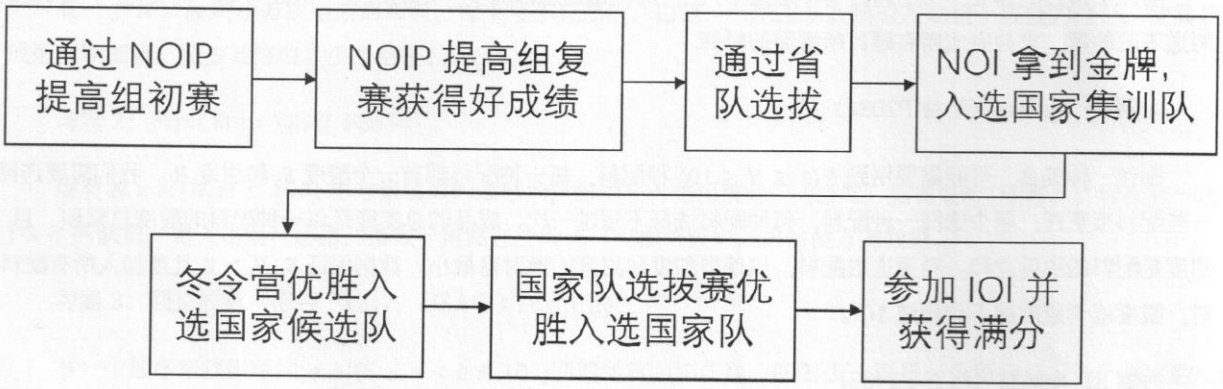
第四章 递推与递归

有些目标是宏大的，比如要在 IOI 赛场中得到满分（俗称 AK IOI）。如果你现在还是一个普通的学生，那么想达成这个目标太难了。但把这样宏大的目标分解为很多个子任务，就没觉得那么复杂了。要想 AK IOI，只需要入选国家队，参加 IOI 即可。那怎么成为入选国家队呢？参加中国队选拔赛并通过面试答辩即可。使用同样的思路往前倒推，直到最后只剩下最基础的任务（比如认真的读完这章内容并完成练习），做完这样的小任务就很简单了。



【图：如何 AK IOI[22]】

像这样将一个很大的任务分解成规模小一些的子任务，子任务分成更小的子任务，直到遇到初始条件，最后整理归纳解决大任务的思想就是递推与递归思想，不过这两者还是有一些区别。这一章涉及的内容是动态规划思想与分治策略的基础，大家也要认真学习啦，说不定目标就真的达到了。

第1节 递推思想

例 1：数楼梯（洛谷 P1255）

楼梯有 $N(N \leq 5000)$ 阶，上楼可以一步上一阶，也可以一步上二阶，计算共有多少种不同的走法。

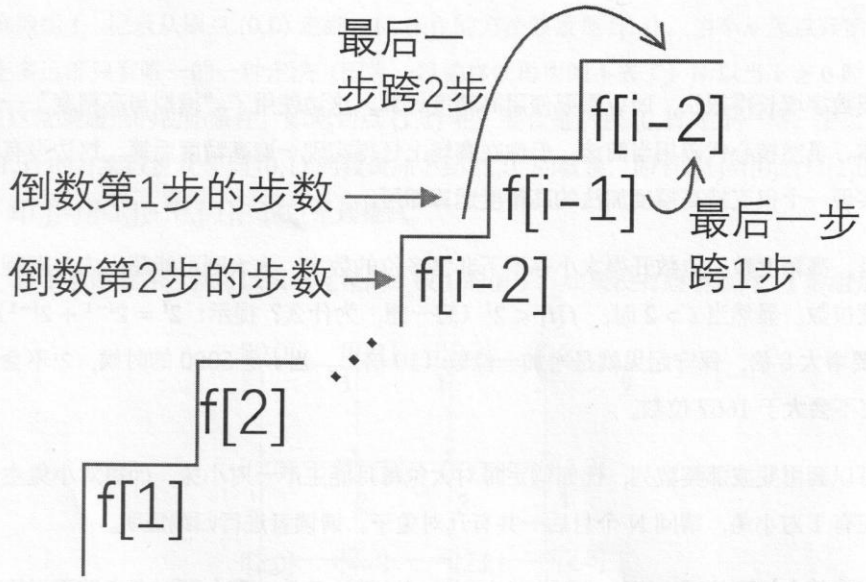
分析：假设想求 1000 个台阶的走法数量，这看起来是个非常宏伟的目标！最简单暴力的思路就是使用回溯法来枚举所有走法，但是速度非常慢，如果数据范围稍大就会超时。

