动态规划

01背包

有 n 件物品和一个容量为 W 的背包，第 i 件物品的体积为 $w[i]$，价值为 $v[i]$，求解将哪些物品装入背包中使总价值最大。

思路：

当放入一个价值为 $w[i]$ 的物品后，价值增加了 $v[i]$，于是我们可以构建一个二维的 $dp[i][j]$ 数组，装入第 i 件物品时，背包容量为 j 能实现的 **最大价值**，可以得到 **转移方程** $dp[i][j] = max(dp[i-1][j], dp[i-1][j-w[i]] + v[i])$。

```
for (int i = 1; i <= n; i++)
    for (int j = 0; j <= W; j++){
        dp[i][j] = dp[i - 1][j];
        if (j >= w[i])
            dp[i][j] = max(dp[i][j], dp[i - 1][j - w[i]] + v[i]);
    }
```

我们可以发现，第 i 个物品的状态是由第 $i-1$ 个物品转移过来的，每次的 j 转移过来后，第 $i-1$ 个方程的 j 已经没用了，于是我们想到可以把二维方程压缩成 **一维** 的，用以 **优化空间复杂度**。

```
for (int i = 1; i <= n; i++)  //当前装第 i 件物品
    for (int j = W; j >= w[i]; j--)  //背包容量为 j
        dp[j] = max(dp[j], dp[j - w[i]] + v[i]);  //判断背包容量为 j 的情况下能是实现总价值最大是多少
```

完全背包

有 n 件物品和一个容量为 W 的背包，第 i 件物品的体积为 $w[i]$，价值为 $v[i]$，每件物品有**无限个**，求解将哪些物品装入背包中使总价值最大。

思路:

思路和**01背包**差不多，但是每一件物品有**无限个**，其实就是从每 **种** 物品中取 $0, 1, 2, \ldots$ 件物品加入背包中

```
for (int i = 1; i <= n; i++)
    for (int j = 0; j <= W; j++)
        for (int k = 0; k * w[i] <= j; k++)     //选取几个物品
            dp[i][j] = max(dp[i][j], dp[i - 1][j - k * w[i]] + k * v[i]);
```

实际上，我们可以发现，取 k 件物品可以从取 $k-1$ 件转移过来，那么我们就可以将 k 的循环优化掉

```
for (int i = 1; i <= n; i++)
    for (int j = 0; j <= W; j++){
        dp[i][j] = dp[i - 1][j];
        if (j >= w[i])
            dp[i][j] = max(dp[i][j], dp[i][j - w[i]] + v[i]);
    }
```

和 01 背包 类似地压缩成一维

```
for (int i = 1; i <= n; i++)
    for (int j = w[i]; j <= W; j++)
        dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
```

多重背包

有 n **种**物品和一个容量为 W 的背包，第 i **种**物品的体积为 $w[i]$，价值为 $v[i]$，数量为 $s[i]$，求解将哪些物品装入背包中使总价值最大。

思路：

对于每一种物品，都有 $s[i]$ 种取法，我们可以将其转化为**01背包**问题

```
for (int i = 1; i <= n; i++){
    for (int j = W; j >= 0; j--)
        for (int k = 0; k <= s[i]; k++){
            if (j - k * w[i] < 0) break;
            dp[j] = max(dp[j], dp[j - k * w[i]] + k * v[i]);
        }
```

上述方法的时间复杂度为 $O(n*m*s)$。

```
for (int i = 1; i <= n; i++){
    scanf("%lld%lld%lld", &x, &y, &s);  //x 为体积， y 为价值， s 为数量
    t = 1;
    while (s >= t){
        w[++num] = x * t;
        v[num] = y * t;
        s -= t;
        t *= 2;
    }
    w[++num] = x * s;
    v[num] = y * s;
}
for (int i = 1; i <= num; i++)
    for (int j = W; j >= w[i]; j--)
        dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
```

尽管采用了 **二进制优化**，时间复杂度还是太高，采用 **单调队列优化**，将时间复杂度优化至 $O(n*m)$

```
#include <bits/stdc++.h>
using namespace std;
const int N = 2e5 + 10;
int n, W, w, v, s, f[N], g[N], q[N];
int main(){
    ios::sync_with_stdio(false);cin.tie(0);
    cin >> n >> W;
    for (int i = 0; i < n; i ++ ){
        memcpy ( g, f, sizeof f);
```

```
        cin >> w >> v >> s;
        for (int j = 0; j < w; j ++ ){
            int head = 0, tail = -1;
            for (int k = j; k <= W; k += w){
                if ( head <= tail && k - s * w > q[head] ) head ++ ;//保证队列长度 <= s
                while ( head <= tail && g[q[tail]] - (q[tail] - j) / w * v <= g[k] - (k - j) / w * v ) tail -- ;//保证队列单调递减
                q[ ++ tail] = k;
                f[k] = g[q[head]] + (k - q[head]) / w * v;
            }
        }
    }
    cout << f[W] << "\n";
    return 0;
}
```

