# 程序设计实习

## 算法基础

张勤健
zqj@pku.edu.cn

北京大学信息科学技术学院

2024 年 5 月 17 日

# 例题 01 数字三角形

```
      7
    3  8
   8  1  0
 2  7  4  4
4  5  2  6  5
```

在上面的数字三角形中寻找一条从顶部到底边的路径，使得路径上所经过的数字之和最大。路径上的每一步都只能往左下或右下走。只需要求出这个最大和即可，不必给出具体路径。

三角形的行数大于 1 小于等于 100，数字为 0 - 99

(http://bailian.openjudge.cn/practice/1163)

输入格式

```
5 //三角形行数。下面是三角形
7
3 8
8 1 0
2 7 4 4
4 5 2 6 5
```

# 例题 01 数字三角形

用二维数组存放数字三角形。
$D(r,j)$: 第 $r$ 行第 $j$ 个数字 ($r$,$j$ 从 1 开始算)
$MaxSum(r,j)$: 从 $D(r,j)$ 到底边的各条路径中，最佳路径的数字之和。
问题：求 $MaxSum(1,1)$

# 例题 01 数字三角形

用二维数组存放数字三角形。
$D(r,j)$: 第 $r$ 行第 $j$ 个数字 ($r$,$j$ 从 1 开始算)
$MaxSum(r,j)$: 从 $D(r,j)$ 到底边的各条路径中，最佳路径的数字之和。
问题：求 $MaxSum(1,1)$

典型的递归问题。

$D(r,j)$ 出发，下一步只能走 $D(r+1,j)$ 或者 $D(r+1,j+1)$。故对于 N 行的三角形：

$$MaxSum(r,j)=\begin{cases} D(r,j) & r=N \\ \max\{MaxSum(r+1,j),MaxSum(r+1,j+1)\}+D(r,j) & r\neq N \end{cases}$$

# 例题 01 数字三角形

```cpp
#include <iostream>
#include <algorithm>
#define MAX 101
using namespace std;
int D[MAX][MAX];
int n;
int MaxSum(int i, int j) {
  if (i == n) return D[i][j];
  int x = MaxSum(i+1,j);
  int y = MaxSum(i+1,j+1);
  return max(x,y)+D[i][j];
}
int main(){
  int i,j;
  cin >> n;
  for (i=1; i<=n; i++)
    for (j=1; j<=i; j++)
      cin >> D[i][j];
  cout << MaxSum(1,1) << endl;
  return 0;
}
```

# 例题 01 数字三角形

## 重复计算

```
                7(1)
            3(1)    8(1)
        8(1)    1(2)    0(1)
    2(1)    7(3)    4(3)    4(1)
4(1)    5(4)    2(6)    6(4)    5(1)
```

如果采用递规的方法，深度遍历每条路径，存在大量重复计算。则时间复杂度为 $2^n$, 对于 $n = 100$ 行，肯定超时

# 例题 01 数字三角形

## 重复计算

```
                7(1)
            3(1)    8(1)
        8(1)    1(2)    0(1)
    2(1)    7(3)    4(3)    4(1)
4(1)    5(4)    2(6)    6(4)    5(1)
```

如果采用递规的方法，深度遍历每条路径，存在大量重复计算。则时间复杂度为 $2^n$, 对于 $n = 100$ 行，肯定超时

改进
如果每算出一个 $MaxSum(r, j)$ 就保存起来，下次用到其值的时候直接取用，则可免去重复计算。那么可以用 $O(n^2)$ 时间完成计算。因为三角形的数字总数是 $n(n + 1)/2$

# 例题 01 数字三角形

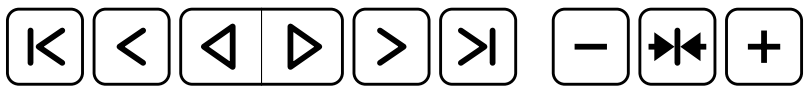
```cpp
#include <iostream>
#include <algorithm>
using namespace std;
#define MAX 101
int D[MAX][MAX];    int n;
int maxSum[MAX][MAX];
int MaxSum(int i, int j){
  if( maxSum[i][j] != -1 ) return maxSum[i][j];
  if(i==n) {
    maxSum[i][j] = D[i][j];
    return maxSum[i][j];
  }
  int x = MaxSum(i+1,j);
  int y = MaxSum(i+1,j+1);
  maxSum[i][j] = max(x,y)+ D[i][j];
  return maxSum[i][j];
}
int main(){
  int i,j;
  cin >> n;
  for(i=1;i<=n;i++) {
    for(j=1;j<=i;j++) {
      cin >> D[i][j];
      maxSum[i][j] = -1;
    }
  }
  cout << MaxSum(1,1) << endl;
  return 0;
}
```

# 例题 01 数字三角形

递归转成递推

| | | | | |
|---|---|---|---|---|
| 7 | | | | |
| 3 | 8 | | | |
| 8 | 1 | 0 | | |
| 2 | 7 | 4 | 4 | |
| 4 | 5 | 2 | 6 | 5 |

| | | | | |
|---|---|---|---|---|
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| 4 | 5 | 2 | 6 | 5 |



# 例题 01 数字三角形

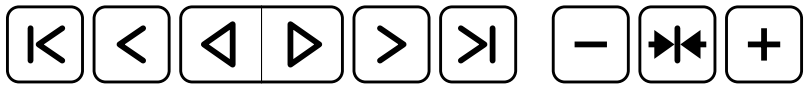
```cpp
#include <iostream>
#include <algorithm>
using namespace std;
#define MAX 101
int D[MAX][MAX];
int n;
int maxSum[MAX][MAX];
int main()  {
  int i,j;
  cin >> n;
  for (i=1; i <=n; i++)
    for (j=1; j <= i; j++)
      cin >> D[i][j];
  for (int i = 1;i <= n; ++ i)
    maxSum[n][i] = D[n][i];
  for (int i = n - 1; i >= 1;  --i)
    for (int j = 1; j <= i; ++j)
      maxSum[i][j] = max(maxSum[i+1][j], maxSum[i+1][j+1]) + D[i][j];
  cout << maxSum[1][1] << endl;
  return 0;
}

