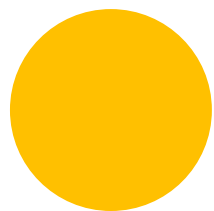
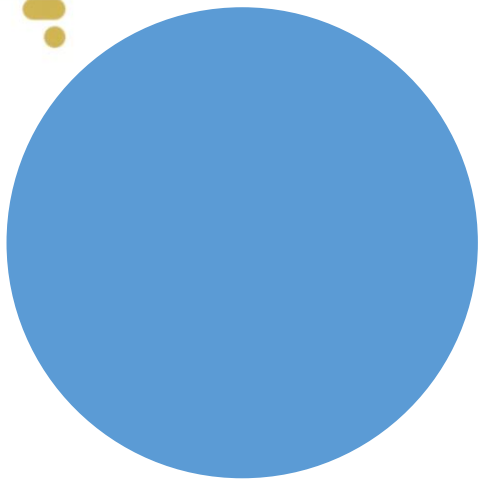




马上开始

35tang-C++竞赛系列三阶课程

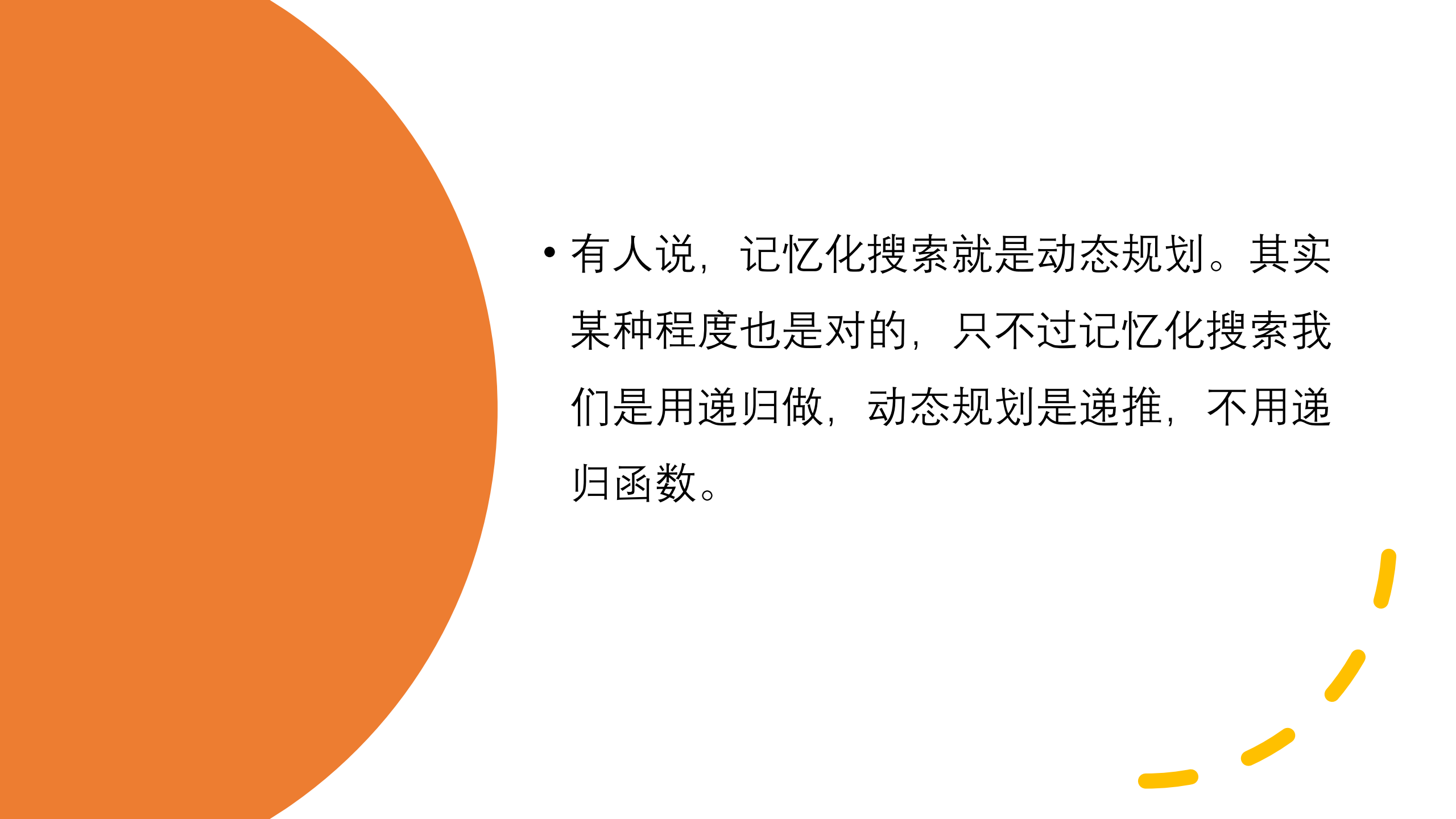


《 35tang-C++竞赛系列三阶课程 》



本节目标

- 了解动态规划算法的原理，掌握基本算法。

- 
- 有人说，记忆化搜索就是动态规划。其实某种程度也是对的，只不过记忆化搜索我们是用递归做，动态规划是递推，不用递归函数。

DP动态规划 dynamic programming



the main concept behind dynamic programming: build larger solutions based on previously found solutions. This building-up of solutions often yields programs that run very quickly.



动态规划其实来源于数学的递推

斐波拉契数列

$$F(n) = F(n-1) + F(n-2)$$

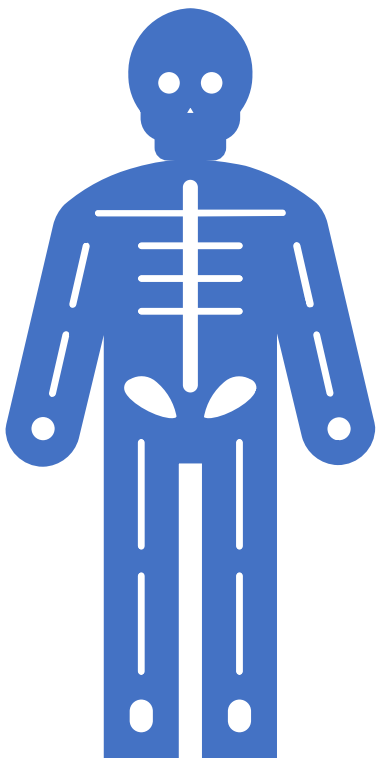
$$F(0) = 0, F(1) = 1$$

```
long long fab(int n)
{
    ans++; //用来统计fab函数执行的次数
    //边界返回  if (n==0) return 0;
    if (n==1) return 1;
