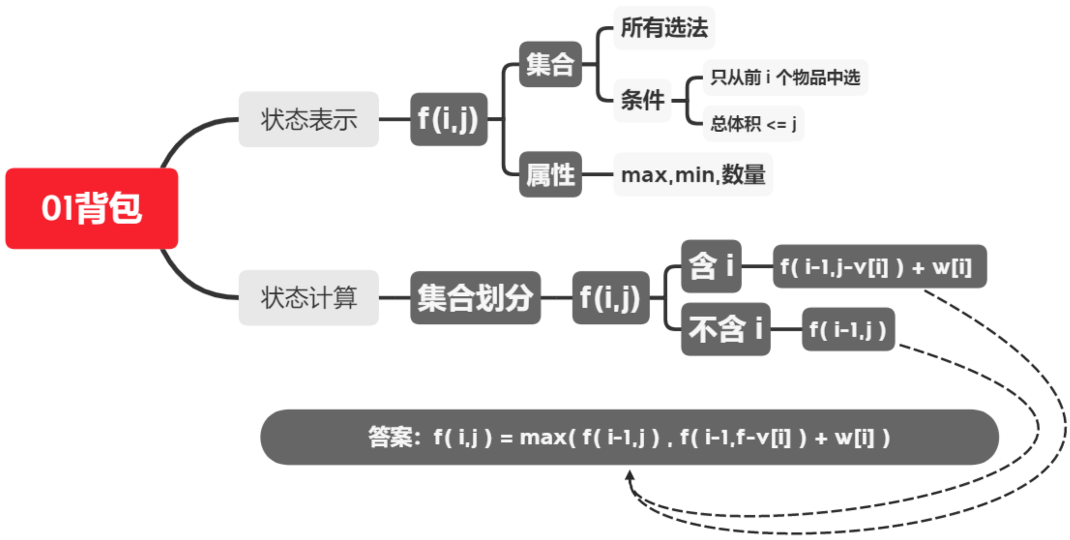
动态规划

背包问题

参考视频：https://www.bilibili.com/video/BV1X741127ZM

- **01背包**

  问题背景：现有`n`件物品，**每件物品都有固定的体积**`v[i]`**和价值**`w[i]`**,每件物品只能装一次**，在规定容量的背包中装下最多的价值



模板题：01背包 采药

朴素代码：

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

const int N = 1010;
int v[N],w[N],n,bag_v,f[N][N]; //f数组存的是考虑前i个物品,背包容量为j时能取到的最大价值

int main()
{
    cin>>n>>bag_v; //读入物品数量和背包容量
```

```
    for(int i=1;i<=n;i++)   cin>>v[i]>>w[i];//依次读入物品体积和价值

    for(int i=1;i<=n;i++)//从1开始枚举，因为装前0个物品的价值都是0，即f[0]
[1~bag_v] = 0
    {
        for(int j=0;j<=bag_v;j++) //枚举背包容量
        {
            f[i][j] = f[i-1][j]; //不装第i个物品的状态一定存在
            if(j>=v[i]) //只有当当前背包容量大于第i个物品的体积时才可以装第i个物品
                f[i][j] = max(f[i-1][j],f[i-1][j-v[i]] + w[i]); //两种状态，哪
个大取哪个
        }
    }

    cout<<f[n][bag_v];//考虑前n个物品，背包容量为bag_v的情况
}
```

可以对上面代码进行优化，压缩空间，压缩空间实际是等价变形那些式子

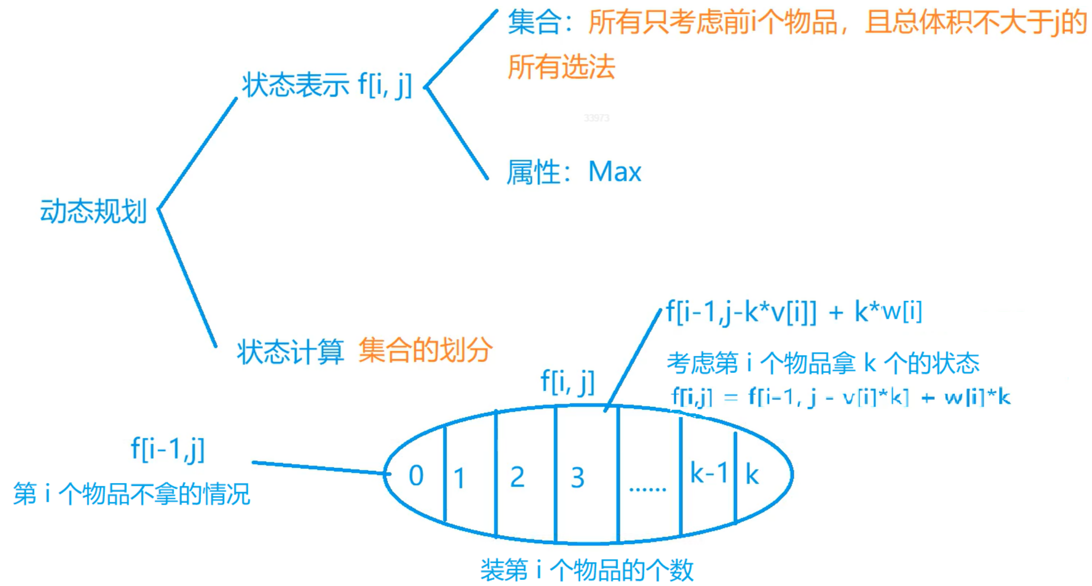
j 从大到小循环的原因是`f[i, j]`要用`f[i - 1, j - v[i]] + w[i]`来更新，从大到小可以保证算`f[j]`时用到的`f[j - v[i]]`存储的是`f[i - 1, j - v[i]]`，而不是`f[i, j - v[i]]`；如果从小到大循环，那么`f[j - v[i]]`会在`f[j]`前被计算出来，那么它就表示`f[i, j - v[i]]`了

一维优化代码：

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

const int N = 1010;
int v[N],w[N],n,bag_v,f[N];
int main()
{
    cin>>n>>bag_v;

    for(int i=1;i<=n;i++)   cin>>v[i]>>w[i];

    for(int i=1;i<=n;i++)
    {
        for(int j=bag_v;j>=0;j--)
        {
            //f[j] = f[j]; 在第i层计算的f[j]实际就是f[i][j],第i-1层计算的就是f[i-
1][j]
            //f[i][j] = f[i-1][j],f[i-1][j]先算出来，所以可以删掉变成f[j]=f[j],
恒等式直接删
            if(j>=v[i])
                f[j] = max(f[j],f[j-v[i]] + w[i]); //等价于f[i][j],f[i][j-
v[i]]+w[i]
                //f[i][j] = max(f[i][j],f[i-1][j-v[i]] + w[i])
   //但第二个参数是要用到f[i-1][j-v[i]] + w[i]的,因为当前为第i层，所以f[i-1][j-
v[i]]是还未计算出来的
   //f[j-v[i]]实际是f[i][j-v[i]]，所以要逆序枚举保证f[j-v[i]]是还未计算出来
        }
    }

    cout<<f[bag_v];
}
```