如果想要走到第 1000 个台阶时，必须要先走到第 998 个台阶或者 999 个台阶，然后一步跨到第 1000 级，所以到第 1000 个台阶的走法数量就是从头到第 998 级的走法数量与从头到第 999 级的走法数量之和，这么想就简单多了。不过还得先知道走到第 998 级和 999 级的走法数量是多少。

---

[22] 当然这只是一个简化版的流程，实际情况下需要的灵感和艰辛不是一两句话就可以说清楚的。

假设从头走到第 i 个台阶的走法数量是 f[i]，根据上面的分析可以得到 f[1000]=f[998]+f[999]。同理 f[999]=f[997]+f[998]……以此类推，可以归纳得到 f[i]=f[i-2]+f[i-1]。



【图：最后一步的走法】

有些读者看到这个可能会感觉有些眼熟。我们在语言入门的循环一章有介绍过这样的东西：后一项等于前面两项之和。这不就是斐波那契数列吗?

即使归纳得到了这个式子（称为递推式），这还不能说明这是一个斐波那契数列。因为根据这个递推式往前追溯，总得有个尽头（称为初始条件）。观察这样一个数列：[1 3 4 7 11 18 29 47 …]，虽然从第三项开始每一项都符合这个递推式，但是这个数列的初始条件和斐波那契数列不一样，所以这个数列和斐波那契数列相差甚远了。

计算数列中的某个元素只需要得到它前面的两项元素就行。如果知道 f[1] 和 f[2] 的值，就可以推导出 f[3]=f[1]+f[2]，继而得到 f[4]=f[2]+f[3]……一直可以计算得到 f[1000]=f[998]+f[999]。获得 f[1] 和 f[2] 的值，作为初始条件就至关重要。

那对于这个问题而言初始条件是什么呢？可以直接进行分析。当只有一个台阶时，跨一步就可以上去了，这是唯一的一种走法；而有 2 个台阶时，可以走两次一步，或者走一次两步，一共两种走法。所以可以确定 f[1]=1，f[2]=2。

现在有了递推式，有了初始条件，就可以获得完整的数列了。经过计算，可以得到这个数列前面几项是 [1 2 3 5 8 13 21 …]，这就是斐波那契数列从第 2 项开始的序列。请注意斐波那契的初始条件是 f[1]=f[2]=1。

像这样知道递推式，也知道初始条件，从初始条件开始往上顺推直到求得目标解的思想就是**递推**。核心代码如下：

```
int main() {
    cin >> N;
    Bigint f[5010];
    f[1] = Bigint(1);
```

```
    f[2] = Bigint(2);
    for (int i = 3; i <= N; i++)
        f[i] = f[i - 2] + f[i - 1];
    f.print();
    return 0;
}
```

由于斐波那契数字增长得很快，所以需要使用高精度计算。这边使用了“模拟与高精度”一章中提供的封装好的大整数结构体。虽然核心代码相当简洁，但你在赛场上还得实现一遍高精度运算。这边没有涉及到高精度乘法所以是只需要实现一个仅支持高精度加法的高精度运算即可。

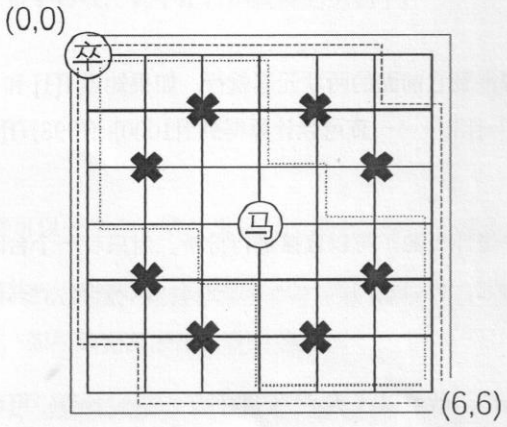
需要注意的是，高精度数组位数开得太小存不下非常多位的数字，太大则可能超出内存限制，所以大概估计一下合理的高精度位数。显然当 $i > 2$ 时，$f[i] < 2^i$（想一想，为什么？提示：$2^i = 2^{i-1} + 2^{i-1}$）。对于2^i来说，i 每增加 3，2^i就要增大 8 倍，保守起见就是增加一位数（10 倍）。当 i 是 5000 的时候，2^i不会超过 5000/3 位数，因此 f[i] 肯定不会大于 1667 位数。

