Part IX 动态规划

一、最大上升子序列和

一个数的序列b_i，当$b_1 < b_2 < \cdots < b_S$的时候，我们称这个序列是上升的。对于给定的一个序列$(a_1, a_2, \cdots, a_N)$，我们可以得到一些上升的子序列$(a_{i_1}, a_{i_2}, \ldots, a_{i_K})$，这里$1 \leq i_1 < i_2 < \cdots < i_K \leq N$。比如，对于序列(1, 7, 3, 5, 9, 4, 8)，有它的一些上升子序列，如(1, 7), (3, 4, 8)等等。这些子序列中序列和最大为18，为子序列(1, 3, 5, 9)的和。你的任务，就是对于给定的序列，求出最大上升子序列和。**注意，最长的上升子序列的和不一定是最大的，比如序列(100, 1, 2, 3)的最大上升子序列和为100，而最长上升子序列为(1, 2, 3)。 输入描述：**

> 输入包含多组测试数据。 每组测试数据由两行组成。第一行是序列的长度$N(1 \leq N \leq 1000)$。第二行给出序列中的N个整数，这些整数的取值范围都在0到10000（可能重复）。

输出描述：

> 对于每组测试数据，输出其最大上升子序列和。

示例1 输入

```
7 1 7 3 5 9 4 8
```

输出

```
18
```

利用动态规划求解，最长上升子序列(LIS)的变形

- 开始假设所有的数自成最大上升子序列，也就是sum[i]=num[i]；
- 之后再从前向后遍历，如果前面某个数小于当前的数，那么以那个数结尾的最大上升子序列的和加上当前的数会构成更大的上升子序列和，找出所有这样的和的最大值作为以当前数为尾的最大上升子序列的和。

数学语言总结如下：

- 状态设计：`sum[i]` 代表以 `num[i]` 结尾的最大上升子序列的和
- 状态转移：`sum[i] = max(sum[i], sum[j] + a[i]) (0 <= j < i, a[j] < a[i])`
- 边界处理：`sum[i]=num[i] (0 <= i < n)`

完整代码实现如下：

```
#include <stdio.h>
#define MAX(a, b) (a > b ? a : b)
#define MAX_SIZE 1005
```

```
int main() {
    int i, j, n;
    int maxsum;
    int num[MAX_SIZE], sum[MAX_SIZE];
    while(scanf("%d", &n) != EOF) {
        maxsum = 0;
        for(i = 0; i < n; i++) {
            scanf("%d", &num[i]);
            sum[i] = num[i];            // 初始化
        }
        for(i = 0; i < n; i++) {
            for(j = 0; j < i; j++) {
                if(num[i] > num[j]) {     //符合递增子序列
                    sum[i] = MAX(sum[j] + num[i], sum[i]);
                }
            }
        }
        maxsum = sum[0];
        for(i = 1; i < n; i++) {
            if(maxsum < sum[i]) {
                maxsum = sum[i];
            }
        }
        printf("%d\n", maxsum);
    }
    return 0;
}
```

时间复杂度为$O(n^2)$

二、最大连续子序列

给定K个整数的序列$\{N_1, N_2, \ldots, N_K\}$，其任意连续子序列可表示为$\{N_i, N_{i+1}, \ldots, N_j\}$，其中$1 \le i \le j \le K$。最大连续子序列是所有连续子序列中元素和最大的一个，例如给定序列{ -2, 11, -4, 13, -5, -2 }，其最大连续子序列为{ 11, -4, 13 }，最大和为20。现在增加一个要求，即还需要输出该子序列的第一个和最后一个元素。

输入描述：

> 测试输入包含若干测试用例，每个测试用例占2行，第1行给出正整数$K(K < 10000)$，第2行给出K个整数，中间用空格分隔。当K为0时，输入结束，该用例不被处理。

输出描述：

> 对每个测试用例，在1行里输出最大和、最大连续子序列的第一个和最后一个元素，中间用空格分隔。如果最大连续子序列不唯一，则输出序号i和j最小的那个（如输入样例的第2、3组）。若所有K个元素都是负数，则定义其最大和为0，输出整个序列的首尾元素。

示例1 输入

> 6 -2 11 -4 13 -5 -2 10 -10 1 2 3 4 -5 -23 3 7 -21 6 5 -8 3 2 5 0 1 10 3 -1 -5 -2 3 -1 0 -2 0