    //递归 return fab(n-1) + fab(n-2);
}
```

主程序如下：

```
int n;
cin>>n;
ans=0;
cout<<fab(n)<<endl;
cout<<"fab total run times:  "<<ans<<endl;
```

- 求fab(n)和求fab(n-1)都会用到fab(n-2)，重复计算？
- 执行一下：n=10的时候，函数fab被调用了177次



记忆化搜索，把算过的值存起来

```
long long fabr[1000]={0};
long long fab1(int n)
{
    ans++;
    //边界返回

    if (n==0) return 0;
    if (n==1) return 1;

    //看看算过没有
    if (fabr[n]!=0) return fabr[n];
    //递归
    return fabr[n]= fab1(n-2)+fab1(n-1);
}
```

n=10的时候只执行了19次

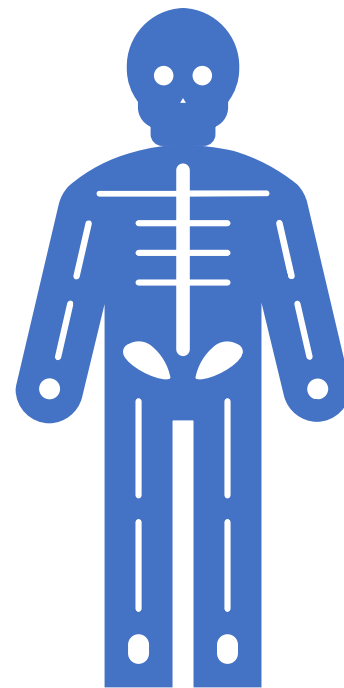
动态规划

动态规划，就是只记录状态的最优值，并用最优值来推导出其他的最优值。也就是说一步一步来做，从小的推导出大的，从前面推导出后面的。

比如这个斐波拉契数列，用动态规划的思路就是：

如果我们知道 $fab(i)$ 之前的 $fab(j), j < i$, 能不能得到 $fab(i)$ 。或者反过来说我们要知道 $fab(i)$ ，需要知道什么？

