🗄️ 动态规划 (Dynamic Programming) 复习笔记合集

目录 (Table of Contents)

---

1. 背包问题

1.1 算法概述与选型指南

- **核心思想**：在容量限制下选择物品，使总价值最大。

算法/模型	适用场景	时间复杂度	典型例题
0/1背包	每个物品只能选一次	$O(NM)$	采药、装箱问题
完全背包	物品可无限选取	$O(NM)$	买书、货币系统
多重背包 (二进制)	物品有数量限制	$O(M \sum \log S)$	庆功会
多重背包 (单调队列)	数量多、卡常	$O(NM)$	多重背包 III
分组背包	每组只能选一个	$O(NM)$	机器分配
二维费用背包	两种限制条件	$O(NMV)$	潜水员
有依赖的背包	选子节点必须先选父节点	$O(NM^2)$	金明的预算方案
背包求方案数	统计最优解数量	$O(NM)$	数字组合
背包求具体方案	输出具体选择方案	$O(NM)$	背包问题求具体方案

1.2 核心模板 (Core Templates)

变量定义：

- `f[j]`：容量为 j 时的最大价值
- `v[i]`：第 i 个物品体积，`w[i]`：第 i 个物品价值

0/1背包 (一维优化)：

```
// 逆序遍历防止重复选取
for (int i = 1; i <= n; i++)
    for (int j = m; j >= v[i]; j--)
        f[j] = max(f[j], f[j - v[i]] + w[i]);
```

完全背包 (一维优化):

```
// 正序遍历允许重复选取
for (int i = 1; i <= n; i++)
    for (int j = v[i]; j <= m; j++)
        f[j] = max(f[j], f[j - v[i]] + w[i]);
```

多重背包 (二进制优化):

```
for (int i = 1; i <= n; i++) {
    int v, w, s;
    cin >> v >> w >> s;
    // 二进制拆分: 1,2,4,...,k
    for (int k = 1; k <= s; k <<= 1) {
        for (int j = m; j >= k * v; j--)
            f[j] = max(f[j], f[j - k * v] + k * w);
        s -= k;
    }
    if (s > 0)
        for (int j = m; j >= s * v; j--)
            f[j] = max(f[j], f[j - s * v] + s * w);
}
```

多重背包 (单调队列优化):

```
const int N = 2e5 + 10;
int g[N], f[N], q[N];

for (int i = 1; i <= n; i++) {
    int v, w, s;
    scanf("%d%d%d", &v, &w, &s);
    memcpy(g, f, sizeof f);
    for (int j = 0; j < v; j++) {  // 按余数分组
        int hh = 0, tt = -1;
        for (int k = j; k <= m; k += v) {
            // 出队:超出窗口范围
            if (hh <= tt && q[hh] < k - s * v) hh++;
            // 维护单调递减队列
            while (hh <= tt && g[q[tt]] - (q[tt]-j)/v*w <= g[k] - (k-j)/v*w) tt--;
            q[++tt] = k;
            // 状态转移: f[k] = max{g[t] + (k-t)/v*w}
```

```
            f[k] = g[q[hh]] + (k - q[hh]) / v * w;
        }
    }
}
```

分组背包:

```
for (int i = 1; i <= n; i++) {       // 枚举组
    int s; cin >> s;
    for (int j = 1; j <= s; j++)     // 读入组内物品
        cin >> v[j] >> w[j];
    for (int j = m; j >= 0; j--)     // 逆序枚举容量
        for (int k = 1; k <= s; k++) // 枚举组内选择
            if (j >= v[k])
                f[j] = max(f[j], f[j - v[k]] + w[k]);
}
```

有依赖的背包 (树形DP):

```
void dfs(int u) {
    for (int i = h[u]; ~i; i = ne[i]) {
        int son = e[i];
        dfs(son);
        // 背包合并:类似分组背包
        for (int j = m - v[u]; j >= 0; j--)
            for (int k = 0; k <= j; k++)
                f[u][j] = max(f[u][j], f[u][j - k] + f[son][k]);
    }
    // 放入父节点
    for (int i = m; i >= v[u]; i--)
        f[u][i] = f[u][i - v[u]] + w[u];
    for (int i = 0; i < v[u]; i++)
        f[u][i] = 0;  // 容量不够无法选
}
```

