算法专题：动态规划（Dynamic Programming）

一：前言

学习基础：数据结构与算法。文章中少部分涉及数据结构的知识不作讲解。

使用情况：

在动态规划中，一个问题必须拥有重叠子问题和最优子结构，才能用动态规划解决。

重叠子问题：一个问题可以被分解为若干个子问题，且这些子问题会重复出现。比如求斐波那契数列时，递归（自顶而下Top-down Approach）会产生大量重叠子问题，此时可以采用**记忆化搜索**记住那些重叠子问题，避免多次计算从而降低时间复杂度。

最优子结构：如果一个问题的最优解可以由子问题的最优解有效的构造出来，那么称这个问题有最优子结构。比如树塔问题自下而上（Bottom-up Approach）进行递推得到最终结果。

比如在常见的最短路问题中，可以采取从目标点向前推和起始点向后推两种策略。

递归和递推：

递归（自顶向下）：

递归是一种自然而然的思考方式，它将一个大问题分解成一个或多个子问题，并通过递归调用来解决子问题。

比如斐波那契数列、汉诺塔问题等。此外，递归可以使代码更加清晰，更接近问题的数学描述。

递推（自底向上）：

递推是从最小的子问题开始，逐步构建问题的解决方案，直到解决整个问题，通常使用迭代或循环来实现。

递推通常更有效率，因为它避免了递归调用的开销，并且可以避免潜在的堆栈溢出问题。递推**适合处理具有重叠子问题性质**的问题，其中子问题的解可以存储并复用。典型的例子包括计算斐波那契数列、求解动态规划问题、计算组合数等。

因此，可以首先尝试使用递推来解决动态规划问题，因为它更容易优化并且通常具有更好的性能。但是，有些问题可能更容易用递归来思考和表达，因此适用递归方法也是有意义的。有时候，甚至可以将两者结合使用，使用递归来理解问题和定义状态转移方程，然后使用递推来实际计算结果，这种方法称为"记忆化搜索"，具体情况一般具体考虑。

优缺点：

时间和空间：动态规划中，采用递归或者递推的算法会使时间复杂度递增，所以往往我们会采取迭代的方案以空间记录其中的数据换取时间复杂度的降低。问题规模的大小往往对应该事件的时间复杂度。

解题步骤：

1.审题观察具有**重叠子问题**和**最优子结构**，判断是否为动态规划问题。

2.找出题中的**状态转移方程**，即如何把大问题分解为小问题。

3.找到**边界**，编写代码具体实现，得出结果。（也可继续分析算法的时间和空间复杂度，对此加强。

4.**回溯**找出问题的答案。

解题思路：

以下介绍中，一般会先根　据人的本能去思考暴力做法，因为即使是动态规划也会与普通的暴力做法有共通的地方，再去观察这个题目的特点，如何联想到动态规划，再使用寻找状态转移方程写代码求解验证。

动态规划和静态规划：

动态规划和静态规划都是优化问题，且二者可以相互转换。

动态规划

• 优点：DP表使得求解过程清晰，更容易获得最优解。

• 缺点：设计DP表难度大；状态空间可能呈指数增长。

静态规划

静态规划也是一种优化问题，包括线性规划和非线性规划。静态规划是从一个或者多个可行解开始，通过不断迭代对现有的可行解进行优化，最后得到最优解。

优点：始终保持一个可行解或近优解，思路相对简单。

缺点：当问题规模较大时，可能面临庞大的计算量。

二：经典题型

1.线性DP

1.1最大连续子序列和

> 题目：给定一个数字序列A1，A2，A3...An，求i，j（1<=i<=j<=n），使Ai+...+Aj最大，输出这个最大和。

对于该题，首先我们想到的是暴力做法，可以采用前缀和。

从A1开始，把A1，A1+A2,A1+A2+A3...；然后再从A2开始，A2，A2+A3...;一直到从An开始，可以用双重for实现，时间复杂度O(n^2)，同时采用一个大小为n*（n+1）/2的数组保存其中的所有数据，再求值。但明显，这样的方法过于繁琐。