- ## 完全背包

完全背包问题和01背包问题很相似，不同点就是完全背包问题的物品个数是无限量的



把 `f[i-1,j-v[i]*k] + w[i]*k` 展开下图橙色部分：

f[i,j-v] + w

状态转移方程 : f[i,j] = Max(f[i-1,j] , $\boxed{\text{f[i-1,j-v]+w, f[i-1,j-2v]+2w , f[i-1,j-3v]+3w,...}}$)

f[i,j-v] = Max(f[i,j-v] , f[i-1,j-2v]+w , f[i-1,j-3v]+2w , ...)

惊奇的发现橙色框那一堆就等于 f[i,j-v]+w

所以： f[i,j] = Max(f[i-1,j] , f[i,j-v] + w)

所以 完全背包的状态转移方程

完全背包：`f[i][j] = max(f[i-1][j],f[i][j-v[i]] + w[i]);`

于 0 1 背包的状态转移方程非常相似， 区别在于 0 1背包是考虑不装 i ，而完全背包考虑

01背包 ： `f[i][j] = max(f[i-1][j],f[i-1][j-v[i]] + w[i]);`

完全背包朴素代码：

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

const int N = 1010;
int v[N],w[N],f[N][N];

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

    for(int i=1;i<=n;i++)     cin>>v[i]>>w[i];

    for(int i=1;i<=n;i++)
        for(int j=0;j<=m;j++)
        {
```

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

    cout<<f[n][m]<<endl;

    return 0;
}
```

模板题： 完全背包 疯狂采药

一维空间优化：

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

const int N = 1010;
int v[N],w[N],f[N];

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

    for(int i=1;i<=n;i++)    cin>>v[i]>>w[i];

    for(int i=1;i<=n;i++)
        for(int j=v[i];j<=m;j++)  //因为 f[i][j] = f[i-1][j]; f[i-1][j]是先更新的，所以这条语句执行后 f[i-1][j] 的值赋给f[i][j],删掉一维，将变成一个恒等式，所以可以直接删，后面也只需删掉一维
            f[j] = max(f[j],f[j-v[i]] + w[i]);

    cout<<f[m]<<endl;

    return 0;
}
```

- ## 多重背包

  有 `N` 种物品和一个容量是 `V` 的背包。

  第 `i` 种物品最多有 S_i 件，每件体积是 V_i，价值是 W_i。

  求解将哪些物品装入背包，可使物品体积总和不超过背包容量，且价值总和最大。
  输出最大价值。

  - **朴素做法**

    思路和完全背包一样，只不过物品的数量受到了限制，完全背包第`i`个物品的个数是无限个，而多重背包的第`i`个物品的个数是`s[i]`个，是有限制的。

    其状态转移方程，与最原始的完全背包一样

    `f[i][j] = max(f[i-1][j],f[i-1][j-v[i]*k] + w[i]*k) //k = 0,1,2,3... s[i]`

    其中`f[i-1][j]`是不装第`i`个物品的所携带价值的最大值。

    朴素版代码：

```
#include<iostream>
using namespace std;
const int N = 110;
int f[N][N];
int v[N],w[N],s[N];

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

    for(int i=1;i<=n;i++)   cin>>v[i]>>w[i]>>s[i];

    for(int i=1;i<=n;i++)
    {
        for(int j=0;j<=m;j++)
            for(int k=0;k<=s[i];k++)
                if(v[i]*k <= j)
                    f[i][j] = max(f[i][j],f[i-1][j-k*v[i]] + w[i]*k);

