

动态规划

- 动态规划
 - 问题索引
 - 线性 DP
 - 数字三角形
 - 基础
 - 变式
 - LIS (最长上升子序列)
 - 基础
 - 变式
 - LCS (最长公共子序列)
 - LCS 变式
 - LICS (最长上升公共子序列)
 - 用多少个最长上升子序列可以覆盖一个给定的序列
 - 求可变序列与给定数同余的方案数
 - 将给定序列分段, 保证每段和不超过给定数情况下, 求每段中「所有数的最大值」之和的最小值
 - 背包
 - 基础
 - 变式
 - 体积定义为「至少」
 - 特殊化的「有依赖」背包问题
 - 一般化的「有依赖」背包背包问题
 - 求方案数与具体方案
 - 「贪心」将无限集缩小为有限集
 - 单调队列优化多重背包
 - 二维费用背包问题
 - 状态机 DP
 - 基础
 - 变式
 - 状态压缩 DP
 - 基础
 - 变式
 - 区间 DP
 - 基础
 - 变式
 - 环形问题的一般化处理思路
 - 树形 DP
 - 基础
 - 变式
 - 数位 DP
 - 基础
 - 单调队列优化
 - 基础

■ 变式

问题索引

单调队列中，只需考虑队头元素：[单调队列优化多重背包](#)

单调队列中，需要考虑所有元素：[将给定序列分段，保证每段和不超过给定数情况下，求每段中「所有数的最大值」之和的最小值](#)

线性 DP

数字三角形

基础

[AcWing 1015. 摘花生](#)

变式

原题链接：[AcWing 1018. 最低通行费](#)

题解链接：[AcWing 1018. 最低通行费](#)

原题链接：[AcWing 1027. 方格取数](#)

这道题本质上是[AcWing 1015. 摘花生](#)的增强版，走的次数变为两次

考虑与上题同样的做法，设 $f[i_1, j_1, i_2, j_2]$ 所表示集合为：所有从 $(1, 1), (1, 1)$ 分别走到 $(i_1, j_1), (i_2, j_2)$ 的方案数，集合的属性为所有方案中的价值最大值

由于同一个数只能取一次，因此当两条路线相交时，对应的数只能被加一次，考虑两条路线相交时的条件：

当两条路线第一次相交时，有 $i_1 = i_2, j_1 = j_2$ ，并且两条路线的**长度**相同，即 $i_1 + j_1 = i_2 + j_2$

设 $i_1 + j_1 = i_2 + j_2 = k$ ，我们时刻用 k 来表示**单条路径的长度**

由于是同时走的，因此**两条路线的长度始终相等**，因此只要 $i_1 = i_2$ ，我们便可以判断二者是否重合

基于上述讨论，我们便可以省去一维，得以重新定义递推函数 $f[k][i_1][i_2]$ ，其中 k 表示**路径长度**

对于点 $(i_1, j_1), (i_2, j_2)$ ，两条路线一共有四种走法：

- 均向下，即 $(i_1, j_1 - 1), (i_2, j_2 - 1)$ 到 $(i_1, j_1), (i_2, j_2)$ ，对应状态为 $f[k - 1][i_1][i_2]$
- 均向右，即 $(i_1 - 1, j_1), (i_2 - 1, j_2)$ 到 $(i_1, j_1), (i_2, j_2)$ ，对应状态为 $f[k - 1][i_1 - 1][i_2 - 1]$
- 一个向下，另一个向右，即 $(i_1 - 1, j_1), (i_2, j_2 - 1)$ 或者 $(i_1, j_1 - 1), (i_2 - 1, j_2)$ 到 $(i_1, j_1), (i_2, j_2)$ ，对应状态为 $f[k - 1][i_1 - 1][i_2]$ 或者 $f[k - 1][i_1][i_2 - 1]$

完整代码：

```
#include <iostream>
using namespace std;
```

```

const int N = 15;
int g[N][N], f[2 * N][N][N];
int n;

int main()
{
    cin >> n;
    int x, y, w;
    while(cin >> x >> y >> w, x || y || w)
        g[x][y] = w;
    for(int k = 2; k <= 2 * n; k++)
    {
        for(int i1 = 1; i1 <= n; i1++)
        {
            for(int i2 = 1; i2 <= n; i2++)
            {
                int j1 = k - i1, j2 = k - i2;
                if(j1 >= 1 && j1 <= n && j2 >= 1 && j2 <= n)
                {
                    int& v = f[k][i1][i2];
                    int t = g[i1][j1];
                    if(i1 != i2) t += g[i2][j2];
                    v = max(v, f[k - 1][i1 - 1][i2 - 1] + t);
                    v = max(v, f[k - 1][i1 - 1][i2] + t);
                    v = max(v, f[k - 1][i1][i2 - 1] + t);
                    v = max(v, f[k - 1][i1][i2] + t);
                }
            }
        }
    }
    cout << f[2 * n][n][n] << endl;
    return 0;
}

```

AcWing 275. 传纸条

本题跟上一题一样，可以直接套上一题的代码

需要注意的是，当两条路线重合时，总可以通过对重合点进行微调，使得调整后的两条路径和的价值**大于等于**未调整前的，这其实是贪心的思路

需要注意的是，由于本题 $1 \leq i \leq m, 1 \leq j \leq n$ ，而 $j = k - i$ ，因此 $i \geq n - k, i \leq k - 1$

由于 k 会增大到 $m + n$ ，又可能导致越界，因此在赋初值时需要取 max ，终点值需要取 min

完整代码：

```

#include <iostream>

using namespace std;