混合背包

放入背包的物品可能只有 **1** 件（01背包），也可能有**无限**件（完全背包），也可能只有**可数的几件**（多重背包）。

思路：

分类讨论即可，哪一类就用哪种方法去 dp。

```
#include <bits/stdc++.h>
using namespace std;
int n, W, w, v, s;
int main(){
    cin >> n >> W;
    vector <int> f(W + 1);
    for (int i = 0; i < n; i ++ ){
        cin >> w >> v >> s;
        if (s == -1){
            for (int j = W; j >= w; j -- )
                f[j] = max(f[j], f[j - w] + v);
        }
        else if (s == 0){
            for (int j = w; j <= W; j ++ )
                f[j] = max(f[j], f[j - w] + v);
        }
        else {
            int t = 1, cnt = 0;
            vector <int> x(s + 1), y(s + 1);
            while (s >= t){
                x[++cnt] = w * t;
                y[cnt] = v * t;
                s -= t;
                t *= 2;
            }
            x[++cnt] = w * s;
            y[cnt] = v * s;
            for (int i = 1; i <= cnt; i ++ )
                for (int j = W; j >= x[i]; j -- )
```

```
                    f[j] = max(f[j], f[j - x[i]] + y[i]);
                }
            }
    cout << f[W] << "\n";
    return 0;
}
```

二维费用的背包

有 n 件物品和一个容量为 W 的背包，背包能承受的最大重量为 M，每件物品只能用一次，第 i 件物品的体积是 $w[i]$，重量为 $m[i]$，价值为 $v[i]$，求解将哪些物品放入背包中使总体积不超过背包容量，总重量不超过背包最大容量，且总价值最大。

思路：

背包的限制条件由一个变成两个，那么我们的循环再多一维即可。

```
for (int i = 1; i <= n; i++)
    for (int j = W; j >= w; j--)  //容量限制
        for (int k = M; k >= m; k--)  //重量限制
            dp[j][k] = max(dp[j][k], dp[j - w][k - m] + v);
```

分组背包

有 n **组**物品，一个容量为 W 的背包，每组物品有若干，同一组的物品最多选一个，第 i 组第 j 件物品的体积为 $w[i][j]$，价值为 $v[i][j]$，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，且使总价值最大。

思路：

考虑每**组**中的**某件**物品选不选，可以选的话，去下一组选下一个，否则在这组继续寻找可以选的物品，当这组遍历完后，去下一组寻找。

```
#include <bits/stdc++.h>
using namespace std;
const int N = 110;
int n, W, s[N], w[N][N], v[N][N], dp[N];
int main(){
    cin >> n >> W;
    for (int i = 1; i <= n; i++){
        scanf("%d", &s[i]);
        for (int j = 1; j <= s[i]; j++)
            scanf("%d %d", &w[i][j], &v[i][j]);
    }
    for (int i = 1; i <= n; i++)
        for (int j = W; j >= 0; j--)
            for (int k = 1; k <= s[i]; k++)
                if (j - w[i][k] >= 0)
                    dp[j] = max(dp[j], dp[j - w[i][k]] + v[i][k]);
    cout << dp[W] << "\n";
    return 0;
}
```

有依赖的背包

有 n 个物品和一个容量为 W 的背包，物品之间有依赖关系，且之间的依赖关系组成一颗 **树** 的形状，如果选择一个物品，则必须选择它的 **父节点**，第 i 件物品的体积是 $w[i]$，价值为 $v[i]$，依赖的父节点的编号为 $p[i]$，若 $p[i]$ 等于 -1，则为 **根节点**。求将哪些物品装入背包中，使总体积不超过总容量，且总价值最大。

思路：

定义 $f[i][j]$ 为以第 i 个节点为根，容量为 j 的背包的最大价值。那么结果就是 $f[root][W]$，为了知道根节点的最大价值，得通过其子节点来更新。所以采用递归的方式。
对于每一个点，先将这个节点装入背包，然后找到剩余容量可以实现的最大价值，最后更新父节点的最大价值即可。

```
#include <bits/stdc++.h>
using namespace std;
const int N = 110;
int n, W, w[N], v[N], p, f[N][N], root;
vector <int> g[N];
void dfs(int u){
    for (int i = w[u]; i <= W; i ++ )
        f[u][i] = v[u];
    for (auto v : g[u]){
        dfs(v);
        for (int j = W; j >= w[u]; j -- )
            for (int k = 0; k <= j - w[u]; k ++ )
                f[u][j] = max(f[u][j], f[u][j - k] + f[v][k]);
    }
}
int main(){
    cin >> n >> W;
    for (int i = 1; i <= n; i ++ ){
        cin >> w[i] >> v[i] >> p;
        if (p == -1) root = i;
        else g[p].push_back(i);
    }
    dfs(root);
    cout << f[root][W] << "\n";
    return 0;
}
```