```

# 例题 01 数字三角形

空间优化

| | | | | |
|---|---|---|---|---|
| 7 | | | | |
| 3 | 8 | | | |
| 8 | 1 | 0 | | |
| 2 | 7 | 4 | 4 | |
| 4 | 5 | 2 | 6 | 5 |

| | | | | |
|---|---|---|---|---|
| 4 | 5 | 2 | 6 | 5 |



没必要用二维 $maxSum$ 数组存储每一个 $MaxSum(r, j)$，只要从底层一行行向上递推，那么只要一维数组 $maxSum[100]$ 即可，即只要存储一行的 $MaxSum$ 值就可以。

# 例题 01 数字三角形

```cpp
#include <iostream>
#include <algorithm>
using namespace std;
#define MAX 101
int D[MAX][MAX];
int n;
int maxSum[MAX];
int main(){
  int i,j;
  cin >> n;
  for (i = 1; i <= n; ++i)
    for (j = 1; j <= i; ++j)
      cin >> D[i][j];
  for (i = 1; i <= n; ++i)
    maxSum[i] = D[n][i];
  for (int i = n - 1; i >= 1;  --i)
    for (int j = 1; j <= i; ++j)
      maxSum[j] = max(maxSum[j], maxSum[j+1]) + D[i][j];
  cout << maxSum[1] << endl;
  return 0;
}

```

# 递归到动规的一般转化方法

递归函数有 n 个参数，就定义一个 n 维的数组，数组的下标是递归函数参数的取值范围，数组元素的值是递归函数的返回值，这样就可以从边界值开始，逐步填充数组，相当于计算递归函数值的逆过程。

# 动规解题的一般思路

1. 将原问题分解为子问题

把原问题分解为若干个子问题，子问题和原问题形式相同或类似，只不过规模变小了。子问题都解决，原问题即解决 (数字三角形例)。

子问题的解一旦求出就会被保存，所以每个子问题只需求解一次。

# 动规解题的一般思路

## 2. 确定状态

- 在用动态规划解题时，我们往往将和子问题相关的各个变量的一组取值，称之为一个“状态”。一个“状态”对应于一个或多个子问题，所谓某个“状态”下的“值”，就是这个“状态”所对应的子问题的解。
- 所有“状态”的集合，构成问题的“状态空间”。“状态空间”的大小，与用动态规划解决问题的时间复杂度直接相关。在数字三角形的例子里，一共有 N×(N+1)/2 个数字，所以这个问题的状态空间里一共就有 N×(N+1)/2 个状态。
  整个问题的时间复杂度是状态数目乘以计算每个状态所需时间.
  在数字三角形里每个“状态”只需要经过一次，且在每个状态上作计算所花的时间都是和 N 无关的常数。
- 用动态规划解题，经常碰到的情况是，K 个整型变量能构成一个状态（如数字三角形中的行号和列号这两个变量构成“状态”）。如果这 K 个整型变量的取值范围分别是 $N_1, N_2, \ldots, N_k$，那么，我们就可以用一个 K 维的数组 $array[N_1][N_2], \ldots, [N_k]$ 来存储各个状态的“值”。

# 动规解题的一般思路

3. 确定一些初始状态（边界状态）的值

以“数字三角形”为例，初始状态就是底边数字，值就是底边数字值。

# 动规解题的一般思路

## 4. 确定状态转移方程

定义出什么是“状态”，以及在该“状态”下的“值”后，就要找出不同的状态之间如何迁移　即如何从一个或多个“值”已知的“状态”，求出另一个“状态”的“值”。状态的迁移可以用递推公式表示，此递推公式也可被称作“状态转移方程”。

数字三角形的状态转移方程:

$$MaxSum[r][j] = \begin{cases} D[r][j] & r = N \\ \max\{MaxSum[r+1][j], MaxSum[r+1][j+1]\} + D[r][j] & r \neq N \end{cases}$$

# 能用动规解决的问题的特点

1. 问题具有最优子结构性质。
   如果问题的最优解所包含的子问题的解也是最优的，我们就称该问题具有最优子结构性质。反过来说就是，我们可以通过子问题的最优解，推导出问题的最优解。
2. 无后效性。
   当前的若干个状态值一旦确定，则此后过程的演变就只和这若干个状态的值有关，和之前是采取哪种手段或经过哪条路径演变到当前的这若干个状态，没有关系。
   某阶段状态一旦确定，就不受之后阶段的决策影响。

# 例题 02：最长上升子序列

一个数的序列 $a_i$，当 $a_1 < a_2 < ... < a_S$ 的时候，我们称这个序列是上升的。对于给定的一个序列 $(a_1, a_2, \ldots, a_N)$，我们可以得到一些上升的子序列 $(a_{i_1}, a_{i_2}, ..., a_{i_K})$，这里 $1 \leq i_1 < i_2 < ... < i_K \leq N$。比如，对于序列 (1, 7, 3, 5, 9, 4, 8)，有它的一些上升子序列，如 (1, 7), (3, 4, 8) 等等。这些子序列中最长的长度是 4，比如子序列 (1, 3, 5, 8).
你的任务，就是对于给定的序列，求出最长上升子序列的长度。

输入
输入的第一行是序列的长度 $N$ $(1 \leq N \leq 1000)$。第二行给出序列中的 $N$ 个整数，这些整数的取值范围都在 0 到 10000。

