动态规划

1、动态规划 DP 基础

```
#include<bits/stdc++.h>//1-1649-1      前缀最大值      w2016010182
using namespace std;
int n,x,sum;
int a[100005],dp[100005];
int main()
{
	cin>>n>>x;
	a[1]=x;
	for(int i=2;i<=n;i++)
	{
		a[i]=(379*a[i-1]+131)%997;
	}
	dp[1]=a[1];
	for(int i=2;i<=n;i++)
	{
		dp[i]=max(dp[i-1],a[i]);
	}
	for(int i=1;i<=n;i++)
	{
		sum=sum+dp[i];
	}
	cout<<sum;
	return 0;
}
```

```cpp
#include<iostream>//1-1649-2      前缀最大值      anselxu
#include<cstring>
#include<algorithm>
#include<cmath>
#include<cstdio>
using namespace std;
int n,x,maxi;
int a[100010];
long long sum;
int main()
{
	cin>>n>>x;
	sum=maxi=a[1]=x;
	for(int i=2;i<=n;i++)
	{
		a[i]=(379*a[i-1]+131)%997;
		if(maxi<a[i])
			maxi=a[i];
		sum+=maxi;
	}
	cout<<sum;
	return 0;
}
```

```cpp
#include<iostream>//2-1650 前缀最小值  leirui
#include<cstdio>
using namespace std;
int a[100005],dp[100005],n,x;
int main()
{
	cin>>n>>x;
	a[1] = x;
	for(int i=2;i<=n;i++)
		a[i] = (379*a[i-1] + 131)%997;

	int vmin = 0;
	dp[1] = a[1] ;
	vmin = min (a[1],a[2]);
	dp[2] = dp[1] + vmin;

	for(int i=3;i<=n;i++)
	{
		vmin = min(a[i],vmin);
		// 在前 i-1 数中最大值和当前数中选最大
		dp[i] = dp[i-1]+vmin;
	}
	cout<<dp[n];
	return 0;
}
```

```cpp
#include<bits/stdc++.h>//3-1651  跳格子  hasome
using namespace std;
const int N=100010;
int a[N],dp[N],n,i,j;
int main()
{
	int a1,b,c;
	cin>>n;
	cin>>a1>>b>>c;
	for(i=1;i<=n;i++)
	{
		int tmp=((long long)a1*i*i+b*i+c)%20000;
		a[i]=tmp-10000;
	}
	dp[1]=a[1];
	for(i=2;i<=n;i++)
	{
		dp[i]=max(dp[i-2]+a[i],dp[i-1]+a[i]);
	}
	cout<<max(dp[i-1],dp[i-2]);
	return 0;
}
```

```cpp
#include<iostream>//4-1652   跳格子 2    leirui
#include<cstdio>
using namespace std;
int dp[100005],x[100005];
int main()
{
	int n,a,b,c;
	cin>>n>>a>>b>>c;
	for (int i = 1; i <= n; i++)
	{
		int tmp = ((long long)a * i * i + b * i + c) % 20000;
		x[i] = tmp - 10000;
	}

	dp[0] = 0;
	dp[1] = x[1];

	for(int i=2;i<=n+1;i++)
	{
		dp[i] = max(dp[i-1],dp[i-2]) + x[i];
	}

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

最大部分和（连续部分和）

```
liuchunhui001            //5-1589
n = int(input())
a = list(map(int,input().split()))
dp = [0 for i in range(n)]
dp[0] = a[0]

for i in range(1,n):
    dp[i] =  max(dp[i - 1] + a[i],a[i])

print(max(dp))
```

解法一：dp 数组存储前 i 个数不选连续的数的最大和。

对于每个数而言，求前 i 个数不选连续的最大和，分两种情况讨论：

情况一：留下第 i-1 个数，第 i 个数就不能留下，问题就转换成，求前 i-1 个数不选连续的最大和。

情况二：不留第 i-1 个数，第 i 个数留下，问题转换成，求 a[i] + 前 i-2 个数不选连续的最大和。

推导出状态转移方程如下：

dp[i] = max(dp[i-1],a[i] + dp[i-2]) 边界：dp[1] = a[1] dp[2] = max(a[1],a[2])

```
#include <bits/stdc++.h>//6-1653-1 取数    javacn
using namespace std;
/*
  推导出动态转移方程如下：
  dp[i] = max(dp[i-1],a[i] + dp[i-2])
  边界：dp[1] = a[1]  dp[2] = max(a[1],a[2])
*/
int i,n,a[100],dp[100];
int main(){
	cin>>n;
	for(i = 1;i <= n;i++)
	{
		cin>>a[i];
	}

	// 交代边界的值
	dp[1] = a[1];
	dp[2] = max(a[1],a[2]);
	// 递推
	for(i = 3;i <= n;i++)
	{
		dp[i] = max(dp[i-1],a[i]+dp[i-2]);
	}

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

解法二：每个元素有 2 种状态，选或者不选，将 2 种状态的最优解都记录一下。

f[i][0]：代表第 i 个元素不选的最优解。

f[i][1]：代表选择第 i 个元素的最优解。

分析： f[i][0]：第 i 个元素不选的状态，可以来自第 i-1 个元素不选，或者选，取最大。

f[i][1]：第 i 个元素选，只能来自第 i-1 个元素不选。

因此得出状态转移方程：

f[i][0] = max(f[i-1][0],f[i-1][1]);

f[i][1] = f[i-1][0] + a[i];

边界：//0 个数选 0 个收益是 0，0 个数选 1 个数的收益是不合法的

f[0][0] = 0,f[0][1] = -INT_MAX;

