本系列文章是学习大佬labuladong算法的过程记录和自我心得。

原文地址：<https://labuladong.gitbook.io/algo>

还可以直接搜索公众号【labuladong】进行关注。

学算法，学技术逗号，最重要的是坚持，明白自己的目标，学无止境，但有急缓，和效率之分。自我一定认清自己的目标，学习要有自主性。

算法是什么？一种用数学来表述的解决问题的方式。用最有效率的方式过关就行。

# 1. 学习算法和刷题的框架思维

从整体到细节，从上到下没从抽象到具体。

## 1.1 数据结构的存储方式

数据结构的存储方式只有两种，数组和链表，诸如散列表、栈、队列、堆、数、图等底层都是数组或者链表。

数组是紧凑型联系存储，链表是指针指向。

## 1.2 数据结构的基本操作

对于数据结构的操作无非就是增删改查。每一种数据结构的存在都是在不同的应用环境下尽可能高效的增删改查。

## 1.3 算法刷题指南

先刷二叉树，二叉树最容易培养框架思维，而且大部分算法技巧，本质上都是树的遍历问题。

细节出错，你得不到正确的答案，但只要有框架，你再错也错不到哪去。

# 2. 动态规划问题

动态规划一般就是求最值，寻求一种最优化方法。求最值就是穷举。

## 2.1 斐波那切数列

0,1,1,2,3,5,8,13,21,34,55

2.1.1 暴力递归

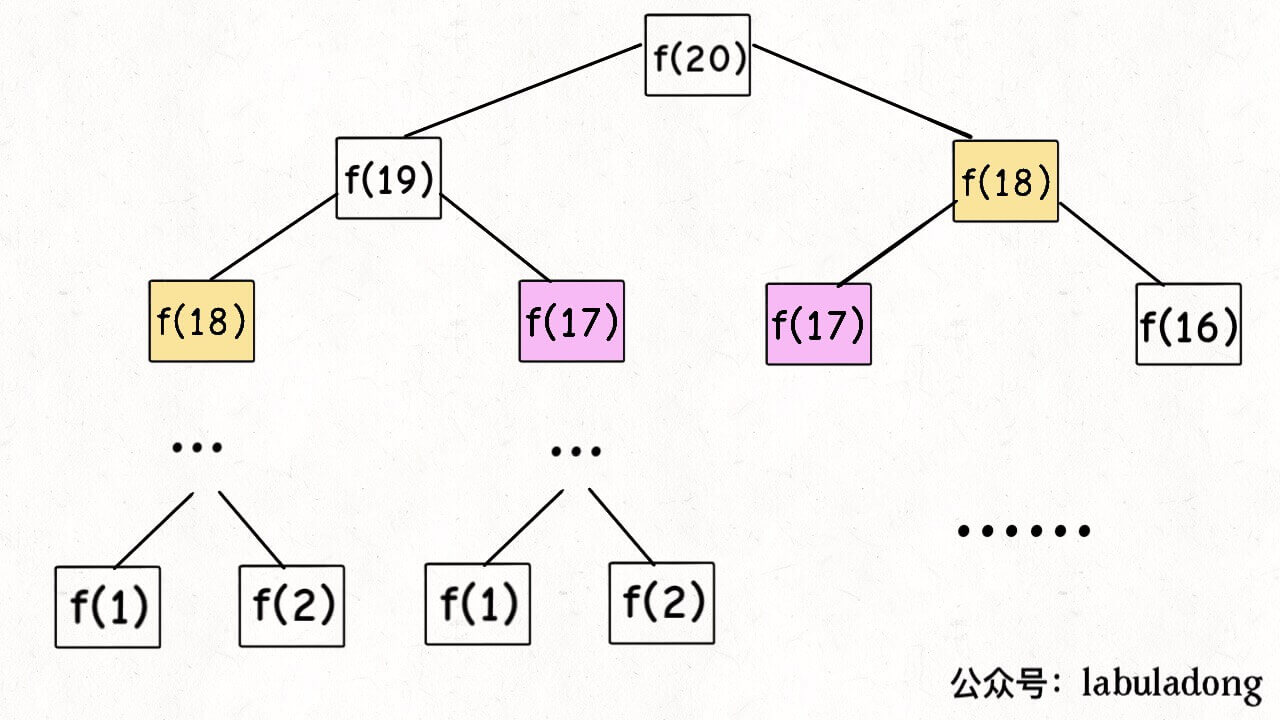
**int fib(int N){**

**if(N==1||N==2) return 1;**

**return fib(N-1)+fib(N-2);**

**}**

遇到递归问题，最好都化出递归图。



递归算法的时间复杂度怎么计算？就是用子问题的个数乘以一个子问题需要的时间。

首先计算子问题个数，就是递归树节点的总数，二叉树节点总数为之术级别，子问题个数为O(2^n)

然后计算解决子问题的时间，本算法中没有循环，只有加法，时间为O(1).