输出
最长上升子序列的长度。
(http://bailian.openjudge.cn/practice/2757)

# 例题 02：最长上升子序列

## 1. 找子问题

“求序列的前 n 个元素的最长上升子序列的长度”是个子问题，但这样分解子问题，不具有“无后效性”
假设 $F(n) = x$, 但可能有多个序列满足 $F(n) = x$。有的序列的最后一个元素比 $a_{n+1}$ 小，则加上 $a_{n+1}$ 就能形成更长上升子序列；有的序列最后一个元素不比 $a_{n+1}$ 小……以后的事情受如何达到状态 $n$ 的影响，不符合“无后效性”

# 例题 02：最长上升子序列

## 1. 找子问题

“求序列的前 n 个元素的最长上升子序列的长度”是个子问题，但这样分解子问题，不具有“无后效性”
假设 $F(n) = x$, 但可能有多个序列满足 $F(n) = x$。有的序列的最后一个元素比 $a_{n+1}$ 小，则加上 $a_{n+1}$ 就能形成更长上升子序列；有的序列最后一个元素不比 $a_{n+1}$ 小……以后的事情受如何达到状态 $n$ 的影响，不符合“无后效性”

换了思路：
" 求以 $a_k$ $(k = 1, 2, 3 \ldots N)$ 为终点的最长上升子序列的长度"
一个上升子序列中最右边的那个数，称为该子序列的“终点”。
虽然这个子问题和原问题形式上并不完全一样，但是只要这 $N$ 个子问题都解决了，那么这 $N$ 个子问题的解中，最大的那个就是整个问题的解。

# 例题 02：最长上升子序列

## 2. 确定状态

子问题只和一个变量– 数字的位置相关。因此序列中数的位置 $k$ 就是“状态”，而状态 $k$ 对应的“值”，就是以 $a_k$ 做为“终点”的最长上升子序列的长度。状态一共有 $N$ 个。

# 例题 02：最长上升子序列

3. 找出状态转移方程

$maxLen(k)$ 表示以 $a_k$ 做为“终点”的最长上升子序列的长度那么：

初始状态：$maxLen(1) = 1$

$$maxLen(k) = \max_{1 \le i < k, a_i < a_k, k \ne 1} \{maxLen(i)\} + 1$$

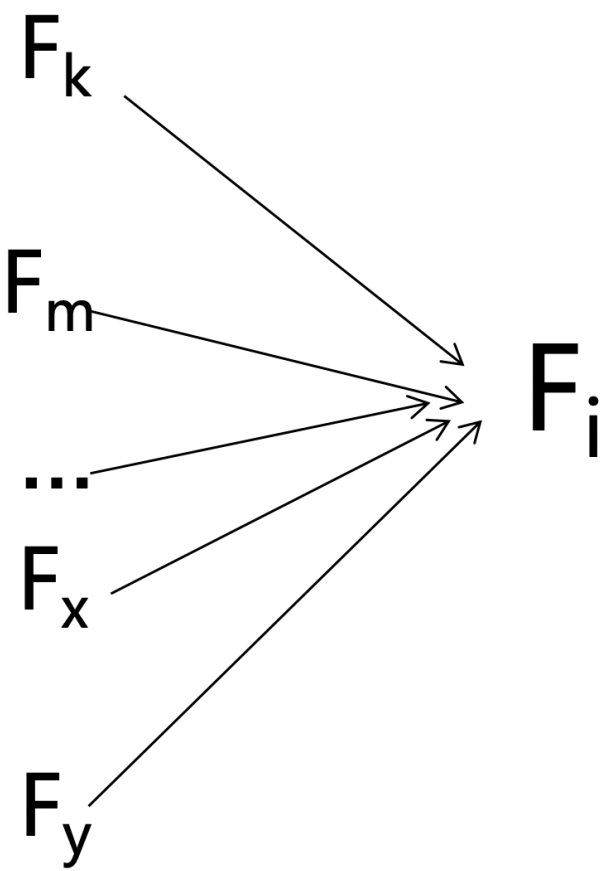
若找不到这样的 $i$, 则 $maxLen(k) = 1$

$maxLen(k)$ 的值，就是在 $a_k$ 左边，“终点”数值小于 $a_k$ ，且长度最大的那个上升子序列的长度再加 1。因为 $a_k$ 左边任何“终点”小于 $a_k$ 的子序列，加上 $a_k$ 后就能形成一个更长的上升子序列。