```

```

const int N = 55;
int g[N][N], f[2 * N][N][N];
int n, m;

int main()
{
    cin >> m >> n;
    for(int i = 1; i <= m; i++)
        for(int j = 1; j <= n; j++) cin >> g[i][j];
    //j = k - i, 1 <= j <= n    ==>
    //1 <= k - i <= n          ==>
    //i >= n - k, i <= k - 1
    for(int k = 2; k <= n + m; k++)
    {
        for(int i1 = max(k - n, 1); i1 <= min(m, k - 1); i1++)
        {
            for(int i2 = max(k - n, 1); i2 <= min(m, k - 1); i2++)
            {
                int j1 = k - i1, j2 = k - i2;
                int t = g[i1][j1];
                if(i1 != i2) t += g[i2][j2];
                for(int a = 0; a <= 1; a++)
                    for(int b = 0; b <= 1; b++)
                        f[k][i1][i2] = max(f[k][i1][i2], f[k - 1][i1 - a][i2 - b]
+ t);
            }
        }
    }
    cout << f[m + n][m][m] << endl;
    return 0;
}

```

LIS (最长上升子序列)

基础

原题链接: [AcWing 895. 最长上升子序列](#)

题解链接: [AcWing 895. 最长上升子序列](#)

AcWing 896. 最长上升子序列 II

考虑贪心思路, 如果我期望一条上升子序列尽可能长, 那么我应当让结尾的数尽可能小, 这样才能够让更多的数接在当前这条子序列的后面

我们设 $q[i]$ 表示长度为 i 的上升子序列的结尾的数, 显然 $q[i]$ 单调上升

当前遍历到的数为 $w[i]$, 我们考虑将 $w[i]$ 加到最后一个「严格」比其小的元素的后面, 由于 $q[i]$ 具有二分性, 因此这一步可以使用二分

完整代码:

```
#include <iostream>

using namespace std;

const int N = 1e5 + 10;

int n;
int w[N], q[N];

int main()
{
    cin >> n;
    for(int i = 1; i <= n ; i++) cin >> w[i];
    int len = 0;
    q[0] = -1e9; //这里我们设定一个初值，不设也可以
    for(int i = 1; i <= n; i++)
    {
        int l = 0, r = len;
        while(l < r)
        {
            int mid = l + r + 1 >> 1;
            if(q[mid] < w[i]) l = mid;
            else r = mid - 1;
        }
        q[l + 1] = w[i];
        len = max(len, l + 1);
    }
    cout << len << endl;
    return 0;
}
```

变式

[AcWing 1017. 怪盗基德的滑翔翼](#)

[AcWing 1014. 登山](#)

[AcWing 482. 合唱队形](#)

[AcWing 1012. 友好城市](#)

[AcWing 1016. 最大上升子序列和](#)

LCS (最长公共子序列)

原题链接: [AcWing 897. 最长公共子序列](#)

题解链接: [AcWing 897. 最长公共子序列](#)

LCS 变式

[LeetCode 1092. 最短公共超序列](#)

LICS (最长上升公共子序列)

[AcWing 272. 最长公共上升子序列](#)

我们首先回顾一下 LIS 与 LCS 的状态定义：

LIS： $f[i]$ 表示集合为所有以 $a[i]$ 为结尾的最长上升子序列，属性为最长上升子序列的长度最大值（其中长度的最小值为 1）

LCS： $f[i][j]$ 表示所有由 $a[0 \cdots i]$ 和 $b[0 \cdots j]$ 组成的公共子序列，属性为公共子序列的长度最大值（由这个区间组成，并不是一定会出现这个区间内的所有数）

划分集合时，二者如下：

LIS：考虑第 i 个数的前一个数下标 j 的可能情况， j 的取值为 $1 \sim i-1$ ，只要满足 $a[j] < a[i]$ ，我们便可以从该状态转移过来

LCS：考虑 $a[i]$ 和 $b[j]$ 是否在公共子序列中，进而可以分出四种情况

本题考虑类似的思路：

$f[i][j]$ 表示集合为：所有由 $a[1 \sim i]$ 和 $b[1 \sim j]$ 构成，以 $b[j]$ 为结尾的最长公共上升子序列，属性为长度的最大值

（这里既可以以 $b[j]$ 结尾，也可以以 $a[i]$ 结尾，前者更为方便）

考虑与 LCS 相似的做法： $a[i]$ 是否在最长公共子序列中

- $a[i]$ 不在最长公共子序列中，有 $f[i][j] = f[i-1][j]$
- $a[i]$ 在最长公共子序列中（首先长度最小为 1），此时必然有： $a[i] = b[j]$ ，并且该条件为 $a[i]$ 在最长公共子序列中的充要条件

此时，我们考虑该公共子序列的前一个数的可能情况，设前一个数下标为 k ，可以进一步将集合划分成：

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

需要说明的是，后者必须为 $i-1$ ，因为子状态是不包含 $a[i]$ 与 $b[j]$

最后我们取 $f[n][i]$ 的最大值即可

完整代码：

```
#include <iostream>
#include <algorithm>

using namespace std;
```

```

const int N = 3010;
int a[N], b[N];
int f[N][N]; // 表示集合：由a[0~i], b[0~j]组成，以b[j]为结尾的公共上升子序列，属性为长度最大值
int n;

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

    for(int i = 1; i <= n; i++)
    {
        for(int j = 1; j <= n; j++)
        {
            f[i][j] = f[i - 1][j]; // a[i]不在公共上升子序列中
            if(a[i] == b[j]) // a[i]在子序列中
            {
                f[i][j] = max(f[i][j], 1); // 上升子序列的长度最小值为1
                for(int k = 1; k < j; k++)
                    if(b[k] < b[j])
                        f[i][j] = max(f[i][j], f[i - 1][k] + 1); // 我们从同时去掉
//不能写成f[i][j]
            }
            f[i][j] = max(f[i][j], f[i][k] + 1)
        }
    }

    int ans = 0;
    for(int i = 1; i <= n; i++) ans = max(ans, f[n][i]);
    cout << ans << endl;

    return 0;
}

