1. 矩阵链乘法问题

问题描述：
给定一个 n 个矩阵的序列〈 A1,A2,A3...An〉,我们要计算他们的乘积：A1A2A3...An，由于矩阵乘法满足结合律，加括号不会影响结果，但是不同的加括号方法，算法复杂度有很大的差别：
考虑矩阵链：〈 A1,A2,A3〉 ，三个矩阵规模分别为 10×100、100×5、5×50
如果按((A1A2)A3)方式，需要做 10* 100* 5=5000 次，再与 A3 相乘，又需要 10* 5* 50=2500，共需要 7500 次运算：
如果按(A1（A2A3）)方式计算，共需要 100* 5* 50+10* 100* 50=75000 次标量乘法，具有 10 倍的差别。可见一个好的加括号方式，对计算效率有很大影响。

动态规划解法：

```
void Matrix_Chain_Order(int p[],int n)
{
    int i,j,L,k,q;
    for (i=1;i<=n;i++)          //先对单个矩阵的链，求解，即所有 m[i][i] =0;
    {
       m[i][i]=0;
    }
    for(L=2;L<=n;L++)         //从两个矩阵链的长度开始，逐次增加矩阵链的长度
         for(i=1;i<=n-L+1;i++)   //在给定 p[]中的矩阵链中，对所有种长度为 L 的情况计算
         {
               j = i+L-1;
               m[i][j] = -1;
               for(k=i;k<=j-1;k++)     //遍历所有可能的划分点 k，计算出最优的划分方案
               {
                  q = m[i][k]+m[k+1][j]+p[i-1]*p[k]*p[j];//计算划分的代价
                  if ( q < m[i][j] || m[i][j] == -1)
                  {
                      m[i][j] = q;         //最优的代价 q 保存在 m[i][j]中
                      s[i][j] = k;         //最优的划分位置 k 保存在 s[i][j]中
                  }
               }
         }

}
```

参考链接：
http://blog.csdn.net/loushixian099/article/details/46344175

2. 最长递增子序列（Longest Increasing Subsequence）

问题描述：
给定一个长度为 N 的数组，找出一个最长的单调自增子序列（不一定连续，但是顺序不能

乱）。例如：给定一个长度为 6 的数组 A{5， 6， 7， 1， 2， 8}，则其最长的单调递增子序列为{5，6，7，8}，长度为 4.

动态规划解法（O(NlogN)）：

```
int BinarySearch(int *array, int value, int nLength)
{
	int begin = 0;
	int end = nLength - 1;
	while(begin <= end)
	{
		int mid = begin + (end - begin) / 2;
		if(array[mid] == value)
			return mid;
		else if(array[mid] > value)
			end = mid - 1;
		else
			begin = mid + 1;
	}
	 return begin;
}

int LIS_DP_NlogN(int *array, int nLength)
{
	int B[nLength];
	int nLISLen = 1;
	B[0] = array[0];

	for(int i = 1; i < nLength; i++)
	{
		if(array[i] > B[nLISLen - 1])
		{
			B[nLISLen] = array[i];
			nLISLen++;
		}
		else
		{
			int pos = BinarySearch(B, array[i], nLISLen);
			B[pos] = array[i];
		}
	}
	return nLISLen;
}
```

参考链接：

http://blog.csdn.net/sgbfblog/article/details/7798737
http://qiemengdao.iteye.com/blog/1660229

3. 最长公共子序列（Longest Common Subsequence）

问题描述：
给定序列 X 和 Y，序列 Z 是 X 的子序列，也是 Y 的子序列，则 Z 是 X 和 Y 的公共子序列。例如 X=<A,B,C,B,D,A,B>，Y=<B,D,C,A,B,A>，那么序列 Z=<B,C,A>为 X 和 Y 的公共子序列，其长度为 3。但 Z 不是 X 和 Y 的最长公共子序列，而序列<B,C,B,A>和<B,D,A,B>也均为 X 和 Y 的最长公共子序列，长度为 4，而 X 和 Y 不存在长度大于等于 5 的公共子序列。

