动态规划初阶

定义

动态规划（dynamic programming），就是把原问题分解成相对简单的子问题的方法。

它常常适用于具有重叠子问题或者最优子结构性质的问题，它将问题转化为多个子问题，并且记录这些子问题的结果，逐步得到最终问题的结果。动态规划所耗时间往往远少于朴素解法。

通常许多子问题非常相似，为此动态规划法试图仅仅解决每个子问题一次，从而减少计算量：一旦某个给定子问题的解已经算出，则将其记忆化存储，以便下次需要同一个子问题解之时直接查表。这种做法在重复子问题的数目关于输入的规模呈指数增长时特别有用。

严格意义上，动态规划只能用来解决最优化问题，但在 OI 中，计数等非最优化问题的递推解法也常被不规范地称作 DP。事实上，动态规划与其它类型的递推的确有很多相似之处，学习时可以注意它们之间的异同。

斐波那契数列

来看看我们熟悉的斐波那契数列的题目，体会什么是动态规划

有一座高度是10级台阶的楼梯，从下往上走，每跨一步只能向上1级或者2级台阶。要求用程序来求出一共有多少种走法。

比如，每次走1级台阶，一共走10步，这是其中一种走法。我们可以简写成 1,1,1,1,1,1,1,1,1,1。

再比如，每次走2级台阶，一共走5步，这是另一种走法。我们可以简写成 2,2,2,2,2。

当然，除此之外，还有很多很多种走法。

我们用$f(n)$来表示到达第n层的走法数量，可以很容易得到它的递归式和边界条件

$$f(n) = f(n-1) + f(n-2),$$

$$f(1) = 1, f(2) = 2$$

通过这些条件我们可以很轻松地写出它的递归版本的解决办法

```
//朴素递归版本
int f(int n){
    if(n == 1) return 1;
    if(n == 2) return 2;
    return f(n-1) + f(n-2);
}
```

朴素递归算法比较费时，$f(n)$会进行多次计算，我们可以采用**记忆化数组**的方式优化，可以减少重复运算。

```
//记忆化递归版本
int a[N]={0,1,2};
int f(int n){
    if(a[n]) return a[n];
    return a[n]=f(n-1) + f(n-2);
}
```

我们采用递推方法来做呢，也就是从下至上。

```
//递推版本
int f[N];
f[1] = 1;
f[2] = 2;
for(int i=3;i<=N;i++){
    f[i] = f[i-1] + f[i-2];
}
```

可以发现其实我们每次计算只需要用到这个数组的三个值$f[i], f[i-1], f[i-2]$，可以优化成**滚动数组**来节约空间，是动态规划中比较常用的一种技巧。

```
for(int i=3;i<=N;i++){
    f[i%3] = f[(i-1)%3] + f[(i-2)%3];
}
```

记忆化搜索 基于递归的思想，即自上而下的解决问题。假设子问题已经得到了解决，并设置终止条件，从而得到问题的解。（想用就用，如果发现没准备好就递归求解）

动态规划 的思想是自下而上的解决问题，由最初的子问题一步步得到最终的结果。即由前面的结果通过状态转移得到后面的结果。

例题

数塔问题

题目描述

有如下所示的数塔，要求从底层走到顶层，若每一步只能走到相邻的结点，则经过的结点的数字之和最大是多少?

输入格式

输入数据首先包括一个整数整数N(1 <= N <= 100)，表示数塔的高度，接下来用N行数字表示数塔，其中第i行有个i个整数，且所有的整数均在区间[0,99]内。

输出格式

从底层走到顶层经过的数字的最大和是多少?

输入样例

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

输出样例

```
30
```

我们用$f[i][j]$表示走到第i层第j个数时最大的数字和，$a[i][j]$表示第i层第j个数，转换一下我们从顶部走到底部

我们可以像前面斐波那契数列那样写出这道题状态转移方程

$$f[i][j]=max(f[i-1][j],f[i-1][j-1])+a[i][j]$$

```
int n;
cin>>n;
for(int i=1;i<=n;i++){
    for(int j=1;j<=i;j++){
        scanf("%d",&a[i][j]);
        f[i][j]=max(f[i-1][j],f[i-1][j-1])+a[i][j];
    }
}
int ans=0;
for(int i=1;i<=n;i++)
    ans=max(ans,f[n][i]);
cout<<ans;
```

最长不下降子序列 (LIS)

题目描述

给出一个数字序列，求其最长不下降子序列的长度。例如， [3,2,4,7,6,8] ，最长不下降子

序列为 [3,4,7,8] 或者 [2,4,7,8] 等长度为 4 的子序列

输入格式

第一行为n，表示n个数（10<=n<=10000）
第二行n个整数，数值之间用一个空格分隔（1<=a(i)<=n）

输出格式

最长不下降子序列的长度

输入样例

```
3
1 2 3
```

输出样例

```
3
```

先定义状态 对于任意的i,定义$f[i]$是以$a[i]$结束的最长不下降子序列的长度，那么显然，问题的解就是$max_{1\leq i\leq n}f[i]$。