```

需要说明的是，这个代码会超时，我们需要着手进行优化

我们注意到以下几点：

- $f[i][j]$ 的赋值只会发生在 $a[i] = b[j]$ 时
- 第二重循环会将 j 从小到大枚举一遍，第三重循环又会枚举一遍，属于重复枚举
- 第三重循环的主要目的是在所有满足 $b[k] < b[j]$, $k \leq j$ 的数中找一个最大值 ($f[i - 1][k] + 1$)，这相当于是求 j 的某个前缀 ($i - 1$ 是固定的) 的最大值

关于求某个数前缀的最大值，我们可以用一个变量 `maxv` 来记录当前的最大值

每次先比较当前遍历到的值与 `maxv` 来得到当前值的前缀最大值，之后我们在用当前值来更新 `maxv`，具体代码如下：

$w[i]$ 为具体序列, $q[i]$ 表示前缀 $1 \sim i$ 中的最大值

```
int maxv = 0;
for(int i = 1; i <= n; i++)
{
    q[i] = max(w[i], maxv);
    if(w[i] > maxv) maxv = w[i];
}
```

本题考虑一样的思路, 由于 $a[i]$ 与 $b[j]$ 是等价的, 因此在内层循环时, 只要 $b[j] < a[i]$ 满足, 我们便可以更新 $maxv$

$maxv$ 当中所记录的是前缀的最大值

完整代码如下:

```
#include <iostream>
#include <cstring>

using namespace std;

const int N = 3010;

int n;
int a[N], b[N], f[N][N];

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

    for(int i = 1; i <= n; i++)
    {
        int maxv = 1;
        for(int j = 1; j <= n; j++)
        {
            f[i][j] = f[i - 1][j];
            if(a[i] == b[j])
                f[i][j] = max(f[i][j], maxv);
            if(b[j] < a[i])
                maxv = max(maxv, f[i - 1][j] + 1);
        }
    }
    int ans = 0;
    for(int i = 1; i <= n; i++)
        ans = max(ans, f[n][i]);
    cout << ans << endl;
    return 0;
}
```


由于这里只用到了 i 与 $i - 1$ 的状态，我们考虑空间优化

直接将二维变为一维需要考虑两个问题：

- 当前值的更新是否依据来自上一层的状态（即对应值是否有被覆盖过）
- 当前值更新后，是否会对其他值造成影响

$f[j]$ 的更新来自于 $f[j]$ 本身与 $maxv$

不难发现， $f[j]$ 所需的更新值不会被任何数所覆盖，我们接着考虑 $f[j]$ 的更新对其他值的影响

$maxv$ 的更新来源于 $f[j]$ 那么当 $f[j]$ 更新时是否会影响到 $maxv$ 呢？

这里有一重保障是， $f[j]$ 的更新条件与 $maxv$ 的更新条件是**互斥**的，二者不会同时发生，自然也谈不上什么影响

完整代码如下：

```
#include <iostream>
#include <cstring>

using namespace std;

const int N = 3010;

int n;
int a[N], b[N], f[N];

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

    for(int i = 1; i <= n; i++)
    {
        int maxv = 1;
        for(int j = 1; j <= n; j++)
        {
            if(a[i] == b[j])
                f[j] = max(f[j], maxv);
            if(b[j] < a[i])
                maxv = max(maxv, f[j] + 1);
        }
    }
    int ans = 0;
    for(int i = 1; i <= n; i++)
        ans = max(ans, f[i]);
    cout << ans << endl;
    return 0;
}
```

最后说明一点：在 $f[i][j]$ 的定义中，我们可以以 $a[i]$ 作为结尾来定义，这样我们就需要将 i 写为内层循环， j 写为外层循环，同样是可以的，但看起来很难受，所以不推荐

用多少个最长上升子序列可以覆盖一个给定的序列

AcWing 1010. 拦截导弹

我们要求的是，对于给定的序列，**最少**需要用多少递减子序列才能将其覆盖

由于我们期望用**最少**的子序列个数来覆盖整个序列，也就是要求每个子序列都**尽可能长**

采取贪心的角度考虑，假设我们当前遍历到的数为 x ，对于前面的数我们已经构成了不同的递减子序列，我们当前考虑的是 x 应该如何处理

直观上来讲，子序列下降速度越慢，那么它就可能越长。也就是每次将 x 接到**所有比其大的数中最小的那个的后面**，这样可以保证子序列下降的速度最慢

如果实在找不到，那么只能令开一条以 x 为结尾的子序列

以上便是贪思路，下面我们给出证明：

假设贪心解对应子序列个数为 A ，最优解对应子序列个数为 B ，只需证明 $A = B$ 即可

由于 B 是最优解，因此 $B \leq A$

不失一般性地，设最优解与贪心解所构成的子序列集合中，子序列 S **第一个不同的数为 x** ， x 前面的数二者均相同

设最优解中 x 前面的数为 a_m ，贪心解中 x 前面的数为 a_t ，由于贪心做法总是将 x 接在所有比其大的数中最小的数的后面，因此有： $a_t \geq a_m$