再重新观察该题，我们要如何降低O(n^2)的时间复杂度，我们将其拆分为子问题，前1个数的最大连续和，前2个数的最大连续和，前3个数的最大连续和.....可知，在求多一个数的连续子序列和时，结合此题，要么就是当前数最大，要么就是加上前几个数的序列，因此我们得到状态转移方程

$$dp[i] = max(A[i], dp[i-1] + A[i])$$

然后我们再比较dp[0],dp[1]...dp[n-1]的最大值即可。

```
#include<iostream>
#include<algorithm>
using namespace std;
const int maxn = 10010;

int A[maxn];
int dp[maxn];

int main() {
    int n;//输入数组长度和数组中的数
    cin >> n;
    for (int i = 0; i < n; i++)cin >> A[i];

    dp[0] = A[0];//状态转移方程
    for (int i = 1; i < n; i++) {
        dp[i] = max(dp[i - 1] + A[i], A[i]);
    }
    //标准库函数max_element，它能够返回数组中的最大元素的迭代器，也就是地址，需要*解引用得到该地址的值。
    cout << *max_element(dp, dp + n) << endl;

    return 0;
}
```

1.2最长上升子序列

> 给定一个长度为n的数列，求数值严格单调递增的子序列的长度最长是多少。

该题和上一题虽然都是线性序列，但是上一题要求的是连续，因此可以从头到尾一个一个来，但是该题的子序列不连续，可以是跳跃状态，因此方法不同。

先思考暴力做法，序列中的每个元素有取和不取两种状态，如果全部考虑，对应2^n种，明显时间复杂度过高。所以考虑其它思路。

其实当前问题在于如何找到该问题的子问题，当然还有，每个上升子序列之间如何比较。1.1中是每个子问题中只有**一个**连续的解答，但是1.2中每个子问题会有**多个上升子序列**，在不到最后时无法确认最长序列。

重点来了，解决方案：这个多个上升子序列的解决方案就是开启第二重循环，对前面的每个dp[i]（dp[i]指的是0-i个种的最长子序列）都进行一次比较，然后取得最大上升子序列的个数。状态转移方程为

$$dp[i] = max(1, dp[j] + 1); (j = 1, 2, 3...i-1, A[j] < A[i])$$

```
#include<iostream>
#include<algorithm>
#include <climits>
using namespace std;
const int maxn = 10010;

int A[maxn];
```

```
int dp[maxn];

int main() {
    int n;//输入数组长度和数组中的数
    cin >> n;
    for (int i = 0; i < n; i++)cin >> A[i];

    dp[0] = 1;//状态转移方程
    int ans = INT_MIN;
    for (int i = 0; i < n; i++) {
        dp[i] = 1;
        for (int j = 0; j <= i; j++) {
            if (A[j] < A[i])dp[i] = max(dp[i], dp[j] + 1);
        }
        ans = max(ans, dp[i]);
    }
    cout << ans << endl;

    return 0;
}
```

1.3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。

有一说一，第一反应，双指针算法，哦不对，没有连续啊，那就不太行了。。那还是回归这个题，我们先看暴力做法，把所有A和B的子序列求出来然后比较，大概（N+M）*2^max（N，M），这样子，时间上肯定太过长了的。

1.3和前两个的最大不同就是现在是两个子序列和子序列多个如何找最长。

针对两个子序列我们可以采取二维数组的形式，dp(i)(j)指的是A的前i个和B的前j个的最长公共子序列；多个子序列可以直接采用在循环时比较出最大的方式。可以得到状态转移方程：

$$dp[i][j] = \begin{cases} dp[i-1][j-1]+1, & A[i] == A[j] \\ max(dp[i-1][j], dp[i][j-1]), & A[i]! = A[j] \end{cases}$$

