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(下)

小象学院 ChinaHadoop.cn 邹博

主要内容

- 动态规划
  - 矩阵连乘问题/Catalan数
  - 子序列数目
  - 无重复字符的最长子串
  - 跳跃问题
  - 最小平方划分
  - 直方图最大矩形面积
  - 最大全一矩形
  - 找零钱问题/背包问题
  - Scramble String
  - Palindrome Partitioning - I/II

矩阵乘积

- 根据矩阵相乘的定义来计算 C=A×B，需要m*n*s次乘法。
- 三个矩阵A、B、C的阶分别是$a_0 \times a_1$, $a_1 \times a_2$, $a_2 \times a_3$，从而(A×B)×C和A×(B×C)的乘法次数是$a_0a_1a_2+a_0a_2a_3$、$a_1a_2a_3+a_0a_1a_3$，二者一般情况是不相等的。
  - 问：给定n个矩阵的连乘积：$A_1 \times A_2 \times A_3 \ldots\ldots \times A_n$，如何添加括号来改变计算次序，使得乘法的计算量最小?
- 此外：若A、B都是n阶方阵，C的计算时间复杂度为$O(n^3)$
  - 问：可否设计更快的算法?
  - 答：分治法：Strassen分块——理论意义大于实践意义。

矩阵连乘的提法

- 给定n个矩阵$\{A_1,A_2,\ldots,A_n\}$，其中A_i与A_{i+1}是可乘的，i=1，2…，n-1。考察该n个矩阵的连乘积：$A_1 \times A_2 \times A_3 \ldots\ldots \times A_n$，确定计算矩阵连乘积的计算次序，使得依此次序计算矩阵连乘积需要的乘法次数最少。
  - 即：利用结合律，通过加括号的方式，改变计算过程，使得数乘的次数最少。

分析

- 将矩阵连乘积$A_iA_{i+1}\ldots A_j$ 记为A[i:j]，这里i≤j
  - 显然，若i==j，则A[i:j]即A[i]本身。
- 考察计算A[i:j]的最优计算次序。设这个计算次序在矩阵A_k和A_{k+1}之间将矩阵链断开，i≤k<j，则其相应的完全加括号方式为

$$(A_iA_{i+1}\ldots A_k)(A_{k+1}A_{k+2}\ldots A_j)$$

- 计算量：A[i:k]的计算量加上A[k+1:j]的计算量，再加上A[i:k]和A[k+1:j]相乘的计算量

最优子结构

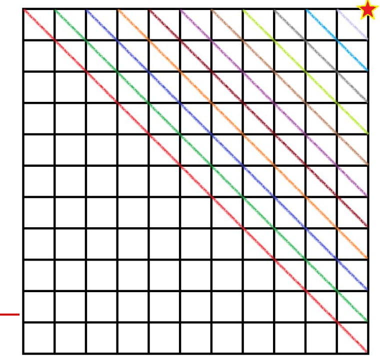
- 特征：计算A[i:j]的最优次序所包含的计算矩阵子链 A[i:k]和A[k+1:j]的次序也是最优的。
- 矩阵连乘计算次序问题的最优解包含着其子问题的最优解。这种性质称为最优子结构性质。
- 最优子结构性质是可以使用动态规划算法求解的显著特征。

状态转移方程 $(A_i A_{i+1} \ldots A_k)(A_{k+1} A_{k+2} \ldots A_j)$

- 设计算A[i:j](1≤i≤j≤n)所需要的最少数乘次数为m[i,j]，则原问题的最优值为m[1,n]；
- 记A_i的维度为 $p_{i-1} \times p_i$
- 当i==j时，A[i:j]即A_i本身，因此，m[i,i]=0；(i=1,2,…,n)
- 当i<j时，$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j$
- 从而：

$$m[i,j] = \begin{cases} 0 & i = j \\ \min\limits_{i \le k < j} \{ m[i,k] + m[k+1,j] + p_{i-1} p_k p_j \} & i < j \end{cases}$$

矩阵连乘问题从算法到实现

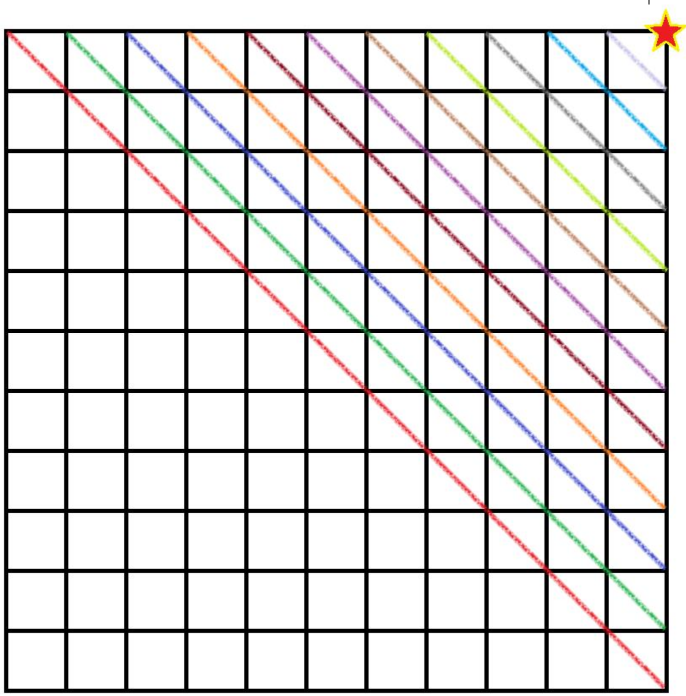


- 由m[i,j]的递推关系式可以看出，在计算m[i,j]时，需要用到m[i+1,j]，m[i+2,j]…m[j-1,j]；

$$m[i,j]=\begin{cases}0 & i=j\\ \min\limits_{i\le k<j}\{m[i,k]+m[k+1,j]+p_{i-1}p_kp_j\} & i<j\end{cases}$$

- 因此，求m[i,j]的前提，不是m[0…i-1;0…j-1]，而是沿着主对角线开始，依次求取到右上角元素。
- 因为m[i,j]一个元素的计算，最多需要遍历n-1次，共O(n^2)个元素，故算法的时间复杂度是O(n^3)，空间复杂度是O(n^2)。

Code



```
//p[0...n]存储了n+1个数，其中，(p[i-1],p[i])是矩阵i的阶；
//s[i][j]记录A[i...j]从什么位置断开；m[i][j]记录数乘最小值
void MatrixMultiply(int* p, int n, int** m, int** s)
{
    int r,i,j,k,t;
    for(i = 1; i <= n; i++)
        m[i][i] = 0;

    //r个连续矩阵的连乘：上面的初始化，相当于r=1
    for(r = 2; r <= n; r++)
    {
        for(i = 1; i <= n-r+1; i++)
        {
            j=i+r-1;
            m[i][j] = m[i+1][j]+ p[i-1]*p[i]*p[j];
            s[i][j] = i;
            for(k = i+1; k < j; k++)
            {
                t = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
                if(t < m[i][j])
                {
                    m[i][j] = t;
                    s[i][j] = k;
                }
            }
        }
    }
}
```

矩阵连乘问题的进一步思考

- n个矩阵连乘，可以分解成i个矩阵连乘和(n-i)个矩阵连乘，最后，再将这两个矩阵相乘。故：

$$P(n)=\begin{cases}1 & n=1\\ \sum_{k=1}^{n-1}P(k)P(n-k) & n>1\end{cases}\Rightarrow P(n)=\Omega\left(\frac{4^n}{\sqrt{\pi}*n^{3/2}}\right)$$

$$P(n)=\frac{1}{n}C_{2n-2}^{n-1}$$

- 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440, 9694845, 35357670, 129644790, 477638700, 1767263190, 6564120420, 24466267020, 91482563640, 343059613650, 1289904147324, 4861946401452……

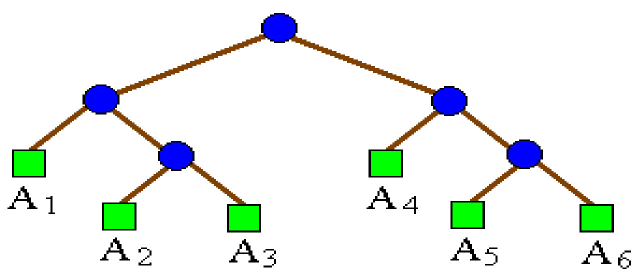
卡塔兰数 Catalan $H(n)=\frac{1}{n+1}C_{2n}^{n}$

- 有N个节点的二叉树共有多少种情形？
- 一个栈(无穷大)的进栈序列为1,2,3,..n,有多少个不同的出栈序列？
- 凸多边形三角化：将一个凸多边形划分成三角形区域的方法有多少种？
- 由左而右扫描由n个1和n个0组成的2n位二进制数，要求在任何时刻，1的累计数不小于0的累计数。求满足这样条件的二进制数的个数。