最终答案：ans = max(f[n][1],f[n][0]);

```
#include<bits/stdc++.h>//6-1653-2    javacn
using namespace std;
const int N = 110;
int f[N][2],a[N];
int n;
int main()
{
      cin>>n;
      for(int i = 1;i <= n;i++)
      {
        cin>>a[i];
      }
    //边界
      f[0][0] = 0,f[0][1] = -INT_MAX;
    //递推
      for(int i = 1;i <= n;i++)
      {
            f[i][0] = max(f[i-1][0],f[i-1][1]);//第 i 个数不取
            f[i][1] = f[i-1][0] + a[i];
      }
      cout<<max(f[n][0],f[n][1]);
      return 0;
}
```

dp 数组：存储状态，以 a[i] 结尾的最长不下降子序列的长度

 归纳动态转移方程：

dp[i]=1

dp[i]=max(dp[j]+1,dp[i])

j 是 1..i-1 之间的数，a[j]<a[i]

```
#include <bits/stdc++.h>//7-1794  最长不下降子序列（LIS）     javacn
using namespace std;

/*
  dp[i]=1
dp[i]=max(dp[j]+1,dp[i])  j 是 1..i-1 之间的数，a[j]<a[i]
*/
//dp：存储以每个数结尾的最长不下降子序列的长度
int a[10100],dp[10100],n,ma;
```

```cpp
int main()
{
    int i,j;
      cin>>n;
      for(i = 1;i <= n;i++)
      {
            cin>>a[i];
      }

      // 求以每个数结尾的最长不下降子序列的长度
      for(i = 1;i <= n;i++)
      {
            dp[i] = 1;
            // 将 a[i] 尝试续到每个数后面，看 dp[i] 能否增加
            for(j = 1;j < i;j++)
            {
                  if(a[j] < a[i])
                  {
                        dp[i] = max(dp[j]+1,dp[i]);
                  }
            }

            ma = max(dp[i],ma);
      }

      cout<<ma;
      return 0;
}
```

a 数组：存放元素

dpa 数组：存储状态，存储以每个点结尾的最长不下降子序列的长度（对于第 i 个人，左边加他自己，可以构成的递增的序列最多有几个数，最多留下几个人）

dpb 数组：存储状态，存储以每个点结尾从右至左的最长不下降子序列的长度（对于第 i 个人，包括自己的情况下，右侧最多留下几个人）

对于每个人而言，以他为中心，最多留下的人数 dpa[i]+dpb[i]-1。

因此，最少出去的人数 = n - 最多留下的人数。

```
#include <bits/stdc++.h>//8-1277     合唱队形求解   javacn
using namespace std;

/*
以第 i 个数为中心点能留下几个人：
计算出从左向右到第 i 个数的最长自增子序列的长度：第 i 个数左侧能留下的人数（含 i）
计算出从右向左到第 i 个数的最长自增子序列的长度：第 i 个数右侧能留下的人数（含 i）
最终能留下的人数 = dpa[i]+dpb[i]-1
留下的人数求最大，n- 最大，得到最少出去的人数
*/
int n,a[110],dpa[110],dpb[110];
int i,j;

int main(){
	cin>>n;
	for(i = 1;i <= n;i++)
	{
		cin>>a[i];
		//边界
		dpa[i] = 1;
		dpb[i] = 1;
	}

	//从左向后计算出最长递增子序列的长度
```

```
    // 从左向后计算出最长递增子序列的长度
    for(i = 2;i <= n;i++)
    {
        // 循环每个数前面的数
        for(j = 1;j < i;j++)
        {
            //a[i]比它前面的数大，才能递增
            if(a[i] > a[j])
            {
                dpa[i] = max(dpa[j]+1,dpa[i]);
            }
        }
    }

    // 从右向左求出最长递增子序列的长度
    for(i = n - 1;i >= 1;i--)
    {
        // 反过来循环第 i 个数后面的数
        for(j = n;j > i;j--)
        {
            if(a[i] > a[j])
            {
                dpb[i] = max(dpb[j]+1,dpb[i]);
            }
        }
    }
    // 计算出最多能留下几个人
    int ma = 0;
    for(i = 1;i <= n;i++)
    {
        ma = max(dpa[i]+dpb[i]-1,ma);
    }
    // 最少要出去的人数
    cout<<n - ma;
}
```

求最长不递减子序列的长度。

```
#include<bits/stdc++.h>//9-1795  拦截导弹    javacn
using namespace std;

int n,a[1010],dp[1010],ma;
int main()
{
	cin>>n;
	for(int i = 1;i <= n;i++)
	{
		cin>>a[i];
		dp[i] = 1;
		for(int j = 1;j < i;j++)
		{
			if(a[j] > a[i])
			{
				dp[i] = max(dp[j]+1,dp[i]);
			}
		}
		ma = max(dp[i],ma);
	}
	cout<<ma;
}
```

```cpp
#include<bits/stdc++.h>//10-1216      数塔问题    javacn
using namespace std;
int a[110][110];
// 思路：将数塔存入二维数组，从倒数第 2 层开始，递推计算出走到每个点最多能够累计的最大数字和，直到第 1 层，就能求出所经过结点的数字和的最大值
int main()
{
	int n;
	cin>>n;
	for(int i=1;i<=n;i++)
	{
		for(int j=1;j<=i;j++)
		{
			cin>>a[i][j];
		}
	}
    // 从倒数第 2 层开始逆推，每个点累加下方的值和下方右边的值中的较大值
	for(int i=n-1;i>=1;i--)
	{
		for(int j=1;j<=i;j++)
		{
			if(a[i+1][j]>a[i+1][j+1])
			{
				a[i][j] = a[i][j] + a[i+1][j];
			}
			else
			{
				a[i][j] = a[i][j] + a[i+1][j+1];
			}
		}
	}
	cout<<a[1][1];
	return 0;
}
```