    }

    cout<<f[n][m];
    return 0;
}
```

- **优化版做法**

  `1,2,4,8,16...,512` 这堆数中可以凑出每个数最多只能选一次能凑出 `0~1023` (和为1023)范围内的任意一个数

  假设第 `i` 组中的物品个数为 `s[i]`，价值和体积为 `v[k] w[k] k<=s[i]` 且 $2^k < s[i] < 2^{k+1}$ ，那么可以将拿去物品的个数的价值全部用 `1,2,4,8...2^(k-1),s[i]-(1+2+4+8+...+2^(k-1) )` 表示出来

  用数组 `total_v[],total_w[]` 来表示拿取 `1个物品i，2个物品i,4个物品i,...,2^(k-1)个物品i,s[i]-(1+2+4+...+2^(k-1) )个物品i ，所具有的价值和体积`

  那么多重背包问题就可以转化成一个 **01背包问题** 了,此时物品就为 `total_v[],total_w[]` 的值的个数

  代码:

```
#include<iostream>
#include<algorithm>
using namespace std;
const int N = 25000,M = 2010; //最多1000物品，每个物品最多2000个，所以
total_v,w[]
int f[M];                       //最多有 1000*log 2000 + 1000种可能，N直接
开25000
int v[N],w[N],cnt;  //w，v  => total_w[],total_v[]

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

```
    for(int i=1;i<=n;i++)
    {
        int a,b,s;
        cin>>a>>b>>s;
        int k=1;
        while(k <= s)
        {
            v[++cnt] = k*a; //存取1,2,4,8...2^k个的v，w
            w[cnt] = k*b;
            s -= k;
            k *= 2;
        }
        if(s > 0)
        {
            v[++cnt] = s*a;//存取s[i]-(1+2+4+8+...+2^k)个的v，w
            w[cnt] = s*b;
        }
    }

    for(int i=1;i<=cnt;i++) //转化为01背包
        for(int j=m;j>=v[i];j--)
            f[j] = max(f[j],f[j-v[i]] + w[i]);

    cout<<f[m]<<endl;

    return 0;
}
```

[多重背包 1](#) [多重背包 2](#)

- ## 分组背包问题

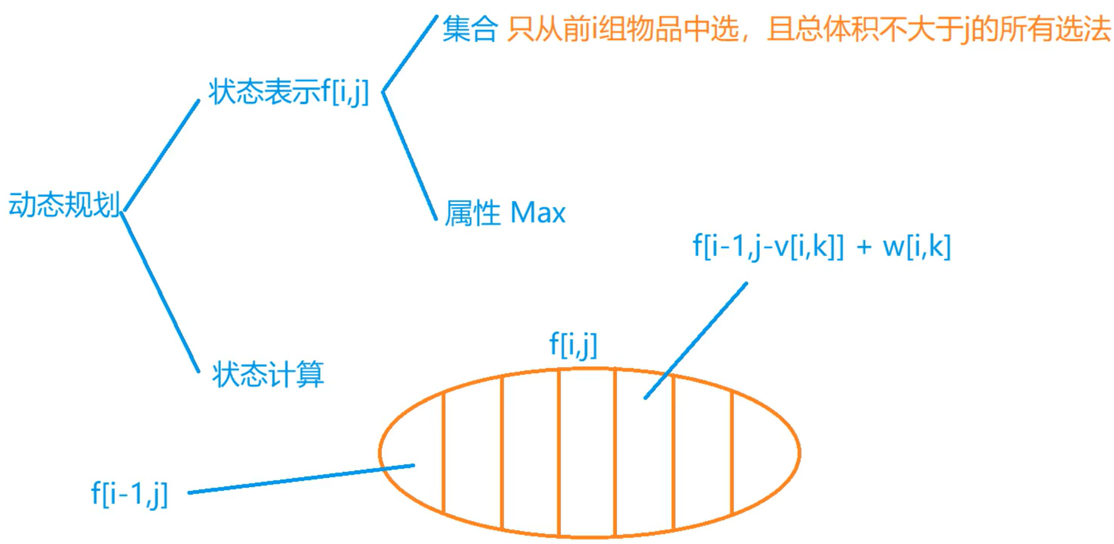
有 N 组物品和一个容量是 V 的背包。

每组物品有若干个，同一组内的物品最多只能选一个。
每件物品的体积是 `v[i][k]`，价值是 `w[i][k]`，其中 `i` 是组号，`k` 是组内编号。

求解将哪些物品装入背包，可使物品总体积不超过背包容量，且总价值最大。

输出最大价值。

它和01背包也有些相似，只不过01背包枚举的是第 `i` 个物品选 `or` 不选，而分组背包问题枚举的是第 `i` 个物品选哪个，就是这个物品更多了。



朴素写法:

```
#include<iostream>
using namespace std;
const int N = 110;
int f[N][N];
int v[N][N],w[N][N],s[N];


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

    for(int i=1;i<=n;i++)
        for(int j=0;j<=m;j++)
        {
            f[i][j] = f[i-1][j];
            for(int k=1;k<=s[i];k++)  //转化成01背包
                if(j>=v[i][k])
                    f[i][j] = max(f[i][j],f[i-1][j-v[i][k]]+w[i][k]);
        }
    cout<<f[n][m]<<endl;

    return 0;
}
```

优化版:

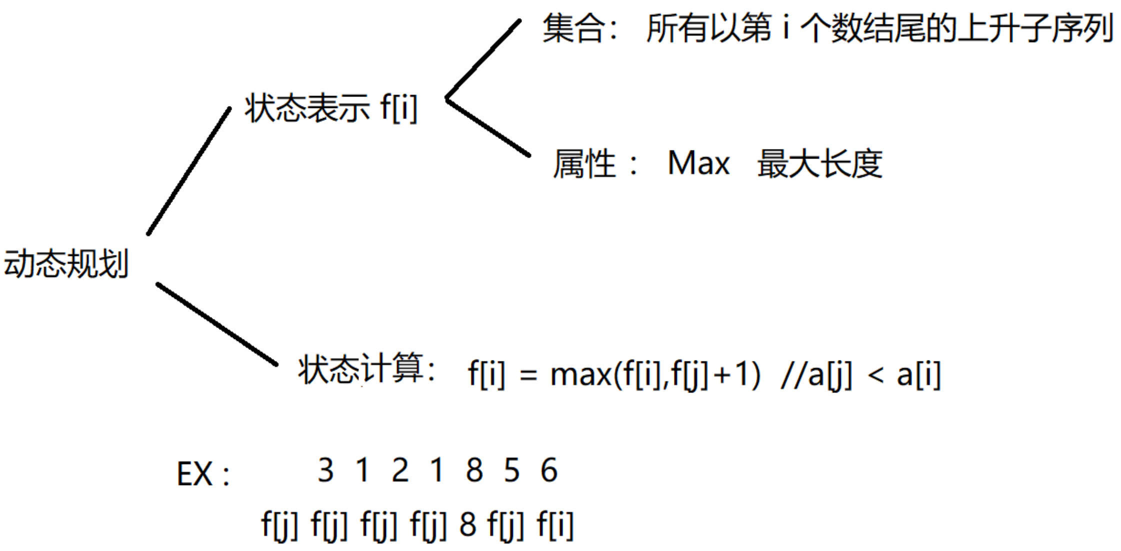
```
#include<iostream>
using namespace std;
const int N = 110;
int f[N];
int v[N][N],w[N][N],s[N];


int main()
{
    int n,m;
    cin>>n>>m;
    for(int i=1;i<=n;i++)
    {
        cin>>s[i];
        for(int k=1;k<=s[i];k++)
        {
            cin>>v[i][k]>>w[i][k];
```

```
        }
    }

    for(int i=1;i<=n;i++)
        for(int j=m;j>=0;j--)
            for(int k=1;k<=s[i];k++) //一维01背包
                if(j>=v[i][k])
                    f[j] = max(f[j],f[j-v[i][k]]+w[i][k]);

    cout<<f[m];

    return 0;
}
```

线性DP

- **最长上升子序列问题**

```
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。
input:
7
3 1 2 1 8 5 6
output:
4
```



首先定义`f[]的集合，和属性，然后将其状态计算方程列出`，这里的状态转移方程是`f[i] = max(f[j]+1,f[i]);`

序列为第`i`个数`a[i]`结尾，前面可以与它构成严格上升的序列的数`a[j]`，如果能构成严格上升的单调序列，那么此时以`a[i]`结尾的序列的长度`f[i]`为以`a[j]`结尾的长度`f[j] + 1`，与所有符合的`a[j]`作比较，找出`f[i]`的最大值

朴素做法(n^2)：

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

```
const int N = 1010;
int f[N],a[N];

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

    for(int i = 1; i <= n; i++)
    {   //f[i] 以a[i]结尾的所有的上升序列
        f[i] = 1; // a[i] 前面的数是 空集
        for(int j = 1; j < i; j++)
        {
            if(a[i] > a[j])
                f[i] = max(f[i],f[j] + 1); //a[i]前面的数是a[j]
        }
    }

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

    cout << res << endl;

    return 0;
}
```

如果要输出序列，则记下转移路径

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

const int N = 1010;
int f[N],a[N],p[N];

void printpath(int k) //输出最长序列
{
    if(k != 0)    printpath(p[k]); //p[]一开始全部初始化为0，且转移过去的j(下标)必定大于0
    if(k>0)
        cout<<a[k]<<' ';
}

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

    for(int i=1;i<=n;i++)
    {
        f[i] = 1;
        for(int j=1;j<i;j++)
            if(a[j] < a[i])
                if(f[i] < f[j]+1)
```

```
                {
                    f[i] = f[j] + 1;
                    p[i] = j; //记录f[i]状态是由哪个f[j]转移过来的
                }
        }
    int k = 0;
    for(int i=1;i<=n;i++)
        if(f[k] < f[i])
            k = i;     //找到最长的序列的结尾下标
    cout<<f[k]<<endl;

    printpath(k);   //输出序列

    return 0;
}
```

贪心做法

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。

输入格式

第一行包含整数N。

第二行包含N个整数，表示完整序列。

输出格式

输出一个整数，表示最大长度。

数据范围

$1 \leq N \leq 100000$,
$-10^9 \leq \text{数列中的数} \leq 10^9$

输入样例:

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

输出样例:

```
4
```

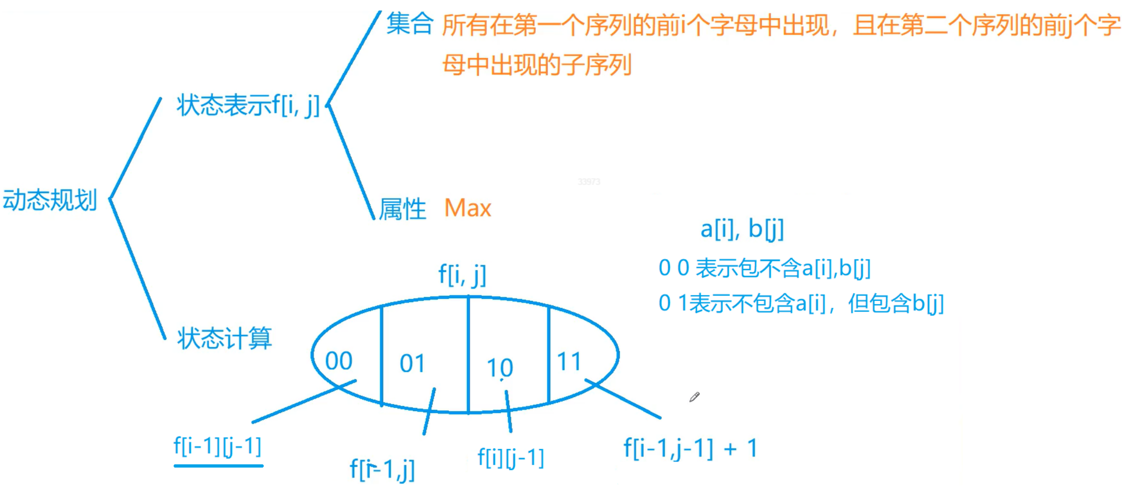
```
#include <iostream>
#include <algorithm>
using namespace std;
const int N = 100010;

int n;
int a[N];
int q[N]; //q[] 存储 长度为i的结尾的数 ex: q[5] = 8 长度为5的上升序列，结尾数字为8
//q[i] 是单调递增的，因为序列是严格单调递增，由q[i]的性质，q[i]也是递增的
int main()
```

```
{
    scanf("%d", &n);
    for (int i = 0; i < n; i ++ ) scanf("%d", &a[i]);

    int len = 0;
    for (int i = 0; i < n; i ++ )
    {
        int l = 0, r = len;
        while (l < r)  //二分找到 最后一个小于a[i]的下标
        {
            int mid = l + r + 1 >> 1;
            if (q[mid] < a[i]) l = mid;
            else r = mid - 1;
        }
        len = max(len, r + 1); //保持区间长度，每次递增 1
        q[r + 1] = a[i]; //q[r+1]第一个比a[i]大的数
    }

    printf("%d\n", len);

    return 0;
}

//STL做法：

#include<iostream>
#include<algorithm>
#include<vector>
using namespace std;
int main(void) {
    int n; cin >> n;
    vector<int>arr(n);
    for (int i = 0; i < n; ++i)cin >> arr[i];

    vector<int>stk;//模拟堆栈
    stk.push_back(arr[0]);

    for (int i = 1; i < n; ++i) {
        if (arr[i] > stk.back())//如果该元素大于栈顶元素,将该元素入栈
            stk.push_back(arr[i]);
        else//替换掉第一个大于或者等于这个数字的那个数
            *lower_bound(stk.begin(), stk.end(), arr[i]) = arr[i];
    }
    cout << stk.size() << endl;
    return 0;
}
```

- **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。
输入格式
第一行包含两个整数N和M。

第二行包含一个长度为N的字符串，表示字符串A。
```

```
第三行包含一个长度为M的字符串，表示字符串B。

字符串均由小写字母构成。

输出格式
输出一个整数，表示最大长度。

数据范围
1≤N≤1000,

输入样例：
4 5
acbd
abedc
输出样例：
3
```



其中，0 1 和 1 0 的两种情况是f[i-1][j],f[i][j-1]的子集，也就是说f[i-1][j]和f[i][j-1]包含这两情况，用这两种状态代替0 1,1 0来计算，在max（）计算时可以重叠，不影响max的结果

与上述相同 f[i-1][j-1] 也被包含在 f[i-1][j]和j[i][j-1] 中因此可以省略 f[i-1][j-1] 的状态计算

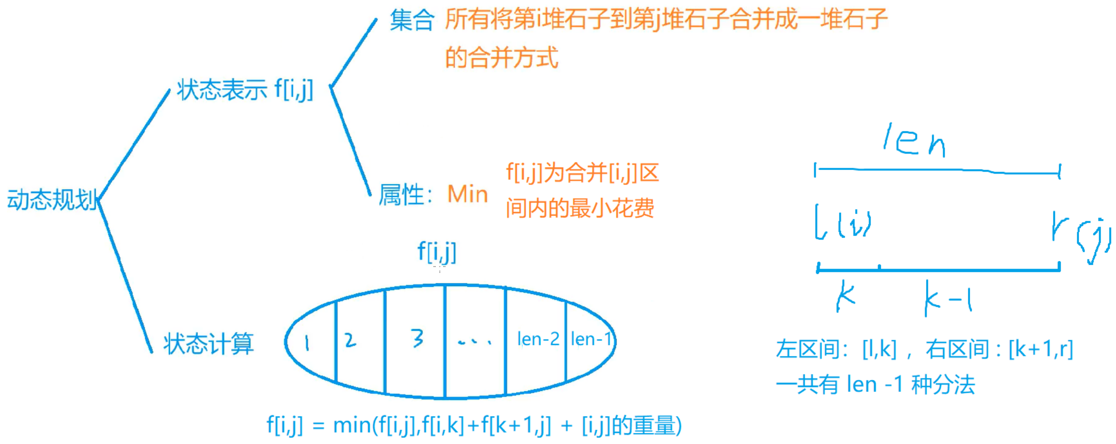
```
#include<iostream>
#include<algorithm>
#include<string>
using namespace std;

const int N = 1010;
int f[N][N];
char a[N],b[N];

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

    cin>>a+1;
    cin>>b+1;

    for(int i=1;i<=n;i++)
        for(int j=1;j<=m;j++)
```

```
        {
            f[i][j] = max(f[i-1][j],f[i][j-1]); //包含了f[i-1][j-1]的情况
            if(a[i] == b[j])
                f[i][j] = max(f[i][j],f[i-1][j-1]+1); //在a[i]==b[j]成立的前提下，计算包含a[i],b[j]的情况 (f[i-1][j-1]+1)
        }

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

区间DP

- **合并石子**