因此，将贪心解中 x 及其之后的所有数均可以接在 a_m 的后面而总子序列个数不发生改变

所有对应任意一个 x ，贪心解总能够转化为最优解，即 $A \leq B$

所以有： $A = B$

在代码的实现上，我们用 $g[i]$ 表示编号为 i ($i \geq 1$) 的递减子序列的结尾的数

我们每次找到第一个比 $w[i]$ 大的数，并将其替换掉（相当于接在该序列的后面，并不会新增子序列个数），这一步可以直接枚举，也可以用二分

由于子序列单调不增，因此 $g[i]$ 数组一定单调上升（这一点很好证明， $g[i]$ 数组如果要新增元素一定是没有比该元素更大的数，因此 $g[i]$ 必然单调上升）

完整代码：

枚举做法：

```
#include <iostream>
```

```

using namespace std;

const int N = 1e3 + 10;

int w[N], f[N], g[N]; // g数组从左往右一定单调上升，表示编号为i的子序列结尾的数
int n;

int main()
{
    while(cin >> w[n]) n++;
    int ans = 0;
    for(int i = 0; i < n; i++)
    {
        f[i] = 1;
        for(int j = 0; j < i; j++)
            if(w[j] >= w[i]) f[i] = max(f[i], f[j] + 1);
        ans = max(ans, f[i]);
    }
    cout << ans << endl;

    int cnt = 0; // 子序列的个数

    for(int i = 0; i < n; i++)
    {
        int idx = 0;
        while(idx < cnt && g[idx] < w[i]) idx++; // 找到第一个比w[i]大的数，并将其替换
掉
        g[idx] = w[i];
        if(idx >= cnt) cnt++;
    }
    cout << cnt << endl;
    return 0;
}

```

二分做法:

```

#include <iostream>

using namespace std;

const int N = 1e3 + 10;

int w[N], f[N], g[N]; // g数组从左往右一定单调上升，表示编号为i的子序列结尾的数
int n;

int main()
{
    while(cin >> w[n]) n++;
    int ans = 0;
    for(int i = 0; i < n; i++)
    {
        f[i] = 1;

```

```

        for(int j = 0; j < i; j++)
            if(w[j] >= w[i]) f[i] = max(f[i], f[j] + 1);
        ans = max(ans, f[i]);
    }
    cout << ans << endl;

    int cnt = 0; //子序列的个数

    for(int i = 0; i < n; i++)
    {
        int l = 0, r = cnt;
        while(l < r)
        {
            int mid = l + r >> 1;
            if(g[mid] >= w[i]) r = mid;
            else l = mid + 1;
        }
        g[l] = w[i];
        if(l >= cnt) cnt++; //如果当前位置已经超过子序列个数，那么直接新开一个
    }
    cout << cnt << endl;
    return 0;
}

```

AcWing 187. 导弹防御系统

这道题是上面那道题的变式

对于每个元素，我们既可以将其加入到上升子序列 **up** 所对应的集合内，也可以将其加入到下降子序列 **down** 所对应的集合内

如果将当前状态看作一个点的话，那么我们每次可以从当前状态转移到另外的两个状态

我们需要的是，在所有的状态中，子序列长度的最小值

这里便可以考虑用 DFS 来做了，因为对于每个状态，我们都需要暴力枚举其对应的两个子状态

我们设计如下的函数接口：

```

void dfs(int u, int cu, int cd);
//u表示当前递归的位置，从1开始，到n结束，因此出口为n+1（我们需要将n递归完毕）
//cu表示上升子序列的个数
//cd表示下降子序列的个数

```

在每次递归遍历中，我们分别将当前递归得到的数 **w[u]** 加入 **up** 或 **down** 中，最后取 **cu + cd** 的最小值即可

完整代码：

```

#include <iostream>

```

```
using namespace std;

const int N = 55;

int n, cnt; // cnt表示最少需要的子序列个数, 初始最大值可以给n
int w[N], up[N], down[N]; // up表示第i个上升子序列结尾的数, down表示第i个下降子序列结尾的数
                                // up单调递减, down单调递增

void dfs(int u, int cu, int cd)
{
    if(cu + cd >= cnt) return; // 当cu+cd大于等于cnt的时候就需要返回了, 因为这种情况一定不合法
    if(u == n + 1) // 这里不能写n, 因为我们需要递归到n+1, 这才表示我们对第n个位置已经遍历完了
    {
        cnt = min(cnt, cu + cd);
        return;
    }

    // 放到递增子序列当中

    int idx = 0;
    while(idx < cu && up[idx] >= w[u]) idx++;
    int back = up[idx];
    up[idx] = w[u];

    if(idx >= cu) dfs(u + 1, cu + 1, cd);
    else dfs(u + 1, cu, cd);

    up[idx] = back;

    idx = 0;
    while(idx < cd && down[idx] <= w[u]) idx++;
    back = down[idx];
    down[idx] = w[u];

    if(idx >= cd) dfs(u + 1, cu, cd + 1);
    else dfs(u + 1, cu, cd);

    down[idx] = back;
}

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

        cnt = n;
        dfs(1, 0, 0);
        cout << cnt << endl;
    }
}
```

```
    return 0;
}
```

求可变序列与给定数同余的方案数

原题链接: [AcWing 1214. 波动数列](#)

我们设数列第一项为 x , 第二项为 $x + d_1$, 第 i 项为 $x + d_1 + d_2 + \cdots + d_{i-1}$, 那么对于长度为 n 的序列和为 $s = x + (x + d_1) + (x + d_1 + d_2) + \cdots + (x + d_1 + d_2 + \cdots + d_{i-1})$, 即:

$$s = nx + (n-1)d_1 + (n-2)d_2 + \cdots + (n-i)d_i + \cdots + d_{n-1}, d_i \in \{a, -b\}$$