不妨假设，已经求得了以$a[1],a[2]...a[i-1]$结尾的最长不下降子序列的长度分别为$f[1],f[2]...f[i-1]$，$f[1]=1$

我们可以得到状态转移方程为：

$f[i]=max_{1\le j\le i}(f[j])+1,1\le j\le i,a[j]\le a[i]$

上述公式，前面的赋值语句表示状态转移方程，后面是状态转移条件。当然上面这个公式并没有涵盖所有情况，当对于某个这样的不存在时， 应当初始化为 1,另外，容易观察得出上述算法复杂度为 $O(n^2)$

```
int n,ans=0;
cin>>n;
for(int i=1;i<=n;i++)
    scanf("%d",&a[i]),
    f[i]=1;
for(int i=1;i<=n;i++){
    for(int j=1;j<i;j++){
        if(a[j]<=a[i]){
            f[i]=max(f[i],f[j]+1);
        }
    }
    ans=max(ans,f[i]);
}
```

另有$O(n^2)$的优化算法读者可以自行网上了解

最长公共子序列 (LCS)

题目描述

给出1-n的两个排列P1和P2，求它们的最长公共子序列。

输入格式

第一行是一个数n；（n是5~1000之间的整数）
接下来两行，每行为n个数，为自然数1-n的一个排列（1-n的排列每行的数据都是1-n之间的数，但顺序可能不同，比如1-5的排列可以是：1 2 3 4 5，也可以是2 5 4 3 1）。

输出格式

一个整数，即最长公共子序列的长度。

输入样例

```
5
3 2 1 4 5
1 2 3 4 5
```

输出样例

```
3
```

先定义状态 对于任意的i,j,定义$f[i][j]$是取a数组前i个元素和b数组前j个元素的最长公共子序列的长度

可以类比之前LIS的解法，分为$a_i=b_j$和$a_i\neq b_j$两种情况

如果是第一种情况，由于我们选定已经匹配了a_i和b_j两个数，所以它的状态转移其实就是从a数组的前$i-1$个元素和b数组前$j-1$个元素匹配的最大值加上一。

如果是第二种情况，a_i和b_j无法匹配，所以只能从前面最优的状态直接转移过来。

我们可以得到状态转移方程为：

$$f[i][j]=\begin{cases}0 & 当i=0或j=0\\ f[i-1][j-1]+1 & 当a[i]=b[j]\\ max(f[i][j-1],f[i-1][j]) & 当a[i]\neq b[j]\end{cases}$$

时间复杂度为$O(n^2)$

```
int n,ans=0;
cin>>n;
for(int i=1;i<=n;i++)
    scanf("%d",&a[i]);
for(int i=1;i<=n;i++)
    scanf("%d",&b[i]);
for(int i=1;i<=n;i++){
    for(int j=1;j<=n;j++){
        if(b[j]==a[i])
            f[i][j]=f[i-1][j-1]+1;
        else
            f[i][j]=max(f[i-1][j],f[i][j-1]);
        ans=max(ans,f[i][j]);
    }
}
```

总结

动态规划的使用需要同时满足以下三点：

1.子问题重叠性：我们必须保证我们分割成的子问题也能按照相同的方法分割成更小的子问题，并且这些子问题的最终分割情况是可以解决的

2.最优子结构：很多情况下动态规划用来求解最优化问题。此时，下一阶段的最优解应该能够由前面各阶段子问题的最优解导出。

3.无后效性：为了保证这些计算能够按顺序，不重复的进行，动态规划要求已经求解的子问题不受后续阶段的影响。

动态规划常用解题步骤：

第一步：将原问题分解成子问题。在这一步重点是分析哪些变量是随着哪些问题规模变小而变小的，哪些变量与问题的规模无关。

第二步：确定状态，在用动态规划解题时，我们往往将和子问题相关的各个变量的一组取值，称之为一个“状态”。

第三步：确定初始状态（边界状态）的值。

第四步：推导出状态转移方程，如何从一个或多个“值”已知的“状态”，求出另一个状态的值。状态的迁移可以用递推公式表示，此递推公式也可被称作“状态转移方程”。

状态确定->入口->状态转移->出口

简单背包问题

根据维基百科，背包问题（Knapsack problem）是一种组合优化的NP完全（NP-Complete，NPC）问题。问题可以描述为：给定一组物品，每种物品都有自己的重量和价格，在限定的总重量内，我们如何选择，才能使得物品的总价格最高。NPC问题是没有多项式时间复杂度的解法的，但是利用动态规划，我们可以以伪多项式时间复杂度求解背包问题。一般来讲，背包问题有以下几种分类：

1. 01背包问题
2. 完全背包问题
3. 多重背包问题

此外，还存在一些其他考法，例如恰好装满、求方案总数、求所有的方案等。本文接下来就分别讨论一下这些问题。

01背包问题

题目描述

给定n个物品，其中第i个物品体积为w_i，价值为v_i。有一个容积为M的背包，要求选择一些物品放入背包，使得物品总体积不超过m的情况下，物品的价值总和最大

暴力

每个物品我们可以选，也可以不选，如何组合下来共有2^n中选择方法。

$dfs(dep,m)$表示在考虑第dep个物品的情况下背包体积还剩m的最大价值

```
int dfs(int dep, int m) {
    if(dep == n+1) return 0;//n个物品处理完了
    if(m < v[dep]) {//如果体积不够放就不放
        return dfs(dep+1, m);
    }
    else {//如果体积能放下就分装和不装两种情况，
        return max(dfs(dep+1, m), dfs(dep+1, j-v[i])+w[i]);
    }
}
```