许多问题也可以套用斐波那契数列，比如假定每对大兔每月能生产一对小兔，而每对小兔生长为大兔也需要一个月，已知现在有 1 对小兔，请问 N 个月后一共有几对兔子。请读者进行归纳证明。

当我们能就某个问题能写出递推式、能确定初始（边界）条件，那么可以考虑使用递推。对于某些数据规模很大的递推任务可以使用矩阵加速提升效率，感兴趣的同学可以自行查阅相关资料。

例 2：过河卒（洛谷 P1002，NOIP2002 普及组）

棋盘上左上方 A 点(0,0) 有一个过河卒，需要走到右下角的目标 B 点(n,m)。卒可以向下或者向右一格。在棋盘上 C 点有一个固定的对手的马，该马所在的点和所有跳跃一步可达的点（直走一格再往马点的远处斜走一格）称为对方马的控制点。因此称之为“马拦过河卒”。求卒从起点到终点所有的路径条数。



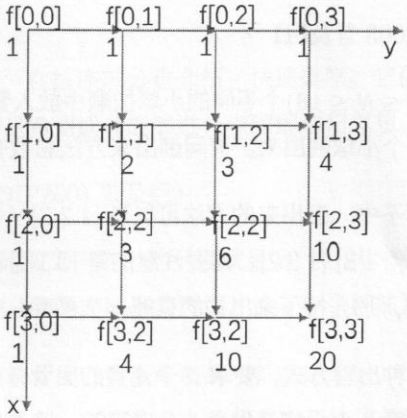
【图：马拦过河卒中的一个例子】

如图，当卒要从起点 (0,0) 往右或下走到终点 (6,6)，而马在 (3,3)。马能跳到的位置已经打上了叉，卒不能走到这些点。在这种情况下，一共有 6 种合法方案。

分析：思路最简单的方式还是枚举往右或往下然后回溯搜索，但是依然会超时。先考虑一个简化版的问题：如果那个马不存在，从左上角到右下角一共有多少种走法？

开立一个二维数组 f，记录从原点 (0,0) 走到坐标 (i,j) 的方法数量是 f[i,j]。当卒从起点开始，笔直往右或者笔直往下，无论走多远都只有唯一的一种走法（因为一旦偏移就再也回不去了）所以当 $k \geq 0$ 时，f[k,0]=f[0,k]=1，这就是递推的初始条件。如何到点 (1,1) 呢？要么是从点 (0,1) 走下一格，要么是从点 (1,0) 往右走一格，因此到(1,1) 的方案数量就是到 (0,1) 的数量加上到 (1,0) 的数量，即 f[1,1]=f[0,1]+f[1,0]。可以归纳得到当 i>0 且 j>0 时，f[i,j]=f[i-1,j]+f[i,j-1]，这就是递推式。