所以，最简单的思路就是 i 从0，1开始计算，把每一个 i 的值都算出来并且保存到数组（比如叫做 dp ），那么求 i 的值实际上就是求 $dp[i] = dp[i-1] + dp[i-2]$



一维的DP时间函数一般是 $O(n)$

```
• long long fabdp(int n)
{
    long long dp[1000];
    dp[0]=0;dp[1]=1;//动态规划也需要初值,就好像记忆化搜索递归的返回
    for (int i=2;i<=n;i++) dp[i]=dp[i-1]+dp[i-2];//递推每一项
    return dp[n];
}
```

//和递归一样，要有初始值：递推总要从已知的开始推

最简单一句话，动态规划就是递推！

上面第二种记忆化搜索也可以算是一种动态规划，只不过是从后往前推，某种角度也是对的。

不过我们用动态规划，一般是线性递推，一般不用递归。动态规划的关键。找出 $dp[i]$ 和 i 前面 dp 元素的关系。比如要求第 i 项的值，看看如果知道 $i-1$ 或者 $i-2...$ 的值如何推出 i 的值，这就是递推。

动态规划

- 总结递推公式
- 比如求第 i 项的值和前面哪些项有关系？什么关系？如果我们把这些项都存在一个数组里面，是不是就用循环，求出每一项的值？
- 比如如果知道 i 项的值等于 $i-1$ 和 $i-2$ 项值的和，就可以
- `for(int i=1;i<=n;i++) dp[i]=dp[i-1]+dp[i-2];`
- 初值设定，比如上面公式：如果 $i=1$ ， $dp[i-2]$ ？所以这句话的 i 其实要从2开始，或者就要加上if语句的特殊判断。数组永远不能访问-1位置。换个角度，既然是递推，从前面推出后面的，我们总要知道最开始是什么？或者说从什么开始推。

