                    dsu.u(i, j)
        res = 0
        for i in range(N):
            if dsu.f(i) == i:
                res += 1
        return res

class DSU(object):
    def __init__(self):
        self.d = range(201)
        self.r = [0] * 201

    def f(self, a):
        return a if a == self.d[a] else self.f(self.d[a])

    def u(self, a, b):
        pa = self.f(a)
        pb = self.f(b)
        if (pa == pb):
            return
        if self.r[pa] < self.r[pb]:
            self.d[pa] = pb
            self.r[pb] += self.r[pa]
        else:
            self.d[pb] = pa
            self.r[pa] += self.r[pb]
```

# 前缀树

前缀树的题目可以使用字典解决，代码还是需要背一下的，C++版本的前缀树如下：

208. Implement Trie (Prefix Tree)这个题是纯考Trie的。参考代码如下：

```
class TrieNode {
public:
    vector<TrieNode*> child;
    bool isWord;

```

```
    TrieNode() : isWord(false), child(26, nullptr) {
    }
    ~TrieNode() {
        for (auto& c : child)
            delete c;
    }
};

class Trie {
public:
    /** Initialize your data structure here. */
    Trie() {
        root = new TrieNode();
    }

    /** Inserts a word into the trie. */
    void insert(string word) {
        TrieNode* p = root;
        for (char a : word) {
            int i = a - 'a';
            if (!p->child[i])
                p->child[i] = new TrieNode();
            p = p->child[i];
        }
        p->isWord = true;
    }

    /** Returns if the word is in the trie. */
    bool search(string word) {
        TrieNode* p = root;
        for (char a : word) {
            int i = a - 'a';
            if (!p->child[i])
                return false;
            p = p->child[i];
        }
        return p->isWord;
    }

    /** Returns if there is any word in the trie that starts with the given prefix. */
    bool startsWith(string prefix) {
        TrieNode* p = root;
        for (char a : prefix) {
            int i = a - 'a';
            if (!p->child[i])
                return false;
            p = p->child[i];
        }
        return true;
    }
private:
    TrieNode* root;
};

/**
 * Your Trie object will be instantiated and called as such:
 * Trie obj = new Trie();
 * obj.insert(word);
 * bool param_2 = obj.search(word);
 * bool param_3 = obj.startsWith(prefix);
 */
```

## 677. Map Sum Pairs

```
class MapSum {
public:
    /** Initialize your data structure here. */
    MapSum() {}

    void insert(string key, int val) {
        int inc = val - vals_[key];
        Trie* p = &root;
        for (const char c : key) {
            if (!p->children[c])
                p->children[c] = new Trie();
            p->children[c]->sum += inc;
            p = p->children[c];
        }
        vals_[key] = val;
    }

    int sum(string prefix) {
        Trie* p = &root;
        for (const char c : prefix) {
            if (!p->children[c])
                return 0;
            p = p->children[c];
        }
        return p->sum;
    }
private:
    struct Trie {
        Trie():children(128, nullptr), sum(0){}
        ~Trie(){
            for (auto child : children)
                if (child) delete child;
            children.clear();
        }
        vector<Trie*> children;
        int sum;
    };

    Trie root;
    unordered_map<string, int> vals_;
};
```

# 图遍历

743. Network Delay Time这个题很详细。

## Dijkstra算法

时间复杂度是O(N ^ 2 + E)，空间复杂度是O(N+E).

```
class Solution:
    def networkDelayTime(self, times, N, K):
        """
        :type times: List[List[int]]
        :type N: int
        :type K: int
        :rtype: int
        """
        K -= 1
        nodes = collections.defaultdict(list)
        for u, v, w in times:
            nodes[u - 1].append((v - 1, w))
        dist = [float('inf')] * N
```

```
        dist[K] = 0
        done = set()
        for _ in range(N):
            smallest = min((d, i) for (i, d) in enumerate(dist) if i not in done)[1]
            for v, w in nodes[smallest]:
                if v not in done and dist[smallest] + w < dist[v]:
                    dist[v] = dist[smallest] + w
            done.add(smallest)
        return -1 if float('inf') in dist else max(dist)
```

## Floyd-Warshall算法

时间复杂度O(n^3)， 空间复杂度O(n^2)。

```
class Solution:
    def networkDelayTime(self, times, N, K):
        """
        :type times: List[List[int]]
        :type N: int
        :type K: int
        :rtype: int
        """
        d = [[float('inf')] * N for _ in range(N)]
        for time in times:
            u, v, w = time[0] - 1, time[1] - 1, time[2]
            d[u][v] = w
        for i in range(N):
            d[i][i] = 0
        for k in range(N):
            for i in range(N):
                for j in range(N):
                    d[i][j] = min(d[i][j], d[i][k] + d[k][j])
        return -1 if float('inf') in d[K - 1] else max(d[K - 1])

```

## Bellman-Ford算法

时间复杂度O(ne)， 空间复杂度O(n)