所以这个算法时间复杂度为二者相乘，指数级别，瞬间爆炸。

我们仔细观察递归树，发现存在大量重复计算，

这就是动态规划问题的第一个性质：重叠子问题。

## 2.2 带备忘录的递归解法

我们要解决重复计算的问题，那么就可以自制一个备忘录，每次算出某个答案，就记录在备忘录中，遇到新问题先去备忘录查一下，如果已经解决，直接用已有答案。

一般情况下，用一个数组来当备忘录。

**int fib(int N){**

**if(N<1) return 0;**

**//备忘录初始化为0**

**vector<int> memo(N+1,0);**

**//进行带备忘录的递归**

**retirn helper(memo,N)**

**}**

**int helper( vector<int>&memo,int n){**

**if (n == 1 || n == 2) return 1;**

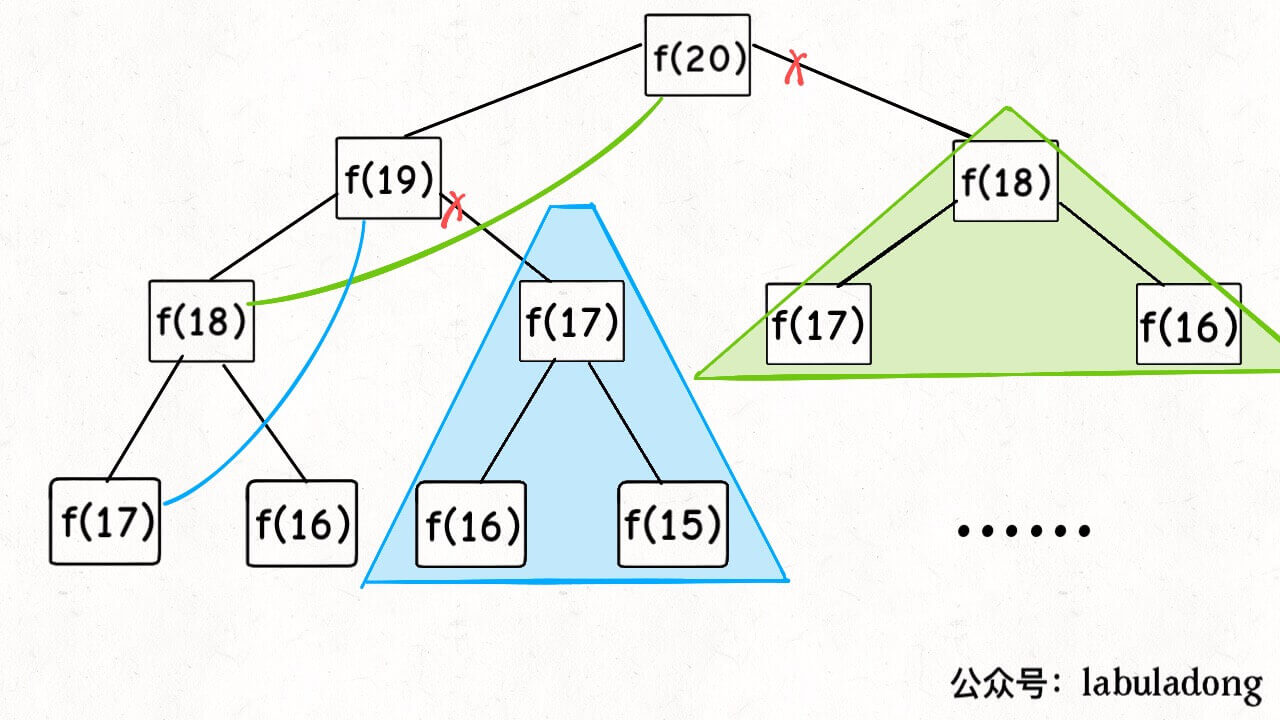
**if(memo[n]!=0) return memo[n];**

**memo[n] =helper(memo,n-1)+helper(memo,n-2);**

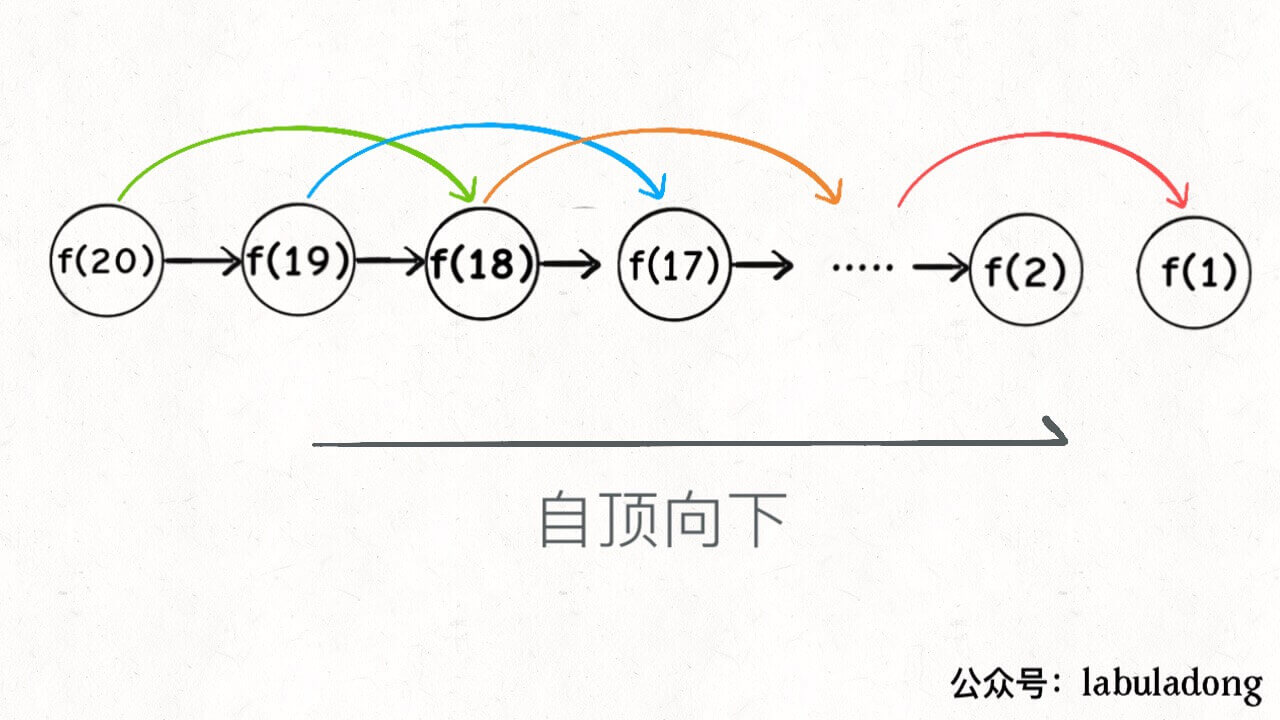
**return memo[n];**

**}**

再画一次递归树，看一下备忘录到底做了什么。



备忘录的作用就是剪枝的



这个方法的子问题个数为O(n).每个问题时间O(1)

所以时间复杂度是O(n).