# 例题 02：最长上升子序列

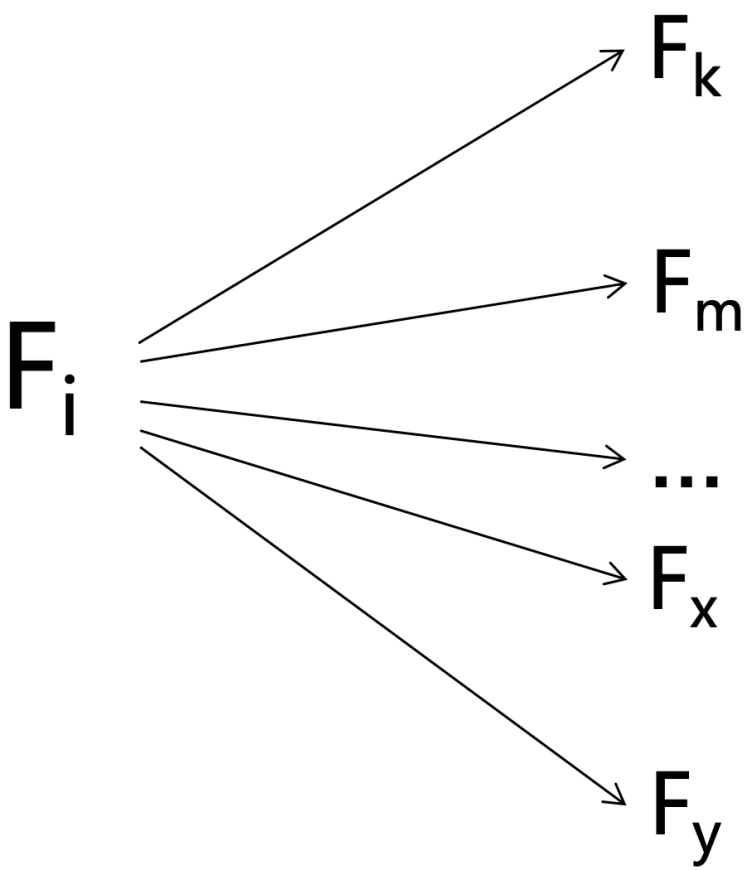
```cpp
#include <iostream>
#include <cstring>
#include <algorithm>
using namespace std;
const int  MAXN =1010;
int a[MAXN];
int maxLen[MAXN];
int main() {
  int N;
  cin >> N;
  for (int i = 1; i <= N; ++i) {
    cin >> a[i];
    maxLen[i] = 1;
  }
  for (int i = 2; i <= N; ++i) {
    //每次求以第 i 个数为终点的最长上升子序列的长度
    for (int j = 1; j < i; ++j)  //察看以第 j 个数为终点的最长上升子序列
      if (a[i] > a[j]) maxLen[i] = max(maxLen[i], maxLen[j] + 1);
  }
  cout << * max_element(maxLen + 1, maxLen + N + 1);
  return 0;
} //时间复杂度 O(N^2)

```

# “人人为我”递推型动规




状态 $i$ 的值 $F_i$ 由若干个值已知的状态值 $F_k, F_m, \ldots, F_y$ 推出，如求和，取最大值……

# 我为人人”递推型动规




状态 $i$ 的值 $F_i$ 在被更新（不一定是最终求出）的时候，依据 Fi 去更新（不一定是最终求出）和状态 i 相关的其他一些状态的值 $F_k, F_m, \ldots, F_y$

# 例题 02：最长上升子序列

```cpp
#include <iostream>
#include <cstring>
#include <algorithm>
using namespace std;
const int  MAXN =1010;
int a[MAXN];
int maxLen[MAXN];
int main() {
  int N;
  cin >> N;
  for (int i = 1; i <= N; ++i) {
    cin >> a[i];
    maxLen[i] = 1;
  }
  for (int i = 1; i <= N; ++i)
    for (int j = i + 1; j <= N; ++j)//看看能更新哪些状态的值
      if (a[j] > a[i]) maxLen[j] = max(maxLen[j], maxLen[i]+1);
  cout << *max_element(maxLen+1, maxLen + N + 1);
  return 0;
} //时间复杂度 O(N^2)
```

# 动规的三种形式

1. 记忆递归型
   优点：只经过有用的状态，没有浪费。递推型会查看一些没用的状态，有浪费
   缺点：可能会因递归层数太深导致爆栈，函数调用带来额外时间开销。空间优化方面比较困难。总体来说，比递推型慢。
2. “我为人人”递推型
   没有什么明显的优势，有时比较符合思考的习惯。个别特殊题目中会比“人人为我”型节省空间。
3. “人人为我”递推型
   在选取最优备选状态的值 $F_m, F_n, \ldots, F_y$ 时，有可能有好的算法或数据结构可以用来显著降低时间复杂度。

# 例题 03：最长公共子序列

给出两个字符串，求出这样的一个最长的公共子序列的长度：子序列中的每个字符都能在两个原串中找到，而且每个字符的先后顺序和原串中的先后顺序一致。

样例输入

```
abcfbc abfcab
programming contest
abcd mnp
```

样例输出

```
4
2
0
```

# 例题 03：最长公共子序列

两个串 $S_1, S_2$

设 $MaxLen(i,j)$ 表示：$S_1$ 的左边 $i$ 个字符形成的子串，与 $S_2$ 左边的 $j$ 个字符形成的子串的最长公共子序列的长度 ($i$,$j$ 从 0 开始算)

$MaxLen(i,j)$ 就是本题的“状态”

假定 $len1 = strlen(S_1), len2 = strlen(S_2)$

那么题目就是要求 $MaxLen(len1, len2)$

$$MaxLen(i,j) = \begin{cases} 0, & i = 0 or j = 0 \\ MaxLen(i-1, j-1) + 1, & i > 0, j > 0, s1[i-1] = s2[j-1] \\ max(MaxLen(i, j-1), MaxLen(i-1, j)), & i > 0, j > 0, s1[i-1] \neq s2[j-1] \end{cases}$$

时间复杂度 $O(len1 * len2)$

# 例题 03：最长公共子序列

```cpp
#include <iostream>
#include <cstring>
using namespace std;
char sz1[1000];
char sz2[1000];
int maxLen[1000][1000];
int main() {
  while (cin >> sz1 >> sz2) {
    int length1 = strlen(sz1);
    int length2 = strlen(sz2);
    int nTmp;
    int i,j;
    for (i = 0; i <= length1; ++i)
      maxLen[i][0] = 0;
    for (j = 0; j <= length2; ++j)
      maxLen[0][j] = 0;
    for (i = 1; i <= length1; ++i)  {
      for (j = 1; j <= length2; ++j) {
        if (sz1[i-1] == sz2[j-1])
          maxLen[i][j] = maxLen[i-1][j-1] + 1;
        else
          maxLen[i][j] = max(maxLen[i][j-1], maxLen[i-1][j]);
      }
    }
    cout <<  maxLen[length1][length2] << endl;
  }
  return 0;
}
```