有了递推式，有了初始条件，就可以求出完整的 f 数组的值了。如果没有碍事的马，f 数组是这样的：



【图：二维数组递推】

如果有些点因为马的把守而不能走呢？其实也没有什么区别，只不过没办法从马的控制点转移到下一个点罢了（换句话说，马的控制点上路径数全部清空）。此外初始条件和递推范围也有一点变化，只需要 f[0,0]=1 即可，同时递推范围就是 $i \geq 0, j \geq 0, ij \neq 0$（想一想，为什么）。代码如下：

```
#include <iostream>
#define MAXN 22
using namespace std;
long long f[MAXN][MAXN] = {0};
int ctrl[MAXN][MAXN], n, m, hx, hy;
int d[9][2] = {{0, 0}, {1, 2}, {1, -2}, {-1, 2}, {-1, -2},
               {2, 1}, {2, -1}, {-2, 1}, {-2, -1}}; // 马的控制范围相对于马位置的偏移量

int main() {
    cin >> n >> m >> hx >> hy;
    for (int i = 0; i < 9; i++) {
        int tmpx = hx + d[i][0], tmpy = hy + d[i][1];
        if (tmpx >= 0 && tmpx <= n && tmpy >= 0 && tmpy <= m) // 判断在棋盘范围内
            ctrl[tmpx][tmpy] = 1; // 记录马的控制点
    }
    f[0][0] = 1 - ctrl[0][0]; // 如果原点就是马控制点，那初始路径数量就是 0，否则是 1
    for (int i = 0; i <= n; i++)
        for (int j = 0; j <= m; j++) {
```

```
            if (ctrl[i][j])continue; // 如果这个点是控制点，那么跳过
            if (i != 0) f[i][j] += f[i - 1][j]; // 也可写成 if(i)，若不在横轴上就加
上面路径数
            if (j != 0) f[i][j] += f[i][j - 1]; // 该点不在纵轴上就加左边的路径数
        }
    cout << f[n][m]; // 输出答案
    return 0;
}
```

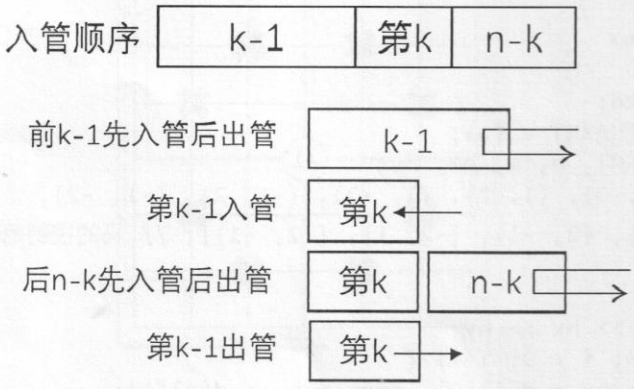
在实现时，需要预处理一下并记录哪些点是马的控制点，然后对所有的点进行递推操作。只有上面或者左边的格点存在，才会累加上面或者左边的方案数。

例 3：栈（洛谷 P1044，NOIP2003 普及组）

有一个单端封闭的管子，将 $N(1 \le N \le 18)$个不同的小球按顺序放入管子的一端。在将小球放入管子的过程中也可以将管子最顶上的一个或者多个小球倒出来。请问倒出来方法总数有多少种?

比如将小球 [1 2 3]依次加入到管子中，倒出来的方法可以是 [1 2 3]（每倒入一个球后立刻拿出来）[3 2 1]（全部倒入球然后依次取出）[2 3 1] [2 1 3] [1 3 2]。需要注意的是 [3 1 2] 是不行的，因为在加入 3 之前，管子里面已经有 1 和 2 了，如果 3 最先出去，那么接下来出去的只能是 2，而 1 被压在最底下。

分析：假设 i 个元素一共有 h[i] 种出管方式。要求 n 个元素的出管方式，但是其中每一个元素（从 1 到 n）都可能可以是最后一个出管的。假设第 k 个小球是最后一个出管的，比 k 早入管且早出管有 k-1 个数，一共有 h[k-1] 种出管方式；比 k 晚入管且早出管有 n-k 个数，一共有 h[n-k] 种出管方式。这种情况下一共就有 h[k-1]×h[n-k] 种出管方式。当 k 取不同值的时候，产生的出管序列也是独立的，所以可以加起来。k 的取值范围可以是 从 1 到 n。所以递推式是 $h(n) = h(0)\times h(n-1) + h(1)\times h(n-2) + \cdots + h(n-1)\times h(0)$，初始条件是 h[0]=h[1]=1。



【图：第 k 个小球的出管方式】

代码非常短小：

```
#include<cstdio>
int main() {
    int n, h[20] = {1, 1};
```

```
    scanf("%d", &n);
    for (int i = 2; i <= n; i++)
        for (int j = 0; j < i; j++)
            h[i] += h[j] * h[i - j - 1];
    printf("%d", h[n]);
    return 0;
}
```

像这种只有一个开口、元素先进后出的管子称为栈，在数据结构线性表一章我们会更详细地介绍栈的性质使用方式。而 h 数组里面的数字就是卡特兰数，前几项是 1,1,2,5,14,42。卡特兰数有很多奇妙的性质，会在《提高篇》中仔细讨论。