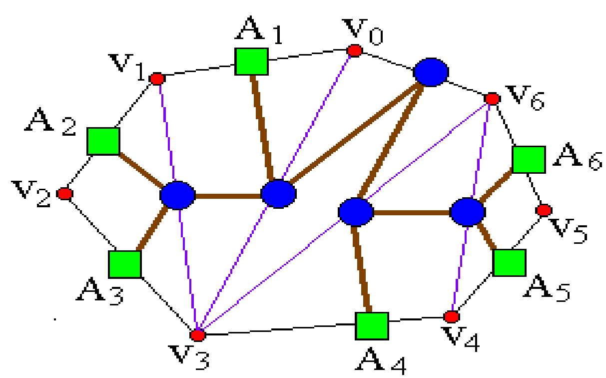
- 注：由h(n)=C(n,2n)/(n+1)很容易求得：h(n)=h(n-1)*(4*n-2)/(n+1)=c(2n,n)-c(2n,n+1)

上述问题的相互联系

- 一个矩阵(一个表达式)的完全加括号方式相应于一棵完全二叉树，称为表达式的语法树。例如，完全加括号的矩阵连乘积((A1(A2A3))(A4(A5A6)))所相应的语法树如图 (a)所示。
- 凸多边形{v0,v1,…vn-1}的三角剖分也可以用语法树表示。例如，图 (b)中凸多边形的三角剖分可用图 (a)所示的语法树表示。
  - 矩阵连乘积中的每个矩阵Ai对应于凸(n+1)边形中的一条边vi-1vi。三角剖分中的一条弦vivj，i<j，对应于矩阵连乘积A[i+1:j]。

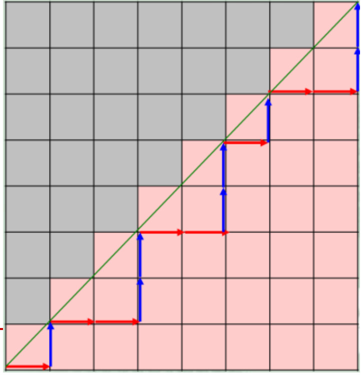


(a)



(b)

附：证明Catalan数公式1/2相等



- 考察问题：n*n棋盘从左下角走到右上角而不穿过主对角线的走法。
- 考虑n*n棋盘，记主对角线为L。从左下角走到右上角不穿过对角线L的所有路径，不算起点，一定有第一次接触到L的位置(可能是终点)，设此位置为M，坐标为(x,x)——设第一个数为横轴坐标。该路径一定从下方的(x,x-1)而来，而起点处第一步也一定是走向(1,0)，两者理由相同——否则就穿过了主对角线。考虑从(1,0)到(x,x-1)的(x-1)*(x-1)的小棋盘中，因为在此中路径一直没有接触过主对角线(M的选取)，所以在此小棋盘中路径也一定没有穿过从(1,0)到(x,x-1)的小棋盘的对角线L1。这样在这个区域中的满足条件的路径数量就是一个同构的子问题，解应该是F(x-1)，而从M到右上角终点的路径数量也是一个同构的子问题，解应该是F(n-x)，而第一次接触到主对角线的点可以从(1,1)取到(n,n)，这样就有F(n)=∑(k=1…n){F(k-1)*F(n-k)}=∑(k=0…n-1){F(k-1)*F(n-k)}。
- 注：抽象成2n个操作组成的操作链，其中A操作和B操作各n个，且要求截断到操作链的任何位置都有：A操作(向右走一步)的个数不少于B操作(向上走一步)的个数。

附：Catalan数公式1和公式2相等的证明

- 现在证明上述问题的解为C(2n,n)/(n+1)
- 思路是先求所有从(0,0)到(n,n)的路径数X，再求所有穿过主对角线L的从(0,0)到(n,n)的路径数Y，用前者减去后者得到所求。
- 从(0,0)到(n,n)的路径数显然是C(2n,n)，一共要走2n步到达右上角，其中向右和向上各n步，总走法是C(2n,n)。
- 考虑一个新增的位置(n-1,n+1)，它位于终点的左上角一个格处，设所有从(0,0)到(n-1,n+1)的路径数为Z，下面要证明Y和Z相等，从而通过求Z来求Y。
  - 考虑从(0,1)到(n-1,n)的对角线L2，对于所有穿过L而到达终点的路径，一定会接触到L2，找出某路径第一次接触到L2的位置M1，将从M1到终点的路径沿L2做对折一定会得到一条从M1到(n-1,n+1)的路径，故每条穿过L到达终点的路径都对应一条到达(n-1,n+1)的路径，即有Y<=Z。
  - 所有从起点到达(n-1,n+1)的路径都一定会穿过L2，找出某路径第一次穿过L2的位置M2，将M2到(n-1,n+1)的路径沿L2对折，就得到一条M2到(n,n)的路径，且该条路径一定穿过L，故每条到达(n-1,n+1)的路径都对应一条穿过L到达终点的路径，即有Z<=Y。
- 故Z==Y。
- Z是显然的从(0,0)到(n-1,n+1)共需走2n步，其中向右n-1步、向上n+1步，故Z=C(2n,n-1)。
- 由以上可知F(n)=X-Y=X-Z=C(2n,n)-C(2n,n-1)=C(2n,n)/(n+1)。

Distinct Subsequences

- 子序列数目
- 给定文本串Text和模式串Pattern，计算文本串Text的子序列中包含模式串Pattern的个数——模式串Pattern以子序列的形式在文本串Text中出现过几次。
  - 如“rabbit”在“rabbbit”中出现过3次。
  - “ab”在“abacab”出现过4次。
    - Text[0,1]= Text[0,5]=Text[2,5]=Text[4,5]=“ab”

动态规划解决子序列数目问题

- 记Pattern[0...j]在Text[0...i]中出现次数为dp[i,j]，借鉴LCS的思想：
- 若Pattern[j]≠Text[i]
  - 则Text[0...i]和Text[0...i-1]对于Pattern[0...j]表达能力相同，即：dp[i,j]=dp[i-1,j]
- 若Pattern[j] = Text[i]
  - Text[0...i-1]表达Pattern[0...j-1]后，最后缀上Text[i]
  - 或者Text[0...i-1]直接表达Pattern[0...j]
  - 即：dp[i,j]=dp[i-1,j-1]+ dp[i-1,j]

状态转移方程和初值

- 写出状态转移方程，得：

$$dp(i,j)=\begin{cases} dp(i-1,j) & Text[i]\neq Pattern[j] \\ dp(i-1,j-1)+dp(i-1,j) & Text[i]=Pattern[j] \end{cases}$$

- 初值：dp(i,0)=1
  - 空串在任何串都出现1次。

滚动数组 $dp(i,j)=\begin{cases} dp(i-1,j) & Text[i]\neq Pattern[j] \\ dp(i-1,j-1)+dp(i-1,j) & Text[i]=Pattern[j] \end{cases}$

- dp(i,j)的更新**只需要前面一行**元素，同时**不**需要记录路径，所以，可以使用**滚动数组**(回忆LCS中的“进一步思考”)，得：

$$dp(j)=\begin{cases} dp(j) & Text[i]\neq Pattern[j] \\ dp(j-1)+dp(j) & Text[i]=Pattern[j] \end{cases}$$

- 初始条件：$dp(0)=1$
  - 空串在空串中出现过1次

Code

```
int DistinctSubsequence(const char* pText, const char* pPattern)
{
    int size1 = (int)strlen(pText);
    int size2 = (int)strlen(pPattern);
    if(size1 < size2)
        return 0;

    int* pSize = new int[size2+1];
    pSize[0] = 1;   //空串在空串中出现1次
    memset(pSize+1, 0, sizeof(int)*size2);

    int i,j;
    for(i = 0; i < size1; i++)
    {
        for(j = size2-1; j >= 0; j--)
        {
            if(pText[i] == pPattern[j])
                pSize[j+1] += pSize[j];
        }
    }
    int s = pSize[size2];
    delete[] pSize;
    return s;
}

int _tmain(int argc, _TCHAR* argv[])
{
    char text[] = "abacab";
    char pattern[] = "ab";
    cout << DistinctSubsequence(text, pattern) << endl;
    return 0;
}
```

Longest Substring Without Repeating Characters

- 无重复字符的最长子串
- 对于给定的字符串，返回它最长的无重复字符的子串的长度，如：字符串“abcabcbb”的无重复最长子串是“abc”，长度为3；字符串“bbbb”的无重复最长子串是“b”，长度为1。
  - 假定字符只包含26个英文小写字母。

从暴力求解开始分析

- 既然计算子串，则设置两个索引i，j分别指向子串首尾，判断该子串是否有重复字符。
  - i，j从0到N-1，子串最长为N，时间复杂度$O(N^3)$ / $O(N^4)$
- 如何判断一个字符串str[i,i+1…j]是否有重复数字？
  - k从i+1到j遍历，判断str[k]是否在str[i…k-1]中出现？
    - 本身已经是$O(N^2)$。
  - 考虑到只有26个字母，所以，使用缓存exist[26]：
    - 初始化为-1
    - k从i+1到j遍历，若str[k]所在的缓存位置exist[str[k]-’a’]为-1，表示str[k]未出现过，则标记exist[str[k]-’a’]=k，继续k+1的考察；若exist[str[k]-’a’]不为-1，表示str[k]出现过，则该子串不是无重复字符的字符串。
  - 以上“缓存”的思路，能否用来解决整个问题的优化？