# 例题 04：最佳加法表达式

有一个由 $1 \ldots 9$ 组成的数字串. 问如果将 $m$ 个加号插入到这个数字串中, 在各种可能形成的表达式中，值最小的那个表达式的值是多少

# 例题 04：最佳加法表达式

解题思路：
假定数字串长度是 $n$，添完加号后，表达式的最后一个加号添加在第 $i$ 个数字后面，那么整个表达式的最小值，就等于在前 $i$ 个数字中插入 $m-1$ 个加号所能形成的最小值，加上第 $i+1$ 到第 $n$ 个数字所组成的数的值（$i$ 从 1 开始算）。

# 例题 04：最佳加法表达式

解题思路：
假定数字串长度是 $n$，添完加号后，表达式的最后一个加号添加在第 $i$ 个数字后面，那么整个表达式的最小值，就等于在前 $i$ 个数字中插入 $m$–1 个加号所能形成的最小值，加上第 $i+1$ 到第 $n$ 个数字所组成的数的值（$i$ 从 1 开始算）。
设 $V(m,n)$ 表示在 $n$ 个数字中插入 $m$ 个加号所能形成的表达式最小值，那么：

$$V(m,n)=\begin{cases} Num(1,n), & m=0 \\ \infty, & n<m+1 \\ \min\limits_{i=m,\ldots,n-1}\{V(m-1,i)+Num(i+1,n)\}, & \text{others} \end{cases}$$

$Num(i,j)$ 表示从第 $i$ 个数字到第 $j$ 个数字所组成的数。数字编号从 1 开始算。此操作复杂度是 $O(j-i+1)$，可以预处理后存起来。

总时间复杂度：$O(mn^2)$ .

# 例题 05：神奇的口袋

有一个神奇的口袋，总的容积是 40，用这个口袋可以变出一些物品，这些物品的总体积必须是 40。
John 现在有 $n$（$1 \leq n \leq 20$）个想要得到的物品，每个物品的体积分别是 $a_1, a_2, \ldots, a_n$。John 可以从这些物品中选择一些，如果选出的物体的总体积是 40，那么利用这个神奇的口袋，John 就可以得到这些物品。
现在的问题是，John 有多少种不同的选择物品的方式。

输入：
输入的第一行是正整数 $n(1 \leq n \leq 20)$，表示不同的物品的数目。接下来的 n 行，每行有一个 1 到 40 之间的正整数，分别给出 $a_1, a_2, \ldots, a_n$ 的值。

输出：
输出不同的选择物品的方式的数目

# 例题 05：神奇的口袋

最直观的想法：枚举

# 例题 05：神奇的口袋

最直观的想法：枚举

枚举每个物品是选还是不选，最多共 $2^n$ 种情况，$n \leq 20$，所以最多 $2^{20}$。

# 例题 05：神奇的口袋-递归解法

```cpp
#include <iostream>
using namespace std;
int a[30];
int N;
int Ways(int w, int k)  { // 从前 k 种物品中选择一些，凑成体积 w 的做法数目
  if (w == 0)   return 1;
  if (k <= 0)   return 0;
  return Ways(w, k - 1) + Ways(w - a[k], k - 1);
}
int main()   {
  cin >> N;
  for (int i = 1; i <= N; ++i)
    cin >> a[i];
  cout << Ways(40,N);
  return 0;
}
```

# 例题 05：神奇的口袋-动规解法

```cpp
#include <iostream>
using namespace std;
int a[40];
int N;
int Ways[50][40];//Ways[i][j] 表示从前 j 种物品里凑出体积 i 的方法数
int main()   {
  cin >> N;
  memset(Ways, 0, sizeof(Ways));
  for (int i = 1; i <= N; ++ i) {
    cin >> a[i];
    Ways[0][i] = 1;
  }
  Ways[0][0] = 1;
  for (int w = 1 ; w <= 40; ++w) {
    for (int k = 1; k <= N; ++k) {
      Ways[w][k] = Ways[w][k-1];
      if (w-a[k] >= 0)  Ways[w][k] += Ways[w-a[k]][k-1];
    }
  }
  cout << Ways[40][N];
  return 0;
}
```

# 例题 05：神奇的口袋

此问题仅在询问容积 40 是否可达，40 是个很小的数，可以考虑对值域空间-即对容积的可达性进行动态规划。

定义一维数组 int sum[41];

依次放入物品，计算每次放入物品可达的容积，并在相应空间设置记录，最后判断 sum[40] 是否可达，到达了几次。

# 例题 05：神奇的口袋-动规解法

```cpp
#include <iostream>
using namespace std;
#define MAX 41
int main() {
  int n, i, j, input;
  int sum[MAX];
  for (i=0; i < MAX; ++i) sum[i] = 0;
  cin >> n;
  for (i = 0; i < n; ++i) {
    cin >> input;
    for(j = 40; j >= 1; --j)
    if (sum[j] > 0 && j + input <= 40)
      sum[j + input] += sum[j];
    //如果 j 有 sum[j] 种方式可达，则每种方式加上 input 就可达 j + input
    sum[input]++;
  }
  cout << sum[40] << endl;
  return 0;
}

```

# 例题 06：Charm Bracelet

有 $N$ 件物品和一个容积为 $M$ 的背包。第 $i$ 件物品的体积 $w[i]$，价值是 $d[i]$。求解将哪些物品装入背包可使价值总和最大。每种物品只有一件，可以选择放或者不放 $(N \leq 3500, M \leq 13000)$。

# 例题 06：Charm Bracelet

用 $F[i][j]$ 表示取前 $i$ 种物品，使它们总体积不超过 $j$ 的最优取法取得的价值总和。要求 $F[N][M]$.