背包问题求方案数

有 n 件物品和一个容量为 W 的背包，每件物品只能用一次，第 i 件物品的重量为 $w[i]$，价值为 $v[i]$，求解将哪些物品放入背包使总重量不超过背包容量，且总价值最大，输出 **最优选法的方案数**，答案可能很大，输出答案模 10^9+7 的结果。

思路：

开一个储存方案数的数组 cnt，$cnt[i]$ 表示容量为 i 时的 **方案数**，先将 cnt 的每一个值都初始化为 1，因为 **不装任何东西就是一种方案**，如果装入这件物品使总的价值 **更大**，那么装入后的方案数 **等于** 装之前的方案数，如果装入后总价值 **相等**，那么方案数就是 **二者之和**

```
#include <bits/stdc++.h>
```

```cpp
using namespace std;
#define LL long long
const int mod = 1e9 + 7, N = 1010;
LL n, W, cnt[N], f[N], w, v;
int main(){
    cin >> n >> W;
    for (int i = 0; i <= W; i ++ )
        cnt[i] = 1;
    for (int i = 0; i < n; i ++ ){
        cin >> w >> v;
        for (int j = W; j >= w; j -- )
            if (f[j] < f[j - w] + v){
                f[j] = f[j - w] + v;
                cnt[j] = cnt[j - w];
            }
            else if (f[j] == f[j - w] + v){
                cnt[j] = (cnt[j] + cnt[j - w]) % mod;
            }
    }
    cout << cnt[W] << "\n";
    return 0;
}
```

背包问题求具体方案

有 n 件物品和一个容量为 W 的背包，每件物品只能用一次，第 i 件物品的重量为 $w[i]$，价值为 $v[i]$，求解将哪些物品放入背包使总重量不超过背包容量，且总价值最大，输出 **字典序最小的方案**

思路：

01 背包求解最优方案中 **字典序最小的方案**，**首先** 我们先求 **01背包**，因为这道题需要输出方案，所以我们 **不能压缩空间**，得保留每一步的方案。
又 由于输出字典序最小的，所以我们应该反着来，从 n 到 1 求解最优解，那么 $dp[1][W]$ 就是最优的解。

```cpp
for (int i = n; i >= 1; i--)
    for (int j = 0; j <= W; j++){
        dp[i][j] = dp[i + 1][j];
        if (j >= w[i])
            dp[i][j] = max(dp[i][j], dp[i + 1][j - w[i]] + v[i]);
    }
```

接下来 就是输出的问题，如何判断这个物品**被选中**，如果 $dp[i][k] = dp[i+1][k-w[i]] + v[i]$，说明选择了第 i 个物品是最优的选择方案。

```cpp
for (int i = 1; i <= n; i++)
    if (W - w[i] >= 0 && dp[i][W] == dp[i + 1][W - w[i]] + v[i]){
        cout << i << " ";
        W -= w[i];
    }
```

数位 DP

```
/* pos 表示当前枚举到第几位
sum 表示 d 出现的次数
limit 为 1 表示枚举的数字有限制
zero 为 1 表示有前导 0
d 表示要计算出现次数的数 */
const int N = 15;
LL dp[N][N];
int num[N];
LL dfs(int pos, LL sum, int limit, int zero, int d) {
    if (pos == 0) return sum;
    if (!limit && !zero && dp[pos][sum] != -1) return dp[pos][sum];
    LL ans = 0;
    int up = (limit ? num[pos] : 9);
    for (int i = 0; i <= up; i++) {
        ans += dfs(pos - 1, sum + ((!zero || i) && (i == d)), limit && (i == num[pos]),
                   zero && (i == 0), d);
    }
    if (!limit && !zero) dp[pos][sum] = ans;
    return ans;
}
LL solve(LL x, int d) {
    memset(dp, -1, sizeof dp);
    int len = 0;
    while (x) {
        num[++len] = x % 10;
        x /= 10;
    }
    return dfs(len, 0, 1, 1, d);
}
```

```
#include<bits/stdc++.h>
#define int long long
using namespace std;
constexpr int MAXN = 24 + 10;
int a[MAXN], mod, f[MAXN][MAXN * 10][MAXN * 10];

int dfs(int pos, int sum, int cur, bool lead0, bool lim) {
    if (!pos)return !lead0 && sum == mod && cur == 0;
    int& now = f[pos][cur][sum];
    if (!lead0 && !lim && ~now)return now;
    int up = lim ? a[pos] : 9, res = 0;
    for (int i = 0;i <= up;++i)
        res += dfs(pos - 1, sum + i, (cur * 10 + i) % mod, lead0 && !i, lim && i ==
up);
    if (!lead0 && !lim)now = res;
    return res;
}

signed main() {
    ios::sync_with_stdio(false);
```

```
    cin.tie(0), cout.tie(0);
    int n;cin >> n;
    int len = 0;
    while (n)a[++len] = n % 10, n /= 10;
    int res = 0;
    for (int i = 1;i <= len * 9;++i) {
        mod = i;memset(f, -1, sizeof f);
        res += dfs(len, 0, 0, 1, 1);
    }
    cout << res;
    return 0;
}
```