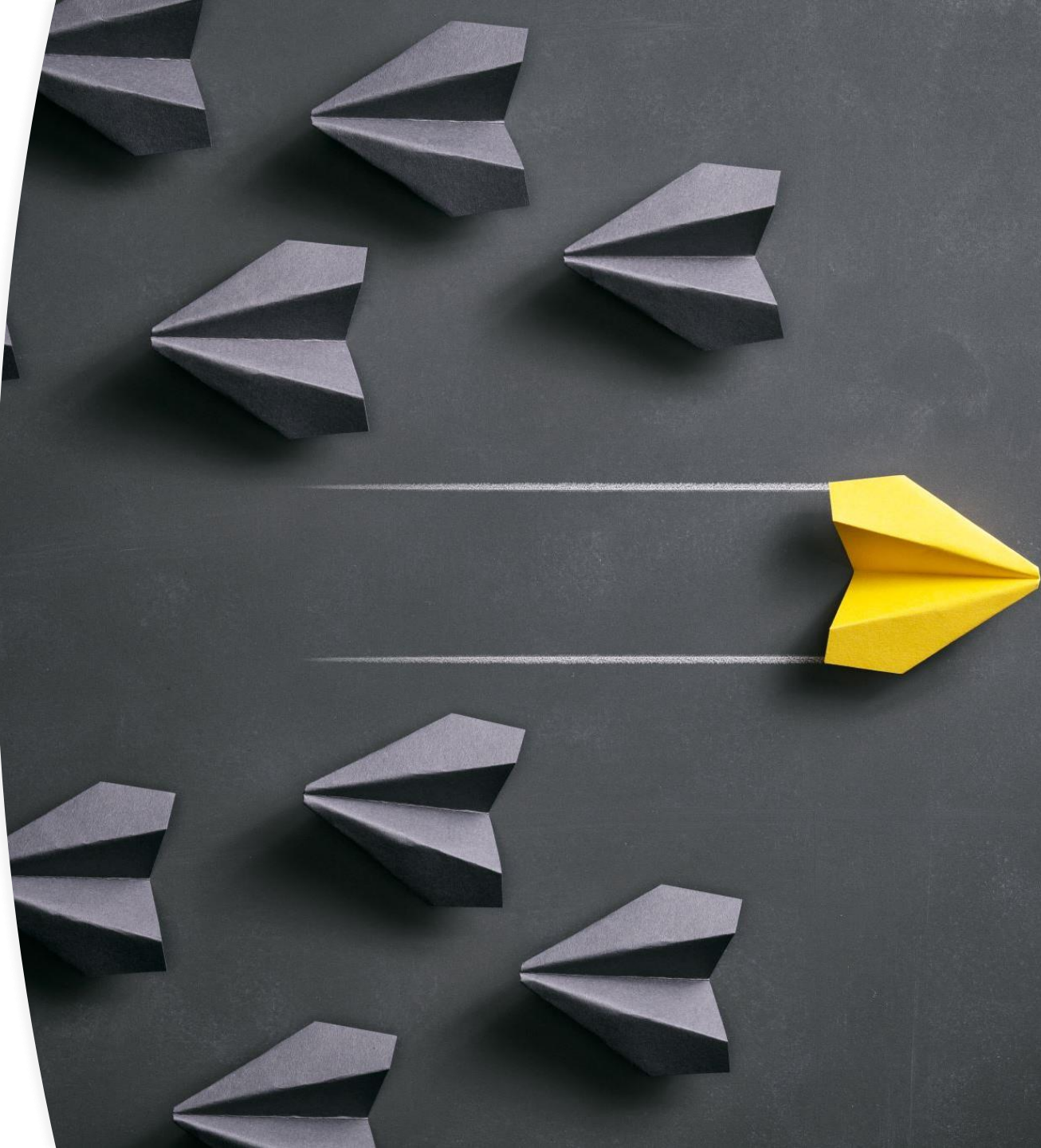
课堂练习：走楼梯

从地面开始，每次上楼梯可以上1个台阶，或者2个台阶，求走n个楼梯一共多少走法？
注意，不能往回跳

数学的概念用了加法和乘法原理，是组合的计算。

关键是递推公式， $dp[i]=dp[i-1]+dp[i-2]$

举个例子现在在楼梯5，那么他要么从前面的楼梯4跳过来，要么是从楼梯3跳过来，那么到达楼梯3的走法+到达楼梯4的走法就是到达楼梯5的走法（加法原理）。



$dp[i]=$

$dp[i-1]+dp[i-2]$

结果数字太大，会溢出，这里要求对100001取模

注意：根据余数定理，每一步都取模和最后结果取模结果是一致的。

每一步取模避免还没有算到最后结果就溢出了。

```
int main()
```

```
{
```

```
    int n;  
    cin>>n ;  
    int dp[10000];  
    dp[0]=1;dp[1]=1;//注意初值设定  
    for(int i=2;i<=n;i++)  
    {  
        dp[i]=(dp[i-1]+dp[i-2])%100001;  
    }  
    cout<<dp[n]<<endl;  
    return 0;
```

```
}
```

如果每一次可以跳
1到k个楼梯呢（注
意，不能往回跳，
往回条状态就乱了，
不能递推了）

```
int main()
{
    int n,k;
    cin>>n>>k;
    int dp[10000];
    dp[0]=1;
    for(int i=1;i<=n;i++)//递推算出每一个dp[i]
    {
        //for(int j=i-1;j>=i-k;j--)
        for(int j=i-1;j>=max(i-k,0);j--)
            //可以从前面k个楼梯跳过来
            dp[i]+=dp[j];
        dp[i]%=100001;
    }
    cout<<dp[n]<<endl;
    return 0;
}
```

