

# 法律声明

□ 本课件包括演示文稿、示例、代码、题库、视频和声音等内容，小象学院和主讲老师拥有完全知识产权的权利；只限于善意学习者在本课程使用，不得在课程范围外向任何第三方散播。任何其他人或机构不得盗版、复制、仿造其中的创意及内容，我们保留一切通过法律手段追究违反者的权利。

□ 课程详情请咨询

■ 微信公众号：小象

■ 新浪微博：ChinaHadoop



# 动态规划(上)

---



小象学院  
ChinaHadoop.cn

邹博

# 主要内容

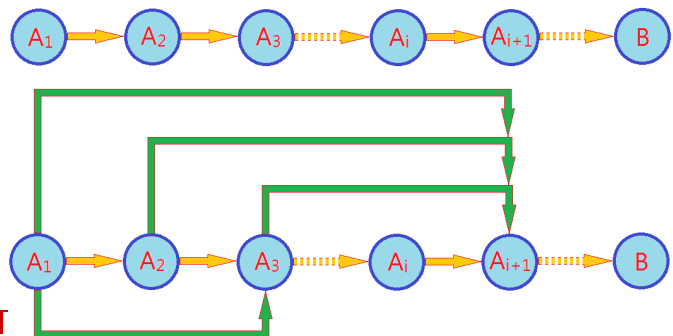
---

- 动态规划和贪心的认识
  - 工具：马尔科夫过程
- 动态规划
  - 最长递增子序列LIS
  - 股票最大收益
  - 任务安排问题
  - 格子取数/走棋盘问题及应用
  - 带陷阱的走棋盘
  - 两次走棋盘问题
  - 字符串的交替连接
  - 词典划分

# 认识论

---

- 认识事物的方法：概念、判断、推理
- 推理中，又分为归纳、演绎。
- 重点考察归纳推理的具体方法。
- 形式化表述：
  - 已知：问题规模为 $n$ 的前提 $A$
  - 求解/求证：未知解 $B$ /结论 $B$
  - 记号：用 $A_n$ 表示“问题规模为 $n$ 的已知条件”



# 对归纳推理的理解

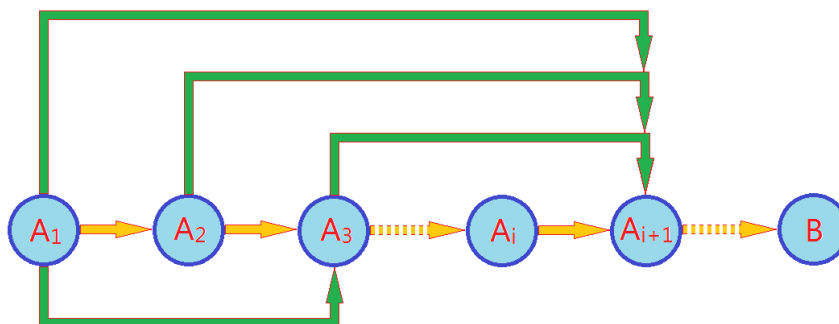
- 若将问题规模降低到0，即已知 $A_0$ ，很容易计算或证明B，则有： $A_0 \rightarrow B$
- 同时，考察从 $A_0$ 增加一个元素，得到 $A_1$ 的变化过程。即： $A_0 \rightarrow A_1$ ；
  - 进一步考察 $A_1 \rightarrow A_2$ ， $A_2 \rightarrow A_3$  .....  $A_i \rightarrow A_{i+1}$
  - 这种方法是(严格的)归纳推理，常常被称作**数学归纳法**。
  - 此时，由于上述推导往往不是等价推导( $A_i$ 和 $A_{i+1}$ 不是互为充要条件)，导致随着 $i$ 的增加，有价值的前提信息越来越少；为避免这一问题，采取如下方案：
    - $\{A_1\} \rightarrow A_2$ ， $\{A_1 A_2\} \rightarrow A_3$  .....  $\{A_1 A_2 \dots A_i\} \rightarrow A_{i+1}$
    - 相对应的，修正后的方法依然是(严格的)归纳推理，有时被称作**第二数学归纳法**。

# 对归纳推理的理解

- 基本归纳法：对于 $A_{i+1}$ ，只需考察前一个状态 $A_i$ 即可完成整个推理过程，它的特点是只要状态 $A_i$ 确定，则计算 $A_{i+1}$ 时不需要考察更前序的状态 $A_1 \dots A_{i-1}$ ，我们将这一模型称为**马尔科夫模型**；

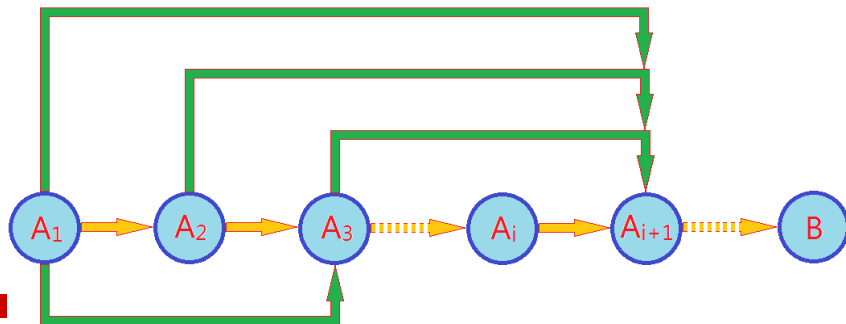


- 高阶归纳法：相应的，对于 $A_{i+1}$ ，需考察前 $i$ 个状态集 $\{A_1 A_2 \dots A_{i-1} A_i\}$ 才可完成整个推理过程，往往称之为**高阶马尔科夫模型**；



- 在计算机算法中，**高阶马尔科夫模型**的推理，叫做“**动态规划**”，而**马尔科夫模型**的推理，对应“**贪心法**”。

# 以上理解的说明



- 无论动态规划还是贪心法，都是根据  $A[0...i]$  计算  $A[i+1]$  的过程
  - 计算  $A[i+1]$  不需要  $A[i+2]$ 、 $A[i+3]$ .....，
  - 一旦计算完成  $A[i+1]$ ，再后面计算  $A[i+2]$ 、 $A[i+3]$ .....时，不会更改  $A[i+1]$  的值。
  - 这即 **无后效性**。
- 亦可以如下理解动态规划：计算  $A[i+1]$  只需要知道  $A[0...i]$  的值，**无需知道  $A[0...i]$  是通过何种途径计算得到的——只需知道它们当前的状态值本身即可。**如果 **将  $A[0...i]$  的全体作为一个整体**，则可以认为动态规划法是 **马尔科夫过程**，而非高阶马尔科夫过程。

# 最长递增子序列LIS

---

- Longest Increasing Subsequence
- 给定长度为N的数组A，计算A的最长的单调递增的子序列(不一定连续)。
  - 如：给定数组A{5,6,7,1,2,8}，则A的LIS为{5,6,7,8}，长度为4。



## 附：使用LCS解LIS问题

---

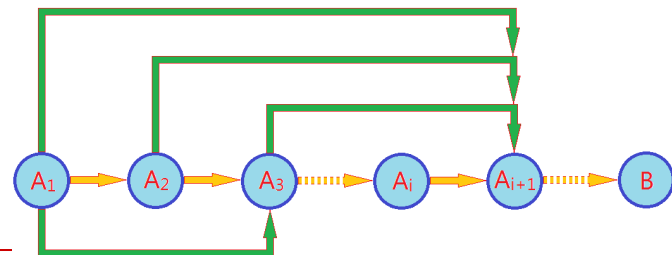
- 原数组为A {5, 6, 7, 1, 2, 8}
- 排序后：A' {1, 2, 5, 6, 7, 8}
- 因为，原数组A的子序列顺序保持不变，而且排序后A'本身就是递增的，这样，就保证了两序列的最长公共子序列的递增特性。如此，若想求数组A的最长递增子序列，其实就是求数组A与它的排序数组A'的最长公共子序列。

# 前缀分析

Array	1	4	6	2	8	9	7
LIS	1	2	3	2	4	5	4

- 0: “1”: 1
- 1: “14”: 2
- 2: “146”: 3
- 3: “12”: 2
- 4: “1468”: 4
- 5: “14689”: 5
- 6: ?

# 最长递增子序列LIS记号



- 长度为 $N$ 的数组记为 $A=\{a_0a_1a_2...a_{n-1}\}$ ;
- 记 $A$ 的前 $i$ 个字符构成的前缀串为 $A_i=a_0a_1a_2...a_{i-1}$ , 以 $a_i$ 结尾的最长递增子序列记做 $L_i$ , 其长度记为 $b[i]$ ;
- 假定已经计算得到了 $b[0,1...,i-1]$ , 如何计算 $b[i]$ 呢?
  - 已知 $L_0L_1.....L_{i-1}$ 的前提下, 如何求 $L_i$ ?