第2节 递归思想

之前已经在语言部分介绍过了递归，在排序部分也介绍了快速排序。它们都使用到了递归。这一节不会详细介绍语言层面的递归程序执行机理，而是希望读者能够进一步理解递归思想。

例 4：数的计算（洛谷 P1028，NOIP2001 普及组）

给出自然数 $n(n \leq 1000)$，最开始时数列中唯一的一项就是 n，可以对这个数列进行下面的操作，生成新的数列。请问最后能生成几种不同的数列？（原题描述不太严谨，换了一个问法）

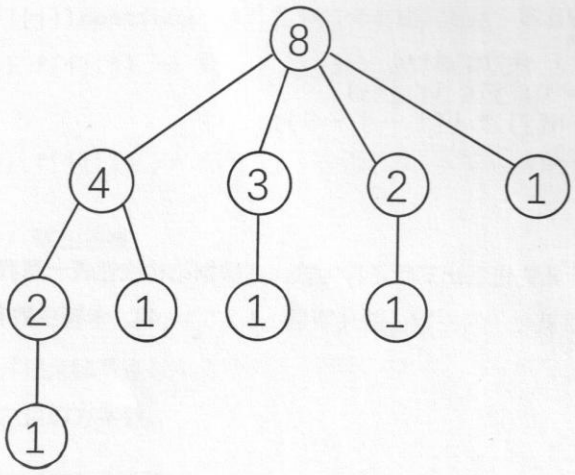
1. 原数列不作任何处理就直接统计为一种合法的数列；
2. 在原数列的末端加入一个自然数，但是这个自然数不能超过该数列最后一个数字的一半；
3. 加入自然数后的新数列，继续按此规则从第一条进行处理，直到不能再加新元素为止。

比如说，输入数字 6，符合这样性质的数列有 [6]、[6 1]、[6 2]、[6 2 1]、[6 3]、[6 3 1]。

分析：对于一个整数 n，如果只考虑前面两点，那么问题就很简单了——答案就是 n/2+1。但是这还没完，题目要求新的数列还要按照同样的规则进行处理，那该怎么办呢？

比如说，数列中最开始只有一个元素 8，在末尾加入一个新元素，列表就可以变成 [8 4]、[8 3]、[8 2]、[8 1]，算上 [8] 一共有 5 种情况。那之后还需要计算更长的数列的方案数怎么办呢？这也很好计算：只需要按照上面这种方法，分别计算 [4]、[3]、[2]、[1] 按照这样的操作能有几种情况，然后累加统计即可。毕竟以 4 开头（3、2、1 同理）的所有合法数列，都可以接续到 8 的后面。

原来是要解决 n=8 的问题，现在分解成了四个规模更小（n=4、3、2、1）但是本质上是同样的子问题；如果要解决 n=4 的问题，基于同样的思想还可以分解成两个规模更小的（n=2、1）但本质相同的子问题；当需要解决 n=2 的问题时，可以分解成 n=1 的问题（只有 n=1 的情况了）；直到 n=1 时，没法继续分解，根据题目说的“不作任何处理就直接统计为一种合法的数列”，可以直接返回只有唯一一种数列，即 [1]。然后返回上一层（n=2）接收到所有小规模问题的答案，合并统计处理获得这个规模下的答案，再继续返回上一层（n=4）……直到求得问题的解。



【图：问题规模的分解】

像这样将一个规模较大的问题，分解成结构相同但是规模更小的问题，继续求解，直到问题不能分解为止，然后规模较大的问题将规模更小问题的答案进行整合得到最终答案的思想就是**递归思想**。

于是可以写出这样的程函数，其中 sol(x)是询问规模为 x 时的答案。

```
int sol(int x) {
    if (x == 1)
        return 1;
    int ans = 1;
    for (int i = 1; i <= x / 2; i++)
        ans += sol(i);
    return ans;
}
```

这样实现函数并没有什么错误，但是并不能通过本题：运行效率很低导致程序超时。这是因为做了很多无效功，比如说 sol(2) 可能由 sol(4) 调用，也有可能被 sol(8) 调用，但是 sol(2) 本身的值是固定不变的，在这里却被重复运行了很多次造成了浪费。