实际上这种解法和迭代的动态规划已经差不多了，只不过这种方法叫作自顶向下，动态规划叫作自底向上。

什么叫自顶向下，就是从一个规模较大的原问题一层一层的分解，直到底层，就叫自顶向下。

自底向上就是从一个小问题，一步一步的向上推。

## 2.3 dp数组的迭代解法

如果我们把备忘录独立成一张表，叫作DP table，在这张表上完成自底向上的推算岂不是更好。

**int fib (int N){**

**vector<int> dp(N+1,0);**

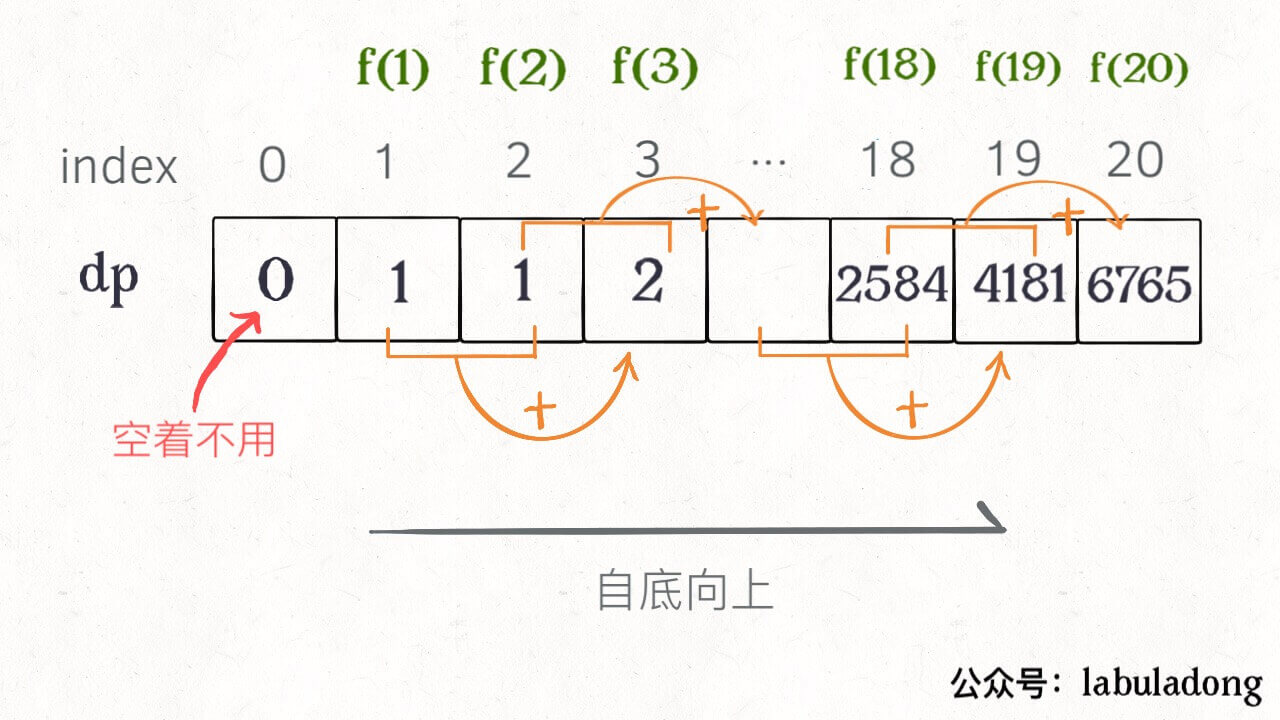
**dp[1] =dp[2]=1**

**for(int i=3;i<=N;i++)**

**dp[i] =dp[i-1]+dp[i-2]；**

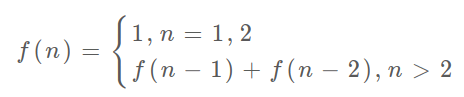
**return dp[N];**

**}**



两种解法差不多，效率也基本相同

状态转移方程，就是描述问题结构的数学形式：



为啥叫状态转移方程，就是为了听起来高大上，把f(n)当做一个状态，这个状态是由f(n-1)和f(n-2)相加转移而来，这就叫状态转移。

这个状态转移方程直接代表着暴力解法。

动态规划问题中最困难的就是写出这个暴力解法。

对于上述问题，还有一个细节可以优化，当前状态只和两个状态有关，所以并不需要那么长的DP table，用两个变量足矣

**int fib(int n) {**

**if (n == 2 || n == 1)**

**return 1;**

**int prev = 1, curr = 1;**

**for (int i = 3; i <= n; i++) {**

**int sum = prev + curr;**

**prev = curr;**

**curr = sum;**

**}**

**return curr;**

**}**

这个技巧就是状态压缩， 每次写代码的时候都可以考虑一下是否可以把DP表压缩大小。

严格来说斐波那契数列问题并不算动态规划，因为没有涉及求最值。

## 2.4 凑零钱的问题

**给你 k 种面值的硬币，面值分别为 c1, c2 ... ck，每种硬币的数量无限，再给一个总金额 amount，问你最少需要几枚硬币凑出这个金额，如果不可能凑出，算法返回 -1 。算法的函数签名如下：**

**// coins 中是可选硬币面值，amount 是目标金额**

**int coinChange(int[] coins, int amount);**

### 2.4.1 暴力递归

这个问题是动态规划问题，具有最优子结构，想要符合最优子结构，子问题之间必须互相独立。

互相独立代表，子问题之间都有最优解，互不干扰。就像你高考想考好大学，每一门都考高分就行，数学和语文的成绩互不干扰。

如何列出正确的状态转移方程？

1，确定 base case。目标金额amout为0时，算法返回0；

2，确定状态，硬币面额由题目给定，目标金额是变的，唯一状态就是目标金额amout

3，确定选择，导致状态产生变化的行为，通过每选择一枚硬币，减少目标金额到0的差额。

4，明确dp函数的定义，dp(n)定义：输入一个目标金额n，返回凑出目标金额n的最少硬币数量。

解法的伪代码如下

**def coinChange（conins:List[int],amout:int）:**

# 定义：要凑出金额 n，至少要 dp(n) 个硬币

**def dp(n)**

# 做选择，选择需要硬币最少的那个结果

**for coin in coins:**

**res = min(res, 1 + dp(n - coin))**

**return res**

**return dp(amout)**

解法的的最终答案，目标金额为0，硬币为0，金额小于0，无解返回-1.

**def coinChange(coins: List[int], amount: int):**

**def dp(n):**

**# base case**

**if n == 0: return 0**

**if n < 0: return -1**

**# 求最小值，所以初始化为正无穷**

**res = float('INF')**

**for coin in coins:**

**subproblem = dp(n - coin)**

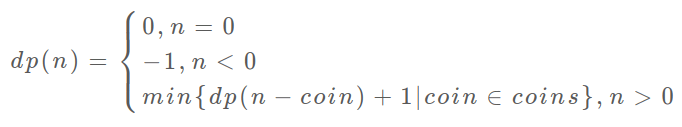
**# 子问题无解，跳过**

**if subproblem == -1: continue**

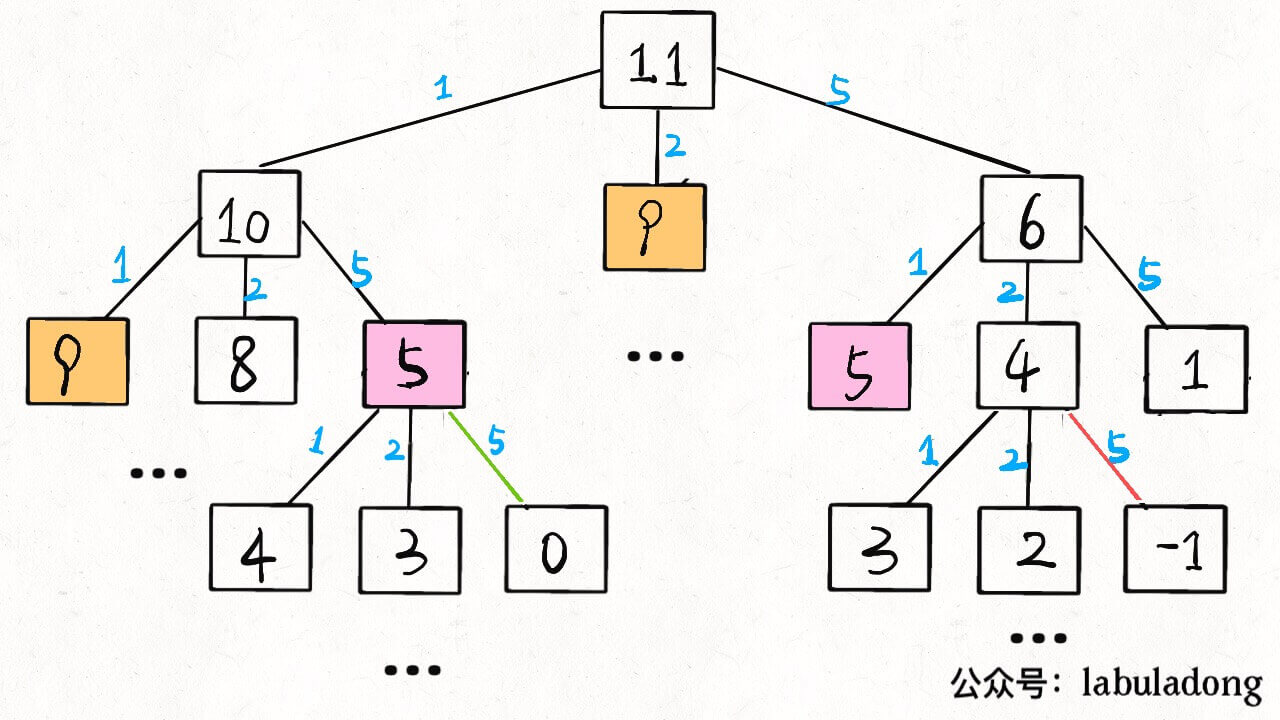
**res = min(res, 1 + subproblem)**

**return res if res != float('INF') else -1**

**return dp(amount)**



消除重叠子问题

****

子问题总数为递归树节点个数，为O(O^k),每个子问题包含一个for循环，复杂度O(K),总时间复杂度O(k\*n^k)