边界条件：

$$F[1][j] = \begin{cases} d[1], & w[1] \leq j \\ 0, & w[1] > j \end{cases}$$

递推：

$$F[i][j] = \begin{cases} F[i-1][j], & j - w[i] < 0 \\ max(F[i-1][j], F[i-1][j-w[i]] + d[i]), & j - w[i] \geq 0 \end{cases}$$

取或不取第 $i$ 种物品，两者选优。

# 例题 06：Charm Bracelet

$$F[i][j] = \begin{cases} F[i-1][j], & j - w[i] < 0 \\ max(F[i-1][j], F[i-1][j-w[i]] + d[i]), & j - w[i] \geq 0 \end{cases}$$

本题如用记忆型递归，需要一个很大的二维数组，会超内存。注意到这个二维数组的下一行的值，只用到了上一行的正上方及左边的值，因此可用滚动数组的思想，只要一行即可。即可以用一维数组，用“人人为我”递推型动规实现。

# 例题 07：滑雪

Michael 喜欢滑雪百这并不奇怪，因为滑雪的确很刺激。可是为了获得速度，滑的区域必须向下倾斜，而且当你滑到坡底，你不得不再次走上坡或者等待升降机来载你。Michael 想知道载一个区域中最长的滑坡。区域由一个二维数组给出。数组的每个数字代表点的高度。下面是一个例子

```
1  2  3  4  5
16 17 18 19 6
15 24 25 20 7
14 23 22 21 8
13 12 11 10 9
```

一个人可以从某个点滑向上下左右相邻四个点之一，当且仅当高度减小。在上面的例子中，一条可滑行的滑坡为 24-17-16-1。当然 25-24-23-...-3-2-1 更长。事实上，这是最长的一条。

输入

输入的第一行表示区域的行数 R 和列数 $C(1 \le R, C \le 100)$。下面是 R 行，每行有 $C$ 个整数，代表高度 $h$，$0 \le h \le 10000$。

输出

输出最长区域的长度。

# 例题 07：滑雪

$L(i,j)$ 表示从点 $(i,j)$ 出发的最长滑行长度。
一个点 $(i,j)$，如果周围没有比它低的点，$L(i,j)=1$

否则

递推公式：$L(i,j)$ 等于 $(i,j)$ 周围四个点中，比 $(i,j)$ 低，且 $L$ 值最大的那个点的 $L$ 值，再加 1
复杂度：$O(n^2)$

# 例题 07：滑雪

解法 1) “人人为我”式递推

$L(i,j)$ 表示从点 $(i,j)$ 出发的最长滑行长度。
一个点 $(i,j)$, 如果周围没有比它低的点，$L(i,j)=1$

将所有点按高度从小到大排序。每个点的 $L$ 值都初始化为 1

从小到大遍历所有的点。经过一个点 $(i,j)$ 时，用递推公式求 $L(i,j)$

# 例题 07：滑雪

解法 2) “我为人人”式递推

$L(i,j)$ 表示从点 $(i,j)$ 出发的最长滑行长度。
一个点 $(i,j)$, 如果周围没有比它低的点，$L(i,j)=1$

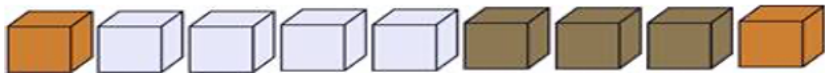
将所有点按高度从小到大排序。每个点的 $L$ 值都初始化为 1

从小到大遍历所有的点。经过一个点 $(i,j)$ 时，要更新他周围的，比它高的点的 $L$ 值。例如：

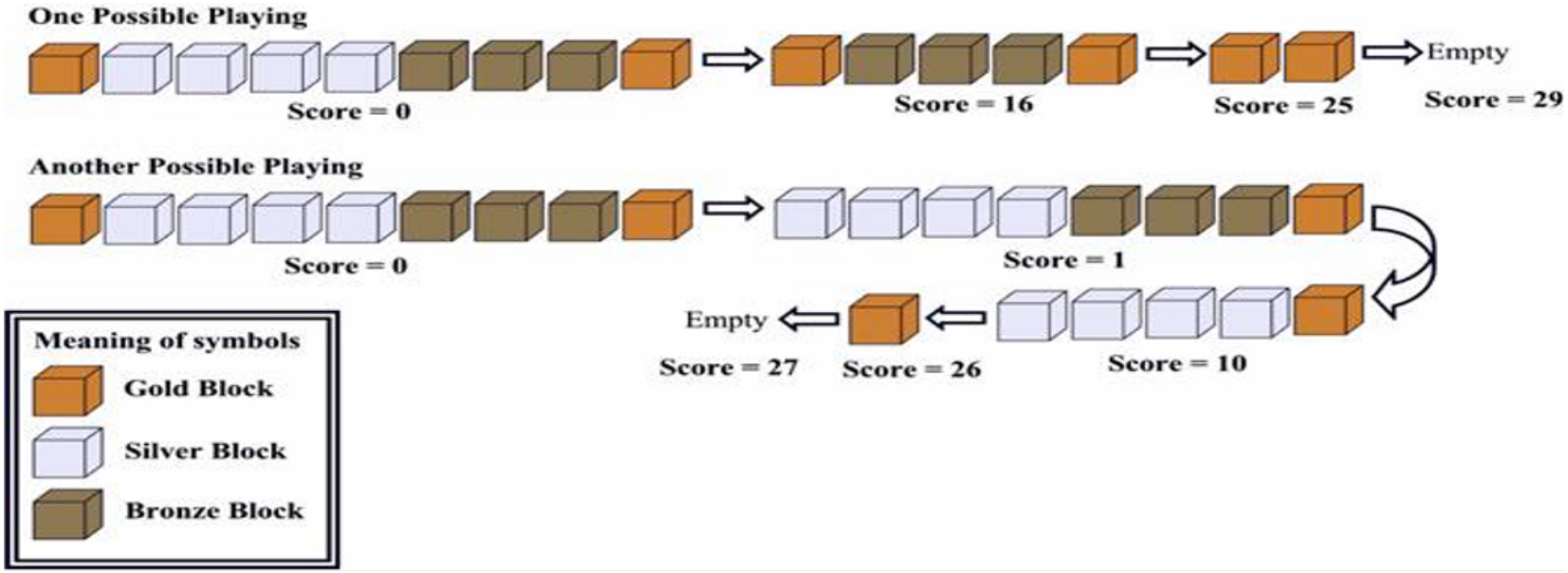
```
if  H(i+1,j) > H(i,j)  // H 代表高度
  L(i+1,j) = max(L(i+1,j),L(i,j)+1)
```