继续分析无重复最大子串的优化方法

- 使用exist['A'~'Z'][N]：
  - exsit['a']表示：字符'a'在字符串str中出现的位置。
- 事实上：只需要记录str[j]在str[0…j-1]最后一次出现的位置k，那么，对于子串str[i…j]：
  - 如果k＞i，则表示str[k]==str[j]，即子串不是无重复串。
- str[0…j-1]最后一次出现的位置不需要提前计算好，边向后查找边更新即可。
- 时间复杂度O(N)，空间复杂度O(1)。
  - 如果把exist[26]当成O(1)的话。

Code

```
const int CHARATER_MAX = 26;
int LongestSubstringUnique(char* str, int size)
{
    int last[CHARATER_MAX]; //记录字符上次出现过的位置
    int start = 0;          //记录当前子串的起始位置
    fill(last, last + CHARATER_MAX, -1);
    int nMax = 0;
    for(int i = 0; i < size; i++)
    {
        if(last[str[i] - 'a'] >= start)  //str[start...i]中出现重复，重新开始记录
        {
            nMax = max(i - start, nMax);
            start = last[str[i] - 'a'] + 1;
        }
        last[str[i] - 'a'] = i;     //记录str[i]最后出现的位置
    }
    return max(size-start, nMax);
}

int _tmain(int argc, _TCHAR* argv[])
{
    char string[] = "abcabcbb";
    LongestSubstringUnique(string, sizeof(string) / sizeof(char) - 1);
    return 0;
}
```

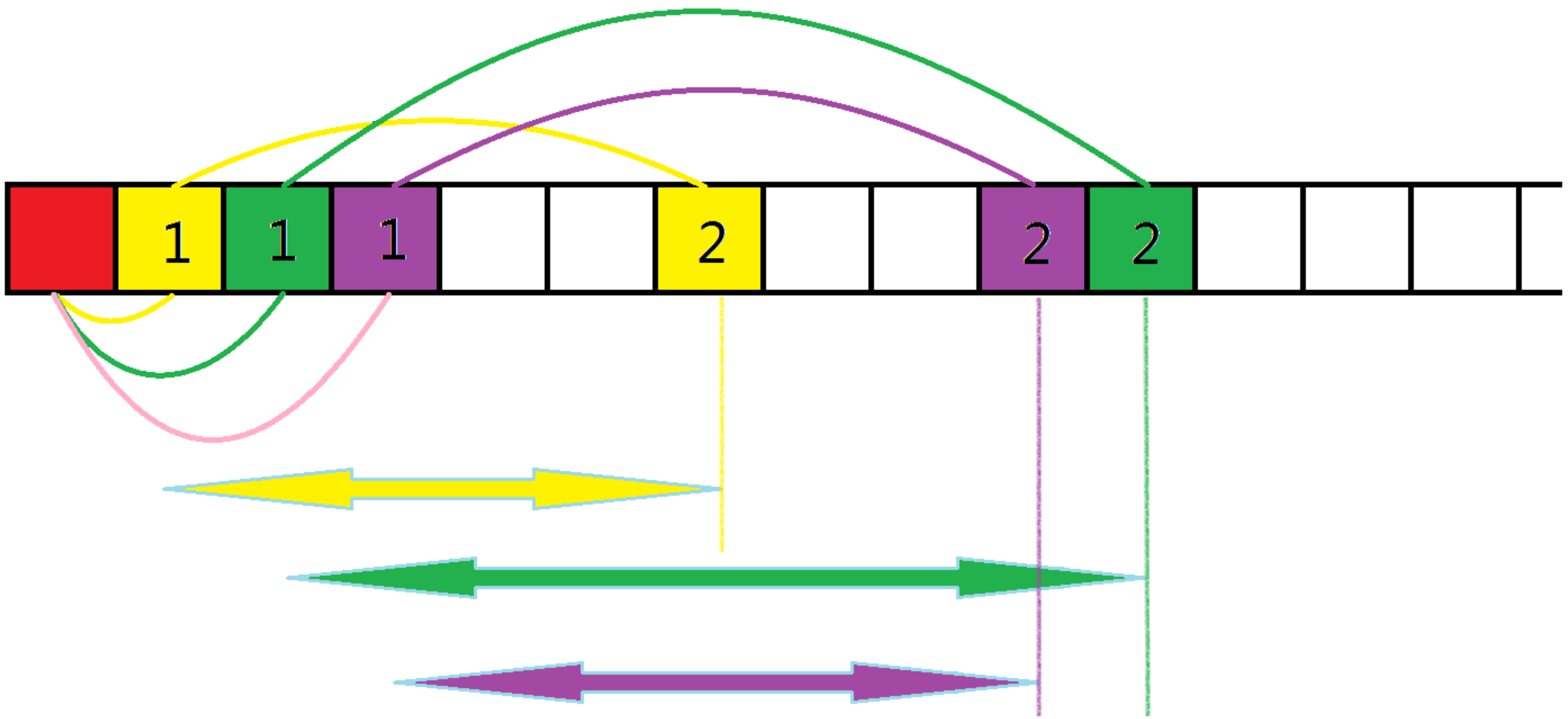
进一步思考

- 若原题目中将每个字符映射成长度为k的单词(k为定值)，则题目换成：
  - Substring with Concatenation of All Words
  - 给定字符串S和单词数组L，L中的单词长度都相等，找出S中所有的子串恰好包含L中所有单词各一次的所有位置。

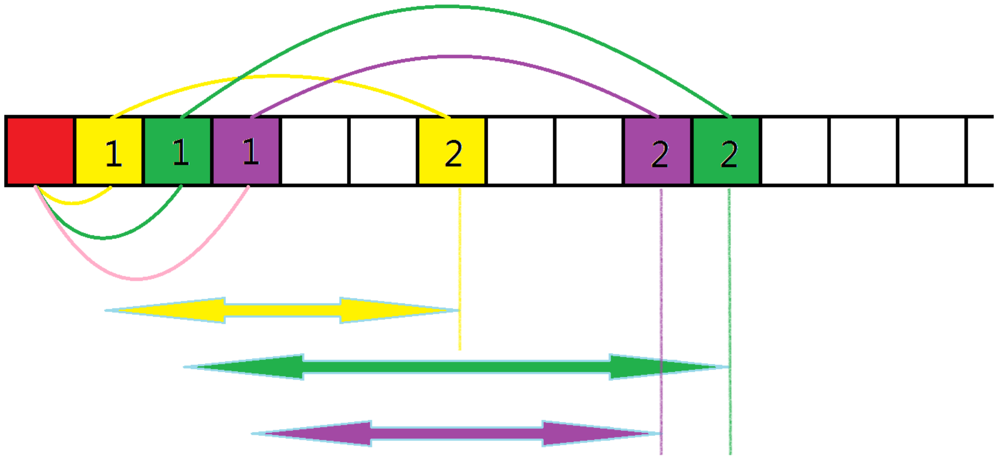
Jump

- 跳跃问题
- 给定非负整数数组，初始时在数组起始位置放置一机器人，数组的每个元素表示在当前位置机器人最大能够跳跃的数目。它的目的是用最少的步数到达数组末端。例如：给定数组A=[2,3,1,1,2]，最少跳步数目是2，对应的跳法是：2→3→2。
- 如：2,3,1,1,2,4,1,1,6,1,7，最少需要几步？

跳跃问题分析



跳跃问题算法步骤



- 初始步数step赋值为0；
- 记当前步的控制范围是[i,j]，则用k遍历i到j
  - 计算A[k]+k的最大值，记做j2；
- step++；继续遍历[j+1,j2]；

Code

```
int Jump(int A[], int n)
{
    if (n == 1)
        return 0;

    int step = 0; //最小步数
    int i = 0;
    int j = 0;  //[i,j]是当前能覆盖的区间
    int k,j2;
    while(j < n)  //覆盖区间尚未包含最后元素
    {
        step++;
        j2 = j;
        for(k = i; k <= j; k++)
        {
            j2 = max(j2, k + A[k]);
            if(j2 >= n-1)   //已经跳跃到最后一步
                return step;
        }
        i = j+1;
        j = j2;
        if(j < i)   //覆盖区间为负，说明无法跳到末尾
            return -1;
    }
    return step;
}

int _tmain(int argc, _TCHAR* argv[])
{
    int A[] = {2,3,1,1,2,4,1,1,6,1,7};
    Jump(A, sizeof(A) / sizeof(int));
    return 0;
}
```

Jump问题“知识挖掘”

- 上述代码的时间复杂度是多少？
  - O(N) or O(N^2)
- 该算法能够天然处理无法跳跃到末尾的情况。
  - 若无法跳到末尾，则返回-1
- 该算法在每次跳跃中，都是尽量跳的更远，并记录j2——属于贪心法；也可以认为是从区间[i,j](若干结点)扩展下一层区间[j+1,j2] (若干子结点)——属于广度优先搜索。
  - 可见，贪心法是需要详细分析才能放心使用。
  - 回忆图论中的概要说明：
    - 广度优先搜索往往和“最少”、“最短”相关联。
- 思考：是否可以使用动态规划解决？
  - 记dp[i]为：到达A[i]时，还剩余多少步没有用。
  - 则：$dp[i+1]=\max(dp[i],A[i])-1$