# 求解LIS

Array	1	4	6	2	8	9	7
LIS	1	2	3	2	4	5	4

- 根据定义， $L_i$ 必须以 $a_i$ 结尾；
- 如果将 $a_i$ 分别缀到 $L_0 L_1 \dots L_{i-1}$ 后面，是否允许呢？
  - 如果 $a_i \geq a_j$ ，则可以将 $a_i$ 缀到 $L_j$ 的后面，得到比 $L_j$ 更长的字符串。
- 从而： $b[i] = \{\max(b[j]) + 1, 0 \leq j < i \text{ 且 } a_j \leq a_i\}$ 
  - 计算 $b[i]$ ：遍历在 $i$ 之前的所有位置 $j$ ，找出满足条件 $a_j \leq a_i$ 的最大的 $b[j] + 1$ ；
  - 计算得到 $b[0 \dots n-1]$ 后，遍历所有的 $b[i]$ ，找出最大值即为最大递增子序列的长度。
- 时间复杂度为 $O(N^2)$ 。

# Code1

```
int LIS1(const int* p, int length)
{
    int* longest = new int[length];
    int i, j;

    for(i = 0; i < length; i++)
        longest[i] = 1;

    int nLis = 1;
    for(i = 1; i < length; i++)
    {
        for(j = 0; j < i; j++)
        {
            if(p[j] <= p[i])
            {
                longest[i] = max(longest[i], longest[j]+1);
            }
        }
        nLis = max(nLis, longest[i]);
    }
    delete[] longest;
    return nLis;
}
```

# LIS的思考

---

□ 思考：如何求最大递增子序列本身？

■ 记录前驱

Array	1	4	6	2	8	9	7
LIS	1	2	3	2	4	5	4

# LIS Code split

```
int _tmain(int argc, _TCHAR* argv[])
{
    int array[] = {1,4,5,6,2,3,8,9,10,11,12,12,1};
    int size = sizeof(array)/sizeof(int);
    int* pre = new int[size];
    int nIndex;
    int max = LIS(array, size, pre, nIndex);
    vector<int> lis;
    GetLIS(array, pre, nIndex, lis);
    delete[] pre;
    cout << max << endl;
    Print(&lis.front(), (int)lis.size());
    return 0;
}
```

```
void GetLIS(const int* array, const int* pre,
           int nIndex, vector<int>& lis)
{
    while(nIndex >= 0)
    {
        lis.push_back(array[nIndex]);
        nIndex = pre[nIndex];
    }
    reverse(lis.begin(), lis.end());
}
```

```
#include <vector>
#include <algorithm>
using namespace std;
```

```
int LIS(const int* p, int length, int* pre, int& nIndex)
{
    int* longest = new int[length];
    int i, j;

    for(i = 0; i < length; i++)
    {
        longest[i] = 1;
        pre[i] = -1;
    }

    int nLis = 1;
    nIndex = 0;
    for(i = 1; i < length; i++)
    {
        for(j = 0; j < i; j++)
        {
            if(p[j] <= p[i])
            {
                if(longest[i] < longest[j]+1)
                {
                    longest[i] = longest[j]+1;
                    pre[i] = j;
                }
            }
        }
        if(nLis < longest[i])
        {
            nLis = longest[i];
            nIndex = i;
        }
    }

    delete[] longest;

    return nLis;
}
```

```
void Print(int* p, int size)
{
    for(int i = 0; i < size; i++)
        cout << p[i] << '\t';
    cout << '\n';
}
```

# $O(N\log N)$ 的最长递增子序列算法

---

- ☐ 对于数组  $A=\{1,4,6,2,8,9,7\}$
- ☐ 1
- ☐ 1,4
- ☐ 1,4,6
- ☐ 1,2,6
- ☐ 1,2,6,8
- ☐ 1,2,6,8,9
- ☐ 1,2,6,7,9



# Code2

```
int LIS(const int* a, int size)
{
    int* b = new int[size];
    int s = 0;
    int i;
    for(i = 0; i < size; i++)
        Insert(b, s, a[i]);
    delete[] b;
    return s;
}

int _tmain(int argc, _TCHAR* argv[])
{
    int a[] = {5, 6, 7, 1, 2, 8};
    int size = sizeof(a) / sizeof(int);
    int len = LIS(a, size);
    cout << len << endl;
    return 0;
}
```

```
void Insert(int* a, int& size, int x)
{
    if(size <= 0)
    {
        a[0] = x;
        size++;
        return;
    }
    int low = 0;
    int high = size-1;
    int mid;
    while(low <= high)
    {
        mid = (low+high)/2;
        if(x < a[mid])
            high = mid-1;
        else if(x >= a[mid])
            low = mid+1;
    }

    if(low >= size)
    {
        a[size] = x;
        size++;
    }
    else
    {
        if(a[low] < x)
            a[low+1] = x;
        else
            a[low] = x;
    }
}
```

# 进一步思考

---

- 如果使用该贪心法，如何计算LIS本身？
- 方案：仍然是记录前驱。

# Code3

```
int LIS(const int* a, int size)
{
    int* b = new int[size];    //足够大的缓冲区
    int s = 0;                //缓冲区有效长度
    int* pre = new int[size]; //前驱
    int i;
    for(i = 0; i < size; i++)
        Insert2(b, s, a, i, pre);

    //计算LIS本身
    int cur = b[s-1]; //LIS的最后一个元素
    i = 0;
    while(cur != -1)
    {
        b[i] = a[cur]; //b被挪用于计算LIS
        cur = pre[cur];
        i++;
    }
    reverse(b, b+s);
    Print(b, s);
    delete[] b;
    delete[] pre;
    return s;
}

int _tmain(int argc, _TCHAR* argv[])
{
    int a[] = {1, 4, 6, 2, 8, 9, 7};
    int size = sizeof(a) / sizeof(int);
    int len = LIS(a, size);
    cout << len << endl;
    return 0;
}
```

```
//在缓冲区b[0...size-1]中插入元素a[i]
void Insert2(int* b, int& size, const int* a, int i, int* pre)
{
    if(size <= 0)
    {
        b[0] = i;
        size++;
        pre[i] = -1;
        return;
    }
    int low = 0;
    int high = size-1;
    int mid;
    while(low <= high)
    {
        mid = (low+high)/2;
        if(a[i] < a[b[mid]])
            high = mid-1;
        else if(a[i] >= a[b[mid]])
            low = mid+1;
    }

    if(low >= size)
    {
        b[size] = i;
        pre[i] = b[size-1];
        size++;
    }
    else
    {
        if(a[b[low]] < a[i])
        {
            b[low+1] = i;
            pre[i] = b[low];
        }
        else
        {
            b[low] = i;
            pre[i] = (low==0) ? -1 : b[low-1];
        }
    }
}
```

# 股票最大收益

- 给定数组 $A[0...N-1]$ ，其中 $A[i]$ 表示某股票第 $i$ 天的价格。如果允许最多只进行一次交易（先买一次，再卖一次），请计算何时买卖达到最大收益，返回最大收益值。
  - 如： $[7,1,5,3,6,4]$ ，则最大收益为 $6-1=5$ 。
  - 如： $[7,6,4,3,1]$ ，则最大收益为0。
  - 一路下跌，则最好的方法是不进行交易。

# 思路分析

---

- 若在第 $i$ 天卖出，则应该在哪天买入更好？
- 答：在 $A[0 \dots i-1]$ 的最小值处买入。

# Code

```
int MaxProfit(const int* prices, int size)
{
    int p = 0;
    int mn = prices[0];
    for (int i = 1; i < size; i++)
    {
        mn = min(mn, prices[i-1]); //p[0...i-1]最小值
        p = max(p, prices[i] - mn);
    }
    return p;
}

int _tmain(int argc, _TCHAR* argv[])
{
    int prices[] = {7, 1, 5, 3, 6, 4};
    cout << MaxProfit(prices, sizeof(prices)/sizeof(int)) << endl;
    return 0;
}
```

# 股票最大收益II

- 给定数组 $A[0...N-1]$ ，其中 $A[i]$ 表示某股票第 $i$ 天的价格。如果允许最多只进行**K次交易**（先买一次再卖一次算一次交易），请计算何时买卖达到最大收益，返回最大收益值。
  - 规定买卖**不能嵌套**，即：买入后，要先卖出才可再买。
- 如： $A=[7,1,5,3,6,4]$ ， $K=3$ ，则在1,3处买入，5,6处卖出，最大收益为7。

# 算法分析

- $dp[k,i]$  表示最多  $k$  次交易，且在第  $i$  天卖出的最大收益。
- 在第  $i$  天，有两种选择：要么卖出股票，要么不卖出股票，从而得到状态转移方程：

$$dp[k][i] = \max \left( \begin{array}{l} dp[k][i-1] \\ dp[k-1][j] + prices[i] - prices[j], j \in [0, i-1] \end{array} \right)$$



# Code

```
int MaxProfitK(const int* prices, int size, int K)
{
    vector<vector<int>> dp(K+1, vector<int>(size, 0));
    int k, i, j;
    for(k = 1; k <= K; k++)
    {
        for(i = 1; i < size; i++)
        {
            dp[k][i] = dp[k][i-1];
            for(j = 0; j < i; j++)
            {
                dp[k][i] = max(dp[k][i], dp[k-1][j] + prices[i]-prices[j]);
            }
        }
    }
    return dp[K][size-1];
}

int _tmain(int argc, _TCHAR* argv[])
{
    int prices[] = {7, 1, 5, 3, 6, 4};
    cout << MaxProfitK(prices, sizeof(prices)/sizeof(int), 3) << endl;
    return 0;
}
```