两个数组同时进行比较，如果相等说明子序列+1，如果不相等取上下两前一个中的最大值即可。

注意：

1）对于你要循环对两个字符串进行处理时，一定不能把某个字符拿出来单独处理，这样就无法参与到循环的变化之中导致结果出错。

2）对于string，STL中的一些容器等，要注意在使用时为其分配一定空间的内存，若要加上其它空间需要重新说明，比如STL中的resize（）等。

```
#include<iostream>
#include<algorithm>
using namespace std;
const int MAX = 1010;

int n, m;
int dp[MAX][MAX];
```

```
8
9  int main() {
10     cin >> n >> m;
11     string A, B;
12     cin >> A >> B;
13 //  这里不能直接后移，因为string中的内存空间只分配了n个m个，多加一个是溢出
14 //  for(int i=n;i>=1;i--)A[n]=A[n-1];
15 //  for(int i=m;i>=1;i--)B[m]=B[m-1];
16
17     for (int i = 1; i <= n; i++) {
18         for (int j = 1; j <= m; j++) {
19             //状态转移方程
20             if (A[i-1] == B[j-1])dp[i][j] = dp[i - 1][j - 1] + 1;
21             if (A[i-1] != B[j-1])dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
22         }
23     }
24     cout << dp[n][m] << endl;
25     return 0;
26 }
```

2.背包问题

2.1 01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 vi，价值是 wi。
> 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。

首先思考暴力做法，由于对于每件物品都具有选择和不选择也就是0和1两个特性，因此时间复杂度在O(2^n)，因此考虑其它方法，这个问题其实很明显是可以分解为一个个子问题的，然后我们再寻找大问题和小问题的解决方案。若dp(i)(j)指的是在容量为j时有i件物品的最好价值，那可以得到以下状态转移方程：

$$dp[i][j] = max(dp[i-1][j], dp[i-1][j-v[i]] + w[i])$$

这个是根据第i个物品在0和1的状态中所得到的方程，如果第i个没有被选择，那么它与i-1时的结果相同，如果它被选择了，我们取一个刚好完全将背包放满的情况的i-1时的值加上当前值的结果，这是在求dp(i)(j)时的方法。所以在求最终结果时，需要求i=5,1<=j<=V时数组dp中的最大值。

不知道有没有人和我一样，在做题分析时陷入误区，将各种j的范围在对不同i时进行分段，然后代码出现很多个if，这其实是没有注意到各种不同段之间的联系，没有按照动态规划中大问题和子问题的思路思考。

```
1  #include<iostream>
2  using namespace std;
3  const int MAXN=1010;
4  #include<algorithm>
5
6  int dp[MAXN][MAXN];
7  int N,V;//物品数量，总容量
8  int v[MAXN],w[MAXN];//单件体积，单件价值
9
10 int main()
```

```
{
    cin>>N>>V;
    for(int i=1;i<=N;i++){
        cin>>v[i]>>w[i];
    }

    for(int i=1;i<=N;i++){
        for(int j=1;j<=V;j++)
            if(j>=v[i])dp[i][j]=max(dp[i-1][j],dp[i-1][j-v[i]]+w[i]);
            else dp[i][j]=dp[i-1][j];
    }
    cout << *max_element(dp[N]+1,dp[N]+V+1) << endl;

    return 0;
}
```

滚动数组为其优化

时间上已经达到O(n^2)，已经优化过，但是我们还能从空间上尝试进行优化。

大家观察上述方法中的状态转移方程，有没有发现第i行的dp数组都是由i-1行中的数据得到的。那么我们是不是可以不用第一维度也就是[i]的维度，在i-1中，已经得到了i-1行的数据，然后接着i行，i行刚好能用i-1行的数据，同时采用一维时也覆盖了上一行，这不就二维变成一维了。运行，啪，出错。