dp

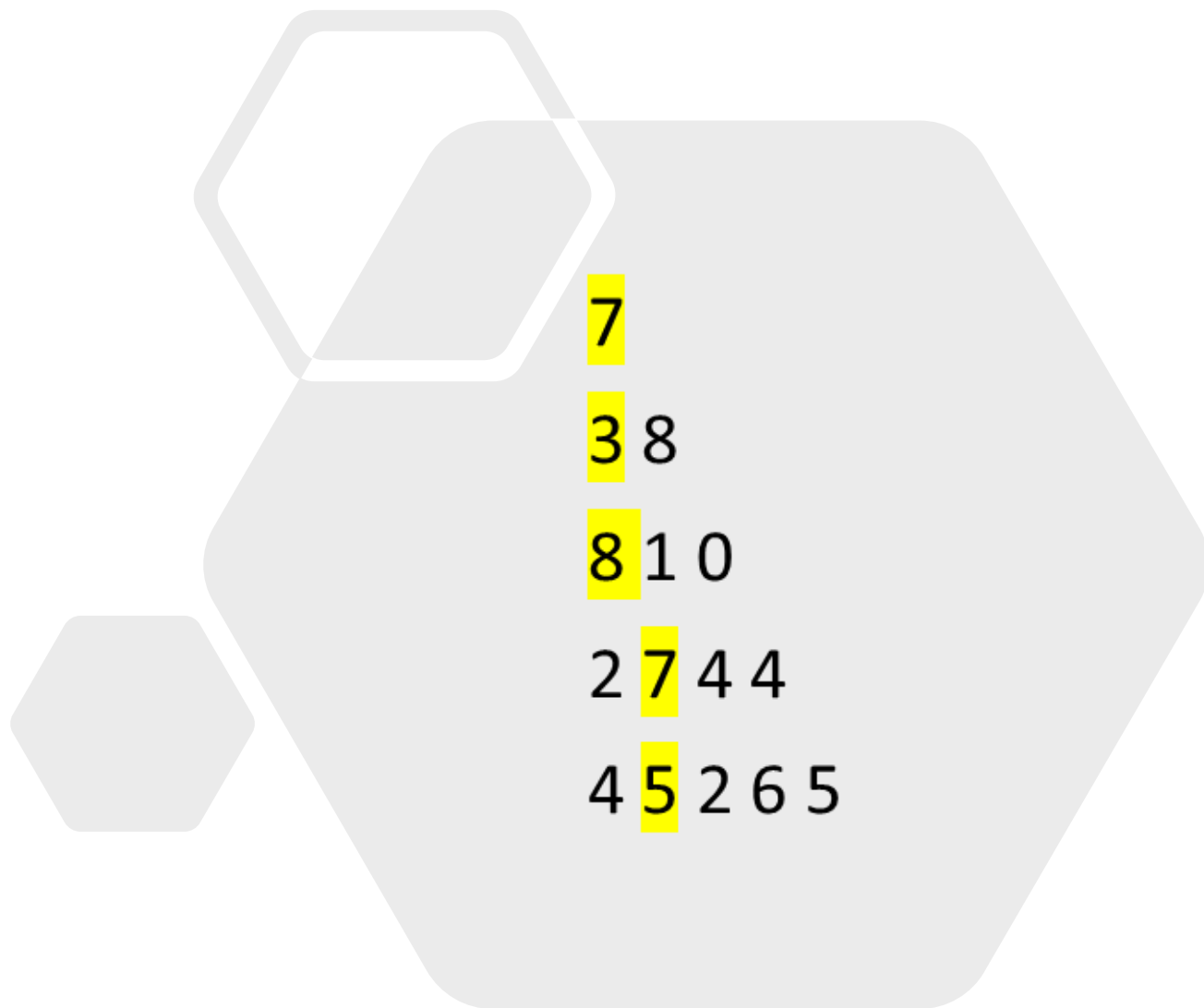
- ✓ dp数组可能是一维数组也可能是二维甚至多维数组。
- ✓ 一定要理解dp数组的定义，每一个元素（项/状态/位置）表示什么。
- ✓ 定义好了之后，找到前后元素的递推关系。
- ✓ 定义初始值。
- ✓ 然后就是循环，按照递推的顺序把dp数组的每一项都填好。

USACO training Section 1.6

PROB Number Triangles

luogu:P1216

- 数字金字塔。
- 写一个程序来查找从最高点到底部任意处结束的路径，使路径经过数字的和最大。每一步可以走到左下方的点也可以到达右下方的点。
- 先看前面DFS记忆化搜索的做法



记忆化搜索1

```
int flag[1001][1001]; //保存当前位置到最后一行的最大值。
int findr(int depth, int col)
{
    if(flag[depth][col] != -1) return flag[depth][col];
    //如果计算过最大值就不要再计算了
    if (depth == R) //到底了，打擂
    {
        return flag[depth][col] = Rarray[depth][col];
    }

    //往左下走和往右下走的最大值加上当前位置就是从当前位置走到底的最大
    return flag[depth][col] = Rarray[depth][col] +
        max(findr(depth+1, col), findr(depth+1, col+1));
}
```

