矩阵连乘

动态规划简化矩阵连乘问题复杂度的方法就是在适当的位置加一个括号。加上括号之后可以简化复杂度。
对于矩阵连乘问题：从矩阵A_i一直乘到矩阵A_j，所需要的最好计算次数就是$p_{i,j}$，那么递推式如下：

$m[i][j]=0(i==j)$或$m[i][j]=min(m[i][k]+m[k+1][j]+p_{i-1}*p_j*p_k)$
$m[i][j]$给出了计算连乘所需要的最优值。其中$i\leq k<j$
因此针对普通的n个矩阵连乘算法，答案就是$m[1,n]$。

该算法主要对i,j,k进行三重循环，复杂度上界就是$O(n^3)$.
在循环过程中可以加入判断m大小的代码，记录下最佳断点k的大小。后续可以直接输出最优方案。

电路布线

~~课本的描述有点太复杂了咱们可以找个更简单的~~
例题：

给定一个二分图，其中左半部包含 n_1 个点（编号 $1\sim n_1$），右半部包含 n_2 个点（编号 $1\sim n_2$），二分图共包含 m 条边。

数据保证任意一条边的两个端点都不可能在同一部分中。

请你求出二分图的最大匹配数。

输入格式

第一行包含三个整数 n_1、 n_2 和 m。

接下来 m 行，每行包含两个整数 u 和 v，表示左半部点集中的点 u 和右半部点集中的点 v 之间存在一条边。

输出格式

输出一个整数，表示二分图的最大匹配数。

核心本是匈牙利算法，但由于课本将其归类在dp,因此我们也只用dp。
问题：求一个二分图的最大匹配数（二分图中不相交的边集合模最大是多大）

匈牙利算法解决
算法本质很简单：
对于两个点集我们设为
$N1,N2$,从其中任意一个的第一个点开始，随机链接对面点集中的任意一个点，形成一条边。如果对面的点还有其他的点相连接(这个点我们设置为n_3，而且n_3还有其他的点链接，那么我们让n_2现在改成与n_1相连接，
复杂度理论上是O（mn) (m n分别是两个点集的点数量)但是事实上复杂度通常实际运行中远远小于mn。

一份代码（但算法分析与设计不考察实现）：

```
#include<iostream>
#include<cstring>
using namespace std;
const int N = 510 , M = 100010;
int n1,n2,m;
int h[N],ne[M],e[M],idx;
bool st[N];
int match[N];

void add(int a , int b)
{
    e[idx] = b, ne[idx] = h[a], h[a] = idx++;
}

void init()
{
    memset(h,-1,sizeof h);
}

int find(int x)
{
    //遍历自己喜欢的女孩
    for(int i = h[x] ; i != -1 ;i = ne[i])
    {
        int j = e[i];
        if(!st[j])//如果在这一轮模拟匹配中,这个女孩尚未被预定
        {
            st[j] = true;//那x就预定这个女孩了
            //如果女孩j没有男朋友，或者她原来的男朋友能够预定其它喜欢的女孩。配对成功
            if(!match[j]||find(match[j]))
            {
                match[j] = x;
                return true;
            }

        }
    }
    //自己中意的全部都被预定了。配对失败。
    return false;
}
int main()
```

```
{
    init();
    cin>>n1>>n2>>m;
    while(m--)
    {
        int a,b;
        cin>>a>>b;
        add(a,b);
    }


    int res = 0;
    for(int i = 1; i <= n1 ;i ++)
    {
         //因为每次模拟匹配的预定情况都是不一样的所以每轮模拟都要初始化
          memset(st,false,sizeof st);
        if(find(i))
          res++;
    }

    cout<<res<<endl;
}
```

dp解决(课本要求方式)
按照课本问题被迫被描述为以下非常麻烦的形式：
我们规定集合$N(i,j)$的意义是节点编号小于等于i而且我们遍历的对面点集的编号小于等于j时，最大匹配对应的集合。最大匹配数我们定义为$size(i,j)$.那么可知：
如果j<= i这个点所连接的对应节点的话，(i, i连接的点)是不在我们的考虑范围内的，也就是说：
$size(i,j) = size(i-1,j).$
如果j>=i所连接的点的编号的话，那么我们做出权衡是否要加入这条边：
$size(i,j) = max(size(i-1,j), size(i-1,pi(i)-1)+1)$。

当i=1时，我们先进行一次初始化，令j<pi(i)时，size(i,j)为0，大于等于时为1。

易得，算法复杂度为O（n^2）

如果要复原出最优解的话，我们再定义traceback算法。
从后往前遍历，用net数组存边。如果$size(i,j)! = size(i-1,j)$，则在net中加入i。如果对应的pi(i)-1>=c（1），则再加入节点1。
复杂度是O（n）。