### 2.4.2 带备忘录的递归

**def coinChange(coins: List[int], amount: int):**

**# 备忘录**

**memo = dict()**

**def dp(n):**

**# 查备忘录，避免重复计算**

**if n in memo: return memo[n]**

**# base case**

**if n == 0: return 0**

**if n < 0: return -1**

**res = float('INF')**

**for coin in coins:**

**subproblem = dp(n - coin)**

**if subproblem == -1: continue**

**res = min(res, 1 + subproblem)**

**# 记入备忘录**

**memo[n] = res if res != float('INF') else -1**

**return memo[n]**

**return dp(amount)**

### 2.4.3 dp数组的迭代解法

**int coinChange(vector<int>& coins, int amount) {**

**// 数组大小为 amount + 1，初始值也为 amount + 1**

**vector<int> dp(amount + 1, amount + 1);**

**// base case**

**dp[0] = 0;**

**// 外层 for 循环在遍历所有状态的所有取值**

**for (int i = 0; i < dp.size(); i++) {**

**// 内层 for 循环在求所有选择的最小值**

**for (int coin : coins) {**

**// 子问题无解，跳过**

**if (i - coin < 0) continue;**

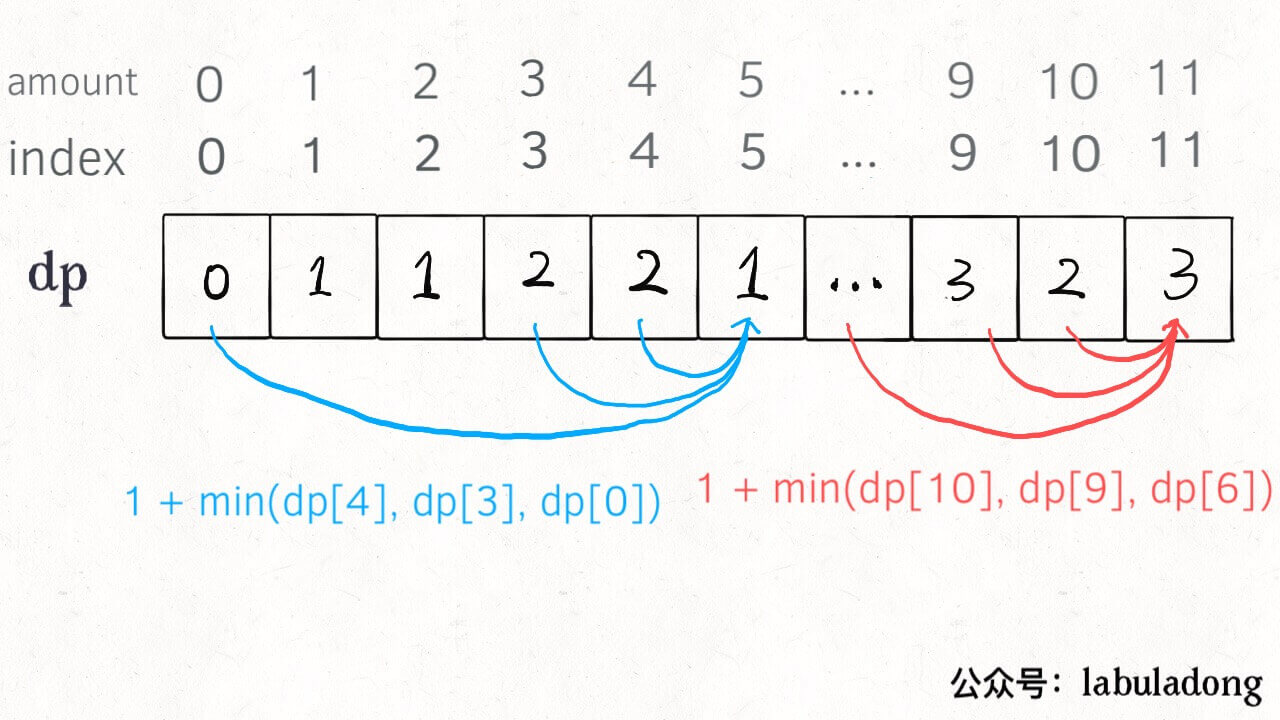
**dp[i] = min(dp[i], 1 + dp[i - coin]);**

**}**

**}**

**return (dp[amount] == amount + 1) ? -1 : dp[amount];**

**}**



计算机解决问题没有任何奇技淫巧，唯一的解决方法就是穷举。算法就是思考如何穷举，好的算法追求的是更聪明的穷举。

# 3. 回溯算法解题套路框架

解决一个回溯问题，实际上就是一个决策树的遍历过程。只需要思考3个问题：

路径：已经做出的选择

选择列表：当前可以做的选择

结束条件：到达决策树底层，无法在做选择的条件

回溯算法的框架：

result = []

def backtrack(路径, 选择列表):

if 满足结束条件:

result.add(路径)

return

for 选择 in 选择列表:

做选择

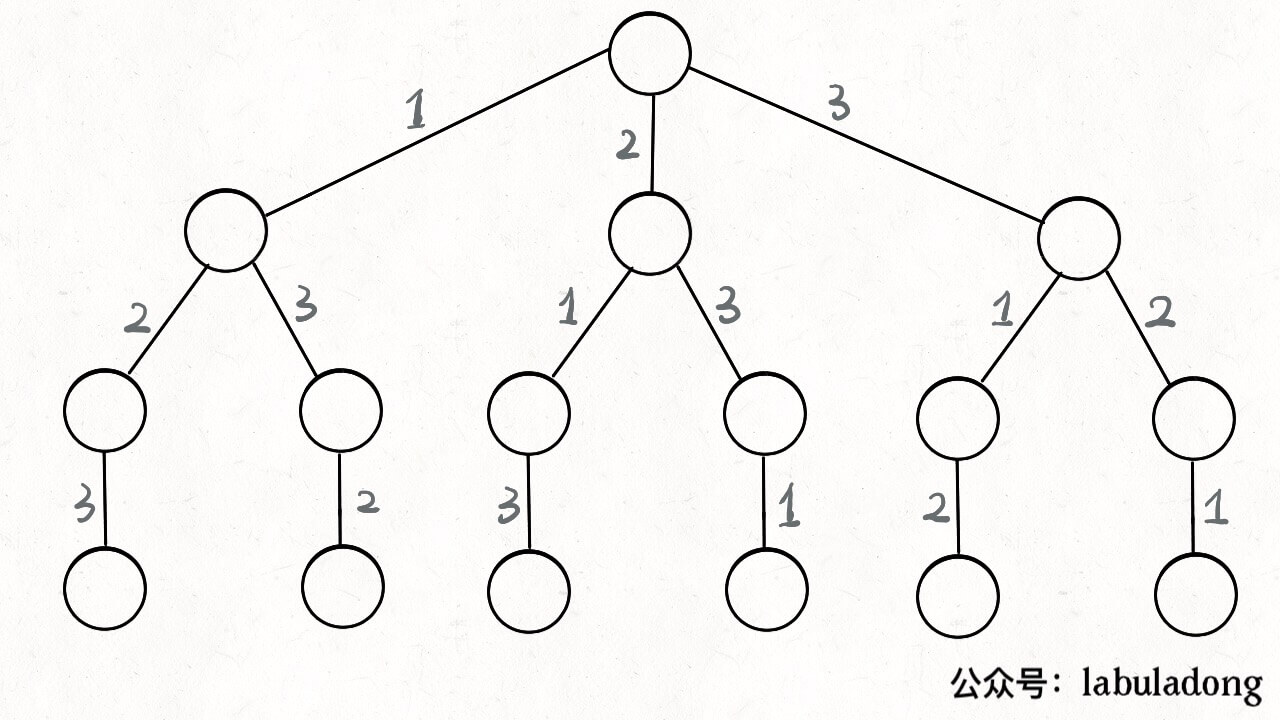
backtrack(路径, 选择列表)