此时问题转变成: 对于给定的每个 s 与 n , 在 d_i 任意取值的情况下, 等式成立的个数

由于 $x \in \mathbb{Z}$, 并且当 d_i 全部唯一确定时, x 也会唯一确定。此时我们需要确定的是, 当 d_i 取哪些值时 x 是合法的 (处于整数范围内) , 因此有如下等式:

$$x = \frac{s - [(n-1)d_1 + (n-2)d_2 + \cdots + (n-i)d_i + \cdots + d_{n-1}]}{n}$$

如果 x 要落在整数范围内, 那么 s 与 $(n-1)d_1 + (n-2)d_2 + \cdots + (n-i)d_i + \cdots + d_{n-1}$ 必须模 n 同余

此时问题转换成: 对于序列 $(n-1)d_1 + (n-2)d_2 + \cdots + (n-i)d_i + \cdots + d_{n-1}$ 与 s 模 n 同余的个数

考虑动态规划, $f[i][j]$ 表示对第 i 个数选择, 模 n 余 j 的方案数

第 i 个数对应 d_i , 系数为 $(n-i)$, 因此:

- 若第 i 个数为 a , 即 $d_i = a$, 有 $(n-1)d_1 + (n-2)d_2 + \cdots + (n-i)a$, 即 $f[i-1][\text{mod}((j - (n-i) * a), n)]$
- 同理, 若第 i 个数为 $-b$, 有 $f[i-1][\text{mod}((j + (n-i) * b), n)]$

最终结果为 $f[n-1][\text{mod}(s, n)]$

完整代码如下:

```
#include <iostream>
using namespace std;

const int N = 1e3 + 10, mod = 1e8 + 7;

int get_mod(int a, int n)
{
    return (a % n + n) % n;
}

int f[N][N];

int n, s, a, b;
```

```

int main()
{
    cin >> n >> s >> a >> b;
    f[0][0] = 1;
    for(int i = 1; i <= n - 1; i++)
    {
        for(int j = 0; j < n; j++)
        {
            f[i][j] = get_mod(f[i][j] + f[i - 1][get_mod(j - (n - i) * a, n)],
mod);
            f[i][j] = get_mod(f[i][j] + f[i - 1][get_mod(j + (n - i) * b, n)],
mod);
        }
    }
    cout << f[n - 1][get_mod(s, n)] << endl;
    return 0;
}

```

将给定序列分段，保证每段和不超过给定数情况下，求每段中「所有数的最大值」之和的最小值

原题链接: [AcWing 299. 裁剪序列](#)

直观考虑，一共 N 个数，之间的空位有 $N - 1$ 个，因此可以分段的选择一共有 2^{n-1} ，考虑动态规划进行优化

设 $f[i]$ 所表示的集合为：**所有将前 i 个数划分方案的集合**，集合的属性为价值的最小值，因此 $f[n]$ 为最终答案

以**最后一段的长度**对整个集合进行划分。对序列 $f[i]$ 而言，设最后一段的长度为 $k(0 \leq k \leq i)$ ，此时有：

$$f[i] = \min_{0 \leq k \leq i} f[i - k] + \max_{i-k+1 \leq j \leq i} A_j$$

设 $j = i - k$ ，上式转换为：

$$f[i] = \min_{0 \leq j \leq i} f[j] + \max_{j+1 \leq k \leq i} A_k$$

容易注意到以下性质：

- $f[i]$ 随着 i 的增大而单调不减

证明：

假定存在两个序列 k_1, k_2 ，长度分别为 L_1, L_2 ，有 $L_2 > L_1$

我们将 k_2 的划分方案平移到 k_1 上，设划分段数为 len ，有： $len_2 \geq len_1$ ，因此对于 k_2 而言，必然有 $f[k_2] \geq f[k_1]$ ，即 $f[i] \leq f[i + 1]$

其次，局部最优值 $f[j] + A_k$ 合法的充要条件为：

- $\sum_{k=j+1}^i A_k \leq m$
- $\sum_{k=j}^i A_k \geq m$
- $A_k = \max_{j+1 \leq l \leq i} A_l$

第一条保证从 $j+1$ 到 i 的和全部小于 m

第二条保证 j 总会取到总和小于 m 的边界

第三条保证 A_k 为 $j+1$ 到 i 中所有数的最大值

下面我们给出第二条的证明（第一和第三可以直接从题目推出来）：

考虑反证法，设存在 $j' < j$ 此时有： $f[j'] + \max_{j' < k \leq i} A_k \leq f[j] + \max_{j < k \leq i} A_k$

由于 $f[i]$ 随 i 增大而单调不减，即 $f[j'] \geq f[j]$

且 $\max_{j' < k \leq i} A_k \geq \max_{j < k \leq i} A_k$ ，因此上述假设不成立

因此若 $j' < j$ ，并且 $A_{j'} < A_j$ ，那么 j' 就是需要淘汰的策略

此时对于区间 $[j, i]$ 而言，内部元素**单调不减**且 i, j 均具有单调性 (i, j 会同步增大) 也就是「双指针」与「单调队列」

其次，由于单调队列中的**所有值**均是局部最优解，**需要全部考虑在内再取最小值**，因此我们需要额外维护一个**允许出现重复元素**的「平衡树」用于存储每个元素所对应的函数值，每次取出最小值即可

在这里我们需要注意一个边界问题，那就是只要当队列中至少存在一个元素时，才能够开始往 `multiset` 中插入元素

这是因为单调队列中单调不增的元素实际上表示的是**边界**，也就是 $f[i]$ 在此处的取值，而对与第二项的最大值而言，需要至少存在两个元素才合法，因此 `multiset` 中的元素个数总会比单调队列中少一个