# 进一步分析

- 代码实现中，交易次数K固定，在计算  $dp[k][i]$  时，需要计算  $i$  次循环；而状态转移方程可以进一步简化：

$$dp[k][i] = \max \left( \begin{array}{l} dp[k][i-1] \\ dp[k-1][j] + prices[i] - prices[j], \quad j \in [0, i-1] \end{array} \right)$$

$$\Rightarrow dp[k][i] = \max \left( \begin{array}{l} dp[k][i-1] \\ prices[i] + \max_{j \in [0, i-1]} (dp[k-1][j] - prices[j]) \end{array} \right)$$

# Code2

```
int MaxProfitK2(const int* prices, int size, int K)
{
    vector<vector<int> > dp(K+1, vector<int>(size, 0));
    int k, i;
    int mx;
    for(k = 1; k <= K; k++)
    {
        mx = dp[k-1][0]-prices[0];
        for(i = 1; i < size; i++)
        {
            dp[k][i] = max(dp[k][i-1], mx+prices[i]);
            mx = max(mx, dp[k-1][i]-prices[i]);
        }
    }
    return dp[K][size-1];
}
```

# 股票最大收益小结

---

- 时间复杂度是如何从 $O(kN^2)$ 降为 $O(k^2N)$ 的？
- 如果需要得到最大收益的买卖时间，如何修改代码？

# 任务安排

- 给定一台有 $m$ 个储存空间的单进程机器；现有 $n$ 个请求：第 $i$ 个请求计算时需要占用 $R[i]$ 个空间，计算完成后，储存计算结果需要占用 $O[i]$ 个空间(其中 $O[i] < R[i]$ )。问如何安排这 $n$ 个请求的顺序，使得所有请求都能完成。
- 如： $m=14$ ， $n=2$ ， $R[1,2]=[10,8]$ ， $O[1,2]=[5,6]$ 。可以先运行第一个任务，计算时占用10个空间，计算完成后占用5个空间，剩余9个空间执行第二个任务；但如果先运行第二个任务，则计算完成后仅剩余8个空间，第一个任务的计算空间就不够了。

# 算法分析

- 第k个任务的计算占用空间加上前面k-1个任务的空间占用量之和，越小越好。从而：

$$\begin{cases} O_1 + O_2 + \dots + O_j + \dots + O_{k-1} + R_k \\ O_1 + O_2 + \dots + O_k + \dots + O_{k-1} + R_j \end{cases}$$

$$\Rightarrow O_j + R_k \leq O_k + R_j$$

$$\Rightarrow R_k - O_k \leq R_j - O_j$$

- 得：将任务按照 $R[i]-O[i]$ 降序排列即可。

# Code

```
int _tmain(int argc, _TCHAR* argv[])
{
    const int N = 2;    //任务数目
    const int M = 14;   //内存数目
    int R[N] = {10, 8};
    int O[N] = {5, 6};
    bool b = IsTaskable(N, M, R, O);
    cout << (b ? "Yes\n" : "No\n");
    return 0;
}
```

```
typedef struct tagTask
{
    int taskID;
    int RO;

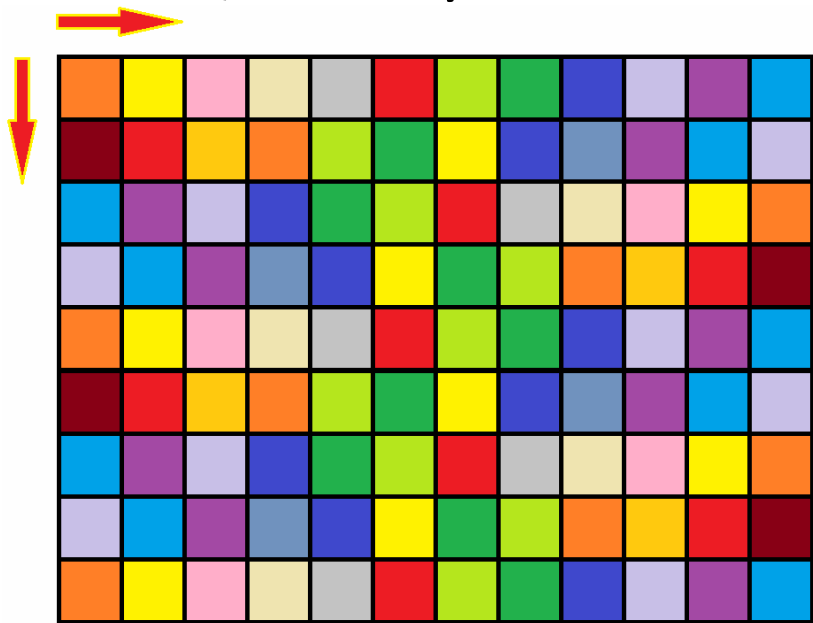
    static bool Compare(const tagTask& t1, const tagTask& t2)
    {
        return t1.RO > t2.RO;
    }
} STask;

bool IsTaskable(int N, int M, const int* R, const int* O)
{
    STask* st = new STask[N];
    int i;
    for(i = 0; i < N; i++)
    {
        st[i].taskID = i;
        st[i].RO = R[i] - O[i];
    }
    sort(st, st+N, STask::Compare);

    int occupy = 0;
    bool bOK = true;
    int k;
    for(i = 0; i < N; i++)
    {
        k = st[i].taskID;
        if(occupy + R[k] > M)
        {
            bOK = false;
            break;
        }
        occupy += O[k];
    }
    delete[] st;
    return bOK;
}
```

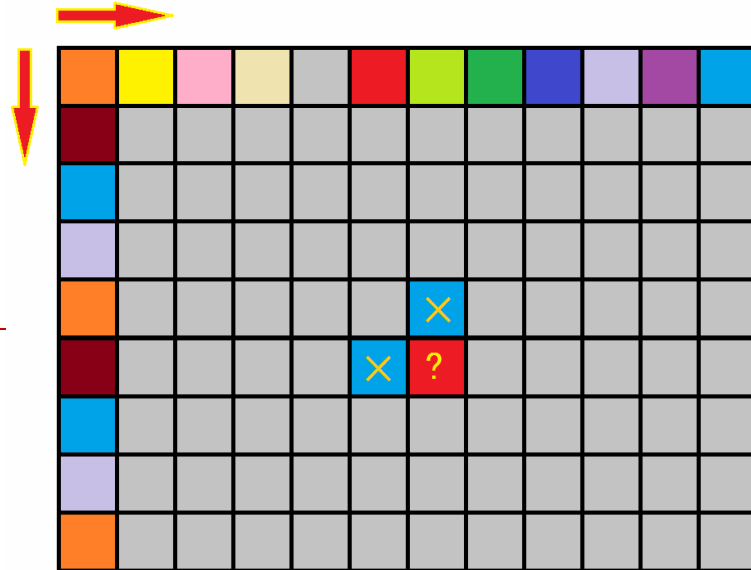
# 走棋盘/格子取数

- 给定 $m \times n$ 的矩阵，每个位置是一个非负整数，在左上角放一机器人，它每次只能朝右和下走，直到右下角，求所有路径中，总和最小的那条路径。





# 状态转移方程



□ 走的方向决定了同一个格子不会经过两次。

■ 若当前位于 $(x,y)$ 处，它来自于哪些格子呢？

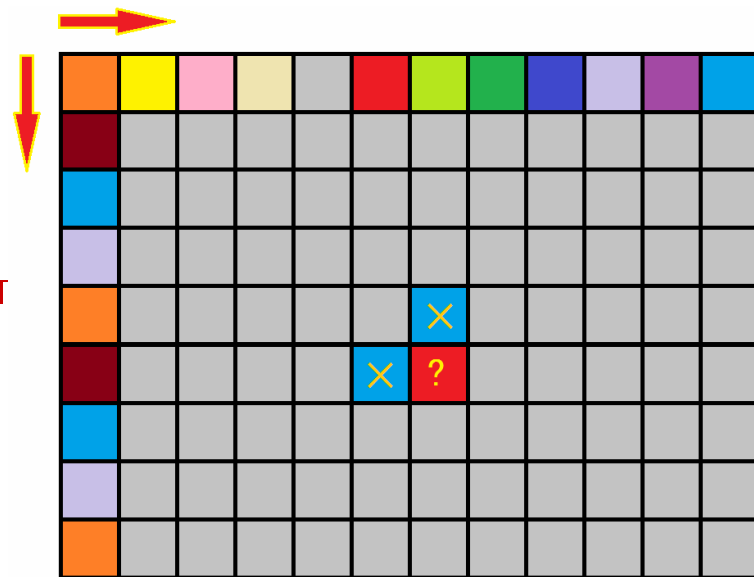
■  $dp[0,0]=a[0,0]$  / 第一行(列)累积

■  $dp[x,y] = \min(dp[x-1,y]+a[x,y], dp[x,y-1]+a[x,y])$

■ 即：  $dp[x,y] = \min(dp[x-1,y], dp[x,y-1]) + a[x,y]$

□ 思考：若将上述问题改成“求从左上到右下的最大路径”呢？

# 状态转移方程



□ 状态转移方程：

$$\begin{cases} dp(i, 0) = \sum_{k=0}^i chess[k][0] \\ dp(0, j) = \sum_{k=0}^j chess[0][k] \\ dp(i, j) = \min(dp(i-1, j), dp(i, j-1)) + chess[i][j] \end{cases}$$