最优子结构：

- LCS问题具有最优子结构
  令 $X =< x_1, x_2, \ldots, x_m >$ 和 $Y =< y_1, y_2, \ldots, y_n >$ 为两个序列，
  $Z =< z_1, z_2, z_3, \ldots, z_k >$ 为 X 和 Y 的任意LCS。则

如果 $x_m = y_n$，则 $z_k = x_m = y_n$ 且 Z_{k-1} 是 X_{m-1} 和 Y_{n-1} 的一个LCS。
如果 $x_m \neq y_n$，那么 $z_k \neq x_m$，意味着 Z 是 X_{m-1} 和 Y 的一个LCS。
如果 $x_m \neq y_n$，那么 $z_k \neq y_n$，意味着 Z 是 X 和 Y_{n-1} 的一个LCS。

从上述的结论可以看出，两个序列的LCS问题包含两个序列的前缀的LCS，因此，LCS问题具有最优子结构性质。在设计递归算法时，不难看出递归算法具有子问题重叠的性质。

设 $C[i,j]$ 表示 X_i 和 Y_j 的最长公共子序列LCS的长度。如果 $i = 0$ 或 $j = 0$，即一个序列长度为0时，那么LCS的长度为0。根据LCS问题的最优子结构性质，可得如下公式：

$$C[i,j] = \begin{cases} 0, & \text{当} i = 0 \text{或} j = 0 \\ C[i-1, j-1] + 1, & \text{当} i, j > 0 \text{且} x_i = y_j \\ MAX(C[i, j-1], C[i-1, j]) & \text{当} i, j > 0 \text{且} x_i \neq y_j \end{cases}$$

动态规划解法：

```
#include <stdio.h>
#include <string.h>
#define MAXLEN 50

void LCSLength(char *x, char *y, int m, int n, int c[][MAXLEN], int b[][MAXLEN])
{
    int i, j;

    for(i = 0; i <= m; i++)
        c[i][0] = 0;
    for(j = 1; j <= n; j++)
        c[0][j] = 0;
```

```
    for(i = 1; i<= m; i++)
    {
        for(j = 1; j <= n; j++)
        {
            if(x[i-1] == y[j-1])
            {
                c[i][j] = c[i-1][j-1] + 1;
                //如果使用'↖'、'↑'、'←'字符，会有警告，也能正确执行。
                //本算法采用 1,3,2 三个整形作为标记
                b[i][j] = 1;                                    }
            else if(c[i-1][j] >= c[i][j-1])
            {
                c[i][j] = c[i-1][j];
                b[i][j] = 3;
            }
            else
            {
                c[i][j] = c[i][j-1];
                b[i][j] = 2;
            }
        }
    }
}

void PrintLCS(int b[][MAXLEN], char *x, int i, int j)
{
    if(i == 0 || j == 0)
        return;
    if(b[i][j] == 1)
    {
        PrintLCS(b, x, i-1, j-1);
        printf("%c ", x[i-1]);
    }
    else if(b[i][j] == 3)
        PrintLCS(b, x, i-1, j);
    else
        PrintLCS(b, x, i, j-1);
}

int main()
{
    char x[MAXLEN] = {"ABCBDAB"};
    char y[MAXLEN] = {"BDCABA"};
```

```
    //传递二维数组必须知道列数，所以使用 MAXLEN 这个确定的数
    int    b[MAXLEN][MAXLEN];
    int    c[MAXLEN][MAXLEN];

    int m, n;

    m = strlen(x);
    n = strlen(y);

    LCSLength(x, y, m, n, c, b);
    PrintLCS(b, x, m, n);

    return 0;
}
```

