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。

# 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

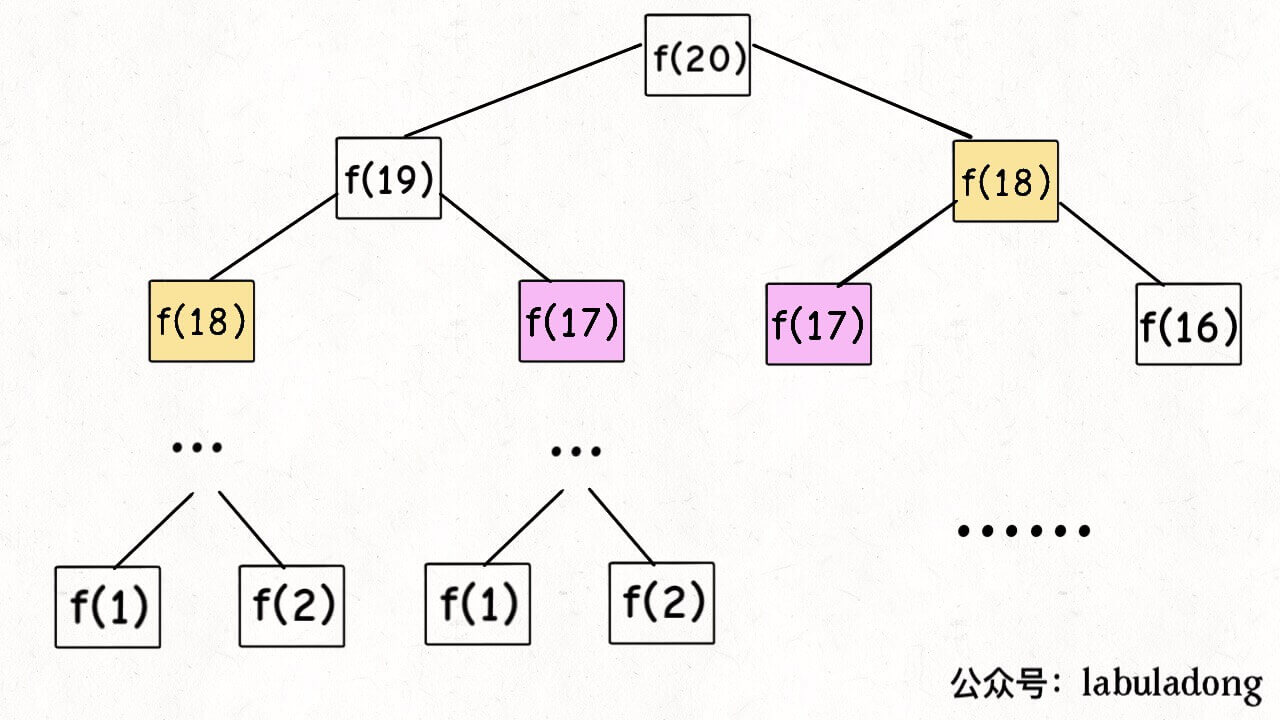
**int fib(int N){**

**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;**

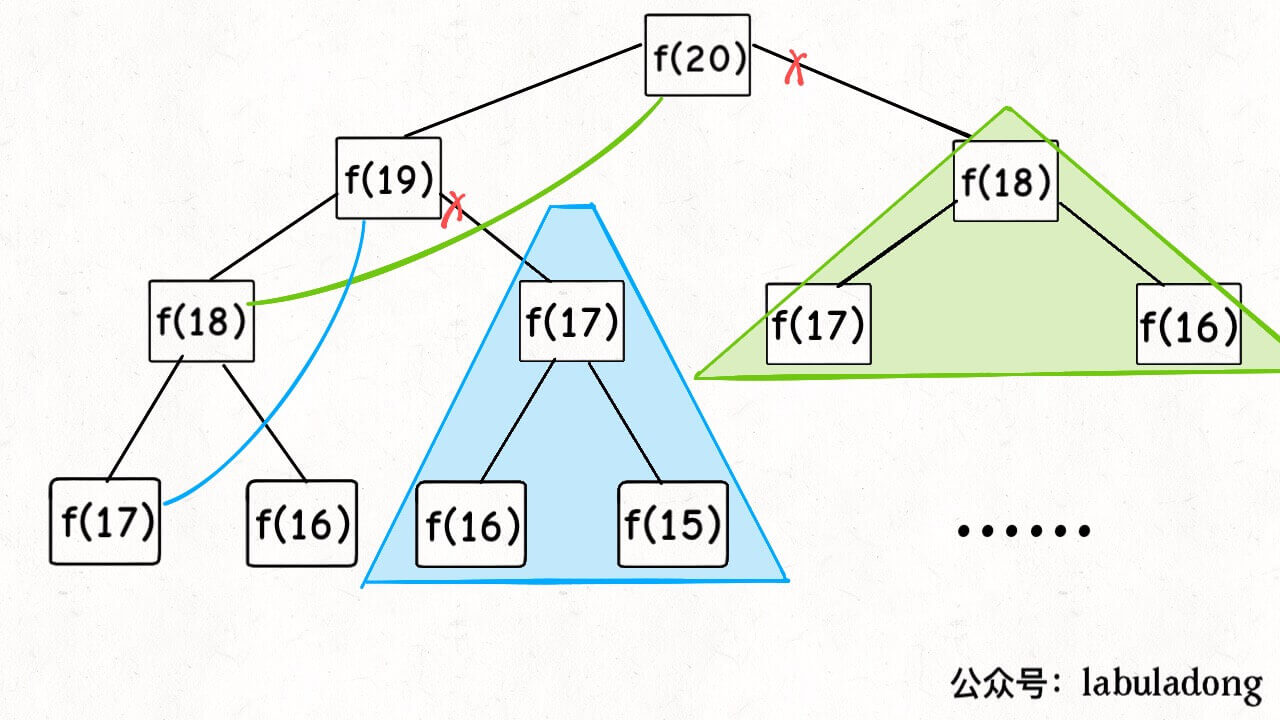