为了防止这种情况，可以开一个数组 f，其每一项 f[i] 就是当问题规模为 i 的时候的答案。首先将数组初始化为 -1[23]，说明 f[i] 还没有被计算过。依然使用同样的办法去求解，只是如果发现已经计算过就直接返回 f[i] 而不必进行接下来的计算了，否则还是按照刚才递归的方式计算，然后将结果存入数组中以便之后再次调用。改进后的代码如下：

```
#include<iostream>
#include<cstring>
using namespace std;
int n, f[1010];
int sol(int x) {
    int ans = 1;
    if (f[x] != -1)
        return f[x];
```

[23] 这里使用 memset 进行初始化，请注意要加 cstring 头文件，而且一般只用于初始化-1 或者 0。

```
    for (int i = 1; i <= x / 2; i++)
        ans += sol(i);
    return f[x] = ans;
}
int main() {
    cin >> n;
    memset(f, -1, sizeof(f));
    f[1] = 1;
    cout << sol(n) << endl;
    return 0;
}
```

这样的话每个数字最多只会只计算一次，因为一旦计算完成就会被存下来，便于日后使用。这样的思想被称为“记忆化搜索”。

有的同学发现本题可以写出递推式 f[i] = 1 + f[1] + f[2] .. + f[i/2]，f[1] = 1。有递推式，也有初始条件，那能否使用上一节介绍的递推求解？完全可以！递推和递归思想往往可以相互转换，请感兴趣的读者尝试使用递推求解本题。

有的情况下进行递推，需要求出初始条件，还需要确定递推顺序（尤其是多维递推时更加复杂），所以这时使用递归思想会容易一些。此外如果本题进行递推的话，需要计算出一些无效状态（比如 n=5、6、7），而递归可以规避这些无效状态以提升计算效率。

例 5：Function（洛谷 P1464）

对于一个递归函数 w(a,b,c)：

1. 如果 $a \le 0$ 或 $b \le 0$ 或 $c \le 0$ 就返回值 1；
2. 如果 $a > 20$ 或 $b > 20$ 或 $c > 20$ 就返回 w(20,20,20)；
3. 如果 $a < b$ 并且 $b < c$ 就返回 w(a,b,c-1)+w(a,b-1,c-1)-w(a,b-1,c)；
4. 其它的情况就返回 w(a-1,b,c)+w(a-1,b-1,c)+w(a-1,b,c-1)-w(a-1,b-1,c-1)；

给出 a,b,c 要求输出 w(a,b,c)。输入的数据在 long long 范围内。

分析：别看数据范围很可怕，实际上就是一个纸老虎。如果输入数据不在 (0,20] 这个范围内，就会强制返回 1 或者 w(20,20,20)。可以非常容易地根据题意写出这个函数。但是基于和上个例子同样的理由，需要建立一个数组将 w 的取值都存下来，以免因为重复计算而超时。具体代码如下：

```
#include <iostream>
using namespace std;
long long f[25][25][25];
long long w(long long a, long long b, long long c) {
    if (a <= 0 || b <= 0 || c <= 0) return 1;
    else if (a > 20 || b > 20 || c > 20) return w(20, 20, 20);
    else if (f[a][b][c] != 0) return f[a][b][c];
    else if (a < b && b < c)
        f[a][b][c] = w(a,b,c-1)+w(a,b-1,c-1)-w(a,b-1,c);
    else
        f[a][b][c] = w(a-1,b,c)+w(a-1,b-1,c)+w(a-1,b,c-1)-w(a-1,b-1,c-1);
    return f[a][b][c];
}
```

```
int main() {
    long long a, b, c;
    while (cin >> a >> b >> c){
        if (a == -1 && b == -1 && c == -1)
            break;
        cout << "w(" << a << ", " << b << ", " << c << ") = ";
        cout << w(a, b, c) << endl;
    }
    return 0;
}
```

本题输出答案的大小并不好估计，可以先尝试使用 long long 类型，但是尝试几个输入后发现本题的输出答案大小并没有那么大，即使用 int 类型也可以通过。此外还需要特别注意输出格式（符号、空格等必须严格和题目要求中的一致），当然也可以使用 printf 来格式化字符串。