空间优化 O(mn)：

```
#include <stdio.h>
#include <string.h>
#define MAXLEN 50

void LCSLength(char *x, char *y, int m, int n, int c[][MAXLEN]) {
    int i, j;

    for(i = 0; i <= m; i++)
        c[i][0] = 0;
    for(j = 1; j <= n; j++)
        c[0][j] = 0;
    for(i = 1; i<= m; i++) {
        for(j = 1; j <= n; j++) {
            //仅仅去掉了对 b 数组的使用，其它都没变
            if(x[i-1] == y[j-1]) {
                c[i][j] = c[i-1][j-1] + 1;
            } else if(c[i-1][j] >= c[i][j-1]) {
                c[i][j] = c[i-1][j];
            } else {
                c[i][j] = c[i][j-1];

            }
        }
    }
}
/*
void PrintLCS(int c[][MAXLEN], char *x, int i, int j) {          //非递归版 PrintLCS
    static char s[MAXLEN];
```

```
	int k=c[i][j];
	s[k]='\0';
	while(k>0){
		if(c[i][j]==c[i-1][j]) i--;
		else if(c[i][j]==c[i][j-1]) j--;
		else{
			s[--k]=x[i-1];
			i--;j--;
		}
	}
	printf("%s",s);
}
*/
void PrintLCS(int c[][MAXLEN], char *x, int i, int j) {
	if(i == 0 || j == 0)
		return;
	if(c[i][j] == c[i-1][j]) {
		PrintLCS(c, x, i-1, j);
	} else if(c[i][j] == c[i][j-1])
		PrintLCS(c, x, i, j-1);
	else {
		PrintLCS(c, x, i-1, j-1);
		printf("%c ",x[i-1]);
	}
}

int main() {
	char x[MAXLEN] = {"ABCBDAB"};
	char y[MAXLEN] = {"BDCABA"};
	//char x[MAXLEN] = {"ACCGGTCGAGTGCGCGGAAGCCGGCCGAA"}; //算法导论上 222 页的
DNA 的碱基序列匹配
	//char y[MAXLEN] = {"GTCGTTCGGAATGCCGTTGCTCTGTAAA"};

	int	c[MAXLEN][MAXLEN];		//仅仅使用一个 c 表

	int m, n;

	m = strlen(x);
	n = strlen(y);

	LCSLength(x, y, m, n, c);
	PrintLCS(c, x, m, n);

	return 0;
```

}

参考链接：
http://blog.csdn.net/so_geili/article/details/53737001

4. 最大连续子序列和/积

问题描述：
Given (possibly negative) integers a1, a2, . . . , an, find the maximum value of subsequence sum. (For convenience, themaximum subsequence sum is 0 if all the integers are negative.)
Example:
For input -2, 11, -4, 13, -5, -2, the answer is 20 (a1 through a3).

动态规划解法：

```
void Max_Subsequence_Sum(int arr[], int length)
{
	int i;
	int left = 0;
	int right = 0;

	int maxsofar = 0;
	int maxendinghere = 0;
	//该数组保存以当前 i 结尾的最大子序列和
	int maxsumhere[length];

	//在每次循环之前，maxendinghere 是结束位置为 i-1 的最大子序列的和，也就是以位置 i-1 结尾的最大子序列和
	//循环开始后，赋值语句修改它以包含截止于位置 i 的最大子序列和，
	//当它变为负值时，就将它重新设为 0（因为截止于 i 的最大子序列现在是空序列了）
	for(i = 0; i < length; i++)
	{
		maxendinghere = max(maxendinghere + arr[i], 0);
		maxsumhere[i] = maxendinghere;
		maxsofar = max(maxsofar, maxendinghere);
	}

	for(i = 0; i < length; i++)
		printf("%d	", maxsumhere[i]);

	printf("\nMax Subsequence Sum Is: %d\n", maxsofar);

	//打印输出最大子序列和开始位置和结束位置
	i = length-1;
	while(maxsumhere[i] != maxsofar)
```

```
            i--;
        right = i;
        for(i; i >= 0; i--)
        {
            if(maxsumhere[i] == 0 || i == 0)
            {
                left = i;
                break;
            }
        }
        if(maxsumhere[i] == 0)
            left++;

        printf("\nLeft Is: %d \n Right Is: %d \n", left,right);
}
```

参考资料：

1. 编程珠玑。第八章
2. https://github.com/LockLee/Algorithms/blob/master/Dynamic_Programming/Maximum_Subsequence_Sum/max_subsequence_sum.c