**if(memo[n]!=0) return memo[n];**

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

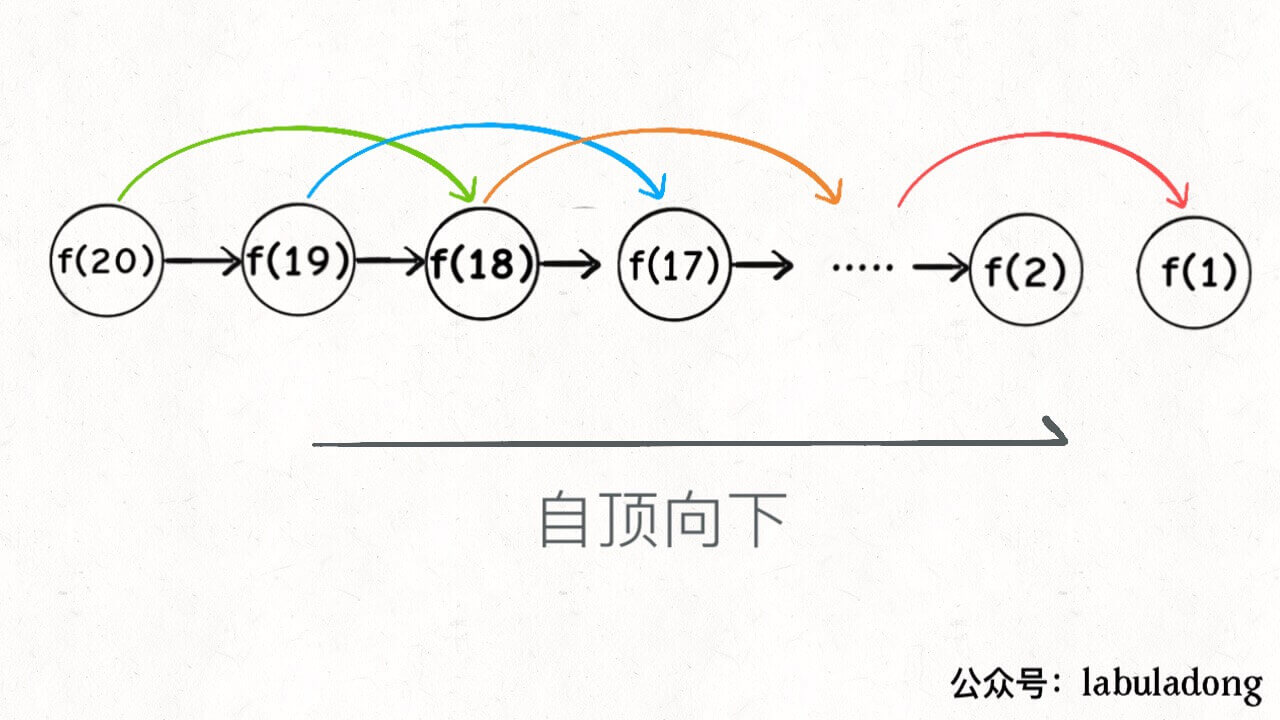
**return memo[n];**

**}**

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);**

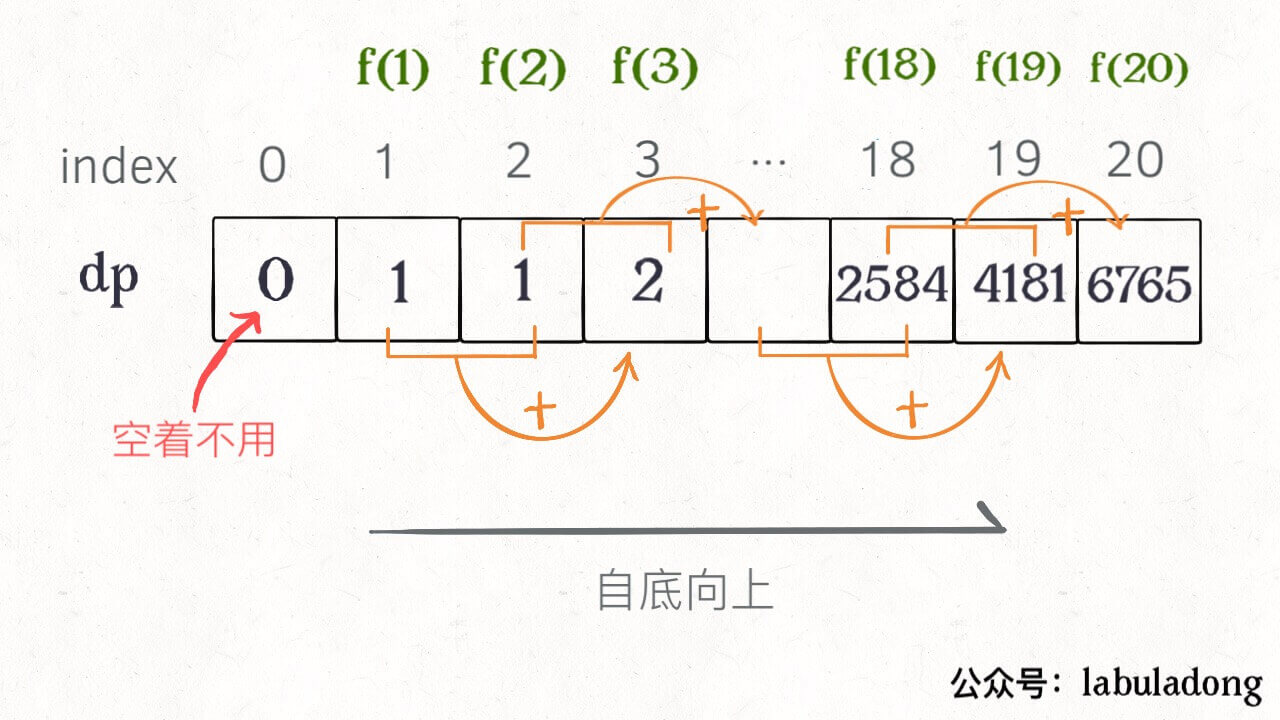
**dp[1] =dp[2]=1**

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

