week6

1.人肉递归低效

2.找到最近最简方法，将其拆解为可重复解决的问题

3.数学归纳法思维(n = 1, n = 2时最基本的条件想明白, n成立如何推到n + 1.)

4.画状态树

5.代码模板要在纸上动笔手写，利于人脑记忆

动态规划

动态规划 = 分治 + 最优子结构

1.动态规划和递归分治没有根本上的区别（关键看有无重复最优子结构）

2.共性：找到重复子问题

3.差异性：最优子结构，中途可以淘汰次优解。

*如果不进行淘汰，傻递归傻分治，经常是指数级时间复杂度。淘汰次优解，复杂度会变为O(n^2)或者O(n)。

动态规划关键点

1.最优子结构 opt[n] = best_of(opt[n – 1], opt[n – 2], …)

2.储存中间状态 opt[i]（面试中最重要）

3.递推公式（比较难的题DP方程不好想）

Fib: opt[i] = opt[i – 1] + opt[i – 2]

二维路径: opt[i, j] = opt[i + 1, j] + opt[i, j + 1](且判断a[i, j]是否为空地)

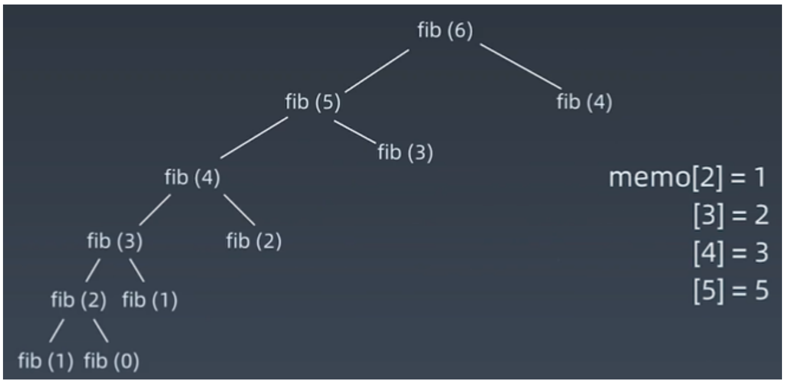
*面试中一般DP方程不是太难找，最重要的是第二步，储存中间状态，重点是要能够定义状态，且把状态定义对。

Fibonacci数列

时间复杂度为O(2^N)

```
int fib(int n) {
  return n <= 1 ? n:fib(n - 1) + fib(n - 2);
}
```

加入缓存，时间复杂度变为O(N)




```
int fib (int i, int[] memo){
  if (n <= 1) return n;
  if (memo[n] == 0) { //n还没有被计算过
    memo[n] = fib(n - 1) + fib(n - 2);
  }
  return memo[n];
}
```

自底向上的方法

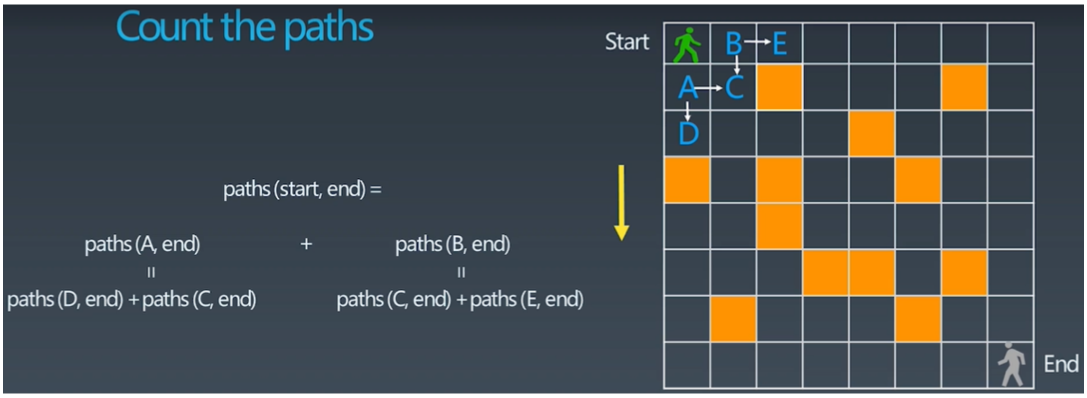
从1，2…开始递推写循环，从最下面开始。

之前的方法是自顶向下，从根到叶子。

```
a[0] = 0; a[1] = 1;
for (int i = 2; i <= n; i++) {
  a[i] = a[i - 1] + a[i - 2];
}
```