撤销选择

核心是for循环里面的递归，递归之前做选择，递归之后撤销选择。

## 3.1 全排列问题

N个不重复的数，全排列有n!个。



从根遍历这棵树，记录路径上的数字,就是所有的全排列。这棵树就是回溯算法的决策树。我们站在每一个节点，都是在做决策。

各种搜索问题其实都是树的遍历问题。

void traverse(TreeNode root){

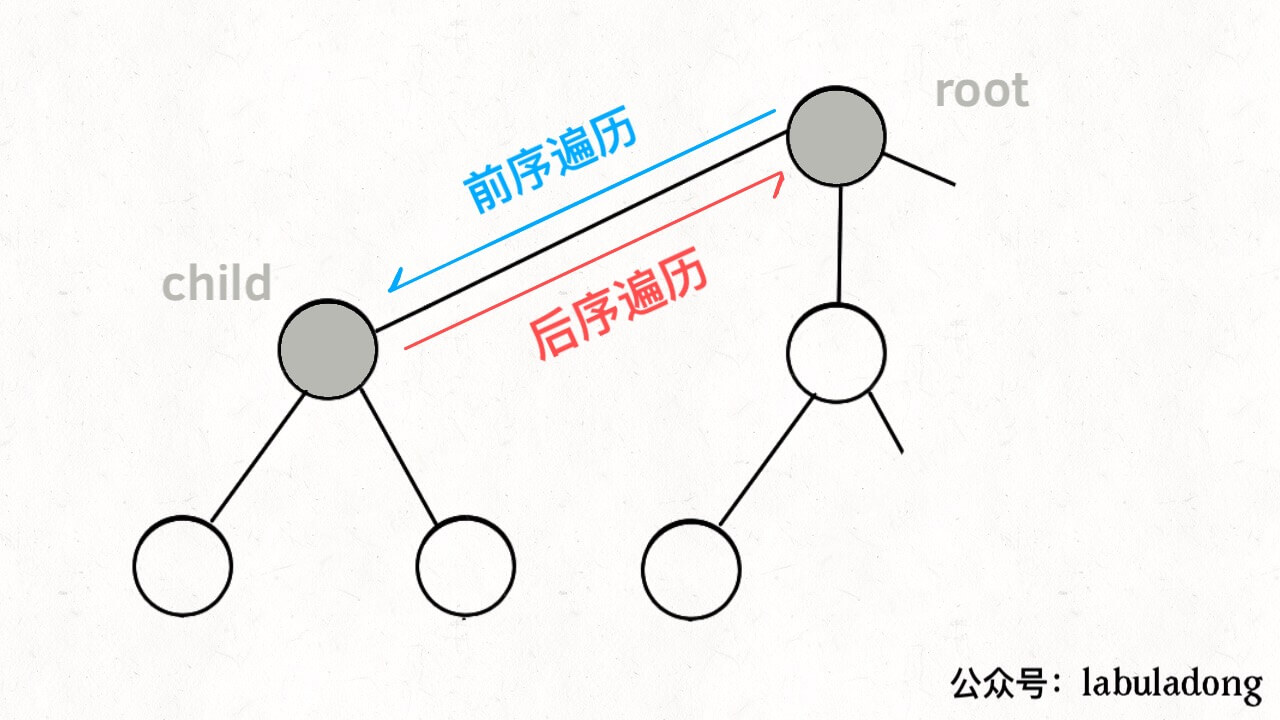
for(TreeNode child :root.childern)

// 前序遍历需要的操作

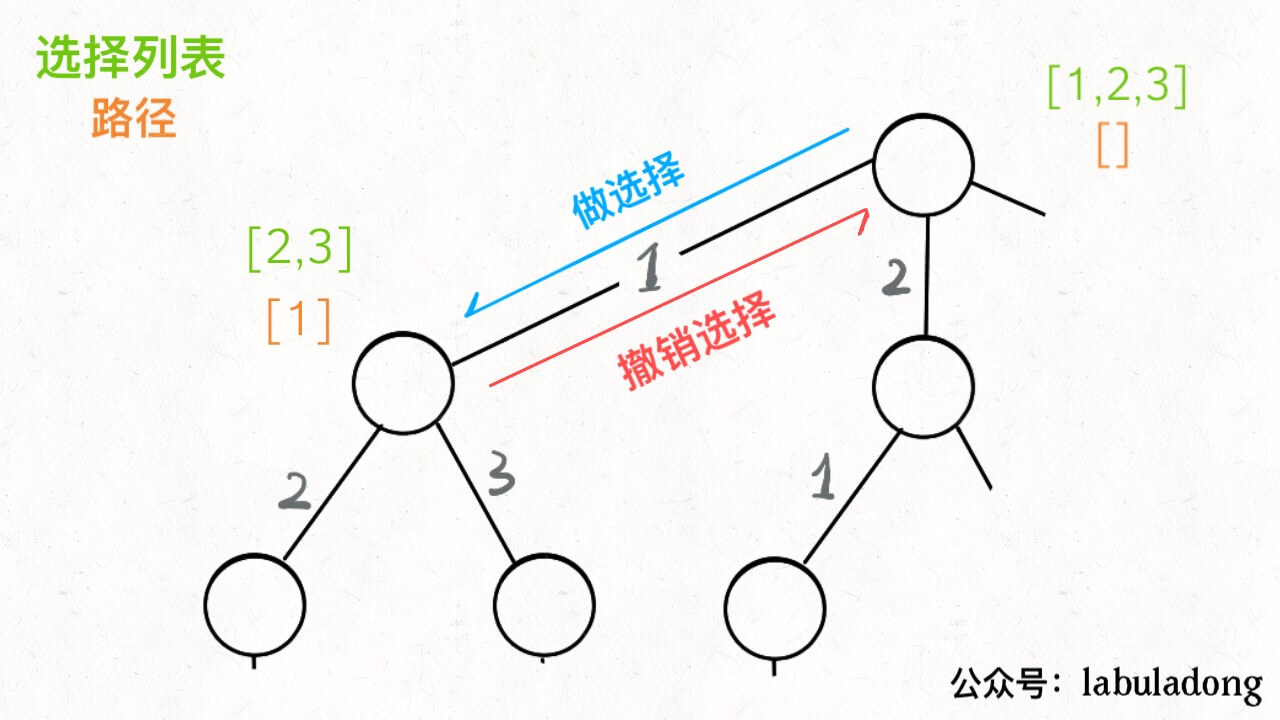
traverse(child);

// 后序遍历需要的操作

}

前序遍历和后序遍历，是两个很有用的时间点

前序遍历代码在进入某个节点之前执行，后序遍历的代码在离开某个节点之后的那个时间点执行。



for 选择 in 选择列表:

# 做选择

将该选择从选择列表移除

路径.add(选择)

backtrack(路径, 选择列表)

# 撤销选择

路径.remove(选择)

将该选择再加入选择列表

我们只要在递归之前作出选择，在递归之后撤销钢材的选择，就能正确得到每个结点的选择列表和路径。

List<List<Integer>> res =new LinkedList<>();

/\*主函数，输入一组不重复的数字，返回他们的全排列\*/

List<List<Integer>> permute(int[] nums) {

// 记录「路径」

LinkedList<Integer> track = new LinkedList<>();

backtrack(nums, track);

return res;

}

// 路径：记录在 track 中

// 选择列表：nums 中不存在于 track 的那些元素

// 结束条件：nums 中的元素全都在 track 中出现

void backtrack(int[] nums, LinkedList<Integer> track) {

// 触发结束条件

if (track.size() == nums.length) {

res.add(new LinkedList(track));

return;

}

for (int i = 0; i < nums.length; i++) {

// 排除不合法的选择

if (track.contains(nums[i]))

continue;

// 做选择

track.add(nums[i]);