从findr(1,1)开始调用或者说从最上面的位置出发。

如果我们把这个递归转换为递推呢？

flag数组现在改个名字叫做dp。把depth换成i，把col换成j。我们会发现

$$dp[i][j] = Rarray[i][j] + \max(\text{findr}(i+1, j), \text{findr}(i+1, j+1))$$

findr(i+1, j)不就是dp[i+1][j]？

findr(i+1, j+1)不就是dp[i+1][j+1]？

于是得到

$$dp[i][j] = Rarray[i][j] + \max(dp[i+1][j], dp[i+1][j+1])$$

i, j从R开始，从大到小，如果知道了dp[i+1][j]和dp[i+1][j+1]就知道了dp[i][j]。

最开始的初始化，最后一行dp[R][]的值就是Rarray[R][]的值

最后的答案是dp[1][1]

DP: 注意递推关系+初始化+不能越界

//dp[i][j]表示i行j列位置到最后一行的最大值

```
void findr1()
```

```
{
```

```
    //注意，初始值dp[R][]=Rarray[R][];
```

```
    for (int i=R-1;i>=1;i--)//倒着来，下面dp[i]需要访问dp[i+1]
```

```
    for (int j=i;j>=1;j--)
```

```
        dp[i][j]=max(dp[i+1][j],dp[i+1][j+1])+Rarray[i][j] ;
```

```
}
```

最终结果是dp[1][1]

记忆化搜索2

```
int flag[1001][1001]; //保存当前位置到(1,1)位置的最大值。
int findr(int depth, int col)
{
    if(flag[depth][col] != -1) return flag[depth][col];
    //如果计算过最大值就不要再计算了
    if (depth == 1) //到最上面了
    {
        return flag[depth][col] = Rarray[depth][col];
    }

    //往左上走和往上走的最大值加上当前位置就是从当前位置走到头的最大
    return flag[depth][col] = Rarray[depth][col] +
        max(findr(depth-1, col), findr(depth-1, col-1));
}

//main函数中打擂寻找最后一行flag[R][]的最大值。
```

如果我们把这个递归转换为递推呢？

flag数组现在改个名字叫做dp。把depth换成i，把col换成j。我们会发现

$$dp[i][j] = Rarray[i][j] + \max(\text{findr}(i-1, j), \text{findr}(i-1, j-1))$$

findr(i-1, j)不就是dp[i-1][j]？

findr(i-1, j-1)不就是dp[i-1][j-1]？

于是得到

$$dp[i][j] = Rarray[i][j] + \max(dp[i-1][j], dp[i-1][j-1])$$

倒过来i, j从1开始，从小到大，如果知道了dp[i-1][j]和dp[i-1][j-1]就知道了dp[i][j]。

最开始的初始化dp[1][1]的值就是Rarray[1][1]的值
最后的答案是最后一行也就是dp[R][]的最大值。

反过来的动态规划，从第一行开始往下推。

最后的答案是最后一行也就是 $dp[R][i]$ 的最大值。

// $dp[i][j]$ 表示i行j列位置到1行1列的最大值

```
void findr1()
{
    //注意，初始值 $dp[0][0]=0$ ;
    for (int i=1;i<=R;i++)
    {
        for (int j=1;j<=i;j++)
             $dp[i][j]=\max(dp[i-1][j-1],dp[i-1][\min(j,i-1)])+Rarray[i][j]$  ;
        //每一行的最大值是上面一行左面和右面的两个
    }
}
```

main函数内：

```
findr1();
sumr=0;
for (int i=1;i<=R;i++) if ( $dp[R][i]>sumr$ )  $sumr=dp[R][i]$ ;
cout<<sumr<<endl;
```

```
dp[i][j]=max(dp[i-1][j-1],dp[i-1][j])+Rarray[i][j] ;
```

[illegible]

优化减去dp数组一个维度:

外循环做到i的时候, $dp[i-1][j]$ 的值上一次i-1的循环中都已经算出来了。如果dp1[]中表示当前i的所有的j的值, 也就是或dp1[j]就是 $dp[i][j]$ 。而 $dp[i-1][j]$ 就是上一次i-1循环的时候的dp1[j]。

```
for (int i=1;i<=R;i++){  
    for (int j=1;j<=i;j++)  
        dp[i][j]=max(dp[i-1][j-1],dp[i-1][j])+Rarray[i][j] ;
```

/*//可以优化减去一个维度

```
for (int i=1;i<=R;i++){
```

```
    for (int j=i;j>=1;j--)//注意, 循环倒过来
```

```
        dp1[j]=max(dp1[j-1],dp1[j])+Rarray[i][j] ;
```

```
    */
```

```
}
```

```
dp[i][j]=max(dp[i-1][j-1],dp[i-1][j])+Rarray[i][j] ;  
dp1[j] =max(dp1[j-1], dp1[j])+ Rarray[i][j] ;
```

为什么j循环要倒过来?

假设现在i是3, j是2.

```
dp1[2] =max(dp1[1], dp1[2])+ Rarray[3][2] ;
```

等价于二维的:

```
dp[3][2] =max(dp[2][1], dp[2][2])+ Rarray[3][2] ;
```

这里=右面的dp1[1],dp1[2]应该是i是2的时候循环算出来的值, 也就是dp[2][1]和dp[2][2]。但是如果j正着循环, 在计算j是2的时候显然j是1的已经计算过了, 那么dp1[1]中就是i是3的时候的值了, 就是dp[3][1]了, 就不对了。

矩阵路径统计

- 有一个类似迷宫的 $m \times n$ 的矩阵结构，机器人从左上角出发需要到达右下角，给出一些障碍（就是不能走的位置），机器人每一步只能朝右或者朝下走一个格子，一共有多少条不同的路径可以到达。

- 输入：

- 第一行3个整数 m, n, q 表示矩阵 m 行 n 列，一共 q 个障碍。后面 q 行，每一行给出两个整数，分别表示一个障碍的行列位置。（ m, n, q 都小于100大于等于1）

- 输出：

- 一个整数，表示答案。

- 示例输入

3 3 1

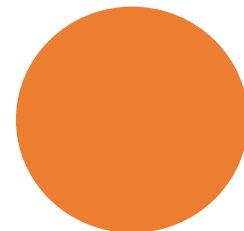
2 2

- 示例输出

2

示例说明，如图，障碍位于2行2列。

S		
	X	
		E



搜索？

- 第一个思路：搜索回溯（每一个位置可能出现在不同的路径，所以需要回溯），由于找所有的路线，所以可以用计数器，每条路都要走到头然后计数器++。但是每一个位置往下2个可能，所以一共 100×100 个位置，就是2的10000次方，肯定爆了。

动态规划

- 或者说记忆化搜索的变形，在上面搜索中我们会发现，如果从后往前看，那么走到某一个位置之后，后面到终点的路径总数实际上是一样的，我们的dfs很多路线的大部分都是重复的。所以可以用记忆化搜索，搞个二维数组，把每一个位置后面有多少不同路径记录下来，那么下一个路径走到这个位置之后就不需要再走了。
- 不用递归，直接递推，就是动态规划。
- 为什么可以用动态规划？就是有递推关系，后一个状态和前一个状态相关，或者说可以从前一个状态推导出来。到达某一个位置的不同路径总数就是到达左面和上面的位置的路径数的和。如果我们定义 $dp[i][j]$ 表示机器人到达 i 行 j 列时候的不同的路径总数，那么如果 i 行 j 列是障碍物，那么 $dp[i][j]=0$ ；否则 $dp[i][j]=dp[i-1][j]+dp[i][j-1]$ ；