5. 硬币找零问题

问题描述：

给予不同面值的硬币若干种种（每种硬币个数无限多），用若干种硬币组合为某种面额的钱，使硬币的的个数最少。假设要用 50、20、10、5、1（元）找出 87 元来，任何人都可以简单地得出：1 张 50、1 张 20、1 张 10、1 张 5 和 2 张 1 元就可以满足。

贪心解法：

输入 money=87，changes={ 50, 20, 10, 5, 1 }，则结果为：{ 1, 1, 1, 1, 2 }。这种方法的特点是：从最大额的零钱开始，逐次凑出需要的数额来，不关心总的数目是否真的是最小。这样的算法也形象地称为“贪心算法”，而找出最少数目的零钱的问题称为“最优化问题”。

```
public static int[] MakeChange(int money, int[] changes)
{
    int[] result = new int[changes.Length];

    for (int i = 0; i < changes.Length; i++)
    {
        result[i] = money / changes[i];
        money = money % changes[i];

        if (money == 0) { break; }
    }
```

```
    return result;
}
```

动态规划解法：
对于数额 n，可做如下考虑：
1）如果 n = 1，则用 1 个 1 元来找零，这就是最优解；
2）如果 n > 1，则对于每个可能的值 i，分别找出 i 元和 n-i 元来。

```
#include<iostream>
using namespace std;
//money 需要找零的钱
//coin 可用的硬币
//硬币种类
void FindMin(int money,int *coin, int n)
{
    //存储 1...money 找零最少需要的硬币的个数
    int *coinNum=new int[money+1]();
    //最后加入的硬币，方便后面输出是哪几个硬币
    int *coinValue=new int[money+1]();
    coinNum[0]=0;

    for(int i=1;i<=money;i++)
    {
        //i 面值的钱找零需要的最少硬币个数
        int minNum=i;
        //这次找零，在原来的基础上需要的硬币
        int usedMoney=0;
        for(int j=0;j<n;j++)
        {
            //找零的钱大于这个硬币的面值
            if(i>=coin[j])
            {
                //if(coinNum[i-coin[j]]+1<=minNum)//所需硬币个数减少了
                /*
                上面的判断语句有问题，在更新时，需要判断 i-coin[j]是否能找的开，如
                果找不开，就不需要更新。
                多谢 zywscq  指正
                */

                //所需硬币个数减少了,并且 i 要能被找开
                //i 要能被找开有两种可能，i 能被直接找开或者 i 能被 coinValue[i-coin[j]]
                   和 coin[j]找开
                //coinValue[i-coin[j]] == 0  表示 coinValue[i-coin[j]]不能被找开
                if(coinNum[i-coin[j]]+1<=minNum&&(i==coin[j]||coinValue[i-coin[j]]!=0))
                {
```

```
                    minNum=coinNum[i-coin[j]]+1;//更新
                    usedMoney=coin[j];//更新
                }
            }
        }
        coinNum[i]=minNum;
        coinValue[i]=usedMoney;
    }

    //输出结果
    if(coinValue[money]==0)
        cout<<"找不开零钱"<<endl;
    else
    {
        cout<<"需要最少硬币个数为："<<coinNum[money]<<endl;
        cout<<"硬币分别为:";
        while(money>0)
        {
            cout<<coinValue[money]<<",";
            money-=coinValue[money];
        }
    }
    delete []coinNum;
    delete []coinValue;
}
int main()
{
    int Money=18;
    int coin[]={1,2,5,9,10};
    FindMin(Money,coin,5);

}
```

参考链接：
http://www.cnblogs.com/anderslly/archive/2011/03/06/making-changes.html
http://blog.csdn.net/kangroger/article/details/36036101