// 进入下一层决策树

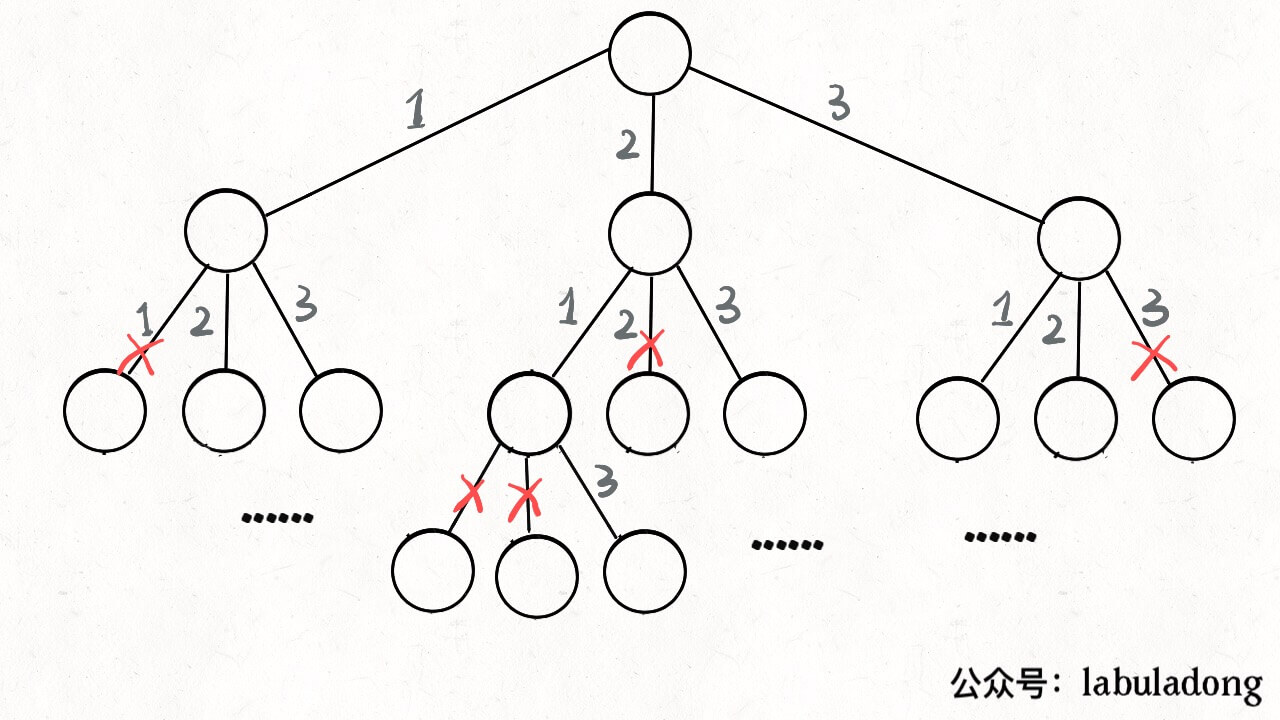
backtrack(nums, track);

// 取消选择

track.removeLast();

}

}



回溯算法是纯暴力穷举，复杂度一般都很高。

## 3.2 N皇后问题

给你一个NXN的棋盘，让你放置N个皇后，让他们不能互相攻击。

ps：皇后可以攻击同一行，同一列

直接套用框架：

vector<vector<string>> res;

/\*输入棋盘边长n，返回所有合法的放置\*/

vector<vector<string>> solveNQueens(int n){

//'.'表示空，'Q'表示皇后，初始化空棋盘

vector<string> board(n,string(n,'.'));

backtrack(board,0);

retrun res;

}

// 路径：board 中小于 row 的那些行都已经成功放置了皇后

// 选择列表：第 row 行的所有列都是放置皇后的选择

// 结束条件：row 超过 board 的最后一行

void backtrack(vector<string>& board, int row) {

// 触发结束条件

if (row == board.size()) {

res.push\_back(board);

return;

}

int n = board[row].size();

for (int col = 0; col < n; col++) {

// 排除不合法选择

if (!isValid(board, row, col))

continue;

// 做选择

board[row][col] = 'Q';

// 进入下一行决策

backtrack(board, row + 1);

// 撤销选择

board[row][col] = '.';

}

}

/\* 是否可以在 board[row][col] 放置皇后？ \*/

bool isValid(vector<string>& board, int row, int col) {

int n = board.size();

// 检查列是否有皇后互相冲突

for (int i = 0; i < n; i++) {

if (board[i][col] == 'Q')

return false;

}

// 检查右上方是否有皇后互相冲突

for (int i = row - 1, j = col + 1;

i >= 0 && j < n; i--, j++) {

if (board[i][j] == 'Q')

return false;

}

// 检查左上方是否有皇后互相冲突

for (int i = row - 1, j = col - 1;

i >= 0 && j >= 0; i--, j--) {

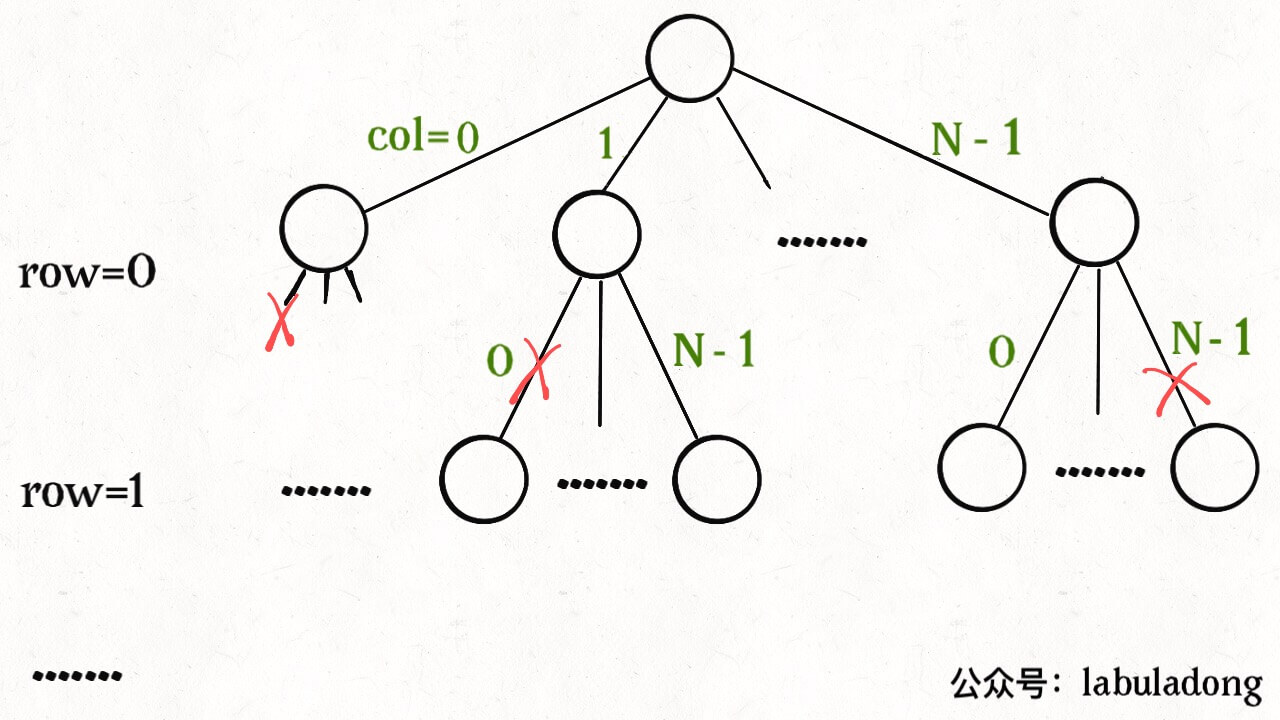
if (board[i][j] == 'Q')

return false;

}

return true;

}



当N=8，就是八皇后问题，这个问题复杂度很高，最坏时间复杂度为O(N^(N+1))

## 3.3 回溯算法总结

回溯算法就是多叉树的遍历问题，关键就是在前序和后序的位置做一些操作。

# 4 BFS算法解题套路框架

DFS算法就是回溯算法，BFS的核心思想就是把一些问题抽象成图，从一个点开始，向四周开始扩散。一般来说，写BFS算法都是用队列这种数据结构，每次都将一个节点周围所有节点加入队列。

BFS和DFS区别就是，BFS找到的路径一定是最短的，但代价就是空间复杂度比DFS大很多。

## 4.1 算法框架

BFS的本质就是让你在一幅图中找到从起点到终点的最近距离.