输出

> 20 11 13 10 1 4 10 3 5 10 10 10 0 -1 -2 0 0 0

题解： 这个问题如果暴力来做，枚举左端点和右端点(即枚举i, j)需要$0(k^2)$的复杂度，而计算$A[i]+\cdots+A[j]$需要$O(k)$的复杂度，因此总复杂度为$O(k^3)$。如果采用记录前缀和的方法(预处理$S[i]=A[0]+A[1]+\cdots+A[i]$，这样$A[i]+\cdots+A[j]=S[j]-S[i-1]$)使计算的时间变为$0(1)$，这样总的复杂度为$O(k^2)$，而这道题k的规模为$10^4$，因此这样的时间复杂度是可以接受的。故完整代码实现如下：

```
#include <stdio.h>
#define MAX_SIZE 10005
#define MAX_NUM 0x3f3f3f3f
#define MAX(a, b) (a > b ? a : b)
int a[MAX_SIZE], sum[MAX_SIZE];

void solve(int k) {
    int i, j;
    int flag = 1, maxSum = -MAX_NUM;
    int start, end;
    for (i = 0; i < k; i++) {
        scanf("%d", &a[i]);
        if (a[i] >= 0) {
            flag = 0;
        }
        sum[i + 1] = sum[i] + a[i];
    }
    if (flag) {
        printf("0 %d %d\n", a[0], a[k - 1]);
    } else {
        for (i = 0; i <= k; i++) {
            for (j = i + 1; j <= k; j++) {
                if (maxSum < sum[j] - sum[i]) {
                    maxSum = sum[j] - sum[i];
                    start = i;
                    end = j - 1;
                }
            }
        }
        printf("%d %d %d\n", maxSum, a[start], a[end]);
    }
}

int main() {
    int k;
    while (scanf("%d", &k) != EOF) {
        solve(k);
    }
```

```
    return 0;
}
```

但是上述代码并不是最优的，而且当k的规模为10^5甚至更大时，那么上述时间复杂度的算法就不适用了，故下面介绍利用动态规划求解的算法。具体如下：

记dp[i]为以元素$a_i(0 \leq i < k)$结尾的最大连续子序列的和，则有两种情况：

- 这个最大和的连续子序列只有一个元素，即以a[i]开始，以a[i]结尾；
- 这个最大和的连续子序列有多个元素，即从前面某处a[j]开始($j < i$)，一直到a[i]结尾。

对于第一种情况，最大和即a[i]本身； 对于第二种情况，最大和即为dp[i-1] + a[i]。 故状态转移方程为：

$$dp[i] = max\{a[i], dp[i-1] + a[i]\}(1 \leq i < k) \quad (\text{边界条件}：dp[0] = a[0])$$

但是题目中要求判断输入全为负数的情况，并要求输出所求最大连续子序列的起点和终点元素，因此再对代码稍加处理即可。

```
#include <stdio.h>
#define MAX_SIZE 10005
int a[MAX_SIZE];
struct DP {
    int left;   //子序列起点
    int right;  //子序列终点
    int val;    //子序列和
} dp[MAX_SIZE];

void solve(int k) {
    int i;
    int maxLeft, maxRight, maxVal;  //记录最大连续子序列起点，终点，和
    dp[0].left = dp[0].right = 0;          // 初始化
    dp[0].val = a[0];
    for(i = 1;i < k;i++) {
        if(a[i] < dp[i - 1].val + a[i]) {
            dp[i].left = dp[i - 1].left;
            dp[i].right = i;
            dp[i].val = dp[i - 1].val + a[i];
        } else{
            dp[i].left = i;
            dp[i].right = i;
            dp[i].val = a[i];
        }
    }
    maxVal = dp[0].val;
    for(i = 1;i < k;i++) {
        if(dp[i].val > maxVal) {
            maxVal = dp[i].val;
            maxLeft = dp[i].left;
            maxRight = dp[i].right;
        }
    }
    printf("%d %d %d\n",maxVal, a[maxLeft], a[maxRight]);
}
```

```
int main(){
    int i, k, flag;
    while(scanf("%d",&k) != EOF && k){
        flag = 1;
        for(i = 0;i < k;i++) {
            scanf("%d",&a[i]);
            if (a[i] >= 0) {
                flag = 0;
            }
        }
        if(flag) {
            printf("0 %d %d\n", a[0], a[k - 1]);
        } else {
            solve(k);
        }
    }
    return 0;
}
```

三、最大子矩阵

已知矩阵的大小定义为矩阵中所有元素的和。给定一个矩阵，你的任务是找到最大的非空(大小至少是1 * 1)子矩阵。 比如，如下4 * 4的矩阵 0 -2 -7 0 9 2 -6 2 -4 1 -4 1 -1 8 0 -2 的最大子矩阵是 9 2 -4 1 -1 8 这个子矩阵的大小是15。

输入描述:

> 输入是一个$N * N$的矩阵。输入的第一行给出$N(0 < N \leq 100)$。 在后面的若干行中，依次（首先从左到右给出第一行的N个整数，再从左到右给出第二行的N个整数……）给出矩阵中的N^2个整数，整数之间由空白字符分隔（空格或者空行）。 已知矩阵中整数的范围都在[-127, 127]。

输出描述:

> 测试数据可能有多组，对于每组测试数据，输出最大子矩阵的大小。

示例1 输入

> 4 0 -2 -7 0 9 2 -6 2 -4 1 -4 1 -1 8 0 -2

输出

> 15

题解： 要求一个二维矩阵的最大子矩阵，首先要会求一维矩阵的最大子矩阵（即一维数组连续最大和），假设原二维矩阵的最大子矩阵所在的行为i到j

- 当i = j时，则最大子矩阵为第i行的连续最大和
- 当i != j时，现在我们已经知道最大子矩阵的行，要求的是其所在的列，我们把从第i行到第j行的所有行相加，得到一个只有一行的一维数组，则该一维数组的连续最大和就是最大子矩阵。

而求一维数组的连续最大和，是一个动态规划问题，具体在第三题里面已经有解答。本题考点主要有两个，第一个是压缩矩阵，压缩矩阵可以减少时间复杂度；第二，最大连续子序列。

故完整代码实现如下：

```
#include <stdio.h>
#include <string.h>
#define MAX(a, b) (a > b ? a : b)
#define N 105
#define INF 0x3f3f3f3f
int martix[N][N];           //存储矩阵
int buf[N];                 //将相邻若干行合并成一行以后的结果
int n, maxSum;              //n是矩阵大小，maxSum是最大矩阵和

// 这里就是求解最大连续子段和
int findMax() {
   int i;
   int result = buf[0], sum = buf[0];
   for (i = 1; i < n; i++) {
       if (sum <= 0) {
           sum = buf[i];            //如果前面位置最大连续子序列和小于等于0，则以当前位置i结尾的最大连续子序列和为buf[i]
       } else {
           sum += buf[i];           //如果前面位置最大连续子序列和大于0，则以当前位置i结尾的最大连续子序列和为它们两者之和
       }

       result = MAX(result, sum);   //更新最大连续子序列和
   }
   return result;
}

int main() {
    int i, j, k;
    while(scanf("%d", &n) != EOF) {
        if(n <= 0) {
            break;
        }
        maxSum = -INF;
        for(i = 0; i < n; i++) {
            for(j = 0; j < n; j++) {
                scanf("%d", &martix[i][j]);
            }
        }
        for(i = 0; i < n; i++) {
            // 数组b表示j ~ n - 1行，对应列元素的和
            // 将二维动态规划问题转化为一维动态规划问题
            memset(buf, 0, sizeof(buf));
            for(j = i; j < n; j++) {
                for(k = 0; k < n; k++) {
                    buf[k] += martix[j][k];
                }
```

```
                maxSum = MAX(findMax(), maxSum);
            }
        }
        printf("%d\n", maxSum);
    }
}
```

四、最小邮票数

> 有若干张邮票，要求从中选取最少的邮票张数凑成一个给定的总值。如，有1分，3分，3分，3分，4分五张邮票，要求凑成10分，则使用3张邮票：3分、3分、4分即可。

输入描述：

> 有多组数据，对于每组数据，首先是要求凑成的邮票总值M<100。然后是一个数N < 20，表示有N张邮票。接下来是N个正整数，分别表示这N张邮票的面值，且以升序排列。

输出描述:

> 对于每组数据，能够凑成总值M的最少邮票张数。若无解，输出0。

示例1 输入

> 10 5 1 3 3 3 4

输出

> 3

题解： 这是一个动态规划问题，是属于背包问题的一种。在该题中采用动态规划，跟0-1背包问题一样，知道每个邮票的值，求恰好满足value的最小个数，**只是这里求的是个数不是最大value**，计算从1到m面值的最小邮票数。更新如下：

```
for(int j = m;j >= num[i];j--) { //must from m to num[i]
    dp[j] = std::min(dp[j], dp[j - num[i]] + 1);
}
```

dp[j]表示总值是j时，需要的邮票个数，邮票i分两种情况，分别是取和不取：

- 1.如果取邮票i，则邮票个数为：`dp[j - num[i]]+1`
- 2.如果不取邮票i，则邮票个数为：`dp[j]`

因此在dp[i][j]时的最优结构是

```
dp[i][j]=min(dp[i - 1][j],dp[i][j - num[i]] + 1)
```

转换为一维的形式就是

```
dp[j]=min(dp[j],dp[j - num[i]] + 1)
```

因为是恰好是某一个值，故初始状态为：

```
dp[0] = 0,dp[1...V] = INF;
```

完整代码实现：

```
#include <iostream>
#include <vector>
#include <algorithm>
#define INF 0x3f3f3f3f
using namespace std;

int main() {
    int m,n;
    while(cin >> m >> n) {
        vector<int> num(n);
        for(int i = 0;i < n;i++) {
            cin >> num[i];
        }
        vector<int> dp(m+1);
        dp[0] = 0;
        for(int i = 1;i <= m;i++) {
            dp[i] = INF;
        }

        for(int i = 0;i < n;i++) {
            for(int j = m;j >= num[i];j--) {
                if(dp[j - num[i]] != INF) {
                    dp[j] = min(dp[j], dp[j - num[i]] + 1);
                }
            }
        }

        if(dp[m] == INF) {
            cout << 0 << endl;
        } else {
            cout<< dp[m] <<endl;
        }
    }
}
```