最长公共子序列

给定两个长度分别为 N 和 M 的字符串 A 和 B，求既是 A 的子序列又是 B 的子序列的字符串长度最长是多少。

输入格式

第一行包含两个整数 N 和 M。

第二行包含一个长度为 N 的字符串，表示字符串 A。

第三行包含一个长度为 M 的字符串，表示字符串 B。

字符串均由小写字母构成。

输出格式

输出一个整数，表示最大长度。

对于序列
$A[n]$与$B[n]$,我们构造二维数组$s[i,j]$,记录的是当我们遍历到$a[i]$与$b[j]$时得到的最长公共子序列长度是多少。
显然答案就是$s[n,n]$。
状态过程中，我们可以得知$s[i,j]$以以下几种方式进行更新。

- $s[i,j]=0(i==0||j==0)$
- $s[i,j]=s[i-1,j-1]+1(a[i]==b[j])$
- $s[i,j]=max(s[i-1,j],s[i,j-1])(a[i]!=b[j])$
- 对于$i==1$并且$j==0$或者$i==0$并且$j==1$时我们将其初始化成1.

一份可行代码如下：

```
# include<bits/stdc++.h>
using namespace std;
typedef long long ll;
ll n,m;
char a[1005];
char b[1005];
ll f[1005][1005];

int main (void)
{

        cin.tie(0);
        cout.tie(0);
        cin>>n>>m;
        cin>>a+1>>b+1;

        for(int i=1;i<=n;i++)
        {
                for(int j=1;j<=m;j++)
```

```
                {
                f[i][j]=max(f[i-1][j],f[i][j-1]);
                if(a[i]==b[j])
                {
                        f[i][j]=max(f[i][j],f[i-1][j-1]+1);

                }
        }
                        }
        cout<<f[n][m]<<endl;
        return 0;

}
```

除此之外我们还可以找出最优解时这个子序列具体是哪些元素。

在上述算法中，我们再开一个B数组。每次更新$s[i,j]$的值时，我们也对$b[i,j]$进行更新。

- 如果$s[i,j] == s[i-1,j-1]$我们就令$b[i,j] = 1$
- 如果$s[i,j] == s[i-1,j]$，我们就令$b[i-1,j] = 2$
- 如果$s[i,j] === s[i,j-1]$,我们就令$b[i,j-1] = 3$

整个过程做完之后，我们开始构造最优解。

从最后元素末尾开始查找。例如$b[n][m]$

- 如果$b[i][j] == 1$，那么说明$s[i,j]$是由$s[i-1][j-1]$加上$a[i]$和$b[j]$得到的，直接输出$a[i]$
- 如果$b[i][j] == 2$，那么说明$s[i,j]$是由$s[i-1,j]$得到的，把递归节点转到$i-1,j$处继续递归
- 如果$b[i][j] == 3$,和2同理类比处理
- 当$i == 0$或者$j == 0$时退出

易得复杂度为$O(m+n)$(设两个序列长n和m)