所以再观察j，j也是从小到大的，所以在求dp(j)的时候，已经把第i-1行的dp(j)更新了，求dp(j+w(i))的数据时必然出错了，它得到的是i行的dp(j)而不是i-1行，那怎么办呢，逆序即可，让j从大往小就没有这个后顾之忧了，j变小也不会有前面数据在用到后面的数据了（dp(i)(j)的求法是在上面和左上，不会用到右侧的数据）。

```
#include<iostream>
using namespace std;
const int MAXN = 1010;
#include<algorithm>

int dp[MAXN];
int N, V; //物品数量，总容量
int v[MAXN], w[MAXN]; //单件体积，单件价值

int main() {
    cin >> N >> V;
    for (int i = 1; i <= N; i++) {
        cin >> v[i] >> w[i];
    }

    for (int i = 1; i <= N; i++) {
        for (int j = V; j >= 1; j--)
            if (j >= v[i])dp[j] = max(dp[j], dp[j - v[i]] + w[i]);
    }
    cout << dp[V] << endl;
    return 0;
}
```

元组法

如该图所示，元组法指的是在每选择是否放入一个物品时，在1-V的容量的情况下，能放就更新相应对应的容量，不能放就不更新。如上图中每个物品的每个容量都会存储，但再次观察，在每行出现了连续的值，这些值的意思是在这个容量范围内从前到后刚好都能存放，所以实际上只需要存储连续的数的第一个也就是关键数据即可。最后得到的dp[N]也就是最好的结果。

这也相当于遍历了每一种可能性。

```
for(int i = 1; i <= N; i++)
    for(int j = V; j >= v[i]; j--)
        if(dp[j] < dp[j-v[i]] + w[i]) {
            dp[j] = dp[j-v[i]] + w[i];
            choice[i][j] = 1;
        }
```

时间复杂度为O(nV)。

2.2完全背包问题

> 有 N种物品和一个容量是 V的背包，每种物品都有无限件可用。第 i种物品的体积是 vi，价值是 wi。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。

和2.1不同的是，这个是无限件，看似无限，但也不能超过最大容量的，可以尝试把数组扩充，也就是对每件物品背包最多能出现的次数，把那些次数对应的加到数组v和w中，转化成01背包问题，不过在空间复杂度上看来就不是那么快乐了。但是，可以把数组扩充改成在循环里面处理。我们根据2.1中的二维数组的状态转移方程，对无限次进行处理。

```
for (int i = 1 ; i <= N ; i++)
        for (int j = 0 ; j <= V ; j++) {
            for (int k = 0 ; k * v[i] <= j ; k++)
                dp[i][j] = max(dp[i][j], dp[i - 1][j - k * v[i]] + k *
w[i]);//无限次的处理
        }
```

运行，啪，TLE（Time Limit Exceeded）这是我见过最常见的错误，毕竟时间复杂度是O(n^2)，所以接下来优化一下，先找到他的状态转移方程，再同样采用滚动数组的思想。由于是无限次，状态转移方程为：

$$dp[i][j] = max(dp[i-1][j], dp[i][j-v[i]] + w[i])$$

由于无限次，在放了第i件时，不需要在i-1次去找，而是仍然在i件处，j从w[i]开始变化。

同样，状态转移方程降维

$$dp[j] = max(dp[j], dp[j-v[i]] + w[i])$$

```
for (int i = 1 ; i <= N ; i++)
        for (int j = v[i] ; j <= V; j++) {
            dp[j] = max(dp[j], dp[j - v[i]] + w[i]);
        }
```

2.3多重背包问题

4. 多重背包问题 I - AcWing题库

```cpp
#include<iostream>
#include<algorithm>
using namespace std;
const int N = 1005;

int dp[N], n, V, v, w, s;

int main() {
    cin >> n >> V;

    while (n--) {
        cin >> v >> w >> s;
        while (s--) {
            for (int j = V; j >= v; j--)
                dp[j] = max(dp[j], dp[j - v] + w);
        }
    }
    cout << dp[V] << endl;
    return 0;
}
```