# 例题 08：方盒游戏

$N$ 个方盒 (box) 摆成一排，每个方盒有自己的颜色。连续摆放的同颜色方盒构成一个方盒片段 (box segment)。下图中共有四个方盒片段，每个方盒片段分别有 $1$、$4$、$3$、$1$ 个方盒



玩家每次点击一个方盒，则该方盒所在方盒片段就会消失。若消失的方盒片段中共有 $k$ 个方盒，则玩家获得 $k \times k$ 个积分。




请问：给定游戏开始时的状态，玩家可获得的最高积分是多少?

# 例题 08：方盒游戏

输入：
第一行是一个整数 $t(1 \leq t \leq 15)$，表示共有多少组测试数据。每组测试数据包括两行

- 第一行是一个整数 $n(1 \leq n \leq 200)$，表示共有多少个方盒
- 第二行包括 $n$ 个整数，表示每个方盒的颜色。这些整数的取值范围是 $[1, n]$

输出：
对每组测试数据，分别输出该组测试数据的序号、以及玩家可以获得的最高积分

样例输入：

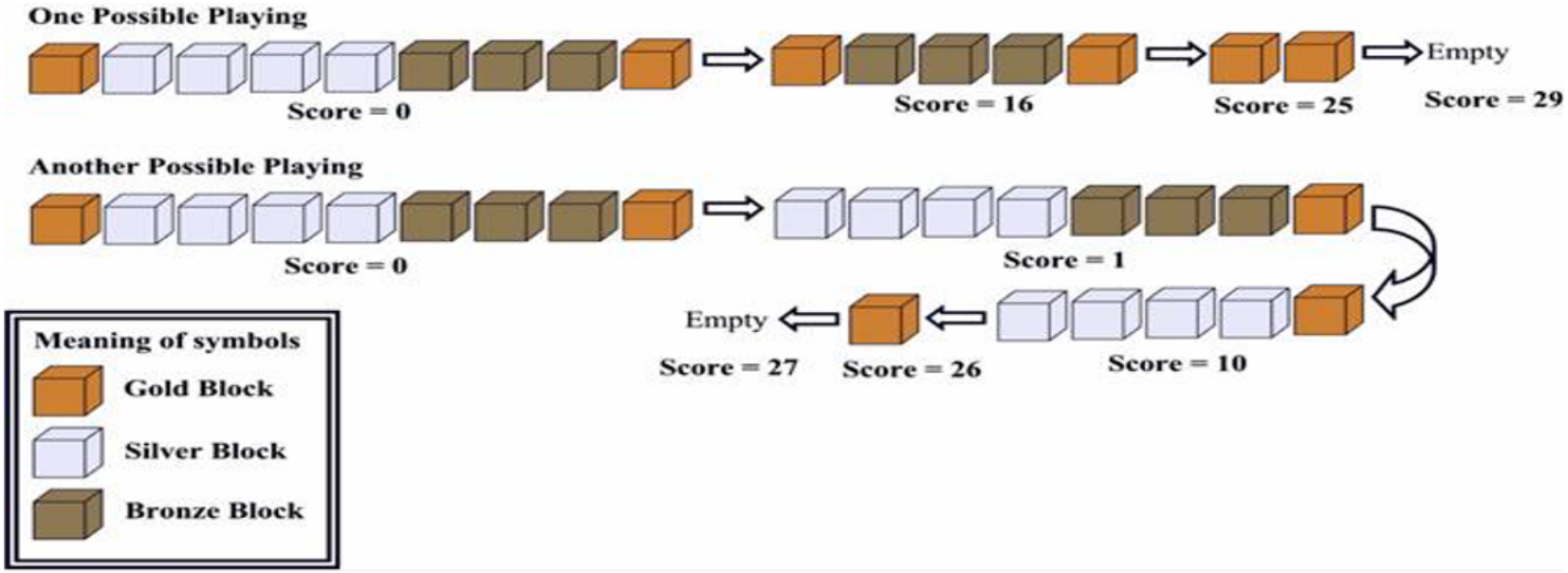
```
2
9
1 2 2 2 2 3 3 3 1
1
1
```

样例输出：

```
Case 1: 29
Case 2: 1
```

# 例题 08：方盒游戏

当同颜色的方盒摆放在不连续的位置时，方盒的点击顺序影响玩家获得的积分




点击下图中红色和蓝色方盒可获得的积分

所有红色方盒合并到同一个片段：$49 + 1 + 36 = 86$

所有蓝色方盒合并到同一个片段：$49 + 16 + 9 = 74$

# 例题 08：方盒游戏

递归问题：
每次点击之后，剩下的方盒构成一个新的方盒队列，新队列中方盒的数量减少了。然后计算玩家从新队列中可获得的最高积分

# 例题 08：方盒游戏

一种思路：
将连续的若干个方块作为一个“大块”(box_segment)
考虑，假设开始一共有 $n$ 个“大块”，编号 $0$ 到 $n-1$
第 $i$ 个大块的颜色是 $color[i]$, 包含的方块数目，即长度, 是 $len[i]$