6.1 0-1 背包问题

问题描述：
有 N 件物品和一个容量为 W 的背包。第 i 件物品的费用是 w[i]，价值是 v[i]。求解将哪些物品装入背包可使价值总和最大。

动态规划解法：

```
#include <iostream>
using namespace std;
#define max(a,b) (((a) > (b)) ? (a) : (b))
int w[]={0,3,6,3,8,6};//商品重量
int v[]={0,4,6,6,12,10};//商品价值
int W = 10; //背包容量
int c[6][11]={0};//c[i][j]表示在商品 1 到 i 中，背包容量为 j 时，最大价值
void Package0_1(int w[],int v[],int W,int n,int c[][11])//
{

    for(int i=1;i<=n;i++)                //逐行填表 c[i][j]
    {
        for (int j=1;j<=W;j++)
        {
            if ( i == 1)            //填写第 1 行时，不参考其他行
            {
                if (j < w[i])
                    c[i][j]=0;
                else
                    c[i][j] = v[i];
            }

            else
            {
                if ( j < w[i])    //背包容量小于商品 i 的重量，商品 i 一定不选
                {
                    c[i][j] = c[i-1][j];
                }
                else
                {
                    c[i][j] = max(c[i-1][j],v[i]+c[i-1][j-w[i]]);//比较选与不选商品 i 的背包总
价值大小
                }
            }
        }
    }

    for(int m =1;m<6;m++)
    {
        for (int n=0;n<11;n++)
        {
            cout<<c[m][n]<<"    ";
        }
        cout<<endl;
```

```
        }

}
void Print_Package0_1(int c[][11])    //构造解
{
        int i=5;
        int j=10;
        cout<<"总价值为"<<c[i][j]<<endl;
        while(i!=1)
        {
                if ( c[i][j] == c[i-1][j] )
                {
                        cout<<"商品"<<i<<"不选"<<endl;
                }
                else
                {
                        cout<<"商品"<<i<<"选"<<endl;
                        j = j - w[i];
                }
                i--;
        }

        if ( c[i][j] == 0)      //
        {
                cout<<"商品"<<i<<"不选"<<endl;
        }
        else
        {
                cout<<"商品"<<i<<"选"<<endl;
        }

}
int main()
{

    Package0_1(w,v,W,5,c);
    Print_Package0_1(c);
    return 0;
}
```

参考链接：

http://love-oriented.com/pack/Index.html

http://blog.csdn.net/luoshixian099/article/details/46572285

http://blog.csdn.net/kangroger/article/details/38864689

http://blog.csdn.net/u014627487/article/details/42969671

6.2　多重背包问题：
问题描述：
已知:有一个容量为 V 的背包和 N 件物品，第 i 件物品的重量是 weight[i]，收益是 cost[i]。
条件:每种物品都有无限件，能放多少就放多少。
问题:在不超过背包容量的情况下，最多能获得多少价值或收益
举例：物品个数 N = 3，背包容量为 V = 5，则背包可以装下的最大价值为 40.

	物品一	物品二	物品三
重量	3	2	2
价值	5	10	20

动态规划解法：
https://github.com/LockLee/Algorithms/tree/master/Dynamic_Programming/Knapsack/Unbounded_knapsack_problem
参考链接：
http://blog.csdn.net/insistGoGo/article/details/11081025
http://blog.csdn.net/qq_34374664/article/details/56015253

7. 最优二叉搜索树

问题描述：
给定 n 个互异的关键字组成的序列 K=<k1,k2,...,kn>，且关键字有序（k1<k2<...<kn），我们想从这些关键字中构造一棵二叉查找树。对每个关键字 ki，一次搜索搜索到的概率为 pi。可能有一些搜索的值不在 K 内，因此还有 n+1 个“虚拟键”d0,d1,...,dn，他们代表不在 K 内的值。具体：d0 代表所有小于 k1 的值，dn 代表所有大于 kn 的值。而对于 i = 1,2,...,n-1,虚拟键 di 代表所有位于 ki 和 ki+1 之间的值。对于每个虚拟键，一次搜索对应于 di 的概率为 qi。要使得查找一个节点的期望代价（代价可以定义为：比如从根节点到目标节点的路径上节点数目）最小，就需要建立一棵最优二叉查找树。