```
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，现在要将这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， 我们可以先合并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11，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。

输入格式
第一行一个数N表示石子的堆数N。

第二行N个数，表示每堆石子的质量(均不超过1000)。

输出格式
输出一个整数，表示最小代价。

数据范围
1≤N≤300
输入样例：
4
1 3 5 2
输出样例：
22
```



对于大部分区间DP问题， 先枚举区间长度，再枚举 左端点 和 右端点

`枚举区间长度 len ，因为相邻两堆合成一堆，所以区间长度为 [2,n] ，不能跳过枚举长度 len，因为合并时，中间有可能会存在 2，3，len-1 堆合成一堆再参与合并 ，所以要知道那一部分小区间对应的最小代价 f[i][j],假如，先将[3,5]合并再于剩下的合并，那么必须得知道 f[3][5]`

`再枚举左端点，因为已知len，所以可以计算出右端点`

`枚举分界点，进行状态转移计算`

代码O(n^3):

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

const int N = 310;
int a[N],s[N];
int f[N][N];

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

    for(int i=1;i<=n;i++)
    {
        cin>>a[i];s[i] = s[i-1] + a[i]; //前缀和
    }

    for(int len = 2;len<=n;len++) //枚举长度
        for(int i = 1; i+len-1<=n ;i++) //枚举左端点
        {
            int l = i , r = i+len-1; //r 右端点
            f[l][r] = 1e9; //初始化，求最小，初始化大一些
            for(int k = l; k<r ;k++) //枚举分界点
                f[l][r] = min(f[l][r],f[l][k]+f[k+1][r] + s[r]-s[l-1]);//前缀和算a[l,r]的总值
        }
    cout<<f[1][n]; //输出区间[1,n]的最小代价

    return 0;
}
```

[石子合并](#)

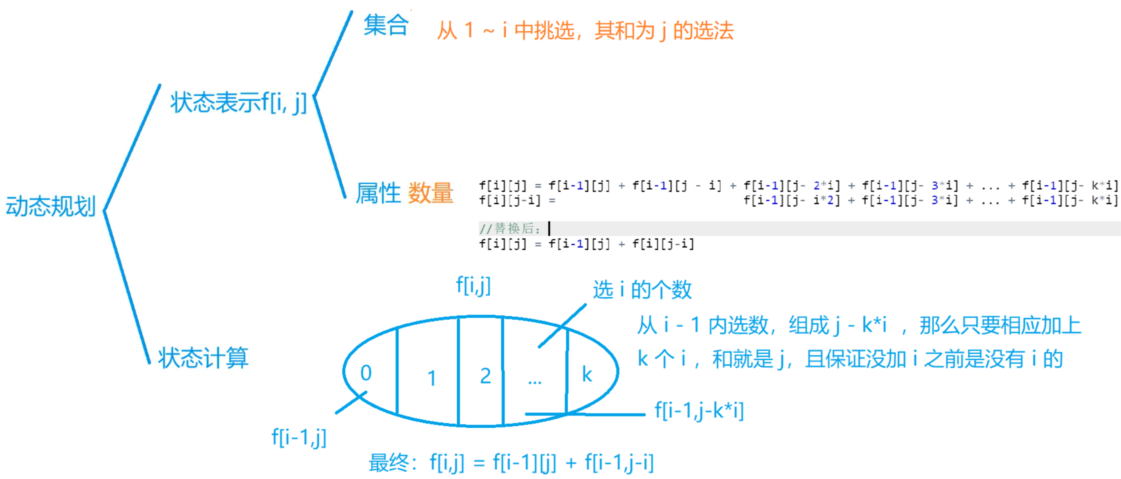
计数类DP

计数类PD，即属性为数量

- 整数划分

```
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。

输入格式
共一行，包含一个整数n。

输出格式
共一行，包含一个整数，表示总划分数量。

由于答案可能很大，输出结果请对109+7取模。

数据范围
1≤n≤1000
输入样例:
5
输出样例:
7
```

完全背包法:



朴素做法:

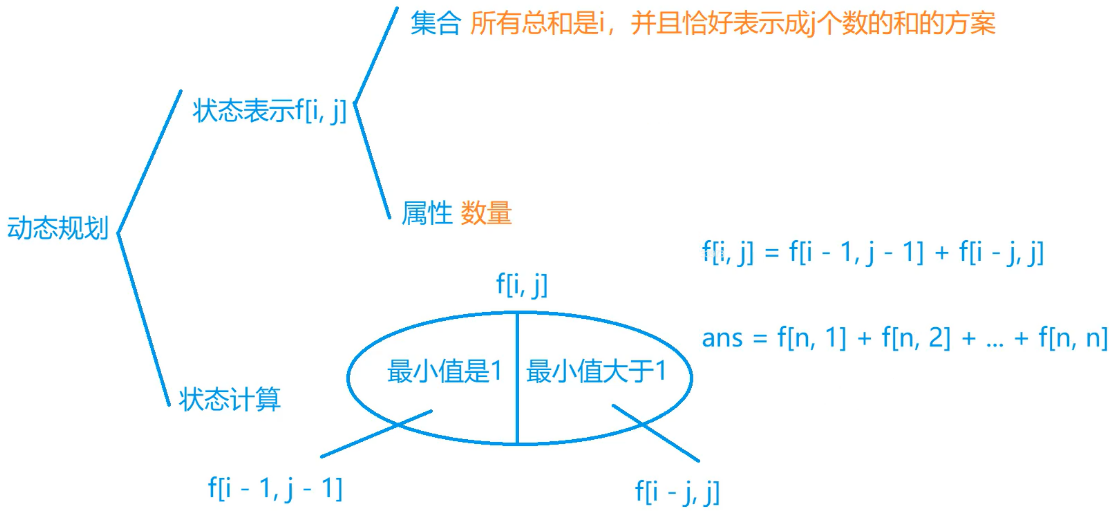
```
#include<iostream>
#include<algorithm>
using namespace std;
const int N = 1010,mod = 1e9 + 7;
int f[N][N];

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

```

```
    f[0][0] = 1;//也可以把f[i][0]全部初始化为1，但是没必要，因为f[i][j] = f[i-1]
[j] 并且 j从0开 会做这个操作

    for(int i=1;i<=n;i++)
        for(int j=0;j<=n;j++)
        {
            f[i][j] = f[i-1][j];
            if(j>=i)
                f[i][j] = (f[i-1][j] + f[i][j-i])%mod;
        }

    cout<<f[n][n];

    return 0;
}
```

优化一维:

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

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

    f[0] = 1;

    for(int i=1;i<=n;i++)
        for(int j=i;j<=n;j++) //完全背包从小到大循环，但状态方程有f[j-i]，所以必须
j>=i
            f[j] = (f[j] + f[j-i])%mod;

    cout<<f[n];

    return 0;
}
```

其他算法




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

using namespace std;

const int N = 1010, mod = 1e9 + 7;

int n;
int f[N][N];

int main()
{
    cin >> n;

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

    int res = 0;
    for (int i = 1; i <= n; i ++ ) res = (res + f[n][i]) % mod;

    cout << res << endl;

    return 0;
}
```

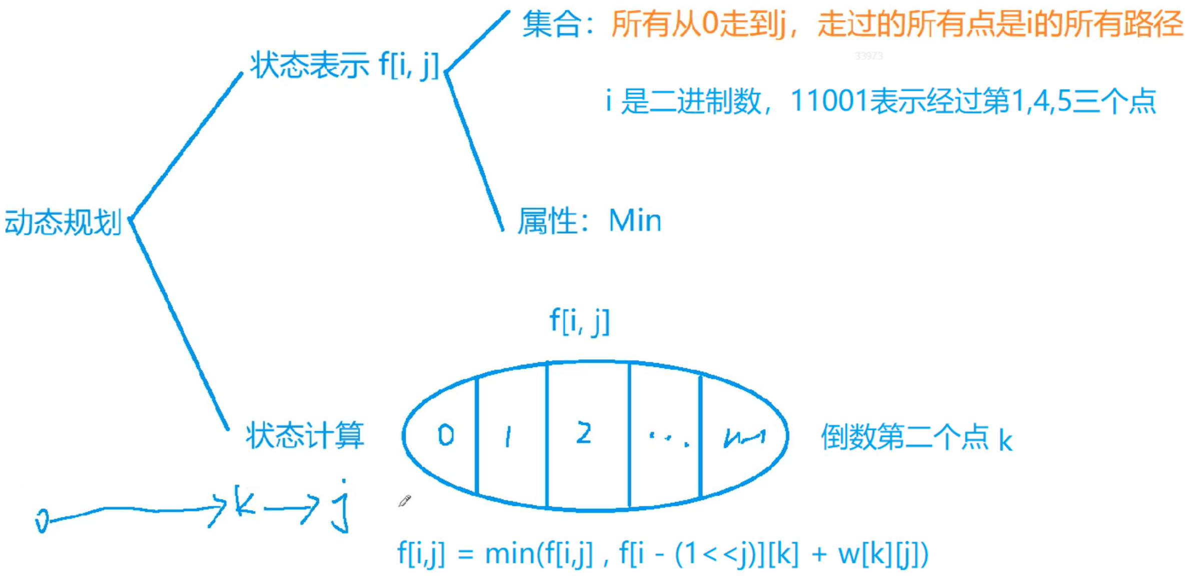
数位统计DP

状态压缩DP

状态压缩主要是用二进制 01 来表示状态

```
#最短Hamilton路径#

给定一张 n 个点的带权无向图，点从 0~n-1 标号，求起点 0 到终点 n-1 的最短Hamilton路径。
Hamilton路径的定义是从 0 到 n-1 不重不漏地经过每个点恰好一次。

输入格式
```

```
第一行输入整数n。

接下来n行每行n个整数，其中第i行第j个整数表示点i到j的距离（记为a[i,j]）。

对于任意的x,y,z，数据保证 a[x,x]=0，a[x,y]=a[y,x] 并且 a[x,y]+a[y,z]>=a[x,z]。

输出格式
输出一个整数，表示最短Hamilton路径的长度。

数据范围
1≤n≤20
0≤a[i,j]≤107
输入样例：
5
0 2 4 5 1
2 0 6 5 3
4 6 0 8 3
5 5 8 0 5
1 3 3 5 0
输出样例：
18
```