它的框架：

//从起点到终点的最近距离

int BFS(Node start ,Node target){

Queue<Node> q;//核心数据结构

Set<Node> visited;//避免走回头路

q.offer(start);//将起点加入队列

visited.add(start);

int step =0;//记录扩散的步数

while (q not empty ){

int sz =q.size();

for(int i=0;i<sz;i++){

Node cur =q.poll();

if(cur is target)

return step;

for(Node x:cur.adj())

if(x not in visited){

q.offer(x);

visited.add(x);

}

}

step++;

}

}

## 4.2 二叉树的最小高度

给定一个二叉树，找出最小深度，最小深度是从根节点到最近叶子结点的最短路径上的节点数量。

叶子节点是指没有子节点的节点。

第一步：明确起点和终点

起点就是root根节点，终点是最靠近根节点的那个叶子节点，叶子节点就是子节点都是null的节点

if （cur.left==null &&cur.right==null）//到达叶子节点

那么上面的框架就可以改造成

  public int minDepth(TreeNode root) {

        if (root == null ) return 0;

        Queue<TreeNode> q = new LinkedList<>();

        q.offer(root);

        //root 本身就是一层，depth初始化为1

        int depth=1;

        while(!q.isEmpty()){

            int sz= q.size();

            /\*将当前队列中的所有节点向四周扩散\*/

            for(int i=0;i<sz;i++){

                TreeNode cur=q.poll();

                //判断是否达到终点

                if(cur.left==null&&cur.right==null)

                    return depth;

                //将cur的相邻节点加入队列

                if(cur.left!=null)

                    q.offer(cur.left);

                if(cur.right!=null)

                    q.offer(cur.right);

            }

            //增加步数

            depth++;

        }

        return depth;

    }

二叉树是很简单的数据结构，再学习复杂问题之前，我们先解答两个问题：

1，为什么BFS可以找到最短距离，DFS不行吗？  
 形象点说，DFS是线，BFS是面，BFS是每增加一步，所有节点都向外迈一步，保证了第一次到达终点的时候，走的步数最少。

而DFS其实也可以找到最短路径，但是时间复杂度相对高很多。它需要靠递归的堆栈记录走过的路径，要找到最短路径要把所有树杈都探索万才可以。

2，BFS那么好，为啥DFS还要存在？

BFS可以找到最短距离，但是空间复杂度高，而DFS的空间复杂度较低

## 4.3 解开密码锁的最少次数



第一步，不管所有限制，让你设计一个算法，穷举所有可能的密码组合，你怎么做？  
四组锁，每一个位置可以向上转，也可以向下转，就有8中可能，每个节点有八个相邻的节点，又让你求最短距离，就是典型的BFS框架

//将s[j]向上拨动一下

String plusOne(String s,int j){

char[] ch=s.toCharArray();

if(ch[j]=='9'){

ch[j]='0';

}else{

ch[j]+=1;

}

return new String(ch);

}

//将s[i] 向下拨动一次

String minusOne(String s,int j){

char[] ch = s.toCharArray();

if (ch[j] == '0')

ch[j] = '9';

else

ch[j] -= 1;

return new String(ch);

}

//BFS 框架，打印出所有可能的密码

void BFS(String target) {

Queue<String> q = new LinkedList<>();

q.offer("0000");

while (!q.isEmpty()) {

int sz = q.size();

/\* 将当前队列中的所有节点向周围扩散 \*/

for (int i = 0; i < sz; i++) {

String cur = q.poll();

/\* 判断是否到达终点 \*/

System.out.println(cur);

/\* 将一个节点的相邻节点加入队列 \*/

for (int j = 0; j < 4; j++) {

String up = plusOne(cur, j);

String down = minusOne(cur, j);

q.offer(up);

q.offer(down);

}

}

/\* 在这里增加步数 \*/

}

return;

}

这段代码有很多问题，

1，会走回头路，

2，没有终止条件

3，没对deadends进行处理。

int openLock(String[] deadends, String target) {

// 记录需要跳过的死亡密码

Set<String> deads = new HashSet<>();

for (String s : deadends) deads.add(s);

// 记录已经穷举过的密码，防止走回头路

Set<String> visited = new HashSet<>();

Queue<String> q = new LinkedList<>();

// 从起点开始启动广度优先搜索

int step = 0;

q.offer("0000");

visited.add("0000");

while (!q.isEmpty()) {

int sz = q.size();

/\* 将当前队列中的所有节点向周围扩散 \*/

for (int i = 0; i < sz; i++) {

String cur = q.poll();

/\* 判断是否到达终点 \*/

if (deads.contains(cur))

continue;

if (cur.equals(target))

return step;

/\* 将一个节点的未遍历相邻节点加入队列 \*/

for (int j = 0; j < 4; j++) {

String up = plusOne(cur, j);

if (!visited.contains(up)) {

q.offer(up);

visited.add(up);

}

String down = minusOne(cur, j);

if (!visited.contains(down)) {

q.offer(down);

visited.add(down);

}

}

}

/\* 在这里增加步数 \*/

step++;