dp[i][j]：代表有 i 个物品，背包容量为 j 时，能够承载的最大价值。
w[i]：代表第 i 个物品的重量。
v[i]：代表第 i 个物品的价值。
二维求解状态转移方程：dp[i][j]=max(dp[i-1][j],v[i]+dp[i-1][j-w[i]])

```
#include <bits/stdc++.h>//11-1282-1  简单背包问题        javacn
using namespace std;

/*
dp[i][j]：代表有 i 个物品，背包容量为 j 时，能够存储的最大价值
对于第 i 个物品：
情况一：背包容量 j< 物品重量 w[j]，放不下，能够存放的价值 dp[i-1][j]
情况二：背包容量 j>= 物品重量 w[j]，放得下
能够得到的最大价值 = max(dp[i-1][j],v[i]+dp[i-1][j-w[i]])
*/
int dp[110][20010];
int n,w[110],v[110],maxw,i,j;
```

```
int main()
{
	cin>>maxw>>n;
	// 读入每个物品的重量和价值
	for(i = 1;i <= n;i++)
	{
		cin>>w[i]>>v[i];
	}

	// 推导
	// 循环 n 个物品
	for(i = 1;i <= n;i++)
	{
		// 循环背包容量从 1~maxw
		for(j = 1;j <= maxw;j++)
		{
			// 如果放不下：背包容量 < 物品重量
			if(j < w[i])
			{
				dp[i][j] = dp[i-1][j];
			}
			else
			{
				// 放得下，就看放进来价值高，还是不放价值高
				dp[i][j] = max(dp[i-1][j],v[i]+dp[i-1][j-w[i]]);
			}
		}
	}

	// 输出 n 个物品，背包容量为 maxw 的情况下最大价值
	cout<<dp[n][maxw];
}
```

一维求解状态转移方程：dp[j]=max(dp[j], dp[j-w[i]]+v[i])

```
#include <bits/stdc++.h>//11-1282-2  简单背包问题        javacn
using namespace std;

int maxw;// 背包承重
int w,v;// 物品的重量和价值
// 讨论：有 i 个物品背包容量为 j 时能够存放的最大价值
int dp[20010];
int n;

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

	// 读入 n 个物品的重量和价值，并计算
	// 循环 n 个物品
	for(int i = 1;i <= n;i++)
	{
		cin>>w>>v;// 读入重量和价值

		// 计算
		// 逆序从背包容量循环到当前物品的重量
		for(int j = maxw;j >= w;j--)
		{
			dp[j] = max(dp[j],v+dp[j-w]);
		}
	}

	cout<<dp[maxw];
	return 0;
}
```

2、DP 进阶

我们可以发现，任何一个位置都只能从左边和右边传过来，这样我们就可以列出我们的方程：dp[i][j]=dp[i-1][j-1]+dp[i-1][j+1]。

dp[i][j]：代表第 i 次传球，第 j 个人获得球的机会数

j!=1&&j!=n：dp[i][j] = dp[i-1][j-1] + dp[i-1][j+1]

j=1 dp[i][j] = dp[i-1][j+1] + dp[i-1][n]，这是计算第 i 次传球，第 1 个人获得球的机会数
j=n dp[i][j] = dp[i-1][j-1] + dp[i-1][1]，这是计算第 i 次传球，第 n 个人获得球的机会数

```
#include <bits/stdc++.h>//1-1801    传球游戏    javacn
using namespace std;

/*
dp[i][j]：代表第 i 次传球，第 j 个人获得球的机会数
j!=1&&j!=n：dp[i][j] = dp[i-1][j-1] + dp[i-1][j+1]
j=1 dp[i][j] = dp[i-1][j+1] + dp[i-1][n]
j=n dp[i][j] = dp[i-1][j-1] + dp[i-1][1]

边界：dp[0][1] = 1
*/

int n,m;
// 代表第 i 次传球，第 j 个人获得球的机会数
int dp[40][40];
```

```cpp
int main()
{
	cin>>n>>m;
	// 边界：第 0 次传球，第 1 个人获得球的机会数
	dp[0][1] = 1;

	// 传 m 次球
	for(int i = 1;i <= m;i++)
	{
		for(int j = 1;j <= n;j++)
		{
			if(j==1) dp[i][j] = dp[i-1][j+1] + dp[i-1][n];
			else if(j==n) dp[i][j] = dp[i-1][j-1] + dp[i-1][1];
				else dp[i][j] = dp[i-1][j-1] + dp[i-1][j+1];
		}
	}

	// 经过 m 次传球，球回到第 1 个人手中的机会数
	cout<<dp[m][1];
	return 0;
}
```