2.4分组背包问题

9. 分组背包问题 - AcWing题库

```cpp
#include<bits/stdc++.h>
using namespace std;

const int N = 110;

int dp[N][N];
int n, m;
int v[N][N], w[N][N], s[N];

int main() {
    cin >> n >> m;
    for (int i = 1; i <= n; i++) {
        cin >> s[i];
        for (int j = 0; j < s[i]; j++)
            cin >> v[i][j] >> w[i][j];
    }

    for (int i = 1; i <= n; i++) {
        for (int j = 0; j <= m; j++) {
            dp[i][j] = dp[i - 1][j];
            for (int k = 0; k < s[i]; k++) {
```

```
                if (j >= v[i][k])dp[i][j] = max(dp[i][j], dp[i - 1][j - v[i][k]] + w[i][k]);
            }
        }
    }

    cout << dp[n][m] << endl;
    return 0;
}
```

3.区间DP

> 设有N堆石子排成一排，其编号为 1,2,3,...,N。每堆石子有一定的质量，可以用一个整数来描述，现在将这 N 堆石子合并成为一堆。每次只能合并**相邻的两堆**，合并的代价为这两堆石子的质量之和，合并后与这两堆石子相邻的石子将和新堆相邻，合并时由于选择的顺序不同，合并的总代价也不相同。
>
> 例如有4堆石子分别为1 3 5 2，我们先合并1 2堆，代价为 4，得到 4 5 2， 又合并1 2 堆，代价为 9，得到9 2 ，再合并得到1堆，总代价为 4+9+11=24；如果第二步是先合并2 3堆，则代价为 7，得到 4 7，最后一次合并代价为11，总代价为 4+7+11=22。
>
> 问题是：找出一种合理的方法，使总的代价最小，输出最小代价。

先是以为是树，每次找到两个最小的相加成一个结点，得到一个树把结点和相加，不过题中说相邻两堆，这个思路不行。

每次找到相邻和的最小值（需要找和保存数据，需要第一次循环找最小，第二次保存数据），还有一个整体循环保证得到一堆，再将每次求的值相加，时间复杂度O(n^3)，但是寻找数据，保存数据的过程很繁琐，难以实现。可以开辟一个二维数组，第二行是第一行的相邻和，再加上其它数据，很复杂。

对于区间DP，它的子问题的错误思路是两堆石子，三堆，四堆…这样子思考。最终结果一定是左边一堆加上右边一堆，左边的那堆也一定由两堆相加，这才是真正的子问题。那子问题如何求解呢，一堆分为左堆右堆相加，左堆可以由两个数，也可以是三个，四个…，两个就是两个数相加的和，所以这样子可以得到动态转移方程：我们用dp(i)(j)表示i到j所需要的代价

$$dp[i][j] = min(dp[i][j], dp[i][k] + dp[k+1][j] + f[j] - f[i-1])$$

f数组是前缀和，得到两堆相加的和，也就是i到j所有数的和，因此得到代码：

```
#include<iostream>
using namespace std;
const int MAXN = 310;

int stone[MAXN], f[MAXN], dp[MAXN][MAXN];

int main() {
    int n;
    cin >> n;
    for (int i = 1; i <= n; i++)cin >> stone[i];

    for (int i = 1; i <= n; i++)f[i] = f[i - 1] + stone[i]; //前缀和

    for (int len = 2; len <= n; len++) { //先长度后端点
        for (int j = 1; j + len - 1 <= n; j++) { //前两个区间包含所有端点
```

```
            int l = j, r = j + len - 1;
            dp[l][r] = 1e8;
            for (int k = l; k < r; k++) {
                dp[l][r] = min(dp[l][r], dp[l][k] + dp[k + 1][r] + f[r] -
f[l - 1]);
            }
        }
    }

    cout << dp[1][n] << endl;
    return 0;
}
```

4.计数类DP

> 一个正整数n可以表示成若干个正整数之和，形如：n=n1+n2+...+nk，其中 n1≥n2≥...≥nk,k≥1。我们将这样的一种表示称为正整数 n的一种划分。现在给定一个正整数n，请你求出n共有多少种不同的划分方法。