□ 滚动数组：

$$\begin{cases} dp(j) = \sum_{k=0}^j chess[0][k] \\ dp(j) = \min(dp(j), dp(j-1)) + chess[i][j] \end{cases}$$

# Code

```
int MinPath(vector<vector<int> >& chess, int M, int N)
{
    vector<int> pathLength(N);
    int i, j;

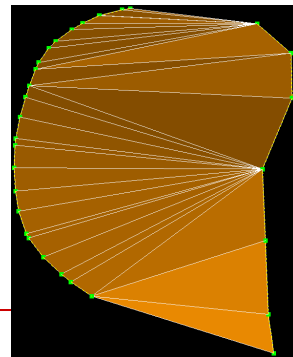
    //初始化
    pathLength[0] = chess[0][0];
    for(j = 1; j < N; j++)
        pathLength[j] = pathLength[j-1] + chess[0][j];

    //依次计算每行
    for(i = 1; i < M; i++)
    {
        pathLength[0] += chess[i][0];
        for(j = 1; j < N; j++)
        {
            if(pathLength[j-1] < pathLength[j])
                pathLength[j] = pathLength[j-1] + chess[i][j];
            else
                pathLength[j] += chess[i][j];
        }
    }
    return pathLength[N-1];
}

int _tmain(int argc, _TCHAR* argv[])
{
    const int M = 10;
    const int N = 8;
    vector<vector<int> > chess(M, vector<int>(N));

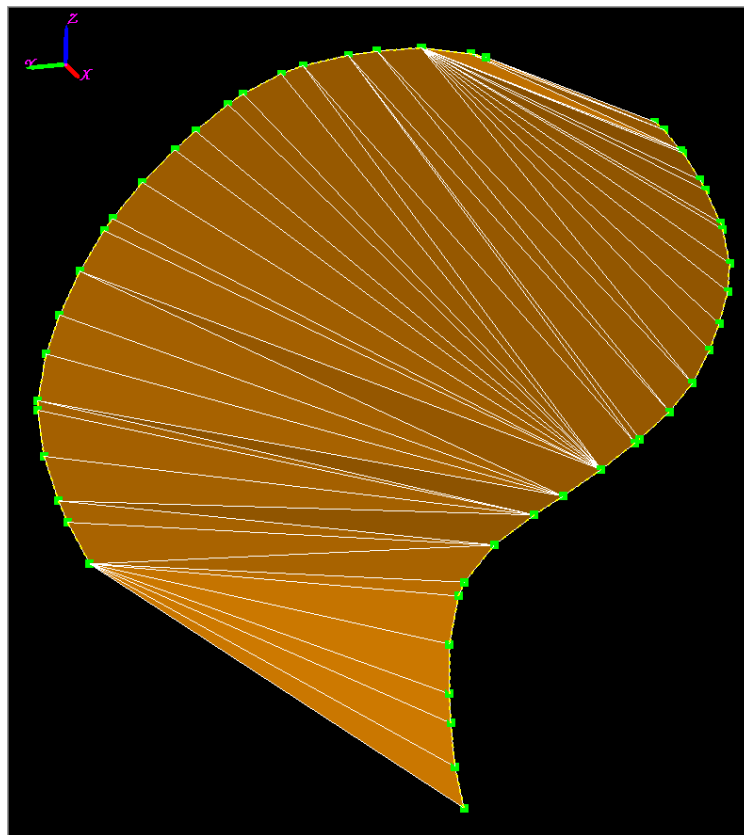
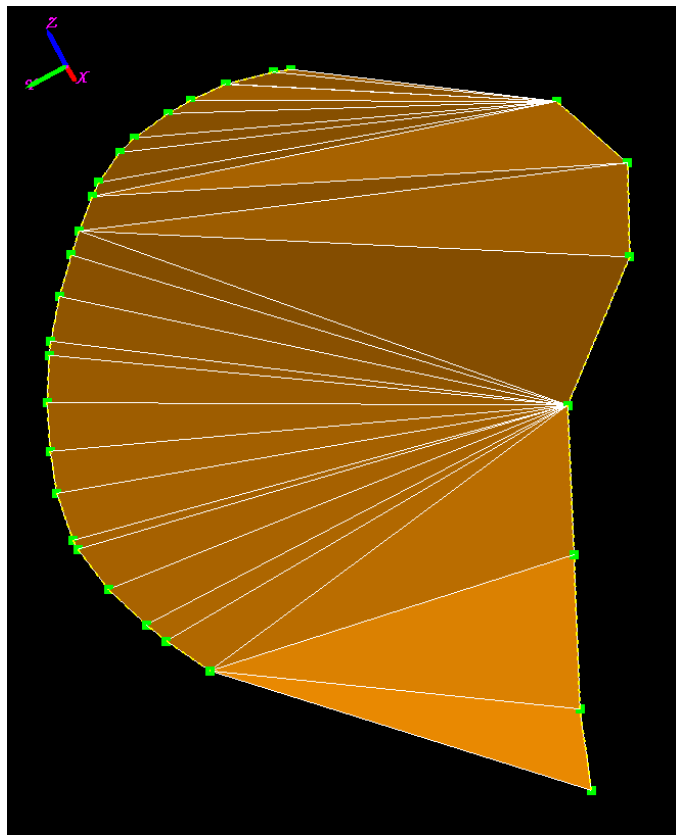
    //初始化棋盘: (随机给定)
    int i, j;
    for(i = 0; i < M; i++)
    {
        for(j = 0; j < N; j++)
            chess[i][j] = rand() % 100;
    }
    cout << MinPath(chess, M, N) << endl;
    return 0;
}
```

# 实践应用



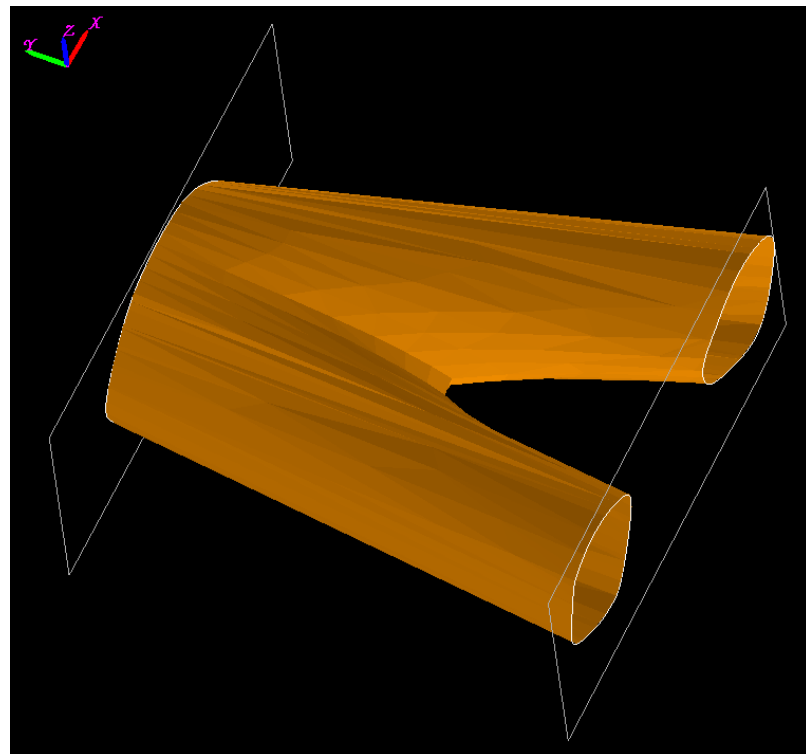
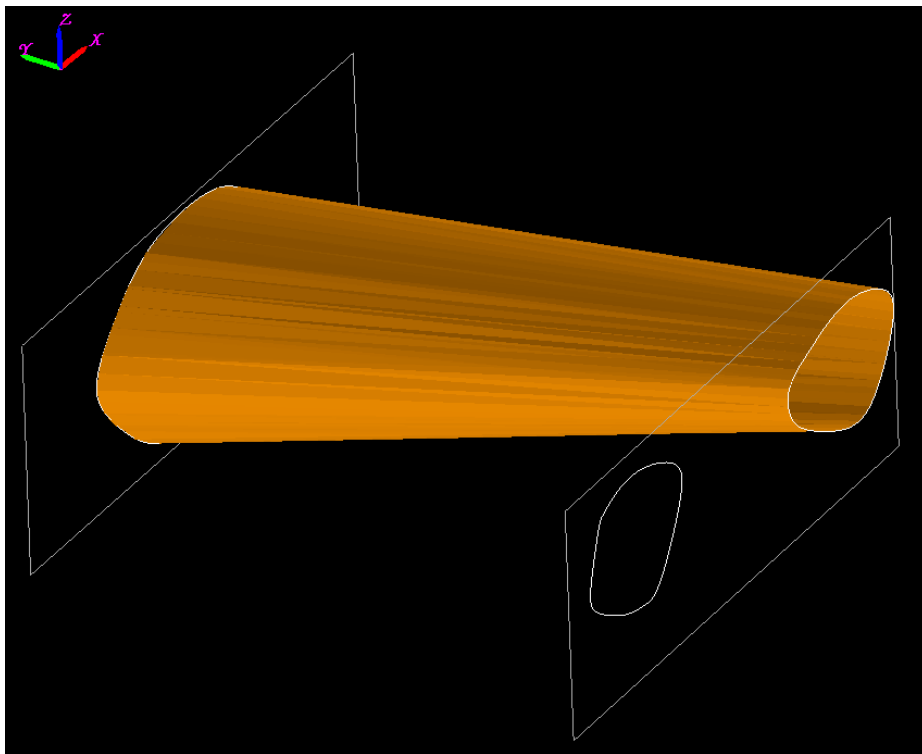
- 给定空间两条曲线，设计经过这两条曲线的曲面，使得该曲面是“最合理”的。
- 探讨如下方案：两条曲线上的点序列，记做  $P1[0...N-1]$  和  $P2[0...M-1]$ ，计算  $P1[i]$  和  $P2[j]$  的距离，从而得到二维表格  $T[N][M]$ 。从  $T[0][0]$  开始出发，只能向右和向下走(曲面不能自相交)，从  $(0,0)$  到  $(N-1,M-1)$  的最短路径，就是曲面的一种“合理连接”。