}

// 如果穷举完都没找到目标密码，那就是找不到了

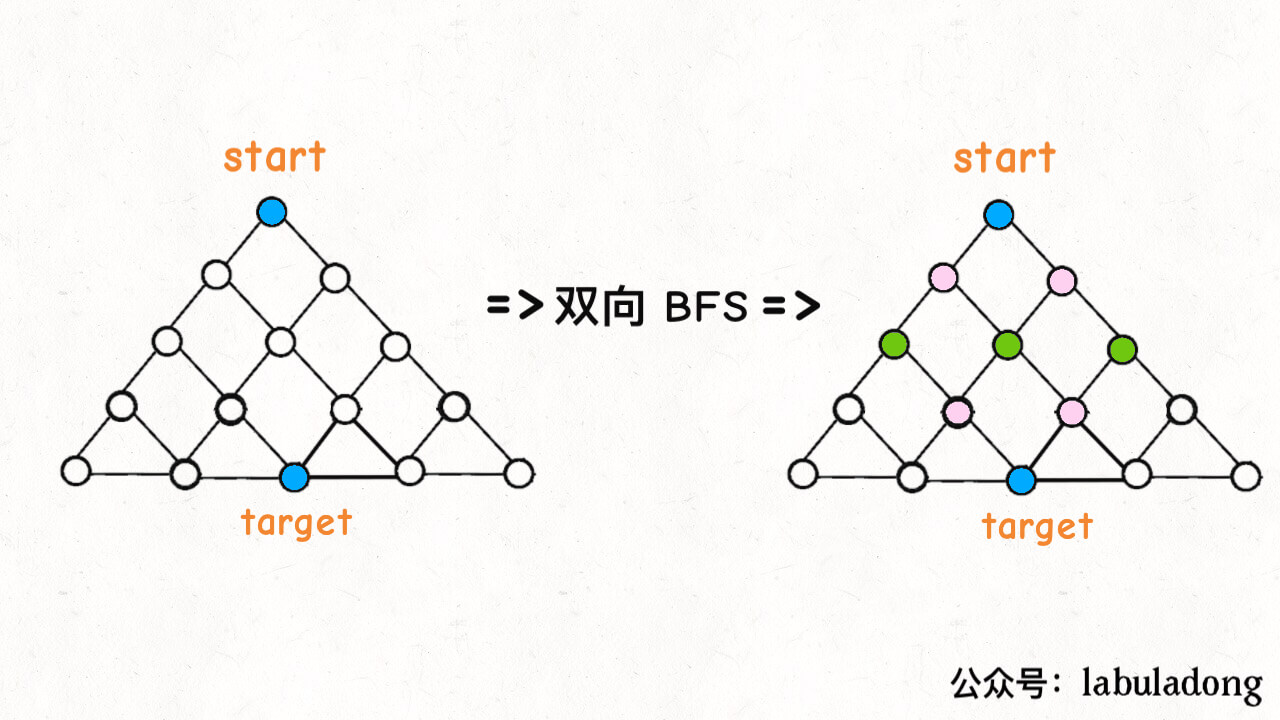
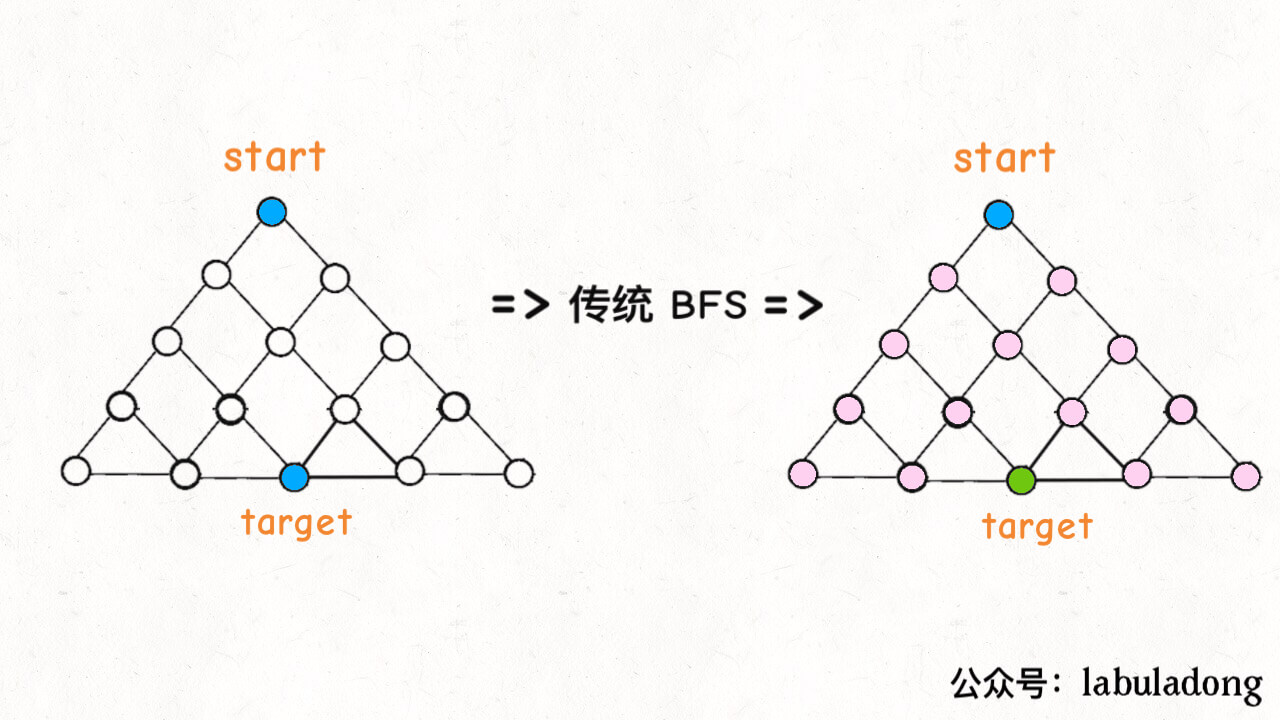
return -1;

}

## 4.4 双向BFS优化

BFS还有一种稍微高级一点的优化思路，双向BFS。

单向的是从起点向四周扩散，遇到终点停止，而双向则是从起点和终点同时扩散，两边有交集的时候停止。



双向BFS要比传统高效。不过必须知道终点在哪里，

int openLock(String[] deadends, String target) {

Set<String> deads = new HashSet<>();

for (String s : deadends) deads.add(s);

// 用集合不用队列，可以快速判断元素是否存在

Set<String> q1 = new HashSet<>();

Set<String> q2 = new HashSet<>();

Set<String> visited = new HashSet<>();

int step = 0;

q1.add("0000");

q2.add(target);

while (!q1.isEmpty() && !q2.isEmpty()) {

// 哈希集合在遍历的过程中不能修改，用 temp 存储扩散结果

Set<String> temp = new HashSet<>();

/\* 将 q1 中的所有节点向周围扩散 \*/

for (String cur : q1) {

/\* 判断是否到达终点 \*/

if (deads.contains(cur))

continue;

if (q2.contains(cur))

return step;

visited.add(cur);

/\* 将一个节点的未遍历相邻节点加入集合 \*/

for (int j = 0; j < 4; j++) {

String up = plusOne(cur, j);

if (!visited.contains(up))

temp.add(up);

String down = minusOne(cur, j);

if (!visited.contains(down))

temp.add(down);

}

}

/\* 在这里增加步数 \*/

step++;

// temp 相当于 q1

// 这里交换 q1 q2，下一轮 while 就是扩散 q2

q1 = q2;

q2 = temp;

}

return -1;

}

双向BFS不再使用队列，而是使用HashSet方便快速判断集合是否有交集。

# 5 二分搜索

## 5.1 二分查找框架

int binarySearch(int[] nums,int target){

int left =0,right=...;

while(...){

int mid =left +(right-left)/2;

if(nums[mi]==target){

...

}else if(nums[mid]<target){

left=...

}else if(nums[mid]>target){

rigtht=...

}

}

return ...;

}

计算mid时需要防止溢出，left + (right - left) / 2 就和 (left + right) / 2 的结果相同，但是有效防止了 left 和 right 太大直接相加导致溢出。

## 5.2 寻找一个数

搜索一个数,如果存在,返回索引,否则返回-1;

int binarySearch(int[] nums,int target){

int left =0;

int right = nums.length-1;

while(left<=right){

int mid =left +(right -left)/2;

if(nums[mid] == target){

return mid;

}else if(nums[mid] <target){

left =mid +1;

}else f( nums[mid] > target){

right = mid-1;

}

}

return -1;

}

## 5.3 寻找左侧边界的二分搜索

\*\*\*

# 6 算法笔试的套路

## 6.1 避实就虚

单链表翻转，比较简单的方法是输入数组中，然后用双指针技巧把它翻转。

## 6.2 巧用随机数

遇到二值的题目，会做的话就正常做，不会做就直接提交结果

看一下通过率，然后写下面代码

// 60% 的概率输出 YES，40% 的概率输出 NO

System.out.println((new Random().nextInt() % 100) < 60 ? "YES" : "NO");

在力所能及的范围内做到极致

## 6.3 代码要分层

main函数负责接收数据，solution函数负责处理数据和输出答案

## 6.4 考前复习策略

尽可能多的看各种各样的题目，直接看别人的答案，看懂思路就好。

没有什么问题是暴力穷举解决不了的。