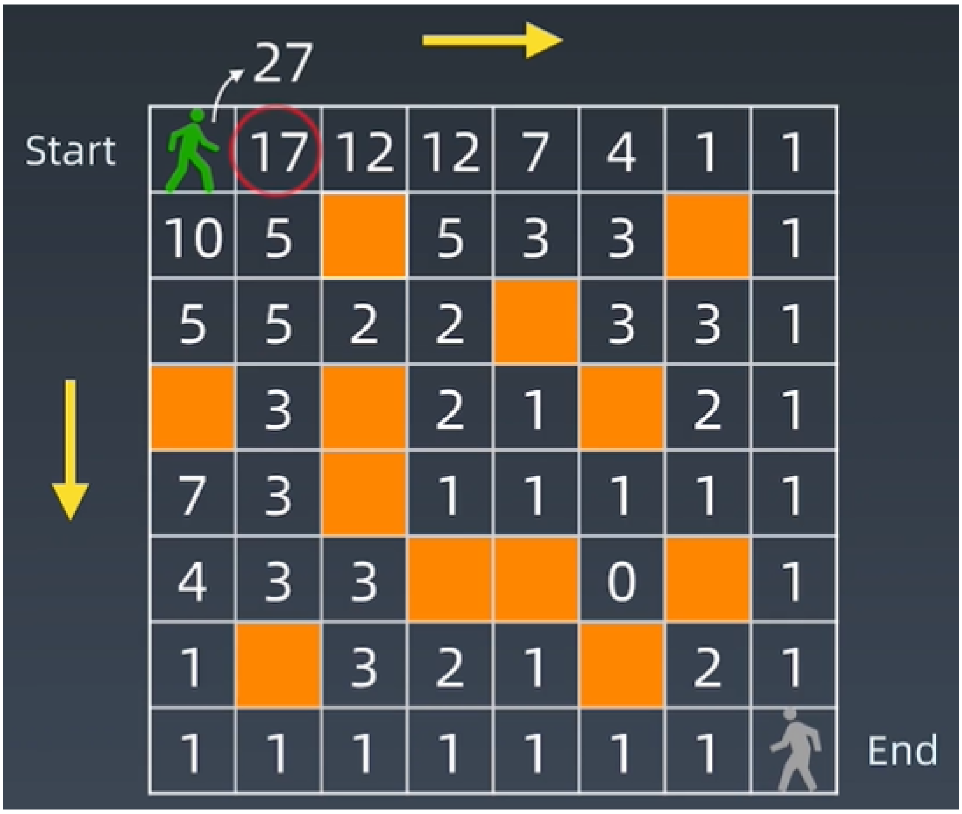
思考：
1.如果一次可以上1，2，3级台阶
2.相邻两步的步伐不能相同

路径计数




```
int countPath (boolean[][] grid, int row, int col) {
  if (!validSquare(grid, row, col)) return 0;
  if (isAtEnd(grid, row, col)) return 1;
  return countPath(grid, row + 1, col) + countPath(grid, row, col + 1);
}
```

自底向上递推



opt[i, j] = opt[i + 1, j] + opt[i, j + 1]

完整逻辑：

if a[i, j] = ’空地’:

    opt[i, j] = opt[i + 1, j] + opt[i, j + 1]

else:

    opt[i, j] = 0

```
class Solution {
    public int uniquePathsWithObstacles(int[][] obstacleGrid) {
        if (obstacleGrid == null || obstacleGrid.length == 0) {
            return 0;
        }

        // 定义 dp 数组并初始化第 1 行和第 1 列。
        int m = obstacleGrid.length, n = obstacleGrid[0].length;
        int[][] dp = new int[m][n];
        //第 1 列的格子只有从其上边格子走过去这一种走法，因此初始化 dp[i][0] 值为 1，存在障碍物时为 0;
        //第 1 行的格子只有从其左边格子走过去这一种走法，因此初始化 dp[0][j] 值为 1，存在障碍物时为 0。
        for (int i = 0; i < m && obstacleGrid[i][0] == 0; i++) {
            dp[i][0] = 1;
        }
        for (int j = 0; j < n && obstacleGrid[0][j] == 0; j++) {
            dp[0][j] = 1;
        }
        // 根据状态转移方程 dp[i][j] = dp[i - 1][j] + dp[i][j - 1] 进行递推。
        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                if (obstacleGrid[i][j] == 0) {
```

```
                    dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
                }
            }
        }
        return dp[m - 1][n - 1];
    }
}
```

自顶向下DFS+记忆化

```
class Solution {
    public int uniquePathsWithObstacles(int[][] obstacleGrid) {
        int row = obstacleGrid.length;
        int col = obstacleGrid[0].length;
        int[][] mem = new int[row][col];
        for (int i = 0; i < row; i++) {
            Arrays.fill(mem[i],-1);
        }
        return dfsUniquePathsWithObstacles(obstacleGrid,0,0,mem);
    }
    private int dfsUniquePathsWithObstacles(int[][] board, int i, int j,int[][] mem) {
        if (i == board.length - 1 && j == board[0].length - 1 && board[i][j] == 0) {
            return 1;
        }
        if (i >= board.length || j >= board[0].length || board[i][j] == 1) {
            return 0;
        }
        if (mem[i][j] != -1) {
            return mem[i][j];
        }
        int total = 0;
        total += dfsUniquePathsWithObstacles(board, i + 1, j, mem);
        total += dfsUniquePathsWithObstacles(board, i, j + 1, mem);
        mem[i][j] = total;
        return total;
    }
}
```

最长公共子序列

关键思维：何如定义动态规划问题的状态。

法一：暴力法

枚举出text1的所有子序列，看是否在text2里。

text1的所有子序列：每个字母取或不取，像括号问题

是否在text2中：对比每个字母，可以有间隔，但是顺序不能变，且只出现一次。

时间复杂度：O(2^N)