当然本题也可以使用上一节介绍的递推方法，只是边界问题和枚举顺序稍不好处理，感兴趣的读者可以自己尝试使用递推实现本题。

例 6：外星密码（洛谷 P1928）

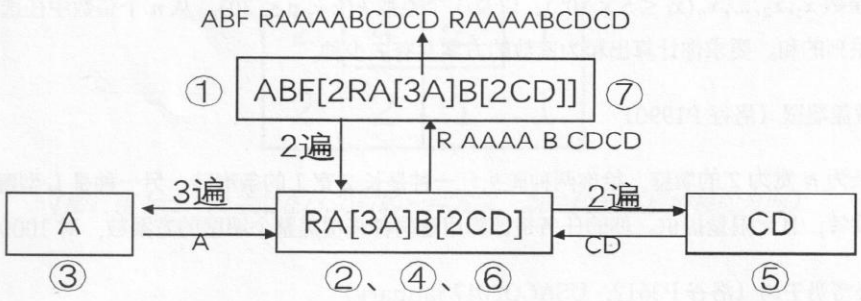
有一种压缩字符串的方式：对于连续的 D($2 \le D \le 99$) 个相同的子串 X 会压缩为 “[DX]” 的形式，而 X 可能可以进行进一步的压缩。比如说字符串 CBCBCBCB 就压缩为[4CB]或者[2[2CB]]。现给出压缩后的字符串，求压缩前的字符串原文。

分析：假设只有一层方括号，那只需要找到方括号，就可以读到重复次数，然后将该重复的部分拼接指定的次数后还原。把一个对方括号作为“压缩区”。如果方括号的“重复部分”里还有方括号呢？没关系，设法把里面的方括号继续展开即可。因此可以写成递归函数，代码如下：

```
#include<iostream>
#include<string>
using namespace std;
string expand() {
    string s = "", X;
    char c; int D;
    while (cin >> c) {// 持续读入字符，直到全部读完
        if (c == '[') { // 发现一个压缩区
            cin >> D; // 读入 D
            X = expand(); // 递归地读入 X
            while (D--) s += X; // 重复 D 次 X 并进行拼接
            // 上面不能写成 while (n--) s+=read();
        }
        else if (c == ']')
            return s; // 压缩区结束，返回已经处理好的 X
        else s += c; // 如果不是'['和']'，那还是 X 的字符，加进去即可
    }
}
int main() {
```

```
    cout << expand();
    return 0;
}
```

可以看看字符串 ABF[4RA[2A]B[3C]]是怎么被一层层展开的：



【图：密码的递归顺序】

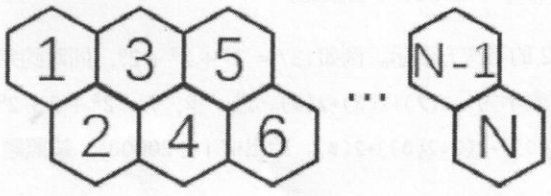
如图，带圈的编号是各个函数的执行顺序。如果感觉层数过多而不好理解，可以先只考虑只有一两层的情况以方便理解。我们已经在语言入门部分介绍了函数递归的机理，觉得理解仍然有困难，可以回去重新学习一下。

如果能将一个大的任务分解成若干个规模较小的任务，而且这些任务的形式与结构和原问题一致，就可以考虑使用递归。当问题规模足够小或者到达了边界条件就要停止递归。分解完问题后还要对这些规模小的任务合并然后返回解，最后逐级上报，解决最大规模的问题。有些问题使用递推策略和递归策略都能解决，但有些问题只能将大问题分割成小问题，但是却很难建立递推式，在这种情况下应当使用递归策略。

第3节 课后习题与试验

习题 1：蜜蜂路线（洛谷 P2437）

一只蜜蜂在下图所示的数字蜂房上爬动，已知它只能从标号小的蜂房爬到标号大的相邻蜂房，蜜蜂从蜂房 M 开始爬到蜂房 N，$M < N \leq 1000$，有多少种爬行路线？



【图：蜂房】

习题 2：小 A 点菜（洛谷 P1164）

餐厅里有 $N(N \leq 100)$ 种菜，第 i 种卖 a_i 元$(a_i \leq 1000)$，每种菜最多只能点一份。现在打算花光 $M(M \leq 10000)$ 元，请问有几种点菜的方法?