动态规划！

对于每个点，计算出其左侧的最高值和右侧的最高值，那么积水量 = min(最侧最高值，右侧最高值) - 该点瓦片高度，以第 2 列为例，左侧最高值是 4，右侧最高值是 5，瓦片高度为 2，那么积水量 = min(4,5)-2=2。

```
#include <bits/stdc++.h>//2-1550-1  房屋积水  javacn
using namespace std;

int n,t;
int a[110];
int l[110];// 求从左向右的前缀最大值
int r[110];// 求从右向左的前缀最大值
```

```
int main()
{
	cin>>n>>t;
	for(int i = 1;i <= n;i++)
	{
		a[i] = t % 10;
		t =  (t * 6807 + 2831) % 201701;
	}

	// 求从左向右的前缀最大值（前 i 个数的最大数）
	l[1] = a[1];
	for(int i = 2;i <= n;i++)
	{
		l[i] = max(l[i-1],a[i]);
	}

	r[n] = a[n];
	for(int i = n - 1;i >= 1;i--)
	{
		r[i] = max(r[i+1],a[i]);
	}

	// 求每个位置的积水量
	int s = 0;
	for(int i = 2;i <= n - 1;i++)
	{
		s = s + min(l[i],r[i]) - a[i];
	}

	cout<<s;
	return 0;
}
```

```cpp
#include <bits/stdc++.h>//2-1550-2    房屋积水 remedy1314
using namespace std;
int n, r;
int a[105], lmax[105], rmax[105],t;
int main()
{
	cin>>n>>r;
	a[1] = r % 10;
	for (int i = 2; i <= n; i++)
	{
		r = (r * 6807 % 201701 + 2831) % 201701;
		a[i] = r % 10;
	}
	for (int i = 1; i <= n; i++)
	{
		lmax[i] = max(lmax[i - 1], a[i]);
	}

	for (int i = n; i >= 1; i--)
	{
		rmax[i] = max(rmax[i + 1], a[i]);
	}

	int ans = 0;
	for (int i = 2; i < n; i++)
	{
		t = min(lmax[i - 1], rmax[i + 1]);

		if(a[i] >= t)    continue;

		ans += t - a[i];
	}
	cout<<ans<<endl;
	return 0;
}
```

```
#include <bits/stdc++.h>//3-1276   挖地雷的算法     javacn
using namespace std;
/*
 dp[i]=a[i]+max(dp[j])  j=i+1 ... n  ij 有通路
 dp[n]=a[n]  边界

 dp: 存储从每个地窖开始最多能挖到的地雷数量
 a: 存储每个地窖的地雷数量
 r: 存储第 i 号地窖去了哪一号地窖
 f: 存储地窖之间的通路关系
*/
int n,i,j,a[210],dp[210],r[210];
bool f[210][210];
```

```
int main()
{
	cin>>n;
	// 读入地雷数量
	for(i = 1;i <= n;i++)
	{
		cin>>a[i];
	}
	int x,y;	// 读入通路关系
	while(true)
	{
		cin>>x>>y;
		if(x == 0 && y == 0) break;
		f[x][y] = true;//x y 之间有通路
	}
	// 边界：从最后一个地窖开始最多能挖到的地雷数就是最后一个地窖的地雷数
	dp[n] = a[n];
	//index：存放 i 号地窖应该去哪个地窖
//count：编号 >i 的地窖中，有通路的情况下，哪个地窖开始挖到的地雷最多，存地雷数
	int index,count;
	// 逆推
	for(i = n - 1;i >= 1;i--)// 从第 i 个地窖开始应该去哪个地窖取决于：
	{	从 i+1~n 之间的地窖中有通路的地窖，从哪个地窖开始挖，得到的地雷最多
		count = 0;
		for(j = i + 1;j <= n;j++)
		{
			if(f[i][j] == true && dp[j] > count)
			{			// 如果 i j 有通路，且从 j 开始挖到的地雷更多
				index = j;
				count = dp[j];
			}
		}
		dp[i] = a[i] + count; // 记录从 i 开始最多能挖到的地雷数量
		r[i] = index;		// 存储第 i 号地窖，去了哪个编号的地窖
	}
```

```
	//求从哪个地窖出发：求 dp 数组最大值的下标
	index = 1;
	for(i = 2;i <= n;i++)
	{
		if(dp[i] > dp[index]) index = i;
	}
	int ans = dp[index];//最多地雷数

	//打印路径
	while(index != 0)
	{
		cout<<index;
		index = r[index];//求出 index 编号的地窖去了哪里
		if(index != 0) cout<<"-";
	}

	cout<<endl<<ans;
	return 0;
}
```

按公司顺序分配机器，第一个阶段把 M 台设备分配给第一个公司，记录下获得的各个盈利，然后把 M 台设备分给前两个公司，和第一个阶段比较记录下来更优各个盈利值，一直到第 N 个阶段把
M 台机器全部分给了 N 个公司。两个阶段之间关系如下讨论。

f[i-1][k]： 表示前 i-1 个公司分配 k 台机器的最大盈利，用 c[i][j] 表示第 i 个公司分配 j 台机器盈利；

f[i-1][k]+c[i][j-k] // 前 i-1 公司分配 k 台机器最大盈利 + 第 i 个公司分配 j-k 台机器的盈利。

状态转移方程：
f[i][j]=max(f[i-1][k]+c[i][j-k])，边界条件：f[0][0]=0。

通过递归输出每个公司分配机器的方案数的解法：

```
#include <bits/stdc++.h>//4-1378-1    机器分配    javacn
#define N 20
using namespace std;
int f[N][N];
int n, m, ans;
int c[N][N];

void print(int i, int j)//打印分配情况
{
	if(i==0) return;
	for(int k = 0; k <= j; k++)
	{
		if(ans == f[i-1][k] + c[i][j-k])
		{
			ans = f[i-1][k];
			print(i-1, k);
			printf( “%d %d\n” , i, j-k);
			break;
		}
	}
}
```

```
int main()
{
	scanf("%d%d", &n, &m);
	for(int i = 1; i <= n; i++)
		for(int j = 1; j <= m; j++)
			scanf("%d", &c[i][j]);

	for(int i = 1; i <= n; i++)
		for(int j = 1; j <= m; j++)
		{
			int maxx = 0;
			for(int k = 0; k <= j; k++)
			{// 递归求各公司分配机器数量
				maxx = max(f[i-1][k] + c[i][j-k], maxx);
			}
			f[i][j] = maxx;
		}
	printf("%d\n", f[n][m]);
	ans = f[n][m];
	print(n, m);
	return 0;
}
```

直接递推求出每个公司分配机器方案数的解法：

```
#include <bits/stdc++.h>//4-1378-2    机器分配    javacn
using namespace std;

/*
f[i][j]：i 个公司在共有 j 台设备的最大利润
f[i][j] = max(f[i-1][k]+c[i][j-k]);
*/
const int N = 20;
int f[N][N];
int n, m, ans;
int c[N][N],r[N];//r：每个公司分到的机器数量

int main()
{
      scanf("%d%d", &n, &m);
      for(int i = 1; i <= n; i++)
            for(int j = 1; j <= m; j++)
                  scanf("%d", &c[i][j]);
    // 循环每个公司
      for(int i = 1; i <= n; i++)
      {
        // 循环机器的额数量
            for(int j = 1; j <= m; j++)
            {
                  int maxx = 0;
                  for(int k = 0; k <= j; k++)  // 递归求各公司分配机器数量
                  {
                        maxx = max(f[i-1][k] + c[i][j-k], maxx);
                  }
                  f[i][j] = maxx;
            }
      }
```

```
    printf("%d\n", f[n][m]);

    ans = f[n][m];
  // 逆序推导出每个公司实际分配了几台机器
    for(int i = n; i >= 1; i--)
    {
      // 枚举第 i 个公司可能分到的机器总数
        for(int k = 0; k <= m; k++)
        {
        // 第 i 个公司分配 j-k 台机器可以获得最大总利润
            if(ans == f[i-1][k] + c[i][m-k])
            {
                r[i] = m - k;
                ans = f[i-1][k];// 接下来计算 i-1 个公司有 m-k 台机器的
                m = k;
                break;
            }
        }
    }

    for(int i = 1;i <= n;i++)
    {
        printf("%d %d\n",i,r[i]);
    }
  return 0;
}
```