法二：找重复性

S1 = ””, S2 = 任意字符串　　--> 空

S1 = ”A”, S2 = 任意字符串　--> 看A是否包含在S2中

S1 = ”....A”, S2 = ”........A”　　--> 求S1前面字符串和S2前面字符串的子序列 + 1

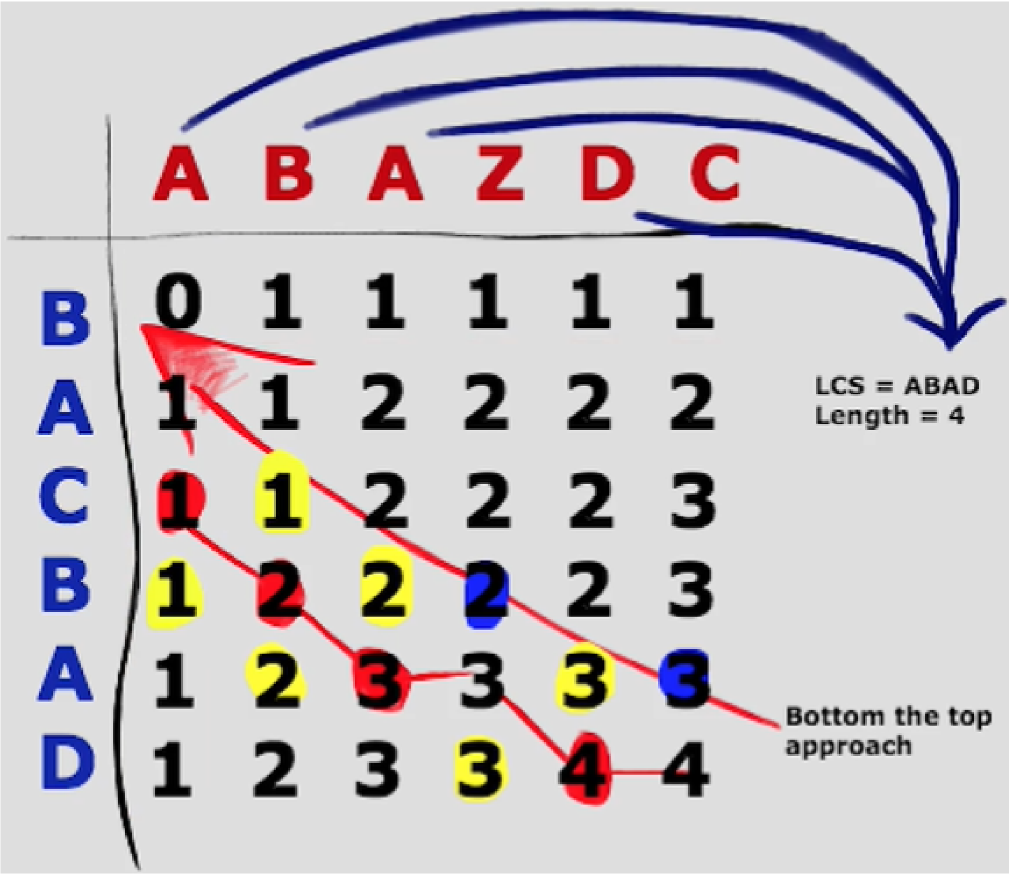
经验1：从最后一个字母开始看

经验2：最后一个字母相同时，求前面的字符串的最长子序列

经验3：两个字符串变化的问题，要变成二维数组的递推，行和列分别是两个字符串

S1[−1] != S2[−1] : LCS[S1, S2] = max(LCS[S1 − 1, S2], LCS[S1, S2 − 1])

S1[−1] == S2[−1] : LCS[S1, S2] = LCS[S1 − 1, S2 − 1] + 1




```python
class Solution:
    def longestCommonSubsequence(self, text1: str, text2: str) -> int:
        if not text1 or not text2:
            return 0
        m = len(text1)
        n = len(text2)
        dp = [[0] * (n + 1) for _ in range(m + 1)]
        for i in range(1, m + 1):
            for j in range(1, n + 1):
                if text1[i - 1] == text2[j - 1]:
                    dp[i][j] = dp[i - 1][j - 1] + 1
                else:
                    dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])
        return dp[m][n]
```

*注意下标，多写多练

*一定要用二维数组，一维数组推不出来

动态规划小结

多记，把细节全部记下来，不断反复，反复好了之后化繁为简，浓缩成一点。
1.打破思维惯性，形成机器思维（找重复性）
2.理解复杂逻辑的关键：分治，画状态树
3.也是职业进阶的要点要领

MIT动态规划课程五步法

1.define subproblems 分治，转化为子问题
2.guess (part of the solution) 猜递推方程
3.relate subproblem solutions 合并子问题的解
4.recurse & memorize or build DP table bottom up 记忆化递归/把DP的状态表建立起来，自底向上递推，两个路径都练一下
5.solve original problem

分治，找重复性和子问题
定义状态空间，可以用记忆化搜索递归/自底向上DP顺推

三角形最小路径和

法一：暴力，递归，n层：left or right 复杂度O(2^N)
法二：DP（时间复杂度n平方）
1.重复性（分治）：problem(i, j) = min(sub(i+1, j), sub(i+1, j+1)) + a[i, j]
2.定义状态数组：f(i, j)
3.DP方程：f[i, j] = min(f[i+1, j], f[i+1, j+1]) + a[i, j]
DP：

```
class Solution:
    def minimumTotal(self, triangle: List[List[int]]) -> int:
        dp = triangle #triangle值赋给dp，dp不需要初始化，方程不用加上a[i,j]
        for i in range(len(triangle) - 2, -1, -1): #i从倒数第二行开始往前，因为倒数第一行不用再计算了。
            for j in range(len(triangle[i])): #纵坐标从最左边开始
                dp[i][j] += min(dp[i + 1][j], dp[i + 1][j + 1])
        return dp[0][0]
```