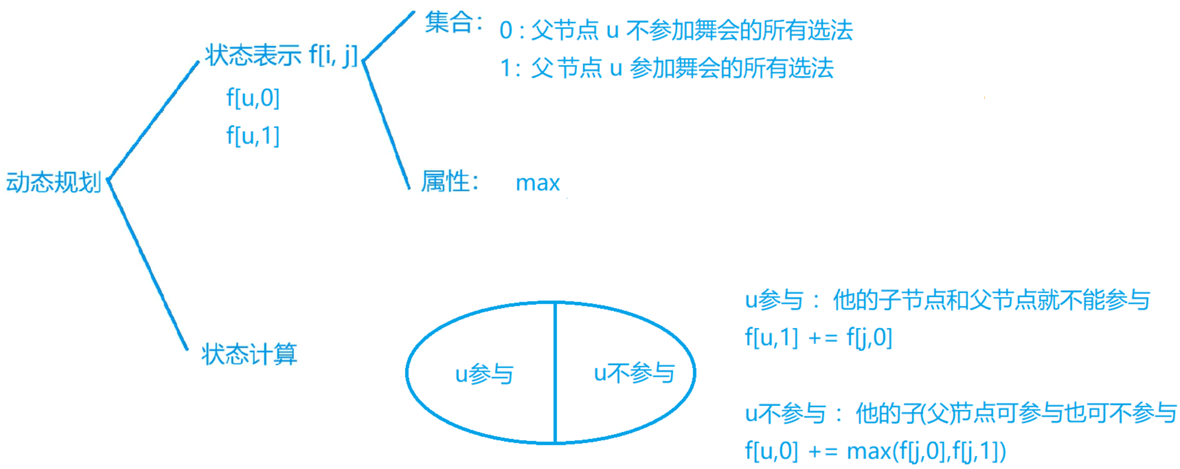

```
#include<iostream>
#include<cstring>
#include<algorithm>
using namespace std;

const int N = 20,M = 1<<N;
int w[N][N],f[M][N];

int main()
{
    int n;
    cin>>n;
    for(int i=0;i<n;i++)
        for(int j=0;j<n;j++)
            cin>>w[i][j];

    memset(f,0x3f,sizeof f); //初始化成无穷大
```

```
    f[1][0] = 0; //0走到0

    for(int i=0;i < 1<<n ;i++) //枚举所有的路径
        for(int j=0;j<n;j++)  //枚举所有的的终点
            if(i>>j & 1) //j这个点必须要在走过的路径中
                for(int k=0;k<n;k++)
                    if((i - (1<<j)) >> k & 1) //除去j点后，k点要在路径 i 中
                        f[i][j] = min(f[i][j],f[i-(1<<j)][k] + w[k][j]);
                        //比较当前路径 i 中走到j点,和当前路径 i剔除j点后,走到k点在走到j点
的花费

    cout<<f[(1<<n)-1][n-1]; //f[111...11][n-1]

    return 0;
}
```

树形DP

```
#没有上司的舞会

Ural大学有N名职员，编号为1~N。

他们的关系就像一棵以校长为根的树，父节点就是子节点的直接上司。

每个职员有一个快乐指数，用整数 Hi 给出，其中 1≤i≤N。

现在要召开一场周年庆宴会，不过，没有职员愿意和直接上司一起参会。

在满足这个条件的前提下，主办方希望邀请一部分职员参会，使得所有参会职员的快乐指数总和最大，求这个
最大值。

输入格式
第一行一个整数N。

接下来N行，第 i 行表示 i 号职员的快乐指数Hi。

接下来N-1行，每行输入一对整数L, K,表示K是L的直接上司。

输出格式
输出最大的快乐指数。

数据范围
1≤N≤6000,
−128≤Hi≤127
输入样例:
7
1
1
1
1
1
1
1
1 3
2 3
```

```
6 4
7 4
4 5
3 5
输出样例：
5
```




```cpp
#include<iostream>
#include<cstring>
#include<algorithm>
using namespace std;

const int N = 6010;

int h[N],e[N],ne[N],idx;
int happy[N],f[N][2];
bool has_father[N];

void add(int a,int b) //邻接表插入
{
    e[idx] = b,ne[idx] = h[a],h[a] = idx++;
}

void dfs(int u)
{
    f[u][1] = happy[u];
    f[u][0] = 0;    //单看u 参与 和 不参与 的情况

    for(int i = h[u];i != -1;i = ne[i])
    {
        int j = e[i];

        dfs(j);

        f[u][0] += max(f[j][1],f[j][0]); //父节点u 不参与
        f[u][1] += f[j][0]; //父节点 u 参与
    }
}

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

```
    for(int i=1;i<=n;i++)    cin>>happy[i];

    memset(h,-1,sizeof h);

    for(int i=1;i<=n-1;i++)
    {
        int a,b;
        cin>>a>>b;
        add(b,a); //b是a的父节点  b->a  a-/->b
        has_father[a] = true;
    }

    int root;

    for(int i=1;i<=n;i++)
        if(!has_father[i])  {root = i;break;}//找根节点

    dfs(root);

    cout<<max(f[root][0],f[root][1]);
    //比较父节点参与 和 不参与两种情况哪个更大

    return 0;
}
```