dp[a][b][c][d]:表示你出了a张爬行牌1,b张爬行牌2,c张爬行牌3,d张爬行牌4时的得分。

起始状态 dp[0][0][0][0]=num[1]，即不出任何爬行卡；之后对于每一张卡片，我都可以选择放与不放， 设当前放的卡 1 数量为 a，卡 2 数量为 b，卡 3 数量为 c，卡 4 数量为 d（以下出现 a~d 均为这个意思），则对于卡一：

比较卡一的放与不放，只需决策卡一的放与不放，即取 dp[a][b][c][d],dp[a-1][b][c][d]+num[r] 的最大值。又由于 a 有一定数量，所以我们可以得出关于 a 的转移方程：

dp[a][b][c][d]=max(dp[a][b][c][d],F[a-1][b][c][d]+num[r])

对于 bcd 以此类推：

```cpp
#include<bits/stdc++.h> //5-1781    乌龟棋    javacn
using namespace std;
int dp[41][41][41][41],num[351],g[5],n,m,x;
int main()
{
    cin>>n>>m;
    for(int i=1;i<=n;i++) cin>>num[i];
    dp[0][0][0][0]=num[1];
    for(int i=1;i<=m;i++)
    {
        cin>>x;
        g[x]++;
    }
    for(int a=0;a<=g[1];a++)
        for(int b=0;b<=g[2];b++)
            for(int c=0;c<=g[3];c++)
                for(int d=0;d<=g[4];d++)
                {
                    int r=1+a+b*2+c*3+d*4;
         if(a!=0)    dp[a][b][c][d]=max(dp[a][b][c][d],dp[a-1][b][c][d]+num[r]);
         if(b!=0)    dp[a][b][c][d]=max(dp[a][b][c][d],dp[a][b-1][c][d]+num[r]);
         if(c!=0)    dp[a][b][c][d]=max(dp[a][b][c][d],dp[a][b][c-1][d]+num[r]);
         if(d!=0)    dp[a][b][c][d]=max(dp[a][b][c][d],dp[a][b][c][d-1]+num[r]);
                }
    cout<<dp[g[1]][g[2]][g[3]][g[4]];
    return 0;
}
```

```
#include <bits/stdc++.h>//6-1796    奶牛沙盘队    javacn
using namespace std;

const int MOD = 100000000; // 余数
const int N = 2010;
/*
1. 状态定义
dp[i][j]：前 i 个数，取若干求和后 % F 为 j 的方案数

2. 状态转移
第 i 个数有 2 种选择，要或者不要，方案数为两者叠加
dp[i][j] = ((dp[i][j] + dp[i - 1][j]) % MOD + dp[i - 1][(j - a[i] + f) % f]) %
MOD;

3. 最终答案
dp[n][0]
*/
long long a[N], dp[N][N]; //cow[i] 指第 i 头奶牛的能力，
int n, f;
```

```
int main()
{
	scanf("%d%d", &n, &f);

		//读入每个数
	for (int i = 1; i <= n; i++)
	{
		scanf("%d", &a[i]);
		a[i] %= f;//提前取余
	}

    //边界条件
	for (int i = 1; i <= n; i++)
	{
		dp[i][a[i]] = 1;
	}

    //01背包求方案数
    //枚举每个数
	for (int i = 1; i <= n; i++)
	{
	  //枚举余数
		for (int j = 0; j <= f - 1; j++)
		{
		dp[i][j]=((dp[i][j]+dp[i-1][j])%MOD + dp[i-1][(j-a[i]+f)%f])%MOD;
		}
	}

	printf("%d", dp[n][0]);

	return 0;
}
```

先求出以每个数结尾的最大和，再取和的前缀最大值作为每个人的特征值（因为不一定以第 i 个数结尾的最大和，是前 i 个数取连续数的最大和）。

接下来根据题意算分数，再计算分数的最大值。

```
#include <bits/stdc++.h>//7-1800    小朋友的数字    javacn
using namespace std;
typedef long long LL;
const int N = 1e6 + 10;
LL n,p,a[N];
//f：存放特征值
LL s[N],f[N],score[N],r,ma = LONG_LONG_MIN;//r 存放每个人的分数
```

```cpp
int main()
{
	cin>>n>>p;
	// 读入 n 个值
	for(int i = 1; i <= n; i++)
	{
		cin>>a[i];
		// 以每个点结束的最大和
		s[i] = max(s[i-1]+a[i],a[i]);
		ma = max(ma,s[i]);
		f[i] = ma % p;// 特征值
	}

	// 第一个人的分数是特征值
	score[1] = f[1];
	LL ans = score[1];
	ma = LONG_LONG_MIN;
	// 计算每个人的得分
	for(int i = 2; i <= n; i++)
	{
		ma = max(ma,f[i-1]+score[i-1]);
		score[i] = ma;
		ans = max(ans,ma) % p;
	}

	cout<<ans;

	return 0;
}
```