完整代码如下：

```
#include <iostream>
#include <cstring>
#include <algorithm>
#include <set>

using namespace std;

typedef long long LL;

const int N = 1e5 + 10;

int w[N], q[N];
LL f[N];
multiset<LL>S;
LL n, m;

//multiset会直接将所有相同的元素全部删除，因此需要迭代器
```



```
void remove(int x)
{
    auto it = S.find(x);
    S.erase(it);
}

int main()
{
    cin >> n >> m;
    for(int i = 1; i <= n; i++)
    {
        cin >> w[i];
        if(w[i] > m)
        {
            cout << "-1" << endl;
            return 0;
        }
    }

    int hh = 0, tt = -1;
    LL sum = 0;
    for(int i = 1, j = 1; i <= n; i++)
    {
        sum += w[i];
        while(sum > m)
        {
            sum -= w[j++];
            if(hh <= tt && q[hh] < j)
            {
                if(hh < tt) //保证队列中至少一个元素之后再去删除set中的元素
                    remove(f[q[hh]] + w[q[hh + 1]]); //此时q[hh]为边界，区间最大值需
要取后一个元素
                hh++;
            }
        }

        while(hh <= tt && w[q[tt]] <= w[i])
        {
            if(hh < tt)
                remove(f[q[tt - 1]] + w[q[tt]]); //此时队尾元素表示局部最大值，边界需
要取前一个元素
            tt--;
        }

        q[++tt] = i; //先将元素插入到队列中，再输出队列中的元素
        if(hh < tt) //以当前队头前一个元素为边界，当前队头认为是整个区间的最大值
            S.insert(f[q[tt - 1]] + w[q[tt]]);

        f[i] = f[j - 1] + w[q[hh]];
        if(S.size()) //当平衡树中不空时，我们取整个的最小值
            f[i] = min(f[i], *S.begin());
    }
    cout << f[n] << endl;
```

```
    return 0;  
}
```

背包

基础

原题链接: [AcWing 2. 01背包问题](#) 题解链接: [AcWing 2. 01背包问题](#)

原题链接: [AcWing 3. 完全背包问题](#) 题解链接: [AcWing 3. 完全背包问题](#)

原题链接: [AcWing 4. 多重背包问题 I](#) 题解链接: [AcWing 4. 多重背包问题 I](#)

原题链接: [AcWing 5. 多重背包问题 II](#) 题解链接: [AcWing 5. 多重背包问题 II](#)

原题链接: [AcWing 9. 分组背包问题](#) 题解链接: [AcWing 9. 分组背包问题](#)

原题链接: [AcWing 7. 混合背包问题](#) 题解链接: [AcWing 7. 混合背包问题](#)

变式

原题链接: [AcWing 423. 采药](#) 题解链接: [AcWing 423. 采药](#)

原题链接: [AcWing 1024. 装箱问题](#) 题解链接: [AcWing 1024. 装箱问题](#)

原题链接: [AcWing 278. 数字组合](#) 题解链接: [AcWing 278. 数字组合](#)

原题链接: [AcWing 1019. 庆功会](#) 题解链接: [AcWing 1019. 庆功会](#)

原题链接: [AcWing 1023. 买书](#) 题解链接: [AcWing 1023. 买书](#)

原题链接: [AcWing 426. 开心的金明](#) 题解链接: [AcWing 426. 开心的金明](#)

原题链接: [AcWing 1013. 机器分配](#)

这道题其实是[\[AcWing 9. 分组背包问题\]](#)和[\[AcWing 12. 背包问题求具体方案\]](#)的结合体（这两题在本文均有题解）

对于分组背包，我们需要枚举**物品组内的选择**，也就是需要三重循环来进行解决

对于本题，可能会出现选 0 个的情况，因此在枚举选择组内物品编号的时候，需要将 0 这个特殊情况加以考虑

完整代码如下：

```
#include <iostream>  
  
using namespace std;  
  
const int N = 20;  
  
int f[N][N], w[N][N];
```

```

int n, m;

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

    for(int i = n; i >= 1; i--)
    {
        for(int j = 1; j <= m; j++)
        {
            for(int k = 0; k <= m; k++)//这里从0开始可以少写一行
                if(j >= k)
                    f[i][j] = max(f[i][j], f[i + 1][j - k] + w[i][k]);
        }
    }
    cout << f[1][m] << endl;
    int k = m;
    for(int i = 1; i <= n; i++)
    {
        for(int j = 0; j <= m; j++)//需要考虑不分配的情况
            if(k >= j && f[i][k] == f[i + 1][k - j] + w[i][j])
            {
                cout << i << " " << j << endl;
                k -= j;
                break;
            }
    }
    return 0;
}

```

以前的题解：[AcWing 1013. 机器分配](#)

[AcWing 1021. 货币系统](#)

[AcWing 532. 货币系统](#)

体积定义为「至少」

[AcWing 1020. 潜水员](#)

特殊化的「有依赖」背包问题

[AcWing 487. 金明的预算方案](#)

一般化的「有依赖」背包问题

[AcWing 10. 有依赖的背包问题](#)

求方案数与具体方案

AcWing 11. 背包问题求方案数

原题链接: [AcWing 12. 背包问题求具体方案](#)

显然这是 01 背包, 不同的是我们需要输出**字典序最小**的方案

考虑状态转移方程:

$$f[i][j] = \max(f[i][j], f[i-1][j-kv] + kw), \quad k = 0, 1$$

这里会出现三种情况:

- $f[i][j] = f[i-1][j]$ 且 $f[i][j] \neq f[i-1][j-v] + w$, 此时说明 $f[i][j]$ 这个状态是通过**不选第 i 个物品这个动作转移过来的**
- $f[i][j] = f[i-1][j-v] + w$ 且 $f[i][j] \neq f[i-1][j]$, 此时说明 $f[i][j]$ 这个状态是通过**选择第 i 个物品这个动作转移过来的**
- $f[i][j] = f[i-1][j] = f[i-1][j-v] + w$, 此时说明 $f[i][j]$ 可以从任意两个状态转移过来

由于我们需要输出具体方案, 因此对于每个状态而言, 我们需要知道它是从哪个状态转移过来的

因此对于第一种情况而言, 由于不选物品 i , 因此我们**不能输出该物品的编号**; 对于第二种情况而言, 由于选择了物品 i , 因此我们**需要输出该物品编号**, 重点在于第三种

由于此时我们的讨论是从后往前输出, 也就是**倒序**, 由于我们需要保证最终的答案**字典序最小**, 因此我们**需要将该编号输出** (如果没有字典序最小的限制, 那么是否输出都不影响)

这其实很容易理解, 我们当前是倒序输出, 因此**先输出的编号一定大于后输出的编号**, 如果从后往前读 (也就是正序方向), 输出该编号后的字典序一定小于不输出该编号的字典序

当然在代码实现上, 我们在 01 背包的处理可以从后往前处理, 这样 $f[1][m]$ 将变为最终答案, 然后在具体方案的输出上, 我们从前往后遍历

具体地, 我们遍历每个节点 i , 考虑节点 i 由 $i+1$ 的哪个状态转移过来, 如果满足条件则输出, 否则跳过

完整代码如下:

```
#include <iostream>

using namespace std;

const int N = 1010;

int f[N][N], v[N], w[N];

int n, m;

int main()
{
    cin >> n >> m;
```

```

for(int i = 1; i <= n; i++) cin >> v[i] >> w[i];
for(int i = n; i >= 1; i--)
{
    for(int j = 1; j <= m; j++)
    {
        f[i][j] = f[i + 1][j];
        if(j >= v[i]) f[i][j] = max(f[i][j], f[i + 1][j - v[i]] + w[i]);
    }
}
int k = m;
for(int i = 1; i <= n; i++)
    if(k >= v[i] && f[i][k] == f[i + 1][k - v[i]] + w[i])
    {
        cout << i << " ";
        k -= v[i]; //需要将k进行移动
    }
return 0;
}

```

以前的题解: [AcWing 12. 背包问题求具体方案](#)

「贪心」将无限集缩小为有限集

[AcWing 734. 能量石](#)

单调队列优化多重背包

[AcWing 6. 多重背包问题 III](#)

二维费用背包问题

原题链接: [AcWing 1022. 宠物小精灵之收服](#)

这道题本质上是 01 背包问题，只不过费用是二维的

一般这种二维费用问题，为了避免字母的混乱，我们统一用 v_i 来表示各个维度的花费

这里我们先套用 01 背包的定义: $f(i, j, k)$ 表示集合为所有考虑前 i 个小精灵，消耗精灵球数量 $\leq j$ ，消耗体力 $\leq k$ 的方案数，属性为所有方案当中精灵的最大数量

我们不难写出如下方程（设总精灵球数量为 V_1 ，总体力为 V_2 ）：

$$f(i, j, k) = \max(f(i - 1, j, k), f(i - 1, j - v_1, k - v_2) + 1)$$

最终的结果需要我们输出最大的精灵数量，即 $f(n, V_1, V_2)$ ，除此以外我们还需要输出剩余体力的最大值

注意到，我们的状态定义是消耗体力的值，因此剩余体力需要用 V_2 减去消耗体力才行

对于这一部分的求法，我们从 $V_2 - 1$ （下面会解释）开始从大到小枚举，只要下一个的函数值是满足条件的，我们便往下走，最后便可以找到消耗体力的最小值

由于受到的伤害使得体力小于等于零时的小精灵是不能收服的，因此最大的体力需要从 $V_2 - 1$ 开始枚举

完整代码如下：

```
#include <iostream>

using namespace std;

const int N = 1010, M = 510;

int f[N][M];

int n, V1, V2;

int main()
{
    cin >> V1 >> V2 >> n;
    for(int i = 1; i <= n; i++)
    {
        int v1, v2;
        cin >> v1 >> v2;
        for(int j = V1; j >= v1; j--)
        {
            for(int k = V2 - 1; k >= v2; k--)
            {
                f[j][k] = max(f[j][k], f[j - v1][k - v2] + 1);
            }
        }
    }
    cout << f[V1][V2 - 1] << " ";
    int k = V2 - 1;
    while(k > 0 && f[V1][k - 1] == f[V1][V2 - 1]) k--; //下一个k满足条件我们才往下走
    cout << V2 - k << endl;
    return 0;
}
```

AcWing 8. 二维费用的背包问题

状态机 DP

基础

[AcWing 1049. 大盗阿福](#)

变式

[AcWing 1057. 股票买卖 IV](#)

[AcWing 1058. 股票买卖 V](#)

状态压缩 DP

基础

[AcWing 1064. 小国王](#)

变式

[AcWing 327. 玉米田](#)

[AcWing 292. 炮兵阵地](#)

[AcWing 524. 愤怒的小鸟](#)

原题链接: [LeetCode 1125. 最小的必要团队](#)

区间 DP

基础

[AcWing 282. 石子合并](#)

变式

环形问题的一般化处理思路

[AcWing 1068. 环形石子合并](#)

[AcWing 320. 能量项链](#)

[AcWing 1069. 凸多边形的划分](#)

[AcWing 479. 加分二叉树](#)

[AcWing 321. 棋盘分割](#)