记忆化递归：

```
class Solution {
    int row;
    Integer[][] memo;
    public int minimumTotal(List<List<Integer>> triangle) {
        row = triangle.size();
        memo = new Integer[row][row];
        return helper(0, 0, triangle);
    }
    private int helper(int level, int c, List<List<Integer>> triangle) {
        if (memo[level][c] != null) {
            return memo[level][c];
        }
        //terminator
```

```
        if (level == row - 1) {
            return memo[level][c] = triangle.get(level).get(c);
        }
        int left = helper(level + 1, c, triangle);
        int right = helper(level + 1, c + 1, triangle);
        return memo[level][c] = Math.min(left, right) + triangle.get(level).get(c);
    }
}
```

三角形最小路径和高票回答：https://leetcode.com/problems/triangle/discuss/38735/Python-easy-to-understand-solutions-(top-down-bottom-up)
都是DP，但是采用不同方法。

最大子序列和

（面试的时候，在白板上也要把以下这些要点罗列出来）
法一：暴力，从非负数开始到非负数，要写两个for loop嵌套，n^2
法二：DP
1.分治（子问题）max_sum(i) = Max(max_sum(i – 1), 0) + a[i]
max_sum(i)表示从后往前看(经验)， 第 i 个元素选的话，从开始到 i 最大的连续序列和是多少
max_sum(i)等于 i – 1 包含进去(前i – 1个元素的最大连续序列和)和不包含 i – 1(即前面都不加进来，从 i 这个位置开始，所以为0)的最大值，加上 i 的值。
最大子序列和 = 当前元素自身最大，或者包含之前后最大
2.状态数组定义 f[i]
f[i]表示从开始到 i 这个元素，且 i 被累加进来的最大连续序列和。
3.DP方程 f[i] = Max(f[i – 1], 0) + a[i]，找出最大的f[i]
(或者用一个sum不断进行累加，如果累加到一个地方为负了，就丢掉，不然就继续保留，直到每一步都找一个最大值)

```
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        dp = nums #把dp数组单独定义出来，不要和nums混着写
        for i in range(1, len(nums)):
            dp[i] = max(dp[i - 1], 0) + nums[i]
            #要么用之前子序列的和，要么自己单干
        return max(dp)
```

零钱兑换(重要!)

题目中，有“最少的硬币个数”字样，有最佳子结构。
若问有多少种不同组合，相当于爬楼梯问题，每次可以走1步，2步，5步，要上11级台阶有多少不同走法。（但爬楼梯1,2,1和1,1,2是不同的，兑换零钱是相同的）