状压 DP

题意：在 $n * n$ 的棋盘里面放 k 个国王，使他们互不攻击，共有多少种摆放方案。国王能攻击到它上下左右，以及左上左下右上右下八个方向上附近的各一个格子，共8个格子。

```
#include <bits/stdc++.h>
using namespace std;
#define LL long long
const int N = 15, M = 150, K = 1500;
LL n, k;
LL cnt[K];    //每个状态的二进制中 1 的数量
LL tot;    //合法状态的数量
LL st[K];    //合法的状态
LL dp[N][M][K];    //第 i 行，放置了 j 个国王，状态为 k 的方案数
int main(){
    ios::sync_with_stdio(false);cin.tie(0);
    cin >> n >> k;
    for (int s = 0; s < (1 << n); s ++ ){  //找出合法状态
        LL sum = 0, t = s;
        while(t){  //计算 1 的数量
            sum += (t & 1);
            t >>= 1;
        }
        cnt[s] = sum;
        if ( (( (s << 1) | (s >> 1) ) & s) == 0 ){  //判断合法性
            st[ ++ tot] = s;
        }
    }
    dp[0][0][0] = 1;
    for (int i = 1; i <= n + 1; i ++ ){
        for (int j1 = 1; j1 <= tot; j1 ++ ){    //当前的状态
            LL s1 = st[j1];
            for (int j2 = 1; j2 <= tot; j2 ++ ){    //上一行的状态
                LL s2 = st[j2];
                if ( ( (s2 | (s2 << 1) | (s2 >> 1)) & s1 ) == 0 ){
                    for (int j = 0; j <= k; j ++ ){
                        if (j - cnt[s1] >= 0)
                            dp[i][j][s1] += dp[i - 1][j - cnt[s1]][s2];
                    }
```

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

最短Hamilton路径

```
using namespace std;

const int N = 20,M = 1 << N;

int n;
int w[N][N];
int f[M][N];//第一维表示是否访问到该点的压缩状态，第二维是走到点j
            //f[i][j]表示状态为i并且到j的最短路径

int main(){
    cin>>n;
    for (int i = 0; i < n; i ++ )
        for (int j = 0; j < n; j ++ )//读入i到j的距离
            cin>>w[i][j];
    memset(f, 0x3f, sizeof f);
    f[1][0]=0;
    for (int i = 0; i < 1 << n; i ++ )//枚举压缩的状态
        for (int j = 0; j < n; j ++ )//枚举到0~j的点
            if(i >> j & 1)//该状态存在j点
                for (int k = 0; k < n; k ++ )//枚举从j倒数第二个点k
                    if(i >> k & 1)//倒数点k存在
                        f[i][j]=min(f[i][j],f[i-(1<<j)][k]+w[k][j]);//状态转移方程，在f[i][j]和状态去掉j的点f[i-(i<<j)][k]+w[k][j]取最小值
    cout<<f[(1<<n)-1][n-1]<<endl;//输出状态全满也就是所有点都经过且到最后一个点的最短距离
    return 0;
}
```

状态转移方程：

```
f[i][j]=min(f[i][j],f[i-(1<<j)][k]+w[k][j]);
```

常用例题

题意：在一篇文章（包含大小写英文字母、数字、和空白字符（制表/空格/回车））中寻找 **helloworld**（任意一个字母的大小写都行）的子序列出现了多少次，输出结果对 10^9+7 的余数。