用 $click\_box(i,j)$ 表示从大块 $i$ 到大块 $j$ 这一段消除后所能得到的最高分

则整个问题就是： $click\_box(0,n-1)$

# 例题 08：方盒游戏

要求 $click\_box(i,j)$ 时，考虑最右边的大块 $j$，对它有两种处理方式，要取其优者：

1. 直接消除它，此时能得到最高分就是：$click\_box(i,j-1)+len[j]^2$
2. 期待以后它能和左边的某个同色大块合并

# 例题 08：方盒游戏

要求 $click\_box(i,j)$ 时，考虑最右边的大块 $j$，对它有两种处理方式，要取其优者：

1. 直接消除它，此时能得到最高分就是：$click\_box(i,j-1)+len[j]^2$
2. 期待以后它能和左边的某个同色大块合并

考虑和左边的某个同色大块合并:

左边的同色大块可能有很多个，到底和哪个合并最好，不知道，只能枚举。假设大块 $j$ 和左边的大块 $k(i \le k < j-1)$ 合并，此时能得到的最高分是多少呢?

# 例题 08：方盒游戏

要求 $click\_box(i,j)$ 时，考虑最右边的大块 $j$，对它有两种处理方式，要取其优者：

1. 直接消除它，此时能得到最高分就是：$click\_box(i,j-1)+len[j]^2$
2. 期待以后它能和左边的某个同色大块合并

考虑和左边的某个同色大块合并:

左边的同色大块可能有很多个，到底和哪个合并最好，不知道，只能枚举。假设大块 $j$ 和左边的大块 $k(i \le k < j-1)$ 合并，此时能得到的最高分是多少呢?
是不是：

$$\max_{i \le k < j-1} \{click\_box(i,k-1)+click\_box(k+1,j-1)+(len[k]+len[j])^2\}$$

# 例题 08：方盒游戏

$$\max_{i \le k < j-1} \{click\_box(i, k-1) + click\_box(k+1, j-1) + (len[k] + len[j])^2\}$$

# 例题 08：方盒游戏

$$\max_{i \le k < j-1} \{click\_box(i, k-1) + click\_box(k+1, j-1) + (len[k] + len[j])^2\}$$

不对！
因为将大块 $k$ 和大块 $j$ 合并后，形成的新大块会在最右边。将该新大块直接将其消去的做法，才符合上述式子，但直接将其消去，未必是最好的，也许它还应该和左边的同色大块合并，才更好

递推关系无法形成，怎么办?

# 例题 08：方盒游戏

需要改变问题的形式。
$click\_box(i,j)$ 这个形式不可取，因为无法形成递推关系
考虑新的形式：
$click\_box(i,j,ex\_len)$
表示:
大块 $j$ 的右边已经有一个长度为 $ex\_len$ 的大块 (该大块可能是在合并过程中形成的，不妨就称其为 $ex\_len$)，且 $j$ 的颜色和 $ex\_len$ 相同，在此情况下将 $i$ 到 $j$ 以及 $ex\_len$ 都消除所能得到的最高分。

于是整个问题就是求：$click\_box(0,n-1,0)$

# 例题 08：方盒游戏

求 $click\_box(i, j, ex\_len)$ 时，有两种处理方法，取最优者假设 $j$ 和 $ex\_len$ 合并后的大块称作 $Q$

1. 将 $Q$ 直接消除，这种做法能得到的最高分就是：$click\_box(i, j-1, 0) + (len[j] + ex\_len)^2$
2. 期待 $Q$ 以后能和左边的某个同色大块合并。需要枚举可能和 $Q$ 合并的大块。假设让大块 $k$ 和 $Q$ 合并，则此时能得到的最大分数是：$click\_box(i, k, len[j] + ex\_len) + click\_box(k+1, j-1, 0)$

递归的终止条件是什么？

```
i == j
```

# 例题 08：方盒游戏

```
1  #include <iostream>
2  #include <cstring>
3  using namespace std;
4  const int M = 210;
5  struct Segment {
6    int color;
7    int len;
8  };
9  Segment segments[M];
10 int score[M][M][M];
11 int ClickBox(int i,int j,int len)  {
12   if( score[i][j][len] != -1) return score[i][j][len];
13   int result = (segments[j].len + len) * (segments[j].len + len);
14   if( i == j ) return result;
15   result += ClickBox(i,j-1,0);
16   for(int k = i; k <= j-1; ++k) {
17     if( segments[k].color != segments[j].color ) continue;
18     int r = ClickBox(k+1,j-1,0);
19     r += ClickBox(i,k,segments[j].len + len);
20     result = max(result,r);
21   }
22   score[i][j][len] = result;
23   return result;
24 }
```

# 例题 08：方盒游戏

```
25  int main() {
26    int T;
27    cin >> T;
28    for(int t = 1; t <= T; ++ t) {
29      int n;
30      memset(score,0xff,sizeof(score));
31      cin >> n;
32      int lastC = 0, segNum = -1;
33      for (int i = 0;i < n; ++i) {
34        int c;
35        cin >> c;
36        if (c != lastC) {
37          segNum ++;
38          segments[segNum].len = 1;
39          segments[segNum].color = c;
40          lastC  = c;
41        } else {
42          segments[segNum].len ++;
43        }
44      }
45      cout << "Case " << t << ": " << ClickBox(0,segNum,0) << endl;
46    }
47    return 0;
48  }
```