最小平方划分

- 一个正整数可以由若干个正整数的平方和表示，求整数201314最小的平方划分数目。
  - 如10，可以写成1^2+3^2，或者$1^2+1^2+2^2+2^2$，显然，1^2+3^2只需要划分为2个完全平方数的和，因此，应该返回2。
  - 如64，本身是完全平方数8^2，返回1。
- 这种平方划分是一定存在的。
  - 如：将任意正整数n划分成n个1显然是一个合理划分(不一定最小)。
  - 类似的有Word Break问题、最小回文划分问题。

贪心？

- 对于给定的整数n，找到比n小的最大的完全平方数k，继续计算n-k……直到1为止。
  - 贪心不能解决问题。
- 如12，第一次找到9，继续计算3。最终得到12=9+1+1+1，划分数目为4；
- 但12=4+4+4，最小划分数目为3。

问题分析

- 如果已经求出了1...n-1的所有数的最小划分，如何求数n的最小划分呢？
- 若$a+k^2=n$，则在a的划分方案上加上数字k，即为n的划分方案。

算法描述

- 已知1…n-1的最小划分，求数n最小划分：
  - 记1…n-1的最小划分为数组split[n]，其中split[i]表示数i的最小划分数目。令n的最小划分数目为x，初始化为n。
  - 记k：遍历1……K，其中，$K=\lfloor\sqrt{n}\rfloor$
    - 若split[k]+1<x，则将x更新为split[k]+1。
  - x即为n的最小划分。
    - 赋值split[n]=x，为计算更大的n做准备。

代码实现

- 可以使用动态规划，在得到1...n-1的最小平方划分的前提下计算n；n从1到N遍历，即最终得到N的最小平方划分。
- 或使用递归；辅助数组split[1...n]的作用是记忆小规模问题，防止重复计算。
- 动态规划和记忆递归的时间复杂度都是$O\left(n\cdot\sqrt{n}\right)$
  - 计算n的划分，迭代次数是$\sqrt{n}$
  - 一共计算n次。

题中题：如何判断完全平方数

- 给定正整数n，如果判断n是否是一个完全平方数？
- 调用系统函数float sqrt(float)，记返回值取整后为a，计算a^2是否等于n。
  - 系统函数本身仍然需要浮点运算(如Taylor展式)
- 使用牛顿迭代法。

复习：平方根算法的证明

- 在任意点x_0处Taylor展开

令$f(x)=x^2-a$，即$f(x)=0$

$$\Rightarrow f(x)=f(x_0)+f'(x_0)(x-x_0)+o(x)$$

$$\Rightarrow f(x)\approx f(x_0)+f'(x_0)(x-x_0)$$

$$\Rightarrow 0=f(x_0)+f'(x_0)(x-x_0)$$

$$\Rightarrow x-x_0=-\frac{f(x_0)}{f'(x_0)}$$

$$\Rightarrow x=x_0-\frac{f(x_0)}{f'(x_0)}$$

将$f(x_0)=x_0^2-a$和$f'(x_0)=2x_0$带入，

$$\Rightarrow x=x_0-\frac{f(x_0)}{f'(x_0)}$$

$$\Rightarrow x=x_0-\frac{x_0^2-a}{2x_0}$$

$$\Rightarrow x=\frac{1}{2}\left(x_0+\frac{a}{x_0}\right)$$

复习：平方根

- 一般若干次(5、6次)迭代即可获得比较好的近似值。

```
float Calc(float x);

int _tmain(int argc, _TCHAR* argv[])
{
    for(int i = 0; i <= 10; i++)
        cout << Calc((float)i) << '\n';
    return 0;
}

float Calc(float a)
{
    if(a < 1e-6)     //负数或者0，则直接返回0
        return 0;

    float x = a / 2;
    float t = a;
    while(fabs(x - t) > 1e-6)
    {
        t = x;
        x = (x + a/x) / 2;
    }
    return x;
}
```

动态规划Code

- 201314=$4^2+227^2+387^2$

```
int GetSquare(int a)
{
    if(a == 1)
        return 1;
    int x = a/2;     //x足够大时，x大于a的平方根
    int t = a;
    while(x < t)          //新值小于原值
    {
        t = x;            //记录原来的值
        x = (x + a/x)/2;    //x值更新
    }
    if(x*x > a)
        return x-1;
    return x;
}

void SquareCut2(int N, int* pre, int* minCut)
{
    int n,k,K,t;
    for(n = 1; n <= N; n++)
    {
        K = GetSquare(n);   //n的平方根下取整
        if(K*K == n)
        {
            minCut[n] = 1;
            pre[n] = 0;
            continue;
        }
        minCut[n] = n;  //默认分成n份
        pre[n] = n-1;   //n份，则前驱是n-1
        for(k = 1; k <= K; k++)
        {
            t = n-k*k;
            if(minCut[n] > minCut[t]+1)
            {
                minCut[n] = minCut[t]+1;
                pre[n] = t;
            }
        }
    }
}
```

```
void Print(int a, const int* pre)
{
    while(a != 0)
    {
        cout << a - pre[a]<<'-'<<GetSquare(a - pre[a])<<'\t';
        a = pre[a];
    }
    cout << endl;
}

int _tmain(int argc, _TCHAR* argv[])
{
    int N = 201314;
    int* pre = new int[N+1];    //pre[i]: i的前一个划分值(0未用)
    int* minCut = new int[N+1]; //minCut[i]: i的最小划分数(0未用)
    SquareCut2(N, pre, minCut);
    Print(N, pre);
    delete[] pre;
    delete[] minCut;
    return 0;
}
```

题外话：四平方和定理

- 每个正整数均可表示为最多4个整数的平方和。
- 欧拉发现的恒等式：

$$
\left(a^2+b^2+c^2+d^2\right)\left(x^2+y^2+z^2+w^2\right)
$$
$$
=(ax+by+cz+dw)^2+(ay-bx+cw-dz)^2
$$
$$
+(az-bw-cx+dw)^2+(aw+bz-cy-dx)^2
$$

- 如果正整数m和n能表示为4个整数的平方和，则其乘积mn也能表示为4个整数的平方和。
  - 只需再证明每个素数可以表示成4个整数的平方和即可。
  - 拉格朗日和欧拉分别在1770年和1773年作出证明。

附：递归Code

- 整数的平方根

```
int GetSquare(int a)
{
	if(a == 1)
		return 1;
	int x = a/2;	//x足够大时，x大于a的平方根
	int t = a;
	while(x < t)		//新值小于原值
	{
		t = x;		//记录原来的值
		x = (x + a/x)/2;	//x值更新
	}
	return x;
}

bool IsSquare(int a, int* nSquare)
{
	if(nSquare[a] == 0)
	{
		int x = a/2;	//x足够大时，x大于a的平方根
		int t = a;
		while(x < t)	//新值小于原值
		{
			t = x;		//记录原来的值
			x = (x + a/x)/2;	//x值更新
		}
		nSquare[a] = (x*x == a) ? 1 : 2;
	}
	if(nSquare[a] == 1)
		return true;
	return false;
}
```

附：递归Code

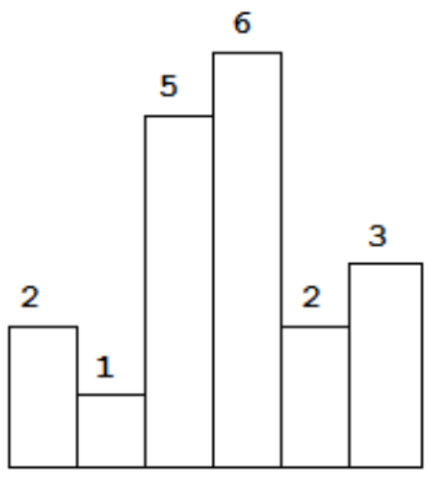
```
int _tmain(int argc, _TCHAR* argv[])
{
    const int N = 200;
    int pre[N+1];        //pre[i]: i的前一个划分值是什么，(0未用)
    int nSquare[N+1];    //bSquare[i]: i是否为平方数，为0表示未计算，1表示是，2表示不是(0未用)
    int minCut[N+1];     //minCut[i]: i的最小划分数目(0未用)
    memset(pre, 0, sizeof(int)*(N+1));
    memset(nSquare, 0, sizeof(int)*(N+1));
    memset(minCut, 0, sizeof(int)*(N+1));
    nSquare[1] = 1;
    SquareCut(N, pre, nSquare, minCut);
    for(int i = 1; i <= N; i++)
    {
        cout << i << ":\t";
        Print(i, pre);
    }
    return 0;
}
```