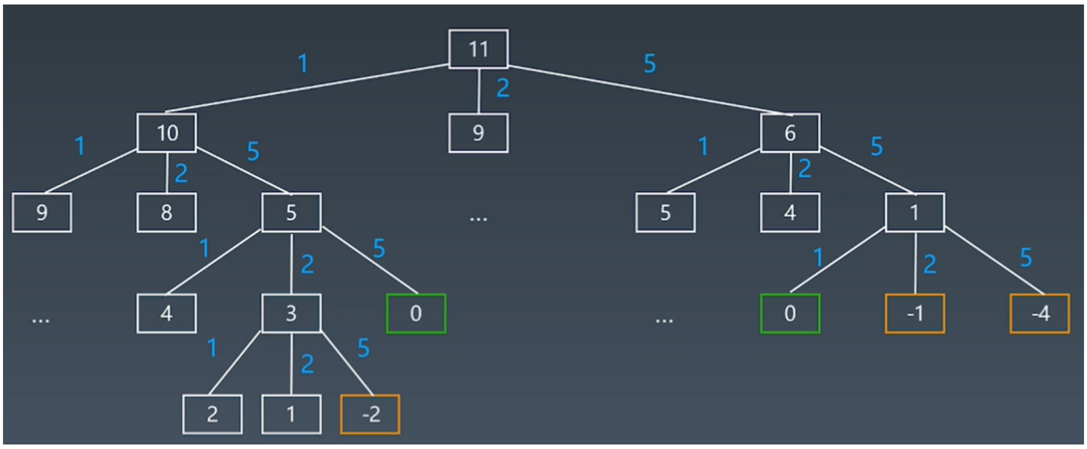
```
coins = [1, 2, 5], amount = 11
```

方法一：
暴力，递归，每次可以取1，2，5，sum减少

```java
public class Solution {
    public int coinChange(int[] coins, int amount) {
        return coinChange(0, coins, amount);
    }

    private int coinChange(int idxCoin, int[] coins, int amount) {
        if (amount == 0) return 0;
        if (idxCoin < coins.length && amount > 0) {
            int maxVal = amount / coins[idxCoin];
            int minCost = Integer.MAX_VALUE;
            for (int x = 0; x <= maxVal; x++) {
                if (amount >= x * coins[idxCoin]) {
                    int res = coinChange(idxCoin + 1, coins, amount - x *
coins[idxCoin]);
                    if (res != -1)
                    minCost = Math.min(minCost, res + x);
                }
            }
            return (minCost == Integer.MAX_VALUE)? -1: minCost;
        }
        return -1;
    }
}
```

方法二：
BFS广度优先遍历，第一次遇到为0的节点，就是所求的最少的硬币数。
节点为0为所求答案，树的深度为所用硬币数量，节点小于0停掉。
递归状态树：




```python
class Solution:
    def coinChange(self, coins: List[int], amount: int) -> int:
        from collections import deque
        queue = deque([amount])
        step = 0
        visited = set()
        while queue:
            n = len(queue)
            for _ in range(n):
                tmp = queue.pop()
```

```
                if tmp == 0:
                    return step
                for coin in coins:
                    if tmp >= coin and tmp - coin not in visited:
                        visited.add(tmp - coin)
                        queue.appendleft(tmp - coin)
            step += 1
        return -1
```

方法三：
DFS

```
class Solution:
    def coinChange(self, coins: List[int], amount: int) -> int:
        coins.sort(reverse=True)
        self.res = float("inf")

        def dfs(i, num, amount):
            if amount == 0:
                self.res = min(self.res, num)
                return
            for j in range(i, len(coins)):
                # 剩下的最大值都不够凑出来了
                if (self.res - num) * coins[j] < amount:
                    break
                if coins[j] > amount:
                    continue
                dfs(j, num + 1, amount - coins[j])

        for i in range(len(coins)):
            dfs(i, 0, amount)

        return self.res if self.res != float("inf") else -1
```

方法四：
DP
1.subproblems
2.DP array：f(n) = min(f(n – k)) + 1, for k in [1, 2, 5]
f(n)凑到面值n最少的硬币数，加1因为k也算一个硬币。
3.DP方程
自底向上法：

```
class Solution:
    def coinChange(self, coins: List[int], amount: int) -> int:
        # 自底向上
        # dp[i] 表示金额为i需要最少的硬币
        # dp[i] = min(dp[i], dp[i - coins[j]]) j所有硬币
        dp = [float("inf")] * (amount + 1)
        dp[0] = 0
        for i in range(1, amount + 1):
```

```
            dp[i] = min(dp[i - c] if i - c >= 0 else float("inf") for c in coins) + 1
        return dp[-1] if dp[-1] != float("inf") else -1
```

自顶向下法:

```
class Solution:
    def coinChange(self, coins: List[int], amount: int) -> int:
        import functools
        @functools.lru_cache(None)
        def helper(amount):
            if amount == 0:
                return 0
            return min(helper(amount - c) if amount - c >= 0 else float("inf") for c in coins) + 1
        res = helper(amount)
        return res if res != float("inf") else -1
```

打家劫舍

a[i]：0…i–1 间房子能偷到的最大值：返回a[n–1]

技巧：再开一维数组，因为不知道第 i 间房子会不会偷，用一维数组无法表示

a[i][0, 1] 0 – i不偷，1– i偷

a[i, 0] = max(a[i – 1, 0], a[i – 1, 1])

a[i, 1] = a[i – 1, 0] + nums[i]

```
class Solution {
    public int rob(int[] nums) {
        if (nums == null || nums.length == 0) return 0;

        int n = nums.length;
        int[][] a = new int[n][2];
        a[0][0] = 0;
        a[0][1] = nums[0];
        for (int i = 1; i < n; ++i) {
            a[i][0] = Math.max(a[i - 1][0], a[i - 1][1]);
            a[i][1] = a[i - 1][0] + nums[i];
        }
        return Math.max(a[n - 1][0], a[n - 1][1]);
    }
}
```

a[i]：0…i间房子，num[i]必偷的最大值