0-1背包问题

有 N 件物品和一个容量是 V 的背包。每件物品只能使用一次。

第 i 件物品的体积是 v_i，价值是 w_i。

求解将哪些物品装入背包，可使这些物品的总体积不超过背包容量，且总价值最大。
输出最大价值。

输入格式

第一行两个整数，N，V，用空格隔开，分别表示物品数量和背包容积。

接下来有 N 行，每行两个整数 v_i, w_i，用空格隔开，分别表示第 i 件物品的体积和价值。

输出格式

输出一个整数，表示最大价值。

简而言之问题就是对于一连串的物品我们对每一个物品都有两个选择--把他装进背包或者不装。我们的目的是在尽可能地填满背包的前提下，背包中物品价值的最大值。

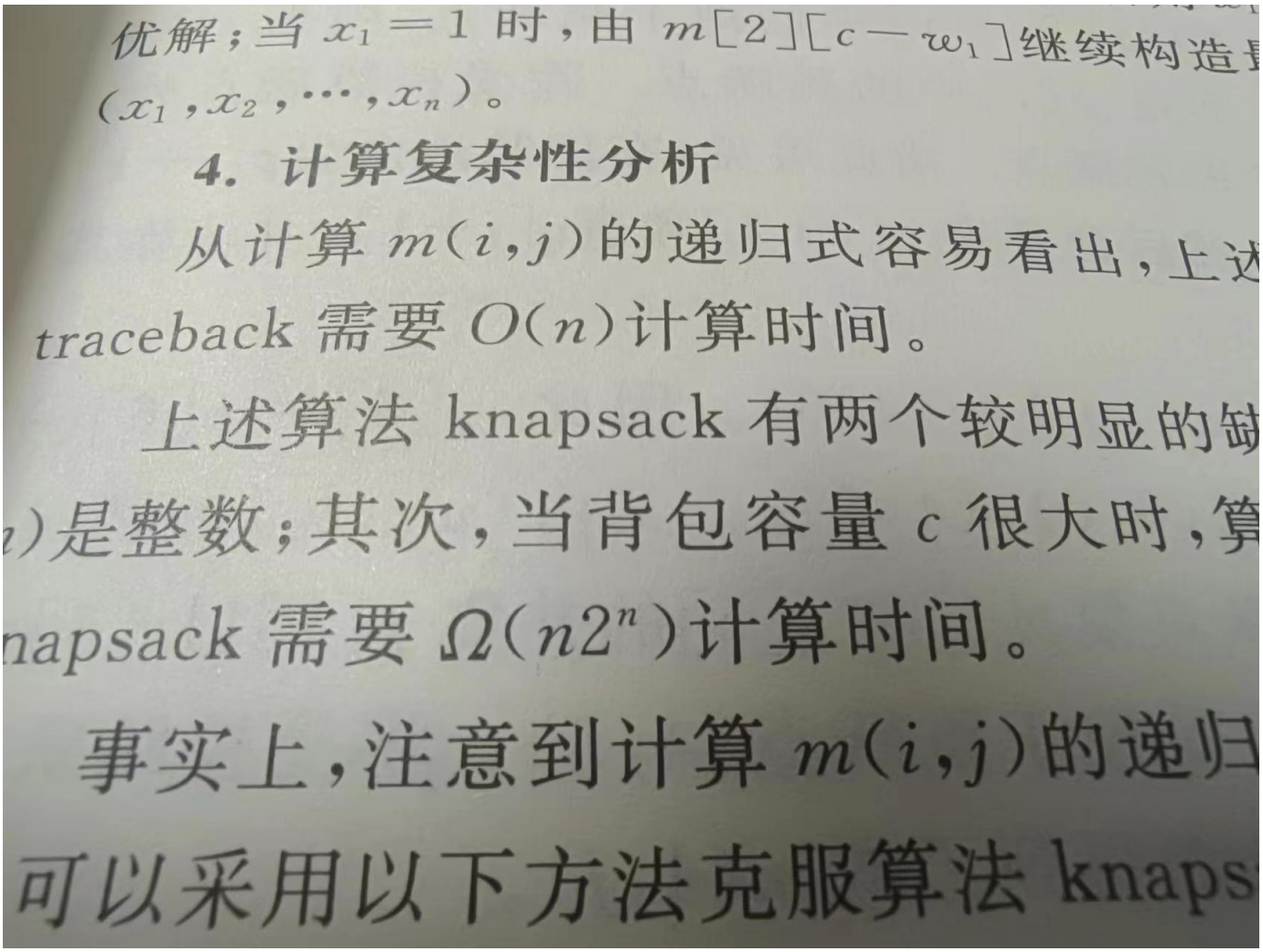
那么我们对每个物品有两种选择--装进去，不装。每次迭代也考虑以下情况：

- 能装我们就先看看，是装了更大呢还是不装好；
- 不能装就不用想跳过

我们构造二维数组
$s[i,j]$，表示我们当前考虑到了第i个物品，而且我们总共当前背包容积装了重量j的东西
那么迭代式子就是：
$s[i,j] = s[i+1,j]$(装不下)
$s[i,j] = max(s[i+1,j], s[i+1][j-w_i] + v[i])$(我们比比是装还是不装)

在这里我们发现，每次迭代最多用到i的上一层结果，因此我们可以将二维数组压缩成为一维数组，大大提高了代码的可读性，同时大幅减少了空间占用。

在这里实现方法我们选用更简单的方法（书上的实现方法大概率不会考，因为我认为任何一个稍微还有一点点学术信仰的学校都不会在他们的算法考试里考一个$n2^n$的实现）



> 传说中的指数级dp!!!!

我们用这串简化后的代码：

```
# include<bits/stdc++.h>
using namespace std;
typedef long long ll;

const int N=1010;
int dp[N];
int w[N],v[N];
int n,m;

int main (void)
{
        cin>>n>>m;
        for(int i=1;i<=n;i++)
        cin>>v[i]>>w[i];

        for(int i=1;i<=n;i++)
```

```
        for(int j=m;j>=v[i];j--)
        dp[j]=max(dp[j],dp[j-v[i]]+w[i]);
        cout<<dp[m]<<endl;

        return 0;

}
```

书上的算法经过一次简化之后复杂度是$O(2^n)$。最好还是背一下。
在此的代码复杂度为$O(nm)$(粗略计算，如有不对恳请更正)