动态规划解法：
//最优二叉查找树

```
#include <iostream>

using namespace std;

const int MaxVal = 9999;

const int n = 5;
//搜索到根节点和虚拟键的概率
double p[n + 1] = {-1,0.15,0.1,0.05,0.1,0.2};
double q[n + 1] = {0.05,0.1,0.05,0.05,0.05,0.1};
```

```
int root[n + 1][n + 1];//记录根节点
double w[n + 2][n + 2];//子树概率总和
double e[n + 2][n + 2];//子树期望代价

void optimalBST(double *p,double *q,int n)
{
	//初始化只包括虚拟键的子树
	for (int i = 1;i <= n + 1;++i)
	{
		w[i][i - 1] = q[i - 1];
		e[i][i - 1] = q[i - 1];
	}

	//由下到上，由左到右逐步计算
	for (int len = 1;len <= n;++len)
	{
		for (int i = 1;i <= n - len + 1;++i)
		{
			int j = i + len - 1;
			e[i][j] = MaxVal;
			w[i][j] = w[i][j - 1] + p[j] + q[j];
			//求取最小代价的子树的根
			for (int k = i;k <= j;++k)
			{
				double temp = e[i][k - 1] + e[k + 1][j] + w[i][j];
				if (temp < e[i][j])
				{
					e[i][j] = temp;
					root[i][j] = k;
				}
			}
		}
	}
}

//输出最优二叉查找树所有子树的根
void printRoot()
{
	cout << "各子树的根： " << endl;
	for (int i = 1;i <= n;++i)
	{
		for (int j = 1;j <= n;++j)
		{
```

```
            cout << root[i][j] << " ";
        }
        cout << endl;
    }
    cout << endl;
}

//打印最优二叉查找树的结构
//打印出[i,j]子树，它是根 r 的左子树和右子树
void printOptimalBST(int i,int j,int r)
{
    int rootChild = root[i][j];//子树根节点
    if (rootChild == root[1][n])
    {
        //输出整棵树的根
        cout << "k" << rootChild << "是根" << endl;
        printOptimalBST(i,rootChild - 1,rootChild);
        printOptimalBST(rootChild + 1,j,rootChild);
        return;
    }

    if (j < i - 1)
    {
        return;
    }
    else if (j == i - 1)//遇到虚拟键
    {
        if (j < r)
        {
            cout << "d" << j << "是" << "k" << r << "的左孩子" << endl;
        }
        else
            cout << "d" << j << "是" << "k" << r << "的右孩子" << endl;
        return;
    }
    else//遇到内部结点
    {
        if (rootChild < r)
        {
            cout << "k" << rootChild << "是" << "k" << r << "的左孩子" << endl;
        }
        else
            cout << "k" << rootChild << "是" << "k" << r << "的右孩子" << endl;
    }
```

```
        printOptimalBST(i,rootChild - 1,rootChild);
        printOptimalBST(rootChild + 1,j,rootChild);
}

int main()
{
        optimalBST(p,q,n);
        printRoot();
        cout << "最优二叉树结构：" << endl;
        printOptimalBST(1,n,-1);
}
```

参考链接：
http://blog.csdn.net/xiajun07061225/article/details/8088784

8. Wine Profit

Problem Description：
"Imagine you have a collection of N wines placed next to each other on a shelf. For simplicity, let's number the wines from left to right as they are standing on the shelf with integers from 1 to N, respectively. The price of the ith wine is pi. (prices of different wines can be different).
Because the wines get better every year, supposing today is the year 1, on year y the price of the ithwine will be y*pi, i.e. y-times the value that current year.
You want to sell all the wines you have, but you want to sell exactly one wine per year, starting on this year. One more constraint - on each year you are allowed to sell only either the leftmost or the rightmost wine on the shelf and you are not allowed to reorder the wines on the shelf (i.e. they must stay in the same order as they are in the beginning).
You want to find out, what is the maximum profit you can get, if you sell the wines in optimal order?"
So, for example, if the prices of the wines are (in the order as they are placed on the shelf, from left to right): p1=1, p2=4, p3=2, p4=3. The optimal solution would be to sell the wines in the order p1, p4, p3, p2 for a total profit 1 * 1 + 3 * 2 + 2 * 3 + 4 * 4 = 29.