该题有两个思路，一是当成完全背包问题，将体积为1-n的n个物品刚好放入背包中。第二个是数学中拆分法的数学思想。

完全背包：

dp(i)(j)指在前i个物品中选择i个物品体积为j。它的状态转移方程应为：

$$dp[i][j] = dp[i-1][j] + dp[i-1][j-i] + dp[i-1][j-2*i] \cdots dp[i-1][j-s*i];$$
$$dp[i][j-i] = dp[i-1][j-i] + dp[i-1][j-2*i] \cdots dp[i-1][j-s*i];$$
$$\text{因此：} dp[i][j] = dp[i-1][j] + dp[i][j-i];$$

优化空间可以得到：

$$dp[j] = dp[j] + dp[j-i];$$

问题一：完全背包中的体积是可以小于j的，但是这个题只能恰好为j，为什么还是完全背包？

答：这个题中的每个物品的体积是连续的数，所以得带的最优解一定刚好为j。假设最优解不为j，多了s个空间，那这个空间一定能再放物品，此时有多余空间的放法就不为最优解了。

问题二：优化空间的原理？

答：在你求第i行时，你所需要的第j列的数据第i行还没有进行更新，所以得到的仍然是第i-1行的数据；你所需要的j-i列的数据只要j是从小到大循环，就可以得到第i行j-i列的数据。

见代码：

```
#include <iostream>
using namespace std;
#include <algorithm>
const int MAXN = 1010, mod = 1e9 + 7;

int n;
int dp[MAXN];

int main() {
```

```
    cin >> n;

    dp[0] = 1;
    for (int i = 1; i <= n; i++)
        for (int j = i; j <= n; j++)
            dp[j] = (dp[j] + dp[j - i]) % mod;

    cout << dp[n] << endl;
    return 0;
}
```

拆分法：

充分展示了什么叫做数学思想。在拆分一个数时，将其分为一定有1和一定没有1的两种情况，

dp(i)(j)表示总个数为j，总和为i的个数

$$dp[i][j] = dp[i-1][j-1] + dp[i-j][j]$$

其中，dp(i-1)(j-1)指的是在（一定有1）总个数少1，总和少1的情况下的总个数；dp(i-j)(j)指的是（一定没有1）在总和少j也就是这个数组得到的划分结果中的每个数加1才是真正划分结果，所以一定没有1；

```
#include <iostream>
using namespace std;
#include <algorithm>
const int MAXN = 1010, mod = 1e9 + 7;

int n;
int dp[MAXN][MAXN];

int main() {
    cin >> n;
    dp[1][1] = 1;
    for (int i = 2; i <= n; i++) {
        for (int j = 1; j <= i; j++)
            dp[i][j] = (dp[i - 1][j - 1] + dp[i - j][j]) % mod;
    }

    int res = 0;
    for (int i = 1; i <= n; i++)res = (res+dp[n][i])%mod;
    cout << res << endl;
    return 0;
}
```

5.矩阵乘法链

> 对于M1×M2×…×Mq，Mi的为维数为ri×ri+1。对优化的乘法顺序，使得计算该乘法链所用的乘法数最少。

难度不大，就类似于一个上楼梯的问题，自底而上。

就举个例子来讲：

> (3×6) × (6×2) × (2×8) —— (M1×M2×M3)
>
> r= (3, 6, 2, 8)

状态转移方程为：

$$c(i,j) = min(c(i,k) + c(k+1,j) + ri \times r(k+1) \times r(j+1))（i <= k < j）;$$

这里的i，j，k指的是M1，M2…的下标。

这个同样是找出所有情况，但是数组的方式以至于在找到相同问题时不用重复计算。同样它的子问题的最优解和大问题的最优解一致，满足最优子结构和重叠子问题的特征。

6.数位统计DP

> 给定两个整数a和b，求a和b之间的所有数字中0~9的出现次数。例如，a=1024，b=1032，则a和b之间共有 9 个数如下：1024 1025 1026 1027 1028 1029 1030 1031 1032，其中0出现10次，1出现10次，2 出现7次，3 出现3次等等...