背包求方案数:

```
memset(f, -0x3f, sizeof f);
f[0] = 0; g[0] = 1;  // g[j]表示方案数

for (int i = 0; i < n; i++) {
    cin >> v >> w;
    for (int j = m; j >= v; j--) {
        int maxv = max(f[j], f[j - v] + w);
        int s = 0;
        if (maxv == f[j]) s = g[j];
        if (maxv == f[j - v] + w) s = (s + g[j - v]) % mod;
```

```
            f[j] = maxv, g[j] = s;
        }
    }
```

背包求具体方案:

```
// 倒序DP方便输出方案
for (int i = n; i >= 1; i--)
    for (int j = 0; j <= m; j++) {
        f[i][j] = f[i + 1][j];  // 不选i
        if (j >= v[i] && f[i][j] < f[i + 1][j - v[i]] + w[i])
            f[i][j] = f[i + 1][j - v[i]] + w[i];  // 选i
    }

// 输出方案
int j = m;
for (int i = 1; i <= n; i++)
    if (j >= v[i] && f[i][j] == f[i + 1][j - v[i]] + w[i]) {
        cout << i << " ";
        j -= v[i];
    }
```

1.3 进阶技巧与模型变形

- **初始化技巧**:
  - 恰好装满: `f[0]=0, 其余=INF(最小值) 或 -INF(最大值)`
  - 不要求装满: `f[0..m]=0`
- **体积/价值为负数**: 平移坐标或改变符号
- **求最小值**: `max` 改 `min`，初始化 `0x3f3f3f3f`
- **输出字典序最小方案**: 倒序DP + 正序选择
- **多重背包转0/1背包**: 二进制优化优于朴素拆分
- **可行性问题**（能否凑出体积 j）: `f[j]` 用 `bool` 类型

1.4 ⚠ 细节与致命坑点

坑点	说明
初始化值	`0x3f3f3f3f` (约 10^9) 用于最小值初始化; `-0x3f3f3f3f` 用于最大值初始化; 切勿用 `0x7f` 会导致溢出
逆序 vs 正序	0/1背包逆序，完全背包正序，搞混会导致重复选取或漏选
数组大小	二维背包需要 $N \times M$，注意内存是否超限
long long	价值总和可能超 `int`，特别是完全背包
模数处理	方案数取模时注意 `s = (s + g[j-v]) % mod`，防止负数用 `(s + mod) % mod`

坑点	说明
单调队列边界	`q[hh] < k - s * v` 出队条件，注意是 `<` 而非 `<=`

---

2. 状态机模型

2.1 算法概述与选型指南

- **核心思想**：将问题抽象为有限状态之间的转移，用DP描述状态转移过程。

算法/模型	适用场景	时间复杂度	典型例题
两状态模型	选/不选两种状态	$O(N)$	大盗阿福（相邻不能选）
股票买卖	有冷却期/手续费限制	$O(NK)$	股票买卖 IV/V
K状态模型	复杂状态转移	$O(NK)$	设计密码

2.2 核心模板 (Core Templates)

变量定义：

- `f[i][0]`：第 i 天不持有（或不选）的最大收益
- `f[i][1]`：第 i 天持有（或选）的最大收益

大盗阿福（相邻不能偷）：

```
const int N = 1e5 + 10, INF = 0x3f3f3f3f;
int f[N][2], w[N];

// f[i][0]: 前i个不选第i个的最大值
// f[i][1]: 前i个选第i个的最大值
f[0][0] = 0, f[0][1] = -INF;
for (int i = 1; i <= n; i++) {
    f[i][0] = max(f[i-1][0], f[i-1][1]);  // 不选i: 可以从选/不选转移
    f[i][1] = f[i-1][0] + w[i];            // 选i: 只能从不选转移
}
```