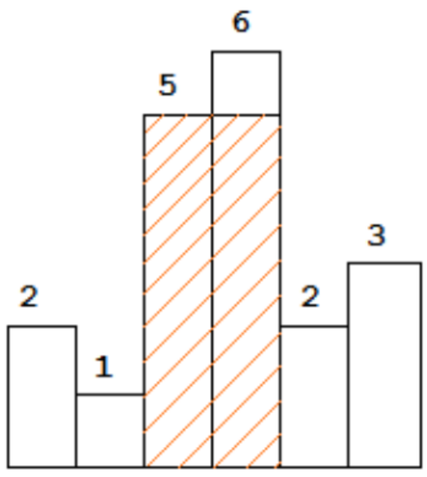
附：递归Code

```
int SquareCut(int a, int* pre, int* nSquare, int* minCut)
{
    if(minCut[a] == 0)  //a尚未计算过
    {
        if(IsSquare(a, nSquare))    //如果a是完全平方数
        {
            minCut[a] = 1;
            pre[a] = 0;
        }
        else    //否则，则切分
        {
            minCut[a] = a;
            pre[a] = a-1;
            int c;
            int i = 1;
            while(i*i < a)
            {
                c = SquareCut(a-i*i, pre, nSquare, minCut) + 1;
                if(c < minCut[a])
                {
                    minCut[a] = c;
                    pre[a] = a-i*i;
                }
                i++;
            }
        }
    }
    return minCut[a];
}
```

直方图矩形面积

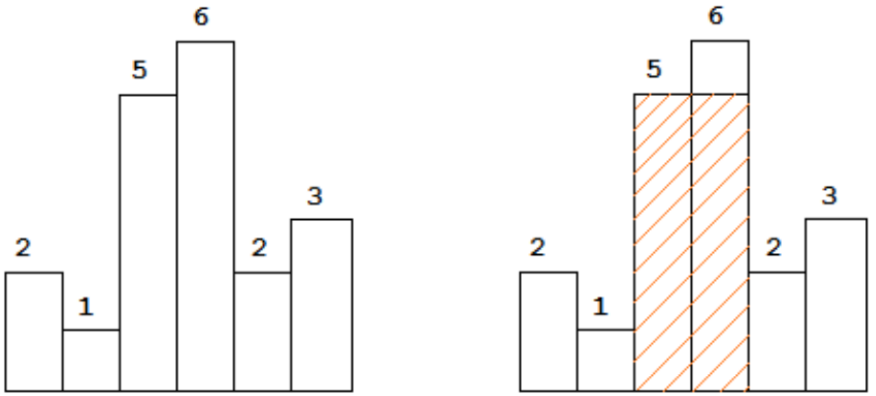
- 给定n个非负整数，表示直方图的方柱的高度，同时，每个方柱的宽度假定都为1；试找出直方图中最大的矩形面积。如：给定高度为：2,1,5,6,2,3，最大面积为10。





暴力求解

- 将直方图的数组记做a[0…size-1]；
- 计算以方柱a[i]为右边界的直方图中，遍历a[0…i]，依次计算可能的高度和面积，取最大者；
- i从0遍历到size-1；
- 时间复杂度为$O(N^2)$。



分析

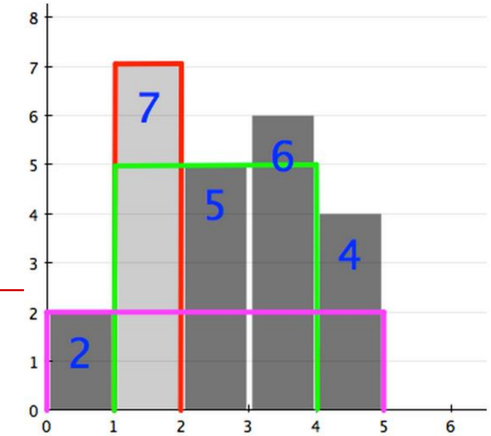
- 显然，若a[i+1]≥a[i]，则以a[i]为右边界的矩形Rect(width,height)，总可以添加a[i+1]带来的矩形Rect(1,height)，使得面积增大
- 只有当a[i+1]＜a[i]时，才计算a[i]为右边界的矩形面积。
    - trick：为了算法一致性，在a[0…size-1]的最后，添加a[size]=0，保证a[size-1]为右边界的矩形得到计算。

算法思想

- 从前向后遍历a[0…size](末尾添加了0)，若a[i]>a[i-1]，则将a[i]放入缓冲区；
- 若a[i]≤a[i-1]，则计算缓冲区中能够得到的最大矩形面积。

- 从a[i]>a[i-1]可以得出：
  - 缓冲区中放入的值是递增的
  - 每次只从缓冲区取出最后元素和a[i]比较——栈。

直方图最大矩形面积法分析



- 以2、7、5、6、4为例：
- 假设当前待分析的元素是4，由刚才的分析得知，栈内元素是2,5,6，其中，6是栈顶。
  - 此时，栈顶元素6＞4，则6出栈
    - 出栈后，新的栈顶元素为5，5和4的横向距离差为1；以6为高度，1为宽度的矩形面积是6*1=6
  - 此时，栈顶元素5＞4，则5出栈
    - 出栈后，新的栈顶元素为2，2和4的横向距离差为3；以5为高度，3为宽度的矩形面积是5*3=15
  - 此时，栈顶元素2≤4，则将4压栈，i++，同样的方法继续考察直方图后面的值。

说明

- 显然，为了能够方便的计算“横向距离”，压入栈的是方柱的索引，而非方柱的高度本身。
  - 这种trick在实践中经常使用。

Code

```
int LargestRectangleArea(vector<int>& height)
{
    height.push_back(0);    //确保原数组height的最后一位能够得到计算

    stack<int> s;
    int answer = 0;
    int temp;   //临时变量
    for (int i = 0; i < (int)height.size(); )
    {
        if (s.empty() || height[i] > height[s.top()])
        {
            s.push(i);
            i++;
        }
        else
        {
            temp = s.top();
            s.pop();
            answer = max(answer,height[temp]*(s.empty() ? i : i-s.top()-1));
        }
    }
    return answer;
}
```

Maximal Rectangle

- 最大全一矩形
- 给定二维布尔矩阵，元素只能取0或者1，找出只包含元素1并且面积最大的矩阵，返回它的面积。

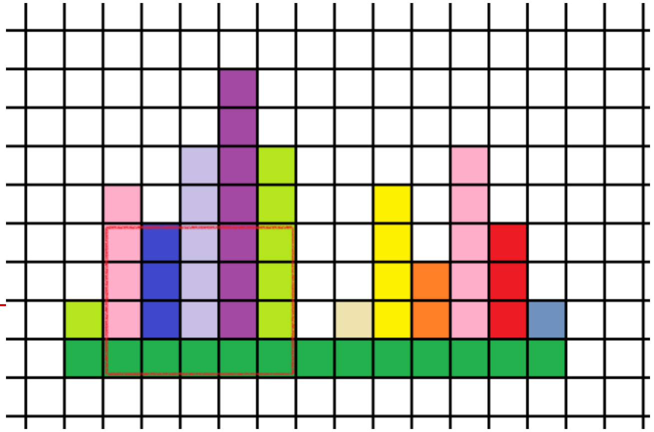
0	1	0	1	1	1	1	0	0
0	1	0	1	0	0	0	0	0
0	1	1	1	1	1	1	0	0
0	1	1	1	1	1	0	0	0
0	1	0	1	1	1	1	0	0
0	1	0	1	0	1	1	0	0
0	1	1	0	1	0	0	0	0

问题分析

$$
\begin{matrix}
0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\
0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0
\end{matrix}
$$

- 首先想到的是动态规划。
- 记以(i,j)为右下角的矩形最大面积为dp(i,j)
  - 在向右、向下扩展时，发现状态信息“不够”。
- 记以(i,j)为右下角的矩形，其左上角为(x,y)，其最大面积记做dp(i,j,x,y)
  - 在向右、向下扩展时，发现状态信息仍然“不够”。

问题求解思路分析



- 分析以(i,j)为右下角的子矩阵：
- k从j-1到0遍历，直到chess[i][j]为0；
  - 在第i行，得到全一子数组chess[i][k+1...j]；
- r从j到0，计算chess[i][r]为底的最高的全1子数组。
- 考察chess[i][r]：
  - k从i-1到0遍历，直到chess[k][r]为0；
  - 在第r列，得到全一子数组chess[k+1...i][r]；
- 形成如下锯齿型结构，下面需要计算该结构的最大矩形面积。
- 结论：直方图最大矩形面积问题！