看分类讨论，情况有点多，也并不像标准的动态规划，可以跳过这个。

```cpp
#include <iostream>
using namespace std;
#include <algorithm>
#include <vector>
const int N = 10;
//对求某个位数上其中的x（0~9）出现的次数
/*
  abc[i]efg
  num[i] < x,0
  num[i] == x,0~efg(efg+1)
  num[i] > x,0~999(1e3)
 */
int get(vector<int> num, int l, int r) {
    int res = 0;
    //便于在求第l位到r位的数值
    for (int i = l; i >= r; i--)res = res * 10 + num[i];
    return res;
}

int power10(int x) {
    int res = 1;
    while (x--)res *= 10;
    return res;
}
//x在1~n上中出现的总次数
int count(int n, int x) {
    if (!n)return 0; //排除n为0的情况

    vector<int>num;//存储n中的每位数
    while (n) {
        num.push_back(n % 10);
        n /= 10;
    }
    n = num.size();
```

```
35
36      int res = 0;
37      for (int i = n - 1 - !x; i >= 0; i--) { //x==0时直接从最高位的下一位开始计数，
    最高位一定没有0
38          if (i < n - 1) { //不是最高位时的算法
39              //处理abc
40              res += get(num, n - 1, i + 1) * power10(i);
41              if (!x)res -= power10(i);
42          }
43
44          if (num[i] == x)res += get(num, i - 1, 0) + 1;
45          else if (num[i] > x)res += power10(i);
46      }
47      return res;
48  }
49
50  int main() {
51      int a, b;
52      while (cin >> a >> b, a) {
53          if (a > b)swap(a, b);
54
55          for (int i = 0; i <= 9; i++)
56              cout << count(b, i) - count(a - 1, i) << ' ';
57          cout << endl;
58      }
59
60      return 0;
61  }
```

7.all-pair最短路问题

> 现有一张有向带权图，请求一个初始节点到最终结点的最短路径。
>
> （每两个结点的路径就是两个结点之间边的权重）。权重可为正可为负。

注意和单源最短路径区分开来。

单源最短路可以用dijkstra算法求解，求出一个顶点到其它所有顶点的最短路径是O(n^2)。

但是allpair最短路问题加上了负的权重，dijkstra算法不再适用。

针对此题的解法：
Cij(k)表示节点i到j的中间节点编号不超过k的最短路长度。

按照最短路包不包括节点k得到状态转移方程：

$$Cij(k) = min(Cij(k-1), Cik(k-1) + Ckj(k-1));$$

边界为

Cii(k)=0 for all k；　　Cij(0)=Wij 或者∞；

方程其实好理解，有个关键点在于为什么all pair最短路问题能够想到以包不包括k为切入点思考到这个状态转移方程的?

先放一下伪代码：

```
//Floyd-Warshall算法
n=|V|
C(0)=W
for k=1 to n
    let C(k)=[Cij(k)] be a new martrix
    for i=1 to n
        for j=1 to n
            Cij(k)=min{ Cij(k-1) , Cik(k-1)+Ckj(k-1) };`
return C(n);
//C(0),C(1)···都是一系列矩阵，后面的矩阵由前面的矩阵得到
//最后一个矩阵就可以得到各店之间的最短路径
//C(0)指的基本上就是一个点到达另一个点，如果这两个点直接相连时的长度，不直接相连就是无穷
//C(1)指的在前面的基础上，可能经过1的路线
//C(2)...C(n),指在前面所有的基础上，可能经过n的路线
//到达最后时相当于把可能经过所有点的路线都走了一遍
//这就是这个方法可行的原因。
//巧妙地运用了每条路径之间的特点，从而不用像dijkstra限制在正权重之中。
```