Solution of Dynamic Programming：

```
#include <stdio.h>
#include <stdlib.h>

//用于存放最大利润，从 m[1][1]开始
int m[10][10] = {0};
//用于存放出售策略，从 s[1][1]开始
int s[10][10] = {0};
```

```
#define max(a, b) (a>b) ? a : b

/*
 *求最大利润函数
 *
 *参数 p[]: 表示每瓶酒的售价，从 p[0]开始，p[0]表示第一瓶酒的售价
 *        n:  表示共有 n 瓶酒
 *
 *m[i][j]表示 Pi 到 Pj 瓶酒的最大利润
 *s[i][i]表示 Pi 到 Pj 瓶酒当前应出售的酒编号
 */
void Wines_Profit(int p[], int n)
{
    int i, j, L, k;
    int profit;
    int year;

    for(i = 1; i <= n; i++)
    {
        m[i][i] = n * p[i-1];
        s[i][i] = i;
    }

    for(L = 2; L <= n; L++)
    {
        for(i = 1; i <= n-L+1; i++)
        {
            j = i+L-1;
            year = n-L+1;
            if((m[i][j-1] + year*p[j-1]) > (m[i+1][j]+year*p[i-1]))
            {
                m[i][j] = m[i][j-1] + year*p[j-1];
                s[i][j] = j;
            }
            else
            {
                m[i][j] = m[i+1][j] + year*p[i-1];
                s[i][j] = i;
            }
        }
    }
}

/*
```

```
 *打印输出最大利润方案
 *
 *参数     p[]: 表示每瓶酒的售价，从 p[0]开始，p[0]表示第一瓶酒的售价
 *        m[][]: 表示 Pi 到 Pj 瓶酒的最大利润
 *        s[][]: 表示 Pi 到 Pj 瓶酒当前应出售的酒编号
 *        i: 表示从第 i 瓶酒开始
 *             j：表示到第 j 瓶酒结束
 *
 */
void Print_Optimal_Profit(int p[], int m[][10], int s[][10], int i, int j)
{
    int tmp = s[i][j];
    int year = 1;

    while(i != j)
    {
        printf("P%d    ", s[i][j]);
        tmp = m[i][j] - (year++)*p[s[i][j]-1];
        if(m[i+1][j] == tmp)
            i++;
        else
            j--;
    }

    printf("P%d    ", s[i][j]);
    printf("\n");
}

void main()
{
    int i, j;
    int count = 0;

    int p[] = {2, 3, 5, 1, 4};
    int length = sizeof(p)/sizeof(p[0]);

    Wines_Profit(p, length);

    printf("Max Profit:%d\n",m[1][length]);

    Print_Optimal_Profit(p, m, s, 1, 5);
}
```

参考链接：

https://www.hackerearth.com/zh/practice/algorithms/dynamic-programming/introduction-to-dynamic-programming-1/tutorial/

9. 编辑距离（Edit Distance）
给定 2 个字符串 a, b. 编辑距离是将 a 转换为 b 的最少操作次数，操作只允许如下 3 种：

1. 插入一个字符，例如：fj -> fxj
2. 删除一个字符，例如：fxj -> fj
3. 替换一个字符，例如：jxj -> fyj

动态规划解法：

```
int edit_distance(char *a, char *b)
{
	int lena = strlen(a);
	int lenb = strlen(b);
	int d[lena+1][lenb+1];
	int i, j;

	for (i = 0; i <= lena; i++) {
		d[i][0] = i;
	}
	for (j = 0; j <= lenb; j++) {
		d[0][j] = j;
	}

	for (i = 1; i <= lena; i++) {
		for (j = 1; j <= lenb; j++) {
			// 算法中 a, b 字符串下标从 1 开始，c 语言从 0 开始，所以 -1
			if (a[i-1] == b[j-1]) {
				d[i][j] = d[i-1][j-1];
			} else {
				d[i][j] = min_of_three(d[i-1][j]+1, d[i][j-1]+1, d[i-1][j-1]+1);
			}
		}
	}

	return d[lena][lenb];
}
```

参考链接：
http://www.dreamxu.com/books/dsa/dp/edit-distance.html

常见的动态规划问题分析与求解：

http://www.cnblogs.com/wuyuegb2312/p/3281264.html#q1a1
http://www.jianshu.com/p/8fae6529dafa
http://www.cnblogs.com/Anker/archive/2013/03/15/2961725.html