```
class Solution:
    def networkDelayTime(self, times, N, K):
        """
        :type times: List[List[int]]
        :type N: int
        :type K: int
        :rtype: int
        """
        dist = [float('inf')] * N
        dist[K - 1] = 0
        for i in range(N):
            for time in times:
                u = time[0] - 1
                v = time[1] - 1
                w = time[2]
                dist[v] = min(dist[v], dist[u] + w)
        return -1 if float('inf') in dist else max(dist)

```

# 最小生成树

1135. Connecting Cities With Minimum Cost

## Kruskal算法

```
class Solution {
public:
    static bool cmp(vector<int> & a,vector<int> & b){
        return a[2] < b[2];
    }

    int find(vector<int> & f,int x){
        while(x != f[x]){
            x = f[x];
        }
        return x;
    }

    bool uni(vector<int> & f,int x,int y){
        int x1 = find(f,x);
        int y1 = find(f,y);
        f[x1] = y1;

        return true;
    }

    int minimumCost(int N, vector<vector<int>>& conections) {
        int ans = 0;
        int count = 0;
        vector<int> father(N+1,0);

        sort(conections.begin(),conections.end(),cmp);
        for(int i = 0;i <= N; ++i){
            father[i] = i;
        }

        for(auto conect : conections){
            if(find(father,conect[0]) != find(father,conect[1])){
                count++;
                ans += conect[2];
                uni(father,conect[0],conect[1]);
                if(count == N-1){
                    return ans;
                }
            }
        }

        return -1;
    }
};
```

## Prim算法

```
struct cmp {
    bool operator () (const vector<int> &a, const vector<int> &b) {
        return a[2] > b[2];
    }
};

class Solution {
public:
    int minimumCost(int N, vector<vector<int>>& conections) {
        int ans = 0;
        int selected = 0;
        vector<vector<pair<int,int>>> edgs(N+1,vector<pair<int,int>>());
        priority_queue<vector<int>,vector<vector<int>>,cmp> pq;
        vector<bool> visit(N+1,false);

```

```
        /*initial*/
        for(auto re : conections){
            edgs[re[0]].push_back(make_pair(re[1],re[2]));
            edgs[re[1]].push_back(make_pair(re[0],re[2]));
        }

        if(edgs[1].size() == 0){
            return -1;
        }

        /*kruskal*/
        selected = 1;
        visit[1] = true;
        for(int i = 0;i < edgs[1].size(); ++i){
            pq.push(vector<int>({1,edgs[1][i].first,edgs[1][i].second}));
        }

        while(!pq.empty()){
            vector<int> curr = pq.top();
            pq.pop();

            if(!visit[curr[1]]){
                visit[curr[1]] = true;
                ans += curr[2];
                for(auto e : edgs[curr[1]]){
                    pq.push(vector<int>({curr[1],e.first,e.second}));
                }
                selected++;
                if(selected == N){
                    return ans;
                }
            }
        }

        return -1;
    }
};
```

# 拓扑排序

BFS方式：

```
class Solution(object):
    def canFinish(self, N, prerequisites):
        """
        :type N,: int
        :type prerequisites: List[List[int]]
        :rtype: bool
        """
        graph = collections.defaultdict(list)
        indegrees = collections.defaultdict(int)
        for u, v in prerequisites:
            graph[v].append(u)
            indegrees[u] += 1
        for i in range(N):
            zeroDegree = False
            for j in range(N):
                if indegrees[j] == 0:
                    zeroDegree = True
                    break
            if not zeroDegree: return False
            indegrees[j] = -1
            for node in graph[j]:
```

```
                indegrees[node] -= 1
        return True
```

DFS方式：

```
class Solution(object):
    def canFinish(self, N, prerequisites):
        """
        :type N,: int
        :type prerequisites: List[List[int]]
        :rtype: bool
        """
        graph = collections.defaultdict(list)
        for u, v in prerequisites:
            graph[u].append(v)
        # 0 = Unknown, 1 = visiting, 2 = visited
        visited = [0] * N
        for i in range(N):
            if not self.dfs(graph, visited, i):
                return False
        return True

    # Can we add node i to visited successfully?
    def dfs(self, graph, visited, i):
        if visited[i] == 1: return False
        if visited[i] == 2: return True
        visited[i] = 1
        for j in graph[i]:
            if not self.dfs(graph, visited, j):
                return False
        visited[i] = 2
        return True
```

如果需要保存拓扑排序的路径：

BFS方式：