**dp[i] =dp[i-1]+dp[i-2]；**

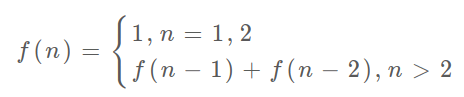
**return dp[N];**

**}**



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)**

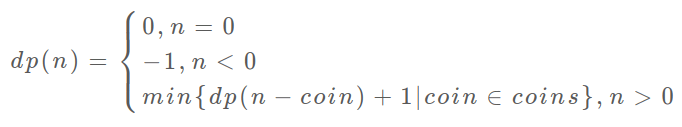
**# 子问题无解，跳过**

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

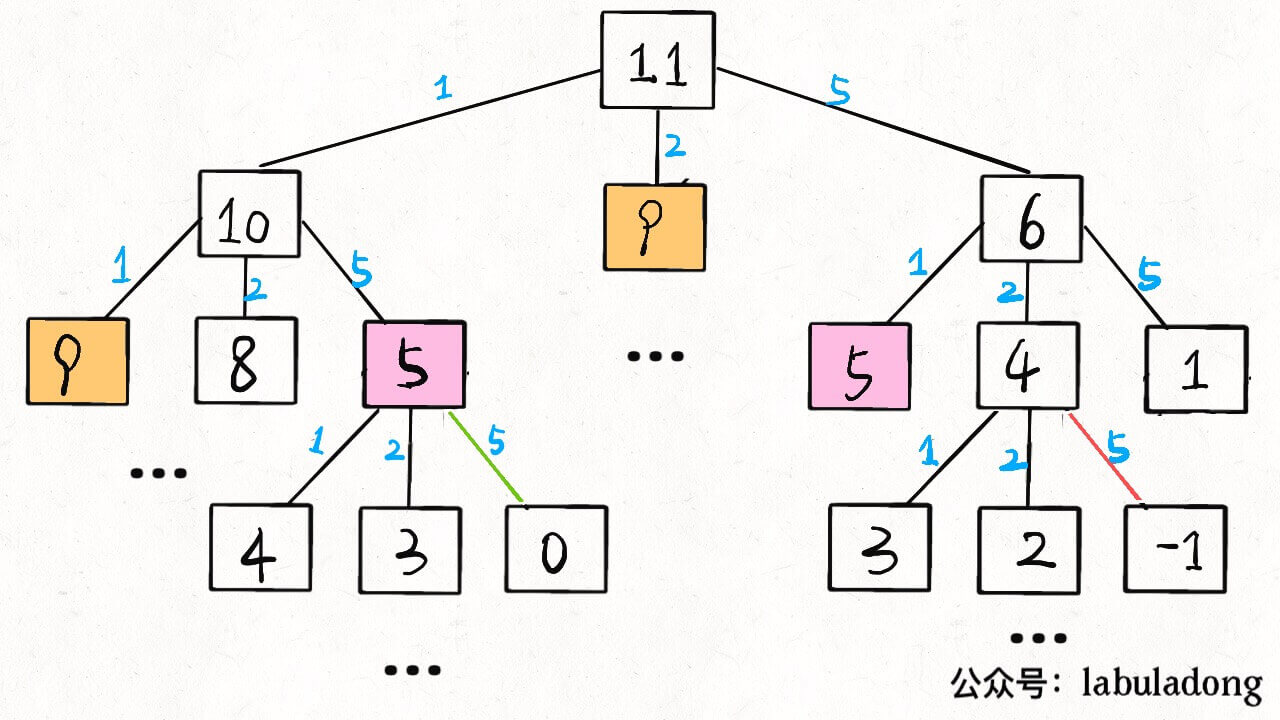
**res = min(res, 1 + subproblem)**

**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;**

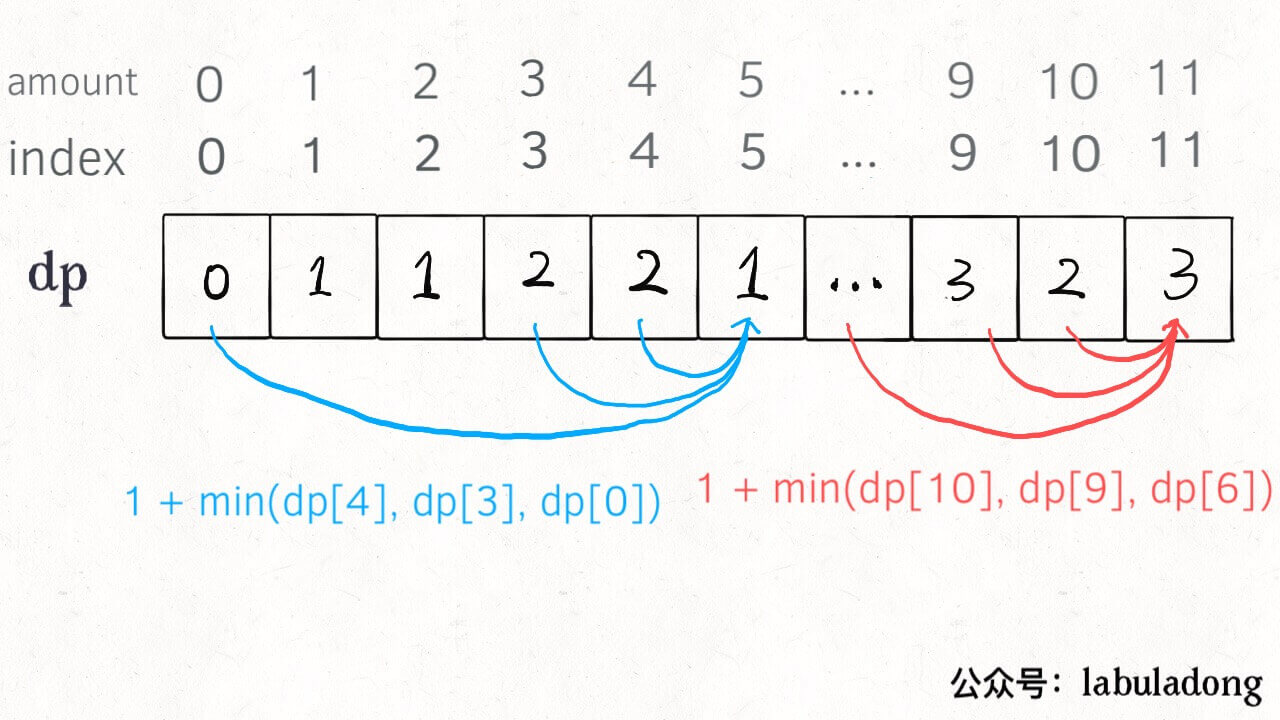
**dp[i] = min(dp[i], 1 + dp[i - coin]);**

**}**

**}**

**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。