# 实践：实践中的应用



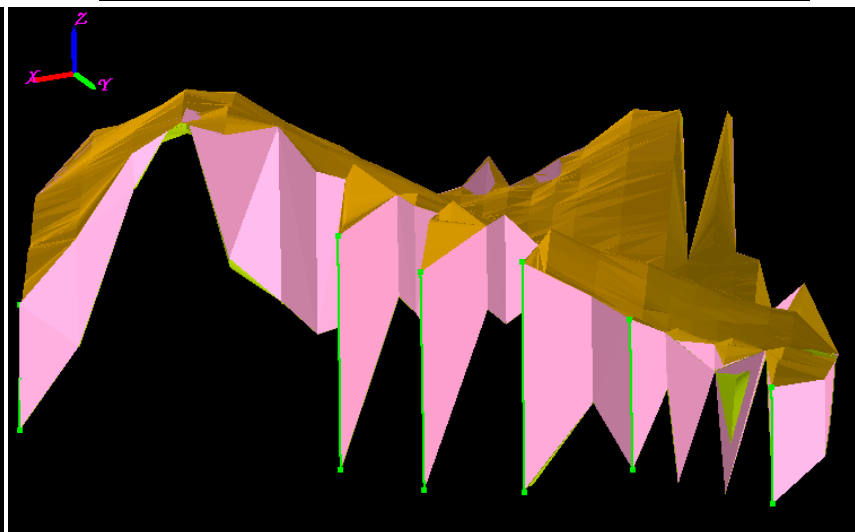
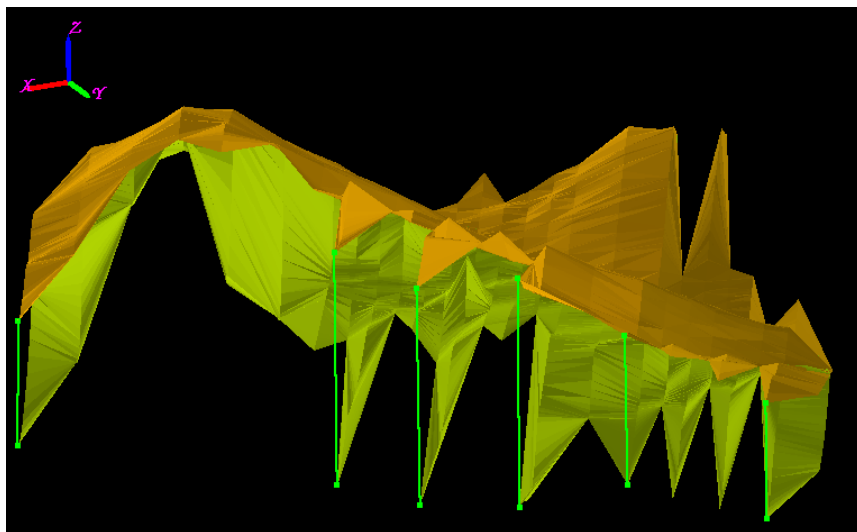
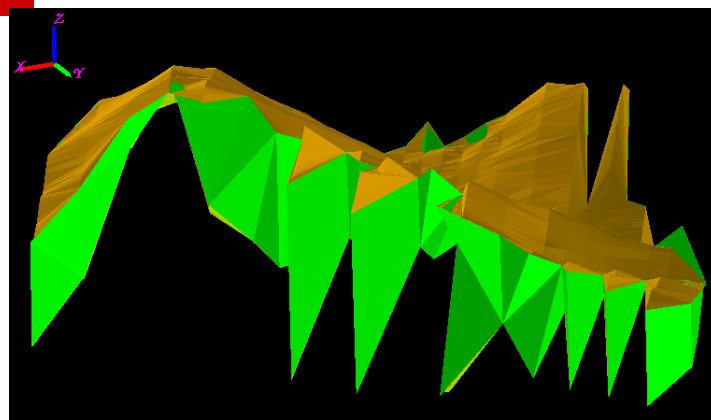
# 如果三维曲线是封闭线...

---

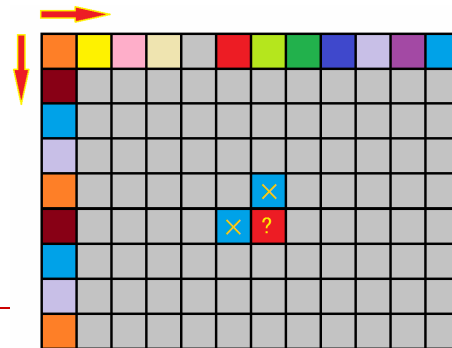


# 经过引导线的曲面——带约束的走棋盘

- 右上：未使用引导线
- 左下：输入的引导线
- 右下：过引导线的曲面



# 动态规划递推式可以得到？



□ 令 $dp(x,y)$ 为当前位于 $(x,y)$ 时有多少种可行路径，则： $dp(x,y)=dp(x-1,y)+dp(x,y-1)$

$$dp(x, y) = dp(x-1, y) + dp(x, y-1)$$

增加“两个坐标值加和” $\rightarrow dp(x+y, x, y) = dp(x+y-1, x-1, y) + dp(x+y-1, x, y-1)$

删除最后一维 $\rightarrow dp(x+y, x) = dp(x+y-1, x-1) + dp(x+y-1, x)$

令 $t=x+y$  $\rightarrow dp(t, x) = dp(t-1, x-1) + dp(t-1, x)$

换个表达方式 $\rightarrow C_t^x = C_{t-1}^{x-1} + C_{t-1}^x$

令 $n=t, m=x$  $\rightarrow C_n^m = C_{n-1}^{m-1} + C_{n-1}^m$



# 带陷阱的走棋盘

- 有一个 $n*m$ 的棋盘网格，机器人最开始在左上角，机器人每一步只能往右或者往下移动。棋盘中有有些格子是禁止机器人踏入的，该信息存放在二维数组blocked中，如果blocked[i][j]为true，那么机器人不能踏入格子(i,j)。请计算有多少条路径能让机器人从左上角移动到右下角。

# 状态转移方程

---

- $dp[i][j]$  表示从起点到  $(i,j)$  的路径条数。
- 只能从左边或者上边进入一个格子
- 如果  $(i,j)$  被占用
  - $dp[i][j]=0$
- 如果  $(i,j)$  不被占用
  - $dp[i][j]=dp[i-1][j]+dp[i][j-1]$