当然，单单一个C矩阵是无法找到它的准确路径的，因此我们需要同时以一个矩阵记录每个过程中的前驱节点，利用回溯的思想。

这个和dijkstra算法中的矩阵记录路径的回溯是思想相似的。

这里的回溯利用前驱矩阵Pij(k)，记录每时每刻的前驱节点。

要是模拟一下floyd算法其实也不难，就是去年数据结构学的对最短路的两种求法，dijkstra和floyd算法。不过去年学的时候没有加上动态规划的状态转移方程。

8.旅行商问题

> 从起始城市出发，花费最小的总权重经过每个城市后回到初始城市的路径和最小花费。

前置:

欧拉路：给定孤立节点图G，若存在一条路，经过图中每边一次且仅一次，该条路称为欧拉路。

欧拉回路：若存在一条回路，经过图中每条边一次且仅一次，该回路称为欧拉回路。

欧拉定理：1.凡是由偶点组成的连通图一定可以一笔画成。2.凡是只有两个奇点的连通图，一定可以一笔画成。

状态转移方程：dp(i,S)指的是i经过S图然后回到初始节点。

$$dp(i,S)=min[w(i,j)+dp(j,S-(sj))](j属于S)$$

状态转移方程的含义是dp（i经过S图回到初始节点）等于i先到某个节点j的距离，加上j节点经过剩余节点回到初始节点，的距离之和。

其中图可以用二进制表示。或者用索引序号表示。 **（标记一下，这个二进制还真没怎么看懂）**

```
int TSP()
{
    for i<-1 to n
        d[i][0]=w[i][0];
    for j<-1 to 2^(n-1)-1
        for i<-1 to n
            if I不属于S
                for each k属于Si，d[i][Si]=min(c[i][k]+d[k](Si-k)|k属于Si);
    for each k属于V[2^(n-1)-1],d[0][2^(n-1)-1]=min(c[0][k]+d[k][2^(n-1)-2]);
    return d[0][2^(n-1)-1];
}
//时间复杂度为：T(n)=O(n*2^n);
// |s|=k时，需要比较k-1次，时间复杂度的和是k取从1到n-2的组合的和.
```

9.网络的无交叉子集

> 简单介绍一个例题，第一二层都是1-10，第一层会跟第二层有一个数对应并且连线记为（i，Ci），i的范围在1-10。
>
> 求一个最大的无交叉子集。

dp的思想。

关键点在于，状态转移方程和回溯找结果

状态转移方程：

$$dp[i][j] = max(dp[i-1][j], dp[i-1][ci-1]+1); (i>1, j>=ci);$$

新加上一个节点要么不放入跟上一层一样，要么加上限制条件再加上当前这条线。

边界：

$$dp[i][j] = 0; (1.j=0; 2.i=1, j<c1;)$$

回溯找具体的无交叉子集：

呼，果然不能乱看，还是要逆向推导，也配合一下回溯的思想。

```
首先看dp[10][10]，在第二层不动的情况下，第一层递减
if dp[i][10]!=dp[[i-1][10]! =dp[10][10] 就可以知道（i，Ci）必在结果中

找下一个点时,在第二层为ci-1的那一列寻找，自底而上直到发现dp[i][Ci-1]!=dp[10][10]-1，此时的i+1就一定在结果中。

一直这样子遍历，直到找到dp[10][10]个时停止。
```

10.子集和数问题

> 设S={s1,s2,...,sn}为n个正数的集合。试找出满足以下条件的和数最大的子集J：si的总和大于c，si的子集的和小于c，其中c是给定的任意常数。该题属于NP-hard问题。
>
> 如集合S={1，3，4，5}，c=7，如何找出子集J，使得J中的元素和最接近于c。

类似于0/1背包问题?

就是类似01背包问题，还有货箱装载问题都是属于0/1背包问题。

附:

1.内容参考

佟鑫宇老师 天津大学智能与计算学部 2023秋 算法设计与分析 ppt

《算法笔记》胡凡 曾磊主编 第十一章

Acwing算法基础课 第五讲活动 - AcWing

2.

考试考察重点：背包问题，矩阵乘法链，最短路问题，旅行商问题，多段图。

3.

全文同样包含个人的主观理解，如有错误，欢迎访问原文链接指正。