3、背包基础

dp[i][j]：代表有 i 个物品，背包容量为 j 时，能够承载的最大价值。
w[i]：代表第 i 个物品的重量。
v[i]：代表第 i 个物品的价值。
二维求解状态转移方程：dp[i][j]=max(dp[i-1][j],v[i]+dp[i-1][j-w[i]])

```
#include <bits/stdc++.h>//1-1282-1  简单背包问题    javacn
using namespace std;

/*
dp[i][j]：代表有 i 个物品，背包容量为 j 时，能够存储的最大价值
对于第 i 个物品：
情况一：背包容量 j< 物品重量 w[j]，放不下，能够存放的价值 dp[i-1][j]
情况二：背包容量 j>= 物品重量 w[j]，放得下
能够得到的最大价值 = max(dp[i-1][j],v[i]+dp[i-1][j-w[i]])
*/
int dp[110][20010];
int n,w[110],v[110],maxw,i,j;
```

```
int main()
{
	cin>>maxw>>n;
	// 读入每个物品的重量和价值
	for(i = 1;i <= n;i++)
	{
		cin>>w[i]>>v[i];
	}

	// 推导
	// 循环 n 个物品
	for(i = 1;i <= n;i++)
	{
		// 循环背包容量从 1~maxw
		for(j = 1;j <= maxw;j++)
		{
			// 如果放不下：背包容量 < 物品重量
			if(j < w[i])
			{
				dp[i][j] = dp[i-1][j];
			}
			else
			{
				// 放得下，就看放进来价值高，还是不放价值高
				dp[i][j] = max(dp[i-1][j],v[i]+dp[i-1][j-w[i]]);
			}
		}
	}

	// 输出 n 个物品，背包容量为 maxw 的情况下最大价值
	cout<<dp[n][maxw];
}
```

一维求解状态转移方程：dp[j]=max(dp[j], dp[j?w[i]]+v[i])

```
#include <bits/stdc++.h>//1-1282-2  简单背包问题    javacn
using namespace std;

int maxw;// 背包承重
int w,v;// 物品的重量和价值
// 讨论：有 i 个物品背包容量为 j 时能够存放的最大价值
int dp[20010];
int n;

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

	// 读入 n 个物品的重量和价值，并计算
	// 循环 n 个物品
	for(int i = 1;i <= n;i++)
	{
		cin>>w>>v;// 读入重量和价值

		// 计算
		// 逆序从背包容量循环到当前物品的重量
		for(int j = maxw;j >= w;j--)
		{
			dp[j] = max(dp[j],v+dp[j-w]);
		}
	}

	cout<<dp[maxw];
	return 0;
}
```

完全背包状态转移方程：

二维写法：f[i][j] = max(f[i-1][j], f[i][j-w[i]]+v[i])

一维写法：f[j]=max(f[j],f[j-w[i]]+v[i])

一维状态转义方程和 01 背包一致，要注意，完全背包要从前往后推导。

```
#include<bits/stdc++.h>//2-1780    采灵芝
using namespace std;
int t,m;
int f[100010];
int ti,vi;// 每个物品的采摘时间和价值

int main()
{
	cin>>t>>m;
	for(int i = 1;i <= m;i++)
	{
		cin>>ti>>vi;
		// 从当前物品的重量（采摘时间）~背包容量（最大时间）循环
		for(int j = ti;j <= t;j++)
		{
			f[j] = max(f[j],f[j-ti]+vi);
		}
	}

	cout<<f[t];// 背包能够存储的最大价值
	return 0;
}
```

解法一：将多重背包的 si 个物品分别装入 w 和 v 数组，直接转换为 01 背包！

```
#include <bits/stdc++.h>//3-1888-1    多重背包（1）   javacn
using namespace std;

/*
01 背包：每种物品有 1 件
完全背包：每种物品有无限件数
多重背包：每种物品有 Si 件
解题思路：将多重背包转换为 01 背包
将 Si 件物品都存起来，转换为有 Si 个物品，每个物品有 1 件
*/
int n,c;//c 背包容量
int v[10010],w[10010];
int dp[110];
int vi,wi,si,k;//k 代表数组下标
```

```cpp
int main()
{
	cin>>n>>c;
	for(int i = 1;i <= n;i++)
	{
		cin>>vi>>wi>>si;
		// 第 i 个物品有 si 件，都存入数组
		for(int j = 1;j <= si;j++)
		{
			k++;
			v[k] = vi;
			w[k] = wi;
		}
	}

	//01 背包
	for(int i = 1;i <= k;i++)
	{
		// 逆序从背包容量循环到当前物品体积
		for(int j = c;j >= v[i];j--)
		{
			dp[j] = max(dp[j],dp[j-v[i]]+w[i]);
		}
	}

	cout<<dp[c];
	return 0;
}
```

解法二：在做 01 背包时，体现一下有 Si 件物品这个条件。

```
#include <bits/stdc++.h>//3-1888-2    多重背包（1）   javacn
using namespace std;
01 背包：每种物品有 1 件
完全背包：每种物品有无限件数
多重背包：每种物品有 Si 件
解题思路：将多重背包转换为 01 背包
将 Si 件物品都存起来，转换为有 Si 个物品，每个物品有 1 件
int n,c;//c 背包容量
int v[110],w[110],s[110];
int dp[110];
int main()
{
      cin>>n>>c;
      for(int i = 1; i <= n; i++)
      {
            cin>>v[i]>>w[i]>>s[i];
      }
    //01 背包
    // 有 n 个物品
      for(int i = 1; i <= n; i++)
      {
            for(int k = 1;k <= s[i];k++)
            {
            // 逆序从背包容量循环到当前物品体积
                  for(int j = c; j >= v[i]; j--)
                  {
                        dp[j] = max(dp[j],dp[j-v[i]]+w[i]);
                  }
            }

      }
    cout<<dp[c];
    return 0;
}
```