# Code

```
int _tmain(int argc, _TCHAR* argv[])
{
    const int M = 4;
    const int N = 5;
    vector<vector<bool>> chess(M, vector<bool>(N));

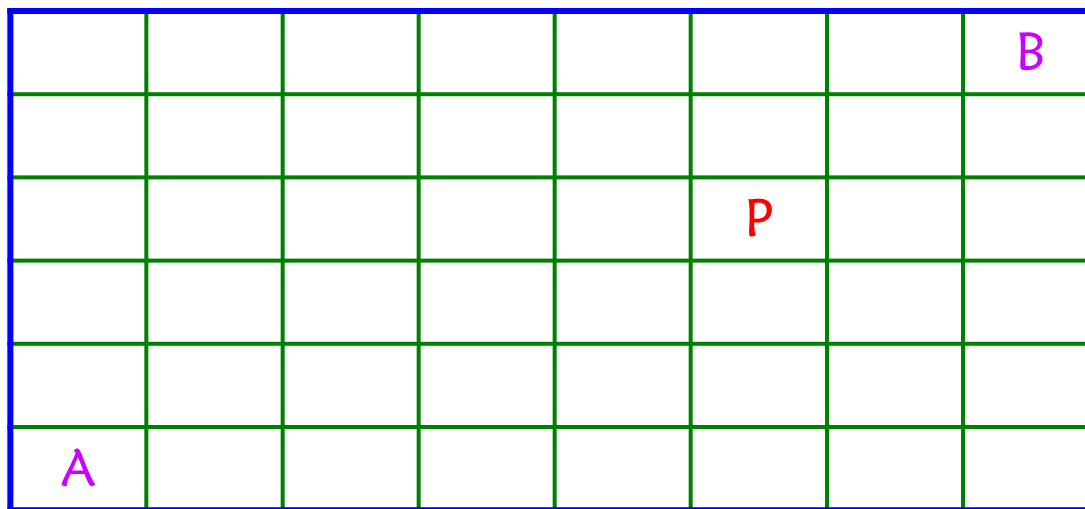
    int i, j;
    for(i = 0; i < M; i++)
    {
        for(j = 0; j < N; j++)
            chess[i][j] = ((rand() % 5) != 0);
    }
    cout << PathNumber(chess) << endl;
    return 0;
}
```

```
int PathNumber(vector<vector<bool>> & chess)
{
    int M = (int)chess.size();    //行
    int N = (int)chess[0].size(); //列
    vector<int> pathNumber(N, 0);
    int i, j;
    pathNumber[0] = chess[0][0] ? 1 : 0;
    for(j = 1; j < N; j++)
    {
        //当前未阻断, 并且前一个值可达
        if(chess[0][j] && (pathNumber[j-1] == 1))
            pathNumber[j] = 1;
    }

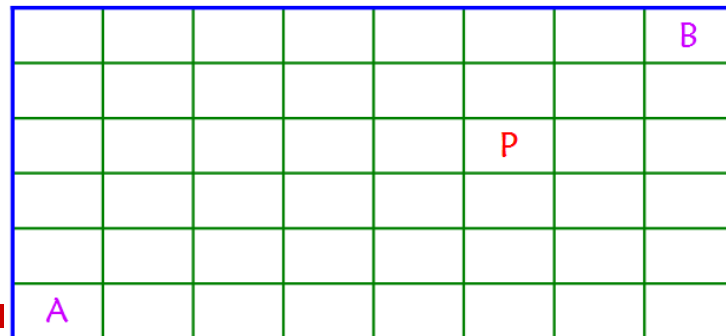
    for(i = 1; i < M; i++)
    {
        if(!chess[i][0])
            pathNumber[0] = 0;
        for(j = 1; j < N; j++)
        {
            if(!chess[i][j]) //当前被阻断
                pathNumber[j] = 0;
            else //Cur=Left+Top
                pathNumber[j] += pathNumber[j-1];
        }
    }
    return pathNumber[N-1];
}
```

# 陷阱走棋盘

- 在 $8 \times 6$ 的矩阵中，每次只能向上或向右移动一格，并且不能经过P。试计算从A移动到B一共有多少种走法。



# 解题过程



- 从A到B共需要移动12步，其中7步向右，5步向上，可行走法数目为  $C_{12}^5 = 792$
- 从A到P共需要8步，其中5步向右，3步向上，可行走法数目为  $C_8^5 = 56$
- 从P到B共需要4步，其中2步向右，2步向上，可行走法数目为  $C_4^2 = 6$
- 则，从A到B经过P的路线有  $56 * 6 = 336$  种；
- 从A到B不经过P的路线有  $792 - 336 = 456$  种。

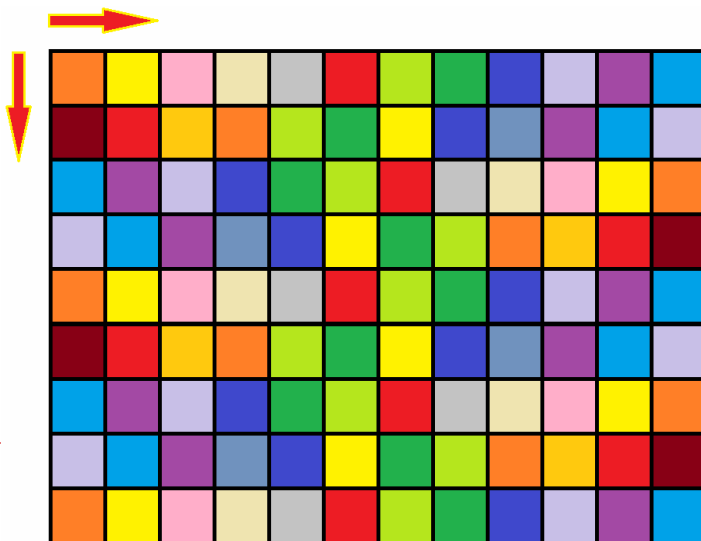
# 方格的可行路径数目

							B
					P		
A							

1	6	21	56	126	196	294	456
1	5	15	35	70	70	98	162
1	4	10	20	35	0	28	64
1	3	6	10	15	21	28	36
1	2	3	4	5	6	7	8
1	1	1	1	1	1	1	1

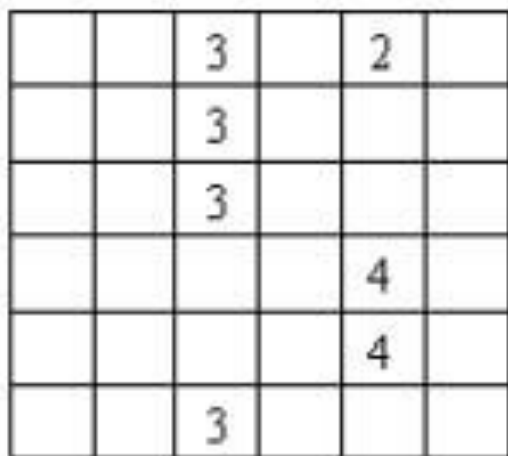
# 两次走棋盘

- 给定 $m \times n$ 的矩阵，每个位置为非负权值，机器人从左上角开始，每次只能朝右和下，走到右下角。然后每次只能朝左和朝上，走回左上角。求权值总和最大的路径。
- 若相同格子走过两次，则该位置权值只算一次。

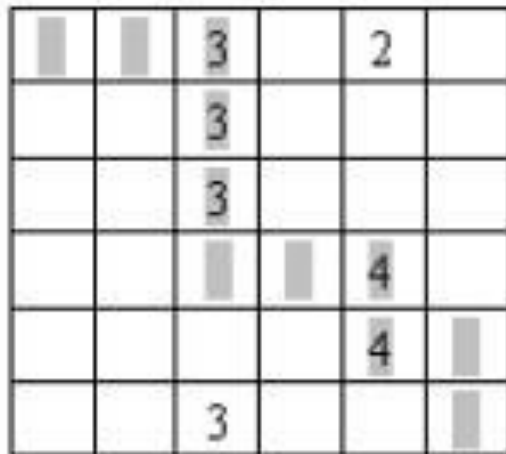


# 分析

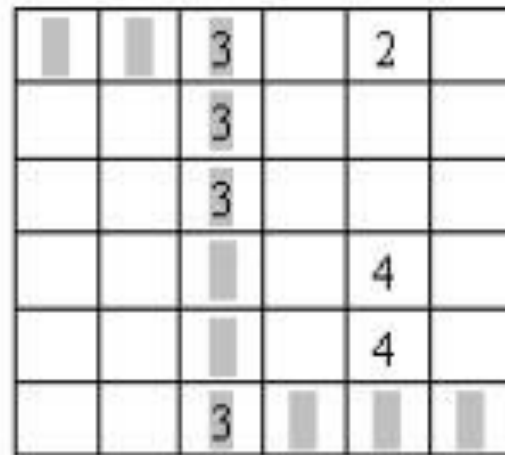
## □ 贪心不能解决问题



图一



图二



图三



# 分析格网棋盘的特点

- 考察 $5 \times 7$ 的矩阵棋盘C，其中， $C[i,j]$ 表示的值 $v$ 表示第 $v$ 步能够到达的位置。

0	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	X

0	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	X

## 举例说明状态的定义

- 在经过8步后，肯定处于C中编号为8的位置。而C中共有3个编号为8的，它们分别是C的第2、3、4行。故假设第1次经过8步走到了C中的第2行，第2次经过8步走到了C中的第3行，用 $dp[8,2,3]$ 表示。
- 用 $dp[s,i,j]$ 记录两次所走的路径获得的最大值，其中s表示走的步数，i和j表示在s步后第1次走的位置和第2次走的位置。由于 $s=m+n-2$ ， $0 \leq i < n$ ， $0 \leq j < m$ ，所以共有 $O(n^3)$ 个状态。

# 状态转移函数的定义

---

0	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	X

# 状态转移函数的定义

---

0	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	X

# 完整的状态转移函数

- if( $i \neq j$ )
  - $DP[s,i,j] = \text{Max}(\$ 
    - $DP[s-1,i-1,j-1],$
    - $DP[s-1,i-1,j],$
    - $DP[s-1,i,j-1],$
    - $DP[s-1,i,j])$
    - $+ W[i,s-i] + W[j,s-j]$