树形 DP

基础

[AcWing 285. 没有上司的舞会](#)

[AcWing 1072. 树的最长路径](#)

[AcWing 1073. 树的中心](#)

变式

[AcWing 1075. 数字转换](#)

[AcWing 1074. 二叉苹果树](#)

[AcWing 323. 战略游戏](#)

AcWing 1077. 皇宫看守

原题链接: [LeetCode 1617. 统计子树中城市之间最大距离](#)

本题需要我们求的是, 子树内最大距离为 d 的所有不同子树数量

子树内的最大距离定义为: 子树内所有点的最大距离 (也就是这棵树的直径)

考虑一个问题: 假设我们当前有一棵子树, 我们应该如果求这棵子树的直径

由于这里边的权值均相同, 设最终的直径长度为 d , 在这里我们对每个节点进行如下操作

- 用 $maxlen$ 表示**当前**已记录的最大长度, 遍历该节点的所有子节点, 得到以子节点为根的最长路径 res
- 用 $maxlen + res$ 来更新 d
- 用 res 更新 $maxlen$
- 遍历完所有子节点后返回 $maxlen$

注意, 中间两步不能调换顺序。由于我们要求经过该点的最长路径, 因此局部最优解为**当前**经过该点的最长路径与**次长**路径之和

这是基于子树已经存在的讨论, 现在的问题是该如何枚举所有的子树

注意到, 一共只有 15 个节点, 因此直接用**排列的方式**递归枚举所有子集, 然后对集合内的所有节点求一次直径即可

完整代码如下:

```
class Solution {
public:
    vector<int> countSubgraphsForEachDiameter(int n, vector<vector<int>>& edges)
    {
        vector<vector<int>> h(n);
        for(auto it : edges)
        {
            int x = it[0] - 1, y = it[1] - 1;
            h[x].push_back(y); // 无向图
            h[y].push_back(x);
        }
        // tree_set 表示目前枚举的集合, st 表示在此次枚举中, 哪些城市是访问过的
        vector<int> ans(n - 1);
        vector<bool> tree_set(n), st(n);
        int d = 0; // 表示当前集合的最大距离
        function<int(int)> dfs = [&](int u) -> int
        {
            st[u] = true;
            int maxlen = 0; // 当前已记录的最大长度
            for(int x : h[u])
            {
                // 遍历u的临边, 如果在当前选定的集合并且没有遍历过的话, dfs一次
                if(tree_set[x] && !st[x])
                {
                    int res = dfs(x) + 1;
                    d = max(d, maxlen + res);
                }
            }
            return maxlen;
        };
        for(int i = 1; i < n; i++)
            ans[i] = d;
        return ans;
    }
};
```



```

        maxlen = max(maxlen, res); //这个一定要放在后面
    }
}
return maxlen;
};

function<void(int)> f = [&](int u) //u表示当前遍历到的位置
{
    if(u == n)
    {
        for(int i = 0; i < n; i++)
        {
            if(tree_set[i]) //遍历子集内每一个节点，求出该子集的最大直径
            {
                fill(st.begin(), st.end(), 0);
                d = 0;
                dfs(i);
                break;
            }
        }
        //如果当前集合的最大距离不为零并且把当前集合全部遍历完毕
        if(d && tree_set == st)
            ans[d - 1]++;
        return;
    }
    //不选当前城市
    f(u + 1);

    //选当前城市
    tree_set[u] = true;
    f(u + 1);
    tree_set[u] = false;
};

f(0); //从0开始遍历所有集合
return ans;
}
};

```

数位 DP

基础

[AcWing 1081. 度的数量](#)

单调队列优化

基础

原题链接: [AcWing 135. 最大子序和](#)

由于我们需要求一段**连续**的子序列的和，考虑用前缀和

定义 $s[i] = w[1] + w[2] + \dots + w[i]$ 。假设所求区间为 $[l, r]$ ，那么区间和为 $s[r] - s[l - 1]$

由于区间长度不超过 m ，因此实际区间为 $[i - m + 1, i]$ ，区间和为 $s[i] - s[i - m]$

当我们枚举 i 时，我们期望 $s[i] - s[i - m]$ 最大，由于 $s[i]$ 固定，因此我们需要让 $s[i - m]$ 最小，即所有局部最优解 ans 为：

$$tmp = s[i] - \min_{0 \leq k \leq m} s[i - k]$$

全局最优解为：

$$ans = \max_{n-m+1 \leq i \leq n} ans_i$$

由于我们需要求一段区间内的最小值，因此后者可以用单调队列优化，时间复杂度变为 $O(n)$

单调队列内部维护元素个数为 $m + 1$ ，所维护元素为前缀和 $s[i]$ ，因此需要预先将 $s[0]$ 插入进队列中

我们在对最终结果赋值时需要保证队列不空，由于我们预先给了队列一个初值，因此需要在调整队列前对 ans 赋值

完整代码：

```
#include <iostream>

using namespace std;
typedef long long LL;
const int N = 3e5 + 10;
int n, m;
LL s[N], q[N];

int main()
{
    cin >> n >> m;
    for(int i = 1; i <= n; i++) cin >> s[i], s[i] += s[i - 1];
    int hh = 0, tt = 0; // 预先将0插入进队列，因为前缀和
    LL ans = -1e18;
    for(int i = 1; i <= n; i++)
    {
        if(hh <= tt && q[hh] < i - m) hh++; // 最后一个元素为 i - m
        ans = max(ans, s[i] - s[q[hh]]);
        while(hh <= tt && s[q[tt]] >= s[i]) tt--;
        q[++tt] = i;
    }
    cout << ans << endl;
    return 0;
}
```

变式