因为对于每个$dfs(dep,m)$我们都进行了大量的重复计算，所以可以使用记忆化数组的方法来优化。记得先将数组初始化。

```
int dfs(int dep, int m) {
    if(dep == n+1) return 0;
    if(f[dep][m]!=-1) return f[dep][m];
    if(m < w[dep]) {
        return f[dep][m]=dfs(dep+1, m);
    }
    else {//如果体积能放下就分装和不装两种情况，
        return d[dep][m]=max(dfs(dep+1, m), dfs(dep+1, j-w[i])+v[i]);
    }
}
```

每个$dfs(dep,m)$我们都只会计算一次，时间复杂度可以优化到$O(n*m)$

递推方法

其实记忆化方法就已经用到了动态规划的思想，我们将其写成递推式子就能写成动态规划了。

我们定义$f[i][j]$为前i个物品共使用j体积的背包所得到的价值的最大值

每一种物品我们只有选和不选另种情况，

选：$f[i][j]=f[i-1][j-w[i]]+v[i]$

不选:$f[i][j]=f[i-1][j]$

```
for(int i=1;i<=n;i++){
    for(int j=1;j<=m;j++){
        if(j>=w[i])
            f[i][j]=max(f[i-1][j],f[i-1][j-w[i]]+v[i]);
        else
            f[i][j]=f[i-1][j];
    }
}
```

我们每次都只用到了两列数组$f[i],f[i-1]$，我们可以使用**滑动数组**的方法来优化空间。

```
for(int i=1;i<=n;i++){
    for(int j=w[i];j<=m;j++){
        if(j>=w[i])
            f[i&1][j]=max(f[(i-1)&1][j],f[(i-1)&1][j-w[i&1]]+v[i&1]);
        else
            f[i&1][j]=f[(i-1)&1][j];
    }
}
```

还能优化吗?

如果我们只用一列数组，将下次操作和使用上次的数据同时在一列中进行呢?

我们可以逆序操作，我们更新掉f[j]状态并不影响之前的数据，所以我们可以优化成一维的。

```
for(int i=1;i<=n;i++){
    for(int j=m;j>=w[i];j--){
        f[j]=max(f[j],f[j-w[i]]+v[i]);
    }
}
```

循环到j时，

f数组的后半部分$f[j\sim m]$处于第i个阶段，也就是我们正在更新的阶段

f数组的前半部分$f[1\sim j-1]$处于第i个阶段，也就是我们将要更新的阶段

可以发现我们更新取前半部分时并不会用到后半部分的数据，满足线性DP的原则，保证了第i物品只会放一次。

完全背包

题目描述

给定N种物品，其中第i种物品的体积为v_i，价值为w_i，并且有无数个。有一个容积为M的背包，要求选择若干个物品放入背包，使得物品总体积不超过M的前提下，物品的价值总和最大

分析

完全背包可以理解为无数多个的01背包问题，我们可以将一种物品一直放，直到更新为止，显然这样时间复杂度太高了。

我们可以想想为什么01背包问题最后我们需要逆序开始，如果从头开始呢?

正序考虑01背包的代码，每次更新了j状态后会保留继续更新下一个状态，实现了无限使用物品。

```
for(int i=1;i<=n;i++){
    for(int j=w[i];j<=m;j++){
        f[j]=max(f[j],f[j-w[i]]+v[i]);
    }
}
```

多重背包

题目描述

给定N种物品，其中第i种物品的体积为v_i，价值为w_i，并且有C_i个。有一个容积为M的背包，要求选择若干个物品放入背包，使得物品总体积不超过M的前提下，物品的价值总和最大。

分析

同样，我们有C个物品，我们直接当作C个独立的物品，每种物品就进行了C次01背包的运算。

```
for(int i=1;i<=n;i++){
    for(int k=1;k<=c[i];k++){
        for(int j=m;j>=w[i];j--){
            f[j]=max(f[j],f[j-w[i]]+v[i]);
        }
    }
}
```

针对这种问题我们可以使用**二进制拆分法**来优化

从$2^0,2^1,2^2,\ldots,2^{k-1}$这$k$个数中选取若干个相加，可以表示出$0\sim2^k-1$之间的任意的整数。我们就可以把$2^k-1$个物品减少为$k$个物品，推广到$2^k+x$的情况，我们再加上一个$x$的物品就可以了。

但是如果C_i*V_i已经超过了背包容积M，我们就可以认为有无数个，当作完全背包问题来解决，能大大减少程序运行次数。

混合背包

多种背包融合，如果熟练掌握上面三种背包问题，混合背包问题也就很简单了。

习题