Code

```
int LargestRectangleArea(int* height, int N)    //height[N]==0
{
    stack<int> s;
    int answer = 0;
    int temp;   //临时变量
    for (int i = 0; i <= N; )
    {
        if (s.empty() || height[i] > height[s.top()])
        {
            s.push(i);
            i++;
        }
        else
        {
            temp = s.top();
            s.pop();
            answer = max(answer,height[temp]*(s.empty() ? i : i-s.top()-1));
        }
    }
    return answer;
}

const int N = 9;
const int M = 7;

int LargestRectangleArea2(int chess[M][N], int M, int N)
{
    int* height = new int[N+1];
    memset(height, 0, sizeof(int)*(N+1));
    //height[N] = 0: 确保原数组height的最后一位能够得到计算
    //height[0...N-1] = 0: 表示当前行没有1

    int i,j;
    int area = 0;
    int cur;
    for(i = 0; i < M; i++)
    {
        for(j = 0; j < N; j++)  //每一行分别处理
        {
            if(chess[i][j] == 0)
            {
                height[j] = 0;
            }
            else
            {
                height[j] += chess[i][j];
            }
        }
        cur = LargestRectangleArea(height, N);
        area = max(cur, area);
    }
    delete[] height;
    return area;
}

int _tmain(int argc, _TCHAR* argv[])
{
    int chess[M][N]
    ={
        {0,1,0,1,1,1,1,0,0},
        {0,1,0,1,0,0,0,0,0},
        {0,1,1,1,1,1,1,0,0},
        {0,1,1,1,1,1,0,0,0},
        {0,1,0,1,1,1,1,0,0},
        {0,1,0,1,0,1,1,0,0},
        {0,1,1,0,1,0,0,0,0},
    };
    LargestRectangleArea2(chess, M, N);
    return 0;
}
```

Code Split

```
int LargestRectangleArea(int* height, int N)    //height[N]==0
{
    stack<int> s;
    int answer = 0;
    int temp;    //临时变量
    for (int i = 0; i <= N; )
    {
        if (s.empty() || height[i] > height[s.top()])
        {
            s.push(i);
            i++;
        }
        else
        {
            temp = s.top();
            s.pop();
            answer = max(answer,height[temp]*(s.empty() ? i : i-s.top()-1));
        }
    }
    return answer;
}
```

```
int _tmain(int argc, _TCHAR* argv[])
{
    int chess[M][N]
    ={
        {0,1,0,1,1,1,1,0,0},
        {0,1,0,1,0,0,0,0,0},
        {0,1,1,1,1,1,1,0,0},
        {0,1,1,1,1,1,0,0,0},
        {0,1,0,1,1,1,1,0,0},
        {0,1,0,1,0,1,1,0,0},
        {0,1,1,0,1,0,0,0,0},
    };
    LargestRectangleArea2(chess, M, N);
    return 0;
}
```

```
int LargestRectangleArea2(int chess[M][N], int M, int N)
{
    int* height = new int[N+1];
    memset(height, 0, sizeof(int)*(N+1));
    //height[N] = 0: 确保原数组height的最后一位能够得到计算
    //height[0...N-1] = 0: 表示当前行没有1

    int i,j;
    int area = 0;
    int cur;
    for(i = 0; i < M; i++)
    {
        for(j = 0; j < N; j++)  //每一行分别处理
        {
            if(chess[i][j] == 0)
            {
                height[j] = 0;
            }
            else
            {
                height[j] += chess[i][j];
            }
        }
        cur = LargestRectangleArea(height, N);
        area = max(cur, area);
    }
    delete[] height;
    return area;
}
```

找零钱

- 给定某不超过100万元的现金总额，兑换成数量不限的100、50、20、10、5、2、1元的纸币组合，共有多少种组合?

该问题的思考过程

- 此问题涉及两个类别：面额和总额。
  - 如果面额都是1元的，则无论总额多少，可行的组合数显然都为1。
  - 如果面额多一种，则组合数有什么变化呢？
- 定义dp[i][j]：使用面额小于等于i的钱币，凑成j元钱，共有多少种组合方法。
  - dp[100][500] = dp[50][500] + dp[100][400]
  - $dp[i][j] = dp[i_{small}][j] + dp[i][j-i]$
    - 不考虑j-i下溢出等边界问题

递推公式　$dp[i][j] = dp[i_{small}][j] + dp[i][j-i]$

- 使用dom[]={1,2,5,10,20,50,100}表示基本面额，i的意义从面额变成面额下标，则：
  - $dp[i][j] = dp[i-1][j] + dp[i][j-dom[i]]$
- 从而：

$$dp[i][j]=\begin{cases} dp[i-1][j]+dp[i][j-dom[i]], & j \ge dom[i] \\ dp[i-1][j], & j < dom[i] \end{cases}$$

- 初始条件：

$$\begin{cases} dp[0][j]=1 \\ dp[i][0]=1 \end{cases}$$

Code

```
int Charge(int value, const int* denomination, int size)
{
    int i;
    int** dp = new int*[size];  //dp[i][j]: 用i面额以下的组合成j元
    for(i = 0; i < size; i++)
        dp[i] = new int[value+1];

    int j;
    for(j = 0; j <= value; j++) //只用面额1元的
        dp[0][j] = 1;

    for(i = 1; i < size; i++)   //先用面额小的，再用面额大的
    {
        dp[i][0] = 1;   //原因：添加任何一个面额，就是一个有效组合
        for(j = 1; j <= value; j++)
        {
            if(j >= denomination[i])
                dp[i][j] = dp[i-1][j] + dp[i][j-denomination[i]];
            else
                dp[i][j] = dp[i-1][j];
        }
    }
    int time = dp[size-1][value];

    for(i = 0; i < size; i++)   //清理内存
        delete[] dp[i];
    delete[] dp;
    return time;
}

int _tmain(int argc, _TCHAR* argv[])
{
    int denomination[] = {1,2,5,10,20,50,100};  //面额
    int size = sizeof(denomination) / sizeof(int);
    int value = 200;
    int c = Charge(value, denomination, size);
    cout << c << endl;
    return 0;
}
```

滚动数组

- 将状态转移方程去掉第一维，很容易使用滚动数组，降低空间使用量。
- 原状态转移方程：

$$dp[i][j]=\begin{cases} dp[i-1][j]+dp[i][j-dom[i]], & j\ge dom[i] \\ dp[i-1][j], & j<dom[i] \end{cases}$$

- 滚动数组版本的状态转移方程：

$$dp[j]=last[j]+dp[j-dom[i]],\quad \left(j\ge dom[i]\right)$$

Code2

```
int Charge2(int value, const int* denomination, int size)
{
    int i;
    int* dp = new int[value+1]; //dp[j]: 凑成j元的组合数
    int* last = new int[value+1];

    int j;
    for(j = 0; j <= value; j++) //只用面额1元的
    {
        dp[j] = 1;
        last[j] = 1;
    }

    for(i = 1; i < size; i++)  //先用面额小的，再用面额大的
    {
        for(j = 1; j <= value; j++)
        {
            if(j >= denomination[i])
                dp[j] = last[j] + dp[j-denomination[i]];
        }
        memcpy(last, dp, sizeof(int)*(value+1));
    }
    int chargeTimes = dp[value];

    delete[] last;
    delete[] dp;
    return chargeTimes;
}
```

继续思考

- 请问：本问题的时间复杂度是多少？
- 在动态规划问题中，如果不求具体解，而只计算解的数目，往往可以使用滚动数组的方式降低空间使用量(甚至空间复杂度)
  - 由于滚动数组减少了维度，甚至代码会更简单
- 思考0-1背包问题和格子取数问题。

Scramble String

- 给定字符串str，可以使用二叉树递归的表示它，如图1的str="great"；
- 将二叉树的任意非叶结点的左右子树互换，可得到图2和图3。
- 我们称经过这样任意次互换得到的字符串互为Scramble String。如"great", "rgeat"，"rgtae"互为Scramble String。
- 给定两个字符串str1和str2，请判断它们是否为Scramble String。

```
    great
   /    \
  gr    eat
 / \    /  \
g   r  e   at
           / \
          a   t
```

```
    rgeat
   /    \
  rg    eat
 / \    /  \
r   g  e   at
           / \
          a   t
```

```
    rgtae
   /    \
  rg    tae
 / \    /  \
r   g  ta   e
       / \
      t   a
```

问题思考

- 若给定的两个字符串长度不相等，不可能是Scramble String；
- 若字符串str1,str2长度都是1；则只需判断str1[0]==str2[0]；
- 若字符串str1,str2长度都是2；则只需判断str1==str2或者reverse(str1)==str2；
- 若字符串str1,str2长度都是n，从1到n-1递归判断。

Code

```cpp
bool IsScramble(const string& str1, const string& str2)
{
    int size = (int)str1.size();
    if(str2.size() != size)
        return false;
    if(size <= 0)
        return true;
    if(size == 1)
        return str1[0] == str2[0];
    for(int k = 1; k < size; k++)
    {
        if(IsScramble(str1.substr(0, k), str2.substr(0, k))
            && IsScramble(str1.substr(k, size-k), str2.substr(k, size-k)) )
            return true;

        if(IsScramble(str1.substr(0, k), str2.substr(size-k, k))
            && IsScramble(str1.substr(k, size-k), str2.substr(0, size-k)) )
            return true;
    }
    return false;
}
```

```cpp
int _tmain(int argc, _TCHAR* argv[])
{
    string str1 = "31425";
    string str2 = "12345";
    //string str1 = "great";
    //string str2 = "rgtae";
    cout << IsScramble(str1, str2) << '\n';
    return 0;
}
```