0	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	X

- Else
  - $DP[s,i,j] = \text{Max}(\$ 
    - $DP[s-1,i-1,j-1],$
    - $DP[s-1,i-1,j],$
    - $DP[s-1,i,j])$
    - $+ W[i,s-i]$

0	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	X

- 其中  $W[x,y]$  表示棋盘位置  $(x,y)$  的权值。

# Code

```
const int N = 202;
const int inf = 1000000000; //无穷大
int dp[N * 2][N][N];
bool IsValid(int step, int x1, int x2, int n)
{
    int y1 = step - x1, y2 = step - x2;
    return ((x1 >= 0) && (x1 < n) && (x2 >= 0) && (x2 < n) && (y1 >= 0) && (y1 < n) && (y2 >= 0) && (y2 < n));
}

int GetValue(int step, int x1, int x2, int n)
{
    return IsValid(step, x1, x2, n) ? dp[step][x1][x2] : (-inf);
}

//dp[step][i][j]表示在第step步两次分别在第i行和第j行的最大得分
int MinPathSum(int a[N][N], int n)
{
    int P = n * 2 - 2; //最终的步数
    int i, j, step;

    //不能到达的位置 设置为负无穷大
    for(i = 0; i < n; ++i)
    {
        for(j = i; j < n; ++j)
        {
            dp[0][i][j] = -inf;
        }
    }
    dp[0][0][0] = a[0][0];

    for(step = 1; step <= P; ++step)
    {
        for(i = 0; i < n; ++i)
        {
            for(j = i; j < n; ++j)
            {
                dp[step][i][j] = -inf;
                if(!IsValid(step, i, j, n)) //非法位置
                    continue;
                //对于合法的位置进行dp
                if(i != j)
                {
                    dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i - 1, j - 1, n));
                    dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i - 1, j, n));
                    dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i, j - 1, n));
                    dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i, j, n));
                    dp[step][i][j] += a[i][step - i] + a[j][step - j]; //不在同一个格子, 加两个数
                }
                else
                {
                    dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i - 1, j - 1, n));
                    dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i - 1, j, n));
                    dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i, j, n));
                    dp[step][i][j] += a[i][step - i]; // 在同一个格子里, 只能加一次
                }
            }
        }
    }
    return dp[P][n - 1][n - 1];
}
```

# Code split

```
for(step = 1; step <= P; ++step)
{
    for(i = 0; i < n; ++i)
    {
        for(j = i; j < n; ++j)
        {
            dp[step][i][j] = -inf;
            if(!IsValid(step, i, j, n)) //非法位置
                continue;
            //对于合法的位置进行dp
            if(i != j)
            {
                dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i - 1, j - 1, n));
                dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i - 1, j, n));
                dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i, j - 1, n));
                dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i, j, n));
                dp[step][i][j] += a[i][step - i] + a[j][step - j];
            }
            else
            {
                dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i - 1, j - 1, n));
                dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i - 1, j, n));
                dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i, j, n));
                dp[step][i][j] += a[i][step - i]; // 在同一个格子里, 只能加一次
            }
        }
    }
}
return dp[P][n - 1][n - 1];
```

```
int MinPathSum(int a[N][N], int n)
{
    int P = n * 2 - 2; //最终的步数
    int i, j, step;

    //不能到达的位置 设置为负无穷大
    for(i = 0; i < n; ++i)
    {
        for(j = i; j < n; ++j)
        {
            dp[0][i][j] = -inf;
        }
    }
    dp[0][0][0] = a[0][0];
```

# 字符串的交替连接

- 输入三个字符串s1、s2和s3，判断第三个字符串s3是否由前两个字符串s1和s2交错而成，即不改变s1和s2中各个字符原有的相对顺序，例如当s1=“aabcc”，s2=“dbbca”，s3=“aadbcbcbac”时，则输出true，但如果s3=“accabdbbca”，则输出false。
- 换个表述：
  - s1和s2是s3的子序列，且 $s1 \cup s2 = s3$



# 问题说明

---

- 若s1和s2没有字符重复：遍历s3的同时，考察是否是s1和s2的字符即可；
- 若字符重复：可以用压栈的方式解决；
- 此外，还能用动态规划，代码更为简洁。

# 状态转移函数

- 为算法表述方便，从1开始数：
- 令 $dp[i,j]$ 表示 $s3[1...i+j]$ 是否由 $s1[1...i]$ 和 $s2[1...j]$ 的字符组成：即 $dp[i,j]$ 取值范围为true/false
  - $s1[i]==s3[i+j]$ 且 $dp[i-1,j]$ 为真，则 $dp[i][j]$ 为真；
  - $s2[j]==s3[i+j]$ 且 $dp[i,j-1]$ 为真，则 $dp[i][j]$ 为真；
  - 其它情况， $dp[i][j]$ 为假。
- 可以使用滚动数组降低空间复杂度。

# Code

```
bool IsInterlace(const char* str1, const char* str2, const char* str)
{
    int M = (int)strlen(str1);
    int N = (int)strlen(str2);
    int S = (int)strlen(str);
    if(M+N != S)
        return false;
    vector<vector<bool> > p(M+1, vector<bool>(N+1));
    int i, j;
    p[0][0] = true;
    for(i = 1; i <= M; i++) //首列
        p[i][0] = (p[i-1][0] && (str1[i-1] == str[i-1]));
    for(j = 1; j <= N; j++) //首行
        p[0][j] = (p[0][j-1] && (str2[j-1] == str[j-1]));

    for(i = 1; i <= M; i++)
    {
        for(j = 1; j <= N; j++)
        {
            p[i][j] = (p[i-1][j] && (str[i+j-1] == str1[i-1]))
                || (p[i][j-1] && (str[i+j-1] == str2[j-1]));
        }
    }
    return p[M][N];
}
```

## Code2

```
bool IsInterlace2(const char* str1, const char* str2, const char* str)
{
    int M = (int)strlen(str1);
    int N = (int)strlen(str2);
    int S = (int)strlen(str);
    if(M+N != S)
        return false;
    if(M < N)
        return IsInterlace2(str2, str1, str);
    vector<bool> p(N+1);
    int i, j;
    p[0] = true;
    for(j = 1; j <= N; j++) //首行
        p[j] = (p[j-1] && (str2[j-1] == str[j-1]));

    for(i = 1; i <= M; i++)
    {
        p[0] = (p[0] && (str1[i-1] == str[i-1]));
        for(j = 1; j <= N; j++)
        {
            p[j] = (p[j] && (str[i+j-1] == str1[i-1]))
                || (p[j-1] && (str[i+j-1] == str2[j-1]));
        }
    }
    return p[N];
}
```

# Word Break

---

## □ 分割词汇

□ 给定一组字符串构成的字典dict和某字符串str，将str增加若干空格构成句子，使得str被分割后的每个词都在字典dict中。返回满足要求的分割str后的所有句子。如：

■ str="catsanddog",

■ dict=["cat","cats","and","sand","dog"]

■ 返回：["cats and dog","cat sand dog"]。

# 分割词汇问题分析

- 记长度为*i*的前缀串 $str[0...i-1]$ 有至少一个可行划分, 用布尔变量 $dp[i]$ 表示, 则:  
$$dp[i] = \exists j (dp[j] \& \& str[j \cdots i-1] \in dict, 0 \leq j \leq i-1)$$
- catsanddog
- 初始条件 $dp[0]=true$ ;
  - 若划分到最后是空串, 则说明该划分是有效的; 即默认空串即在字典中。
- 若只需要计算 $str$ 是否可以划分成句子, 直接返回 $dp[size]$ 即可; 该题目还需要返回所有的划分, 所以, 需要保存“前驱”。
  - 代码中将其记录为棋盘 $chess$ 。

# Code1

## □ DP

```
bool WordBreak1(const set<string>& dict, const string& str)
{
    int size = (int)str.length();
    vector<bool> f(size+1); //f[i]: str[0...i-1]是否在词典中

    f[0] = true;
    int i, j;
    for(i = 1; i <= size; i++) //str[0...i-1]: 长度为i
    {
        for(j = i-1; j >= 0; j--)
        {
            if(f[j] && (dict.find(str.substr(j, i-j)) != dict.end())) //str[j...i-1]
            {
                f[i] = true;
                break;
            }
        }
    }
    return f[size];
}

int _tmain(int argc, _TCHAR* argv[])
{
    set<string> dict;
    dict.insert("cat");
    dict.insert("cats");
    dict.insert("and");
    dict.insert("sand");
    dict.insert("dog");
    dict.insert("dog");
    string str = "catsanddog";
    if(WordBreak1(dict, str))
        cout << "Break is TRUE\n";
    else
        cout << "Break is FALSE\n";
    return 0;
}
```

# Code2

```
bool WordBreak2(const set<string>& dict, const string& str)
{
    int size = (int)str.length();
    if(size == 0)
        return true;

    for(int i = size-1; i >= 0; i--)
    {
        if(WordBreak2(dict, str.substr(0, i))
            &&(dict.find(str.substr(i, size-i)) != dict.end()))
            return true;
    }
    return false;
}
```