可以用动态规划或者记忆化搜索的矩阵

关键：不能往回走：只能往右或者往下。这样，我们的每一个位置的状态就只依赖于他前面的左面或者上面的位置，而这两个位置一定是走过的（前面递推出来的）。

如果矩阵没有限制只能往右或者往下，没有限制这种递增的关系，就不能直接使用记忆化搜索或者动态规划了。这样就是循环依赖的关系了。

- 处理矩阵问题的时候要注意两点：
 1. 一般来讲，题目给出的障碍位置都是从第一行算起的，比如例子里面的2 2位置就是第2行第2列，机器人最开始位于1行1列。而数组缺省都是从0开始放的，所以你要决定你的数组存储矩阵的时候从0开始还是从1开始？如果从0开始放，那么 输入的2 2 位置就是你数组的1 1 位置了。
 2. 一定要小心， $i-1$ ， $j-1$ 会不会变成-1？假设行列都是从0开始的，显然边界不能比0小。

```
int main() {  
    int m,n,q;  
    cin>>m>>n>>q;  
    vector<vector<int> > dp (m, vector<int>(n,0)); //这里也可以用 int dp[101][101], obstacleGrid[101][101];  
    vector<vector<int> > obstacleGrid (m, vector<int>(n,0)); //动态数组初始化为m*n的大小  
    int a,b;  
    for (int i=1;i<=q;i++){ //读入障碍矩阵  
        cin>>a>>b;  
        obstacleGrid[a-1][b-1]=1;//注意， 我们的矩阵从0行0列开始放  
    }  
    if (obstacleGrid[0][0]==0) dp[0][0]=1;  
    for (int i=0;i<m;i++)  
        for (int j=0;j<n;j++){  
            if(obstacleGrid[i][j]==1) dp[i][j]=0;//障碍  
            else { //注意下面的边界处理  
                if(i!=0) dp[i][j]+=dp[i-1][j];  
                if(j!=0) dp[i][j]+=dp[i][j-1];  
            }  
        }  
    }  
    cout<<dp[m-1][n-1]<<endl;  
    return 0;  
}
```

学到了什么？



搜索---递推---递归---记忆化搜索—动态规划



动态规划，就是只记录状态的最优值，并用最优值来推导出其他的最优值。也就是说一步一步来做，从小的推导出大的，从前面推导出后面的。



某种角度说，动态规划就是“倒过来”的记忆化搜索

作业1

- 还记得前面讲过的最大连续子序列和的问题么？复习一下。用动态规划实现。

求最大子序列和

给定整数 a_1, \dots, a_n (可能有负数), 求最大的连续子序列和。

输入, 第一行为整数 n ($n \leq 10^5$), 表示整数的个数, 第二行 n 个整数 a_1, \dots, a_n 空格分开

输出, 一个整数, 为最大连续子序列和

示例输入

6

-2 11 -4 20 -3 -9

示例输出

27

示例输入

6

-2 -9 -5 -1 -7 -9

示例输出

-1

提示: 设 $dp[]$ 数组, $dp[i]$ 表示前 i 项以 $a[i]$ 为结尾的最大连续子序列和, 那么 $dp[i]$ 和 $dp[i-1]$ 有什么关系? 最后打擂台去 dp 数组最大。

作业2：挑战： 矩阵最小和 路径

给出一个 $m \times n$ 的数字矩阵，其中每一个数字都小于100的正整数， $1 \leq m$ ， $n \leq 100$ 。求从左上角位置到右下角位置的所有路径中和最小是多少。所谓的“走”就是一个位置只能移动到他的右面或者下面位置。

输入格式：

第一行两个整数 m 和 n ，接下来 m 行，每一行 n 个整数。表示矩阵。

输出格式：

一行一个，表示最小和。

示例输入：

3 3

1 3 1

1 5 1

4 2 1

示例输出：

7

1	3	1
1	5	1
4	2	1

示例说明：矩阵如图，显然按照黄色部分路径走的话经过的位置的所有数字加起来和最小。

作业2提示

- 显然dp是二维数组，因为每一个位置不同，值不同。如果不理解也可以先写出记忆化搜索来。
- 如何定义dp[i][j]的含义？如果矩阵从0行0列开始，dp[m-1][n-1]是我们要找的答案。那么dp[i][j]的含义？i行j列走到当前位置的路径的最小和。dp[i][j]如何根据前面的两个位置dp[i-1][j]和dp[i][j-1]推出来？
- 关键，递推的时候肯定是i从小到大，j也从小到大，但是，注意需要特殊处理不能让i-1或者j-1变成负的。

由易到难，思维体系训练
实战结合，创新协作培养
兴趣导向，未来职业引领

<https://www.35tang.com>



扫码关注公众号

<https://www.三五堂.com>



添加辅导老师