关于 DP 要理解的关键点：

1、DP 的本质

求有限的集合中的最值（个数）

本质上，DP 代表了走到阶段 i 的所有路线的最优解；

2、DP 需要思考的点：

（1）DP 的状态是什么？状态要求什么：最大、最小、数量？

（2）DP 的状态计算？

状态转义方程；

求解方法：a、递推 b、考虑阶段 i（最后一个阶段的值）的值是如何得来的；

（3）DP 的边界是什么？

关键术语：阶段、状态、决策（状态转移方程）、边界；

以数塔问题（1216: 【基础】数塔问题）为例，理解 DP 的本质，再理解 01 背包的本质（1282: 【提高】简单背包问题）；

经典的 DP 模板题要熟练掌握，熟记状态转义方程！

本题解题的关键点：二进制优化（类似压缩的思想）

（1）有 n 个不同的物品，要讨论 2n 种选择的可能（每个物品选或者不选）；

（2）一个物品有 n 件，虽然要讨论 2n 种选择的可能，但由于 n 个物品是一样的，那么就减少了讨论数量，比如：有 4 个物品，如果是不同物品的选 2 个，选 1 2、2 3 是不同的选择，但如果是相同的物品，选哪两个就都是一样的了。

因此，n 个物品，要讨论的可能就分别是：选 0 个、选 1 个、选 2 个、选 3 个…选 n 个。

（3）要将 0~n 个不同的选择表达出来，比较简单的方法是将 n 二进制化。

比如：整数 7，只需要用 1 2 4 三个数任意组合，就能组合出 0~7 这 8 种可能。

再比如：整数 10，只需要用 1 2 4 3（注意最后一个数），就能组合出 0~10 这 11 种可能，这样 n 这个值就被二进制化了。

因此如果要讨论 10 个一样的物品，就转化为讨论 4 个不同的物品了；而 n 个一样的物品，就转化为 log₂n 个不同的物品进行讨论。

```
dp[j]=max(dp[j], dp[j-v[i]]+w[i])
#include <bits/stdc++.h>//4-1889  多重背包（2）  javacn
using namespace std;
const int N = 20010;
int v[N],w[N],dp[2010];
int n,m;//n 种物品，背包容量为 m
int vi,wi,si;
int k = 0;
```

```
int main()
{
	cin>>n>>m;
	for(int i = 1;i <= n;i++)
	{
		cin>>vi>>wi>>si;
对 si 二进制化，比如：有 10 件一样的物品，我们转换为有 4 件不同的物品：1 2 4 3
		  这 4 种物品的体积分别是：1*vi 2*vi 4*vi 3*vi
		int t = 1;// 权重，表示 2 的次方
		while(t <= si)
		{
			k++;
			v[k] = t * vi;
			w[k] = t * wi;
			si = si - t;
			t = t * 2;
		}
		if(si > 0)		// 如果二进制化有剩余，存入
		{
			k++;
			v[k] = si * vi;
			w[k] = si * wi;
		}
	}
	//01 背包
	for(int i = 1;i <= k;i++)
	{
		for(int j = m;j >= v[i];j--)
		{
			dp[j] = max(dp[j],dp[j-v[i]]+w[i]);
		}
	}
	cout<<dp[m];
	return 0;
}
```

多重背包转换为 01 背包，接下来分 01 背包、完全背包两种情况讨论：

```
#include <bits/stdc++.h>//5-1905      混合背包      javacn
using namespace std;

const int N = 20000;
int v[N],w[N],s[N];
int vi,wi,si;
int k = 0;// 表示存入数组的数据量
int dp[1010];
int n,m;
```

```
int main()
{
	cin>>n>>m;
	for(int i = 1;i <= n;i++)
	{
		cin>>vi>>wi>>si;
		// 如果是多重背包，做二进制拆分
		if(si > 0)
		{
			int t = 1;
			while(t <= si)
			{
				k++;
				w[k] = t * wi;
				v[k] = t * vi;
				s[k] = -1;// 转换为 01 背包
				si = si - t;
				t = t * 2;
			}
			if(si > 0)
			{
				k++;
				w[k] = si * wi;
				v[k] = si * vi;
				s[k] = -1;//01 背包
			}
		}
		else
		{
			k++;
			w[k] = wi;
			v[k] = vi;
			s[k] = si;
		}
	}
```

```
        //计算
        //循环 k 个物品
        for(int i = 1;i <= k;i++)
        {
                //判断是 01 背包还是完全背包
                if(s[i] == -1)
                {
                        for(int j = m;j >= v[i];j--)
                        {
                                dp[j] = max(dp[j],dp[j-v[i]]+w[i]);
                        }
                }
                else
                {
                        for(int j = v[i];j <= m;j++)
                        {
                                dp[j] = max(dp[j],dp[j-v[i]]+w[i]);
                        }
                }
        }

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

4、LIC 和 LCS

dp 数组：存储状态，以 a[i] 结尾的最长不下降子序列的长度

归纳动态转移方程：

dp[i]=1

dp[i]=max(dp[j]+1,dp[i])

j 是 1..i-1 之间的数，a[j]<a[i]

```
#include <bits/stdc++.h>//1-1794      最长不下降子序列（LIS）      javacn
using namespace std;

/*
  dp[i]=1
dp[i]=max(dp[j]+1,dp[i])  j 是 1..i-1 之间的数，a[j]<a[i]
*/
//dp：存储以每个数结尾的最长不下降子序列的长度
int a[10100],dp[10100],n,ma;
```

```
int main()
{
    int i,j;
      cin>>n;
      for(i = 1;i <= n;i++)
      {
            cin>>a[i];
      }

      // 求以每个数结尾的最长不下降子序列的长度
      for(i = 1;i <= n;i++)
      {
            dp[i] = 1;
            // 将 a[i] 尝试续到每个数后面，看 dp[i] 能否增加
            for(j = 1;j < i;j++)
            {
                  if(a[j] < a[i])
                  {
                        dp[i] = max(dp[j]+1,dp[i]);
                  }
            }

            ma = max(dp[i],ma);
      }

      cout<<ma;
      return 0;
}
```

将原来的 dp 数组的存储以每个数结尾的 LIS 序列的长度，修改为存储上升子序列长度为 i 的上升子序列的最小末尾数值。

原理：LIS 长度如果已经确定，那么如果这种长度的子序列的结尾元素越小，后面可能续的元素会更多！