# Code3

```
int WordBreak3(const set<string>& dict, const string& str, vector<int>& f)
{
    int size = (int)str.length();
    for(int i = size-1; i >= 0; i--)
    {
        if(f[i] == 0)
            f[i] = WordBreak3(dict, str.substr(0, i), f);

        if((f[i] == 1) && (dict.find(str.substr(i, size-i)) != dict.end()))
            return 1;
    }
    return -1;
}

int _tmain(int argc, _TCHAR* argv[])
{
    set<string> dict;
    dict.insert("cat");
    dict.insert("cats");
    dict.insert("and");
    dict.insert("sand");
    dict.insert("dog");
    dict.insert("dog");
    string str = "catsandsdog";
    int size = (int)str.length();
    vector<int> f(size+1); //f[i]: str[0...i-1]是否在词典中
    memset(&f.front() + 1, 0, sizeof(int)*size);
    f[0] = 1;
    if(WordBreak3(dict, str, f) == 1)
        cout << "Break is TRUE\n";
    else
        cout << "Break is FALSE\n";
    return 0;
}
```

# Last Code

```
void AddAnswer(const string& str, const vector<int>& oneBreak, vector<string>& answer)
{
    int s = (int)oneBreak.size();
    int size = (int)str.length();
    answer.push_back(str.substr(0, s));
    string& sentence = answer.back();
    sentence.reserve(size+s); //申请足够的内容长度
    int start = 0, end = 0;
    for(int i = s-1; i >= 0; i--) //oneBreak[size-1]==0, 特殊处理
    {
        end = oneBreak[i]; //别忘了, k=oneBreak[i]的值表示在string[k]的前面添加break
        sentence += str.substr(start, end-start);
        sentence += ' ';
        start = end;
    }
    sentence += str.substr(start, size-start); //最后一个break
}

//计算str[0...cur-1]的wordbreak有哪些
void FindAnswer(const vector<vector<bool>>& chess, const string& str, int cur, vector<int>& oneBreak, vector<string>& answer)
{
    if(cur == 0) //叶子
    {
        AddAnswer(str, oneBreak, answer);
        return;
    }
    int size = (int)str.length();
    for(int i = 0; i < cur-1; i++)
    {
        if(chess[cur][i]) //str[i...cur]在词典中
        {
            oneBreak.push_back(i);
            FindAnswer(chess, str, i, oneBreak, answer);
            oneBreak.pop_back();
        }
    }
}

void WordBreak(const set<string>& dict, const string& str, vector<string>& answer)
{
    int size = (int)str.length();
    //chess[i][j]: str[0...i-1]中, 是否可以在第j号元素的前面加break
    vector<vector<bool>> chess(size+1, vector<bool>(size));
    vector<bool> f(size+1); //f[i]: str[0...i-1]是否在词典中

    int i, j;
    f[0] = true; //空串在词典中
    for(i = 1; i <= size; i++) //str[0...i-1]: 长度为i
    {
        for(j = i-1; j >= 0; j--)
        {
            if(f[j] && (dict.find(str.substr(j, i-j)) != dict.end())) //str[j...i-1]
            {
                f[i] = true;
                chess[i][j] = true;
            }
        }
    }
    vector<int> oneBreak; //一种可行的划分
    FindAnswer(chess, str, size, oneBreak, answer); //计算str[0...size-1]的wordbreak有哪些
}

void Print(const vector<string>& answer)
{
    vector<string>::const_iterator itEnd = answer.end();
    for(vector<string>::const_iterator it = answer.begin(); it != itEnd; it++)
        cout << *it << endl;
    cout << endl;
}

int _tmain(int argc, _TCHAR* argv[])
{
    set<string> dict;
    dict.insert("下雨天");
    dict.insert("留客");
    dict.insert("留客天");
    dict.insert("天留");
    dict.insert("留我不");
    dict.insert("我不留");
    dict.insert("留");
    dict.insert("dog");
    string str = "下雨天留客天留我不留";
    vector<string> answer;
    WordBreak(dict, str, answer);
    Print(answer);
    return 0;
}
```

# Main Code

```
//计算str[0...cur-1]的wordbreak有哪些
void FindAnswer(const vector<vector<bool>> & chess, const string& str, int cur,
               vector<int> & oneBreak, vector<string> & answer)
{
    if (cur == 0) //叶子
    {
        AddAnswer(str, oneBreak, answer);
        return;
    }
    int size = (int)str.length();
    for (int i = 0; i < cur-1; i++)
    {
        if (chess[cur][i]) //str[i...cur]在词典中
        {
            oneBreak.push_back(i);
            FindAnswer(chess, str, i, oneBreak, answer);
            oneBreak.pop_back();
        }
    }
}

void WordBreak(const set<string> & dict, const string& str, vector<string> & answer)
{
    int size = (int)str.length();
    //chess[i][j]: str[0...i-1]中, 是否可以在第j号元素的前面加break
    vector<vector<bool>> chess(size+1, vector<bool>(size));
    vector<bool> f(size+1); //f[i]: str[0...i-1]是否在词典中

    int i, j;
    f[0] = true; //空串在词典中
    for (i = 1; i <= size; i++) //str[0...i-1]: 长度为i
    {
        for (j = i-1; j >= 0; j--)
        {
            if (f[j] && (dict.find(str.substr(j, i-j)) != dict.end())) //str[j...i-1]
            {
                f[i] = true;
                chess[i][j] = true;
            }
        }
    }
    vector<int> oneBreak; //一种可行的划分
    FindAnswer(chess, str, size, oneBreak, answer); //计算str[0...size-1]的wordbreak有哪些
}
```

# Aux Code

```
void AddAnswer(const string& str, const vector<int>& oneBreak, vector<string>& answer)
{
    int s = (int)oneBreak.size();
    int size = (int)str.length();
    answer.push_back(string());
    string& sentence = answer.back();
    sentence.reserve(size+s); //申请足够的内容长度
    int start = 0, end = 0;
    for(int i = s-2; i >= 0; i--) //oneBreak[size-1]==0, 特殊处理
    {
        end = oneBreak[i]; //别忘了, k=oneBreak[i]的值表示在string[k]的前面添加break
        sentence += str.substr(start, end-start);
        sentence += ' ';
        start = end;
    }
    sentence += str.substr(start, size-start); //最后一个break
}
```

下雨天. 留客. 天留. 我不留

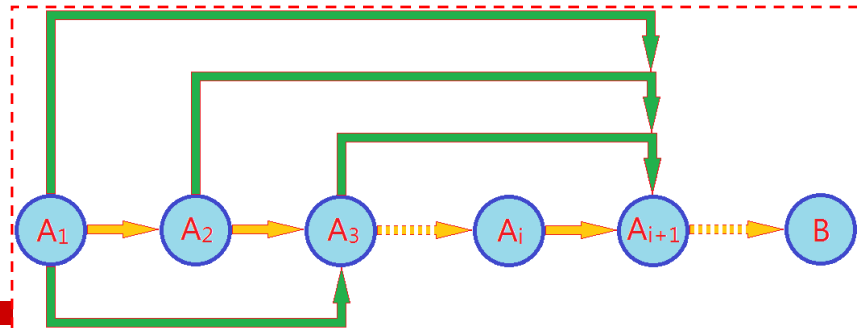
下雨天. 留客天. 留. 我不留

下雨天. 留客天. 留我不. 留

```
void Print(const vector<string>& answer)
{
    vector<string>::const_iterator itEnd = answer.end();
    for(vector<string>::const_iterator it = answer.begin();
        it != itEnd; it++)
        cout << *it << endl;
    cout << endl;
}

int _tmain(int argc, _TCHAR* argv[])
{
    set<string> dict;
    dict.insert("下雨天");
    dict.insert("留客");
    dict.insert("留客天");
    dict.insert("天留");
    dict.insert("留我不");
    dict.insert("我不留");
    dict.insert("留");
    dict.insert("dog");
    string str = "下雨天留客天留我不留";
    vector<string> answer;
    WordBreak(dict, str, answer);
    Print(answer);
    return 0;
}
```

# 总结



- 动态规划是**方法论**，是解决一大类问题的通用思路。  
事实上，很多内容都可以**归结为**动态规划的思想。
- **KMP**中求next数组：已知 $\text{next}[0 \dots i-1]$ ，求 $\text{next}[i]$ ；
- 最长回文子串**Manacher算法**中，已知 $P[0 \dots i-1]$ 求 $P[i]$
- **何时**可以考虑**使用**动态规划：
  - 初始规模下能够方便的得出结论
    - 空串、长度为0的数组、自身等
  - 能够得到问题规模增大导致的变化
    - 递推式——状态转移方程

# 我们在这里

□ <http://wenda.ChinaHadoop.cn>

■ 视频/课程/社区

□ 微博

■ @ChinaHadoop

■ @邹博\_机器学习

□ 微信公众号

■ 小象

■ 大数据分析挖掘



---

感谢大家！

恳请大家批评指正！