```
class Solution(object):
    def findOrder(self, numCourses, prerequisites):
        """
        :type numCourses: int
        :type prerequisites: List[List[int]]
        :rtype: List[int]
        """
        graph = collections.defaultdict(list)
        indegrees = collections.defaultdict(int)
        for u, v in prerequisites:
            graph[v].append(u)
            indegrees[u] += 1
        path = []
        for i in range(numCourses):
            zeroDegree = False
            for j in range(numCourses):
                if indegrees[j] == 0:
                    zeroDegree = True
                    break
            if not zeroDegree:
                return []
            indegrees[j] -= 1
            path.append(j)
            for node in graph[j]:
                indegrees[node] -= 1
        return path
```

DFS方式：

```
class Solution(object):
    def findOrder(self, numCourses, prerequisites):
        """
        :type numCourses: int
        :type prerequisites: List[List[int]]
        :rtype: List[int]
        """
        graph = collections.defaultdict(list)
        for u, v in prerequisites:
            graph[u].append(v)
        # 0 = Unknown, 1 = visiting, 2 = visited
        visited = [0] * numCourses
        path = []
        for i in range(numCourses):
            if not self.dfs(graph, visited, i, path):
                return []
        return path

    def dfs(self, graph, visited, i, path):
        if visited[i] == 1: return False
        if visited[i] == 2: return True
        visited[i] = 1
        for j in graph[i]:
            if not self.dfs(graph, visited, j, path):
                return False
        visited[i] = 2
        path.append(i)
        return True
```

207. Course Schedule

210. Course Schedule II

310. Minimum Height Trees

# 查找子字符串，双指针模板

这是一个模板，里面的map如果是双指针范围内的字符串字频的话，增加和减少的方式如下。

```
int findSubstring(string s){
        vector<int> map(128,0);
        int counter; // check whether the substring is valid
        int begin=0, end=0; //two pointers, one point to tail and one  head
        int d; //the length of substring

        for() { /* initialize the hash map here */ }

        while(end<s.size()){

            if(map[s[end++]]++ ?){  /* modify counter here */ }

            while(/* counter condition */){

                 /* update d here if finding minimum*/

                //increase begin to make it invalid/valid again

                if(map[s[begin++]]-- ?){ /*modify counter here*/ }
            }

            /* update d here if finding maximum*/
```

```
        }
        return d;
    }
```

### 76. Minimum Window Substring

这个题的map是t的字频，所以使用map更方式和上是相反的。

```
class Solution(object):
    def minWindow(self, s, t):
        """
        :type s: str
        :type t: str
        :rtype: str
        """
        res = ""
        left, cnt, minLen = 0, 0, float('inf')
        count = collections.Counter(t)
        for i, c in enumerate(s):
            count[c] -= 1
            if count[c] >= 0:
                cnt += 1
            while cnt == len(t):
                if minLen > i - left + 1:
                    minLen = i - left + 1
                    res = s[left : i + 1]
                count[s[left]] += 1
                if count[s[left]] > 0:
                    cnt -= 1
                left += 1
        return res
```

# 动态规划

## 状态搜索

### 688. Knight Probability in Chessboard

### 62. Unique Paths

### 63. Unique Paths II

### 913. Cat and Mouse

### 576. Out of Boundary Paths

```
class Solution(object):
    def findPaths(self, m, n, N, i, j):
        """
        :type m: int
        :type n: int
        :type N: int
        :type i: int
        :type j: int
        :rtype: int
        """
        dp = [[0] * n for _ in range(m)]
        for s in range(1, N + 1):
            curStatus = [[0] * n for _ in range(m)]
            for x in range(m):
                for y in range(n):
                    v1 = 1 if x == 0 else dp[x - 1][y]
                    v2 = 1 if x == m - 1 else dp[x + 1][y]
                    v3 = 1 if y == 0 else dp[x][y - 1]
                    v4 = 1 if y == n - 1 else dp[x][y + 1]
```

```
                    curStatus[x][y] = (v1 + v2 + v3 + v4) % (10**9 + 7)
            dp = curStatus
        return dp[i][j]
```

# 贪心

贪心算法（又称贪婪算法）是指，在对问题求解时，总是做出在当前看来最好的选择。也就是说，不从整体最优上加以考虑，他所作出的是在某种意义上的局部最优解。贪心算法和动态规划算法都是由局部最优导出全局最优，这里不得不比较下二者的区别

贪心算法：
1.贪心算法中，作出的每步贪心决策都无法改变，因为贪心策略是由上一步的最优解推导下一步的最优解，而上一部之前的最优解则不作保留。
2.由（1）中的介绍，可以知道贪心法正确的条件是：每一步的最优解一定包含上一步的最优解

动态规划算法：
1.全局最优解中一定包含某个局部最优解，但不一定包含前一个局部最优解，因此需要记录之前的所有最优解
2.动态规划的关键是状态转移方程，即如何由以求出的局部最优解来推导全局最优解
3.边界条件：即最简单的，可以直接得出的局部最优解

贪心是个思想，没有统一的模板。