习题 3：选数（洛谷 P1036，NOIP2002 普及组）

已知 n 个整数 $x_1, x_2, \ldots, x_n (x_i \leq 5 \times 10^6)$，以及 1 个整数 $k(k < n \leq 20)$。从 n 个整数中任选 k 个整数相加，可分别得到一系列的和。要求你计算出和为素数的方案共有多少种。

习题 4：覆盖墙壁（洛谷 P1990）

你有一个长为 N 宽为 2 的墙壁，给你两种砖头：一种是长 2 宽 1 的条形砖，另一种是 L 型覆盖 3 个单元的砖头。砖头可以旋转，且无限量提供。你的任务是计算用这两种来覆盖整个墙壁的方案数，对 10000 取余。

习题 5：秘密奶牛码（洛谷 P3612，USACO2017 January）

给定一个长度不超过 30 的字符串，不断对这个将这个字符串后面拼接自身的“旋转字符串”（旋转字符串是指把原字符串的最后一个字符移动到第一个之前），比如 `COW` 拼接后变为 `COWWCO`，再变成 `COWWCOOCOWWC`，这样可以扩展成一个无限长度的字符串。给定 $N(N \leq 10^{18})$，求这个字符串第 N 个字符是什么。第一个字符是 $N = 1$。

习题 6：黑白棋子的移动（洛谷 P1259）

有 $2n(4 \leq n \leq 100)$ 个棋子排成一行，开始为位置白子全部在左边，黑子全部在右边，同时最右边还有 2 个空位。移动棋子的规则是：每次必须同时移动相邻的两个棋子，颜色不限，可以左移也可以右移到空位上去，但不能调换两个棋子的左右位置。每次移动必须跳过若干个棋子（不能平移），要求最后能移成黑白相间的一行棋子。要求编程打印出移动过程。

例如，当$n = 5$时，移动的过程是这样的:

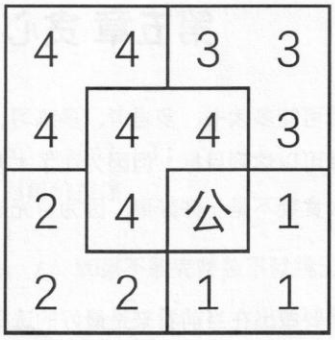
```
step 0:ooooo*****--
step 1:oooo--****o*
step 2:oooo****--o*
step 3:ooo--***o*o*
step 4:ooo*o**--*o*
step 5:o--*o**oo*o*
step 6:o*o*o*--o*o*
step 7:--o*o*o*o*o*
```

习题 7：幂次方（洛谷 P1010，NOIP1998 普及组）

任何一个正整数都可以用 2 的幂次方表示。例如$137 = 2^7 + 2^3 + 2^0$，同时约定方次用括号来表示，即a^b 可表示为 `a(b)`。由此可知，137 可表示为：`2(7)+2(3)+2(0)`。进一步，$7 = 2^2 + 2 + 2^0$ (2^1用 2 表示)，$3 = 2 + 2^0$，所以最后可表示为 `2(2(2)+2+2(0))+2(2+2(0))+2(0)`。给出$n(n \leq 20000)$，按照题目要求输出将 n 变为 2 和 0 组成的幂次方式子。

习题 8：地毯填补问题（洛谷 P1228）

迷宫是一个边长为 $2^k, (0 < k \leq 10)$ 的正方形，公主站在迷宫的一个方格上。要求你使用 L 形覆盖 3 格的小地毯不重不漏的覆盖整个迷宫（除了公主站立的位置）。请输出具体方案，方案可能不唯一。



【图：当 k=2，公主站在(3,3)时的一种方案，数字代表 L 形地毯的方向】

习题 9：南蛮图腾（洛谷 P1498）

南蛮图腾是一种递归图形。当规模为 1 时，南蛮图腾是一个简单的三角形：

```
 /\
/__\
```

规模每增加 1，图形就变得复杂了：把原来规模的图形复制三次，分别放置于上方，左下角和右下角，组成了一个更大的三角形。当规模为 2 的时候，图形是这样的：

```
   /\
  /__\
 /\  /\
/__\/__\
```

给出规模 $n(n \leq 10)$，请画出对应规模的图形。

习题 10：思考一下递推与递归的例题中，哪些可以使用另外一种思路（比如递归的例题是否可以使用递推完成）？如果可以的话，尝试使用另外一种方式完成这些例题。