```
#include <bits/stdc++.h>//2-1893     最长上升子序列 LIS（2）      javacn
using namespace std;

//dp: 长度为 i 的 LIS 的最后一位最小值是多少
int a[100100],dp[100100];
int i,n,l,r,mid;
```

```
int main()
{
	// 读入
	scanf("%d",&n);
	for(i = 1; i <= n; i++)
	{
		scanf("%d",&a[i]);
	}
	dp[1] = a[1];	// 边界
	int len = 1;//LIS 的长度
	for(i = 2; i <= n; i++)		// 从第 2 个数开始求解
	{
		// 如果 a[i] 比 dp 最后一位大，a[i] 直接续上去，增加 LIS 的长度
		if(a[i] > dp[len])
		{
			len++;
			dp[len] = a[i];
		}
		else
		{
	// 二分查找到 dp 数组中第 1 个 >=a[i] 的元素下标，替换（dp 数组一定是递增的）
			l = 1;
			r = len;
			while(l <= r)
			{
				mid = (l + r) / 2;
				if(a[i] <= dp[mid]) r = mid - 1;
				else l = mid + 1;
			}
			// 替换
			dp[l] = a[i];
		}
	}
	printf("%d",len);
}
```

我们可以用 dp[i][j] 来表示第一个串的前 i 位，第二个串的前 j 位的 LCS 的长度，那么递推出状态转移方程：

如果当前的 a [i] 和 b[j] 相同（即是有新的公共元素） 这说明该元素一定位于公共子序列中。因此，现在只需要找：a 数组 1~i-1 和 b 数组 1~j-1 的最长公共子序列：

```
dp[i][j]=max(dp[i][j],dp[i-1][j-1]+1);
```

如果不相同，说明最后一个元素肯定不是公共子序列中的元素，那么考虑找 a 数组 1~i-1 和 b 数组 1~j 的 LCS，或者找：a 数组的 1~i 和 b 数组的 1~j-1 的 LCS，那么，状态转义方程如下：

```
dp[i][j]=max(dp[i-1][j],dp[i][j-1]);
```

```cpp
#include <bits/stdc++.h>//3-1821    最长公共子序列(LCS)(1)    javacn
using namespace std;

/*
a[i]==b[j], 方程：dp[i-1][j-1]+1
a[i]!=b[j], 方程：max(dp[i][j-1],dp[i-1][j])
*/
const int N = 1010;// 常量, 表示数组大小
int a[N],b[N],dp[N][N];
int n,i,j;

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

	// 递推
	for(i = 1;i <= n;i++)
	{
		for(j = 1;j <= n;j++)
		{
			if(a[i] == b[j]) dp[i][j] = dp[i-1][j-1] + 1;
			else dp[i][j] = max(dp[i-1][j],dp[i][j-1]);
		}
	}

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

1、因为两个序列都是 1~n 的全排列，那么两个序列元素互异且相同，也就是说只是位置不同;

2、通过 c 数组将 b 序列的数字在 a 序列中的位置求出；

3、如果 b 序列每个元素在 a 序列中的位置递增，说明 b 中的这个数在 a 中的这个数整体位置偏后，可以考虑纳入 LCS；

4、从而就可以转变成求用来记录新的位置的 c 数组中的 LIS。

```
#include <bits/stdc++.h>//4-1822        最长公共子序列 (LCS)(2)     javacn
using namespace std;
const int N = 100100;
int a[N],b[N],c[N],dp[N];
int n,i;

int main()
{
      cin>>n;
      for(i = 1; i <= n; i++)
      {
            scanf("%d",&a[i]);
            c[a[i]] = i;// 求出 a 数组的每个数的位置
      }

      for(i = 1; i <= n; i++)        scanf("%d",&b[i]);
```

```
    // 求 b 数组的每个数在 a 数组的位置（c[b[i]]）的 LIS
    dp[1] = c[b[1]];// 边界
    int len = 1;
    int l,r,mid;
    // 从第 2 个数开始讨论
    for(i = 2; i <= n; i++)
    {
        // 增加 LIS 的长度
        if(c[b[i]] > dp[len])
        {
            len++;
            dp[len] = c[b[i]];
        }
        else
        {
            l = 1;
            r = len;
            while(l <= r)
            {
                mid = (l + r) / 2;
                if(c[b[i]] <= dp[mid]) r = mid - 1;
                else l = mid + 1;
            }

            dp[l] = c[b[i]];
        }
    }

    cout<<len;
    return 0;
}
```

求最长不递减子序列的长度。

```
#include<bits/stdc++.h>//5-1795  拦截导弹    javacn
using namespace std;

int n,a[1010],dp[1010],ma;
int main()
{
	cin>>n;
	for(int i = 1;i <= n;i++)
	{
		cin>>a[i];
		dp[i] = 1;
		for(int j = 1;j < i;j++)
		{
			if(a[j] > a[i])
			{
				dp[i] = max(dp[j]+1,dp[i]);
			}
		}
		ma = max(dp[i],ma);
	}
	cout<<ma;
}
```

思路：求最少的修改次数， 那就是要找出需要修改的数字，而且越少也好。逆向思维， 找最长的上升子序列（LIS）。然后，用总个数减去上升的，即需要修改的数字。

```
#include<bits/stdc++.h>//6-1902    最少的修改次数    dragoncatter
using namespace std;
const int N = 1e5 + 10 ;
int n, cnt = 1;
int a[N], dp[N];
int main()
{
	cin >> n;
	for (int i = 1; i <= n; i++)
	{
		cin >> a[i];
	}
	dp[1] = a[1];
	for (int i = 2; i <= n; i++)
	{
		if (a[i] > dp[cnt])
		{
			dp[++cnt] = a[i];
		}
		else
		{
			int k = lower_bound(dp+1, dp+cnt + 1, a[i], less<int>()) - dp;
			dp[k] = a[i];
		}
	}
	cout << n - cnt;
	return 0;
}
```