Code2

```
bool IsScramble(const string& str1, const string& str2)
{
    int size = (int)str1.size();
    if(str2.size() != size)
        return false;
    //dp[0]未使用
    vector<vector<vector<bool> > > dp(size+1,
        vector<vector<bool> >(size, vector<bool>(size, false)));
    int i,j,r;
    for(i = 0; i < size; i++)
        for(j = 0; j < size; j++)
            dp[1][i][j] = (str1[i] == str2[j]);

    bool b;
    for(int k = 2; k <= size; k++)
    {
        for(i = 0; i <= size-k; i++)
        {
            for(j = 0; j <= size-k; j++)
            {
                b = false;
                for(r = 1; r < k; r++)
                {
                    if((dp[r][i][j] && dp[k-r][i+r][j+r])
                        ||(dp[r][i][j+k-r] && dp[k-r][i+r][j]))
                    {
                        b = true;
                        break;
                    }
                }
                dp[k][i][j] = b;
            }
        }
    }
    return dp[size][0][0];
}
```

```
int _tmain(int argc, _TCHAR* argv[])
{
    //string str1 = "31425";
    //string str2 = "12345";
    string str1 = "great";
    string str2 = "rgtae";
    cout << IsScramble(str1, str2) << '\n';
    return 0;
}
```

Code3

```
int _tmain(int argc, _TCHAR* argv[])
{
    //string str1 = "31425";
    //string str2 = "12345";
    string str1 = "great";
    string str2 = "rgtae";
    int size = (int)str1.size();
    if(str2.size() != size)
        cout << "-1\n";
    vector<vector<vector<int> > > dp(size+1, vector<vector<int> >(size, vector<int>(size, 0)));
    cout << IsScramble3(str1, str2, 0, 0, size, dp) << '\n';
    return 0;
}
```

Code3

```
int IsScramble3(const string& str1, const string& str2, int i, int j, int k,
    vector<vector<vector<int> > >& dp)
{
    if(k <= 0)
        return 1;
    if(k == 1)
        return ((str1[i] == str2[j]) ? 1 : -1);

    for(int r = 1; r < k; r++)
    {
        if(dp[r][i][j] == 0)
            dp[r][i][j] = IsScramble3(str1, str2, i, j, r, dp);
        if(dp[k-r][i+r][j+r] == 0)
            dp[k-r][i+r][j+r] = IsScramble3(str1, str2, i+r, j+r, k-r, dp);
        if((dp[r][i][j] == 1) && (dp[k-r][i+r][j+r] == 1))
            return 1;

        if(dp[r][i][j+k-r] == 0)
            dp[r][i][j+k-r] = IsScramble3(str1, str2, i, j+k-r, r, dp);
        if(dp[k-r][i+r][j] == 0)
            dp[k-r][i+r][j] = IsScramble3(str1, str2, i+r, j, k-r, dp);
        if((dp[r][i][j+k-r] == 1) && (dp[k-r][i+r][j] == 1))
            return 1;
    }
    return -1;
}
```

Palindrome Partitioning

- 给定一个字符串str，将str划分成若干子串，使得每一个子串都是回文串。在所有可能的划分中，子串数目最少是多少?
  - s="aab"，最少子串数目为2
    - "aa"， "b"
  - "abacdccdaa"：最少子串数目为4
    - "aba"、 "c"、 "dccd"、 "aa"

贪心法是不对的

- 贪心法：
  - 计算字符串S的最长回文子串，假定为A，则将S切分成B|A|C三部分，对B和C分别递归计算最长回文子串。
- 这是不对的。
- 反例：ADACDCAEA
  - 贪心划分：A|D|ACDCA|E|A
  - 最少回文划分：ADA|CDC|AEA

Palindrome Partitioning

- 假设当前已经计算完成前缀串str[0…i-1]的最少划分数目d[0…i-1]，其中d[j]表示前缀串str[0…j]的最少划分数目；
- 则：要划分前缀串str[0…i]，假定划分方案的最后一个子串为str[j+1…i]，显然，根据题意str[j+1…i]为回文串，并且str[0…j]一定是最少划分，即：
  - d[i]=min{d[j]+1|s[j+1…i]是回文串，i-1≥j≥0}
- 注意：
  - str[j…i]是否为回文串，本身是个小的动态规划
    - 若已知str[j+1…i-1]是回文串，则只需判断str[j]==str[i]；
  - 上述整个过程从后向前分析仍然可以得到类似的结论；

题中题

- 在计算str[i,i+1…j]是否是回文串这一子问题中，暴力也是无可厚非的。线性探索：j从i到n-1遍历即可。
- 事实上，可以事先缓存所有的str[i,i+1...j]是回文串的那些记录：用二维布尔数组p[n][n]就够了：p[i][j]的true/false表示了str[i,i+1...j]是否是回文串；
- 它本身是个小的动态规划：
  - 如果已知str[i+1...j-1]是回文串，那么，判断str[i,i+1...j]是否是回文串，只需要判断str[i]==str[j]就可以。

Code1

```
void CalcSubstringPalindrome(const char* str, int size, vector<vector<bool> >& p)
{
    int i,j;
    for(i = 0; i < size; i++)
        p[i][i] = true;
    for(i = size-2; i >= 0; i--)
    {
        p[i][i+1] = (str[i] == str[i+1]);
        for(j = i+2; j < size; j++)
        {
            if((str[i] == str[j]) && p[i+1][j-1])
                p[i][j] = true;
        }
    }
}

int MinPalindrome(const char* str)
{
    int size = (int)strlen(str);
    vector<vector<bool> > p(size, vector<bool>(size));
    CalcSubstringPalindrome(str, size, p);

    int i,j;
    int* minPart = new int[size];
    for(i = 0; i < size; i++)
    {
        if(p[0][i]) //本身是回文串，则不需要分割
        {
            minPart[i] = 1;
            continue;
        }
        minPart[i] = i+1;
        for(j = 0; j < i; j++)
        {
            if((p[j+1][i]) && (minPart[i] > minPart[j]+1))
            {
                minPart[i] = minPart[j]+1;
            }
        }
    }
    int mp = minPart[size-1];
    delete[] minPart;
    return mp;
}
```

思考

- 如果不止计算最小回文划分的切分数目，还要求最小回文划分本身，如何处理？
- 记录前驱：
  - 每一个前缀串最小回文划分的前驱

Code2

```
int _tmain(int argc, _TCHAR* argv[])
{
    const char* str = "abacdccdaa";
    int size = (int)strlen(str);
    int* pre = new int[size];
    cout << MinPalindrome2(str, pre, size);
    vector<string> result;
    Cut(str, pre, size, result);
    delete[] pre;
    Print(result);
    return 0;
}
```

```
void Cut(const char* str, const int* pre, int size,
        vector<string>& result)
{
    int nEnd = size-1;
    int nStart;
    while(pre[nEnd] != -1)
    {
        nStart = pre[nEnd];
        result.push_back(string(str+nStart+1,str+nEnd+1));
        nEnd = nStart;
    }
    result.push_back(string(str,str+nEnd+1));
    reverse(result.begin(), result.end());  //逆序
}
```

```
int MinPalindrome2(const char* str, int* pre, int size)
{
    vector<vector<bool> > p(size, vector<bool>(size));
    CalcSubstringPalindrome(str, size, p);

    //minPart[i]: str[0...i]最少划分数目
    int* minPart = new int[size];
    int nMin;
    int i,j;
    for(i = 0; i < size; i++)
    {
        if(p[0][i]) //本身是回文串，则不需要分割
        {
            minPart[i] = 1;
            pre[i] = -1;
            continue;
        }
        minPart[i] = i+1;
        nMin = -1;
        for(j = 0; j < i; j++)
        {
            if((p[j+1][i]) && (minPart[i] > minPart[j]+1))
            {
                minPart[i] = minPart[j]+1;
                nMin = j;
            }
        }
        pre[i] = nMin;
    }
    int mp = minPart[size-1];
    delete[] minPart;
    return mp;
}
```

Palindrome Partitioning II

- 给定一个字符串str，将str划分成若干子串，使得每一个子串都是回文串。计算str的所有可能的划分。
  - 单个字符构成的字符串，显然是回文串；所以，这个的划分一定是存在的。
- 如：s="aab"，返回
  - "aa"，"b"；
  - "a"，"a"，"b"。

Palindrome Partitioning II

- 思考：若当前计算得到了str[0…i-1]的所有划分，可否添加str[i…j]，得到更大的划分呢？
  - 显然，若str[i…j]是回文串，则可以添加。
- 剪枝：在每一步都可以判断中间结果是否为合法结果。
  - 回溯+剪枝——如果某一次发现划分不合法，立刻对该分支限界。
  - 一个长度为n的字符串，最多有n-1个位置可以截断，每个位置有两种选择，因此时间复杂度为$O(2^{n-1})=O(2^n)$。