字符串 DP ，构建一个二维 DP 数组，$dp[i][j]$ 的 i 表示文章中的第几个字符，j 表示寻找的字符串的第几个字符，当字符串中的字符和文章中的字符相同时，即找到符合条件的字符， `dp[i][j] = dp[i - 1][j] + dp[i - 1][j - 1]` ，因为字符串中的每个字符不会对后面的结果产生影响，所以 DP 方程可以优化成一维的， 由于字符串中有重复的字符，所以比较时应该从后往前。

```
#include <bits/stdc++.h>
using namespace std;
#define LL long long
const int mod = 1e9 + 7;
char c, s[20] = "!helloworld";
LL dp[20];
int main(){
    dp[0] = 1;
    while ((c = getchar()) != EOF)
        for (int i = 10; i >= 1; i--)
            if (c == s[i] || c == s[i] - 32)
                dp[i] = (dp[i] + dp[i - 1]) % mod;
    cout << dp[10] << "\n";
    return 0;
}
```

题意：（最长括号匹配）给一个只包含‘(’，‘)’，‘[’，‘]’的非空字符串，“()”和“[]”是匹配的，寻找字符串中最长的括号匹配的子串，若有两串长度相同，输出靠前的一串。

设给定的字符串为 s ，可以定义数组 $dp[i], dp[i]$ 表示以 $s[i]$ 结尾的字符串里最长的括号匹配的字符。显然，从 $i - dp[i] + 1$ 到 i 的字符串是括号匹配的，当找到一个字符是‘)’或‘]’时，再去判断第 $i - 1 - dp[i - 1]$ 的字符和第 i 位的字符是否匹配，如果是，那么 `dp[i] = dp[i - 1] + 2 + dp[i - 2 - dp[i - 1]]` 。

```
#include <bits/stdc++.h>
using namespace std;
const int maxn = 1e6 + 10;
string s;
int len, dp[maxn], ans, id;
int main(){
    cin >> s;
    len = s.length();
    for (int i = 1; i < len; i++){
        if ((s[i] == ')' && s[i - 1 - dp[i - 1]] == '(' ) || (s[i] == ']' && s[i - 1 - dp[i - 1]] == '[')){
            dp[i] = dp[i - 1] + 2 + dp[i - 2 - dp[i - 1]];
            if (dp[i] > ans) {
                ans = dp[i];  //记录长度
                id = i;  //记录位置
            }
        }
    }
    for (int i = id - ans + 1; i <= id; i++)
        cout << s[i];
    cout << "\n";
    return 0;
}
```

题意：去掉区间内包含“4”和“62”的数字，输出剩余的数字个数

```
int T,n,m,len,a[20];//a数组用于判断每一位能取到的最大值
```

```
ll l,r,dp[20][15];
ll dfs(int pos,int pre,int limit){//记搜
    //pos搜到的位置，pre前一位数
    //limit判断是否有最高位限制
    if(pos>len) return 1;//剪枝
    if(dp[pos][pre]!=-1 && !limit) return dp[pos][pre];//记录当前值
    ll ret=0;//暂时记录当前方案数
    int res=limit?a[len-pos+1]:9;//res当前位能取到的最大值
    for(int i=0;i<=res;i++)
        if(!(i==4 || (pre==6 && i==2)))
            ret+=dfs(pos+1,i,i==res&&limit);
    if(!limit) dp[pos][pre]=ret;//当前状态方案数记录
    return ret;
}
ll part(ll x){//把数按位拆分
    len=0;
    while(x) a[++len]=x%10,x/=10;
    memset(dp,-1,sizeof dp);//初始化-1（因为有可能某些情况下的方案数是0）
    return dfs(1,0,1);//进入记搜
}
int main(){
    cin>>n;
    while(n--){
        cin>>l>>r;
        if(l==0 && r==0)break;
        if(l) printf("%lld\n",part(r)-part(l-1));//[l,r](l!=0)
        else printf("%lld\n",part(r)-part(l));//从0开始要特判
    }
}
```

SOSdp 高维前缀和

子集向超集转移

```
for(int j = 0; j < n; j++)
    for(int i = 0; i < 1 << n; i++)
        if(i >> j & 1) f[i] += f[i ^ (1 << j)];
```

超集向子集转移

```
for(int j = 0; j < n; j++)
    for(int i = (1 << n) - 1; i >= 0 ; i--)
        if(!(i >> j & 1)) f[i] += f[i ^ (1 << j)]
```

汉明权重

```
for (int i = 0; (1<<i)-1 <= n; i++) {
    for (int x = (1<<i)-1, t; x <= n; t = x+(x&-x), x = x ? (t|((((t&-t)/(x&-x))>>1)-1)) : (n+1)) {
        // todo
    }
}
```

/END/