Code

```
void FindSolution(const char* str, int size, int nStart, vector<vector<string> >& all,
                  vector<string>& solution, const vector<vector<bool> >& p)
{
    if(nStart >= size)
    {
        all.push_back(solution);
        return;
    }
    for(int i = nStart; i < size; i++)
    {
        if(p[nStart][i])
        {
            solution.push_back(string(str+nStart, str+i+1));
            FindSolution(str, size, i+1, all, solution, p);
            solution.pop_back();
        }
    }
}

void MinPalindrome3(const char* str, vector<vector<string> >& all)
{
    int size = (int)strlen(str);
    vector<vector<bool> > p(size, vector<bool>(size));
    CalcSubstringPalindrome(str, size, p);

    vector<string> solution;    //一个解
    FindSolution(str, size, 0, all, solution, p);
}
```

```
int _tmain(int argc, _TCHAR* argv[])
{
    const char* str = "abacdccdaa";
    vector<vector<string> > all;
    MinPalindrome3(str, all);
    PrintAll(all);
    return 0;
}
```

```
Count: 16
1: a | b | a | c | d | c | c | d | a | a
2: a | b | a | c | d | c | c | d | aa
3: a | b | a | c | d | cc | d | a | a
4: a | b | a | c | d | cc | d | aa
5: a | b | a | c | dccd | a | a
6: a | b | a | c | dccd | aa
7: a | b | a | cdc | c | d | a | a
8: a | b | a | cdc | c | d | aa
9: aba | c | d | c | c | d | a | a
10: aba | c | d | c | c | d | aa
11: aba | c | d | cc | d | a | a
12: aba | c | d | cc | d | aa
13: aba | c | dccd | a | a
14: aba | c | dccd | aa
15: aba | cdc | c | d | a | a
16: aba | cdc | c | d | aa
```

继续思考：动态规划

- 若已知：str[0…i-1]的所有回文划分φ(i)，
- 如何求str[0…i]的所有划分呢？
  - 如果子串str[j…i]是回文串，则将该子串和φ(j)共同添加到φ(i+1)中。
- 算法：
  - i从0到n，依次调用以下两步
    - 将集合φ(i+1)置空；
    - 遍历j(0≤j＜i)，若str[j,j+1…i]是回文串，则将{str[j…i]，φ(j)}添加到φ(i+1)中；
  - 最终返回φ(n)。

Code

```
void Add(vector<vector<string> >& to, const vector<vector<string> >& from, const string& sub)
{
    if(from.empty())
    {
        to.push_back(vector<string>(1, sub));
        return;
    }
    to.reserve(from.size());
    for(vector<vector<string> >::const_iterator it = from.begin(); it != from.end(); it++)
    {
        to.push_back(vector<string>());
        vector<string>& now = to.back();
        now.reserve(it->size()+1);
        for(vector<string>::const_iterator s = it->begin(); s != it->end(); s++)
            now.push_back(*s);
        now.push_back(sub);
    }
}

void MinPalindrome4(const char* str, vector<vector<string> >& all)
{
    int size = (int)strlen(str);
    vector<vector<bool> > p(size, vector<bool>(size));
    CalcSubstringPalindrome(str, size, p);

    //prefix[i]：长度为i的前缀串的所有划分方案
    vector<vector<string> >* prefix = new vector<vector<string> >[size];
    prefix[0].clear();  //仅为了强调长度为0的串，划分解为空
    int i,j;
    for(i = 1; i <= size; i++)  //考察str[0...i-1]
    {
        for(j = 0; j < i; j++)
        {
            if(p[j][i-1])
            {
                Add((i == size) ? all : prefix[i], prefix[j], string(str+j, str+i));
            }
        }
    }
    delete[] prefix;
}
```

```
Count: 16
1: aba | c | dccd | aa
2: a | b | a | c | dccd | aa
3: aba | c | d | cc | d | aa
4: a | b | a | c | d | cc | d | aa
5: aba | cdc | c | d | aa
6: a | b | a | cdc | c | d | aa
7: aba | c | d | c | c | d | aa
8: a | b | a | c | d | c | c | d | aa
9: aba | c | dccd | a | a
10: a | b | a | c | dccd | a | a
11: aba | c | d | cc | d | a | a
12: a | b | a | c | d | cc | d | a | a
13: aba | cdc | c | d | a | a
14: a | b | a | cdc | c | d | a | a
15: aba | c | d | c | c | d | a | a
16: a | b | a | c | d | c | c | d | a | a
```

Code *split*

```
void MinPalindrome4(const char* str, vector<vector<string> >& all)
{
    int size = (int)strlen(str);
    vector<vector<bool> > p(size, vector<bool>(size));
    CalcSubstringPalindrome(str, size, p);

    //prefix[i]：长度为i的前缀串的所有划分方案
    vector<vector<string> >* prefix = new vector<vector<string> >[size];
    prefix[0].clear();  //仅为了强调长度为0的串，划分解为空
    int i,j;
    for(i = 1; i <= size; i++)  //考察str[0...i-1]
    {
        for(j = 0; j < i; j++)
        {
            if(p[j][i-1])
            {
                Add((i == size) ? all : prefix[i], prefix[j], string(str+j, str+i));
            }
        }
    }
    delete[] prefix;
}
```

Code *split*

```cpp
void Add(vector<vector<string> >& to, const vector<vector<string> >& from, const string& sub)
{
    if(from.empty())
    {
        to.push_back(vector<string>(1, sub));
        return;
    }
    to.reserve(from.size());
    for(vector<vector<string> >::const_iterator it = from.begin(); it != from.end(); it++)
    {
        to.push_back(vector<string>());
        vector<string>& now = to.back();
        now.reserve(it->size()+1);
        for(vector<string>::const_iterator s = it->begin(); s != it->end(); s++)
            now.push_back(*s);
        now.push_back(sub);
    }
}
```

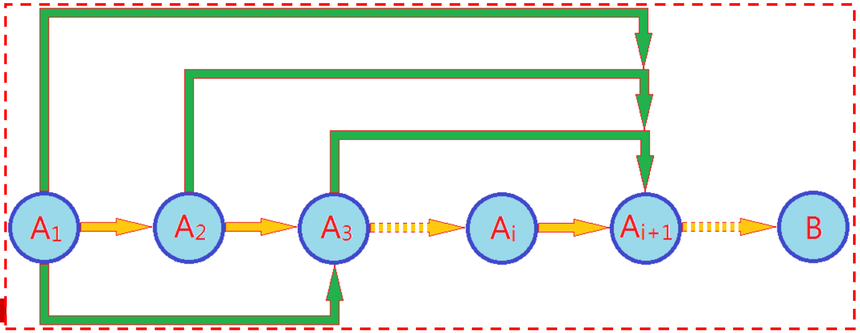
Palindrome Partitioning思考

- 与之类似的：
  - 给定仅包含数字的字符串，返回所有可能的有效IP地址组合。如："25525511135"，返回 "255.255.11.135"，"255.255.111.35"。
  - 该问题只插入3个分割位置。
  - 只有添加了第3个分割符后，才能判断当前划分是否合法。
    - 如：2.5.5.25511135，才能判断出是非法的。
      - 当然，它可以通过"25511135"大于"255.255"等其他限界条件"事先"判断。

DFS与DP深刻认识

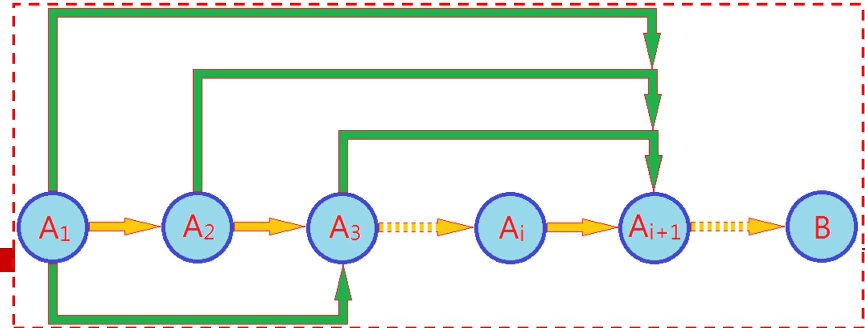
- DFS的过程，是计算完成了str[0…i]的切分，然后递归调用，继续计算str[i+1,i+2…n-1]的过程；
- 而DP中，假定得到了str[0…i-1]的所有可能切分方案，如何扩展得到str[0…i]的切分；
  - 注：上述两种方法都可从后向前计算得到对偶的分析。
- 从本质上说，二者是等价的：最终都搜索了一颗隐式树。
  - DFS显然是深度优先搜索的过程，而DP更像层序遍历。
  - 如果只计算回文划分的最少数目，动态规划更有优势；如果计算所有回文划分，DFS的空间复杂度比DP略优。

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next[0…i-1]，求next[i]；
  - 最长回文子串Manacher算法中，已知P[0…i-1]求P[i]
  - 隐马尔科夫HMM前向-后向算法、Viterbi算法
- 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

总结



- 无后效性
  - 计算A[i]时只读取A[0...i-1]，不修改——历史
  - 计算A[i]时不需要A[i+1...n-1]的值——未来
- 在实践中往往忽略无后效性：
  - 问题本身决定了它是成立的：格子取数问题
  - 通过更改计算次序可以达到该要求：矩阵连乘问题
- 哪些题目不适合用动态规划？
  - 状态转移方程的推导，往往陷入局部而忽略全局。在重视动态规划的同时，别忘了从总体把握问题。
    - 思考：围棋正方形问题

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
  - 小象学院
  - 大数据分析挖掘

感谢大家！

恳请大家批评指正！