股票买卖 IV（最多K笔交易）：

```
const int M = 110, N = 1e5 + 10, INF = 0x3f3f3f3f;
int f[N][M][2];  // [天数][交易次数][是否持有]

memset(f, -0x3f, sizeof f);
for (int i = 0; i <= n; i++) f[i][0][0] = 0;  // 0次交易收益为0

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

```
        // 不持有: 昨天就不持有 或 昨天持有今天卖出
        f[i][j][0] = max(f[i-1][j][0], f[i-1][j][1] + w[i]);
        // 持有: 昨天就持有 或 昨天不持有今天买入(开启第j笔交易)
        f[i][j][1] = max(f[i-1][j][1], f[i-1][j-1][0] - w[i]);
    }

// 答案: max{f[n][j][0]} 最后一天不持有的最大值
```

股票买卖 V（含冷冻期）：

```
// 0: 不持有且不在冷冻期  1: 持有  2: 不持有且在冷冻期(昨天刚卖)
f[i][0] = max(f[i-1][0], f[i-1][2]);      // 昨天就不持有 或 冷冻期结束
f[i][1] = max(f[i-1][1], f[i-1][0] - w[i]); // 继续持有 或 买入
f[i][2] = f[i-1][1] + w[i];                // 昨天持有今天卖出
```

2.3 进阶技巧与模型变形

- **增加冷冻期**：增加一个"冷冻期"状态
- **增加手续费**：卖出时减去手续费
- **交易次数限制**：增加一维记录已用交易次数
- **必须卖出**：答案取 `f[n][0]` 而非 `max(f[n][0], f[n][1])`
- **无限交易**：去掉交易次数维度，`f[i][0/1]` 即可

2.4 ⚠ 细节与致命坑点

坑点	说明
负无穷初始化	不可能状态用 `-INF` 标记，防止错误转移
边界状态	`f[i][0][0] = 0` 表示0次交易不持有，收益为0
答案选取	股票问题答案通常是 `f[n][*][0]`（不持有状态）
状态定义清晰	务必明确 `f[i][j]` 的确切含义，避免状态混淆

---

3. 树形DP

3.1 算法概述与选型指南

- **核心思想**：在树上进行DFS，自底向上或自顶向下传递DP状态。

算法/模型	适用场景	时间复杂度	典型例题
树的最长路径	求树的直径	$O(N)$	树的最长路径
树的中心	求使最大距离最小的点	$O(N)$	树的中心
树形背包	选m个节点的最大值	$O(NM^2)$	二叉苹果树

算法/模型	适用场景	时间复杂度	典型例题
树的最小点覆盖	选择最少点覆盖所有边	$O(N)$	战略游戏、皇宫看守
换根DP	需要以每个点为根计算	$O(N)$	树的中心

3.2 核心模板 (Core Templates)

变量定义：

- `d1[u]`：从 u 向下的最长路径，`d2[u]`：从 u 向下的次长路径
- `p1[u]`：最长路径经过的子节点，`u1[u]`：从 u 向上的最长路径

树的最长路径（直径）：

```
int dfs(int u, int father) {
    int dist = 0, d1 = 0, d2 = 0;
    for (int i = h[u]; ~i; i = ne[i]) {
        int j = e[i];
        if (j == father) continue;
        int d = dfs(j, u) + w[i];
        dist = max(dist, d);
        // 维护最长和次长
        if (d >= d1) d2 = d1, d1 = d;
        else if (d > d2) d2 = d;
    }
    ans = max(ans, d1 + d2);  // 直径 = 最长 + 次长
    return dist;
}
```

换根DP（树的中心）：

```
// 第一遍DFS：计算向下的最长/次长路径
dfs_d(int u, int father) {
    d1[u] = d2[u] = -INF;
    for (int j : children) {
        int d = dfs_d(j, u) + w;
        if (d >= d1[u]) d2[u] = d1[u], d1[u] = d, p1[u] = j;
        else d2[u] = max(d2[u], d);
    }
    if (d1[u] == -INF) d1[u] = d2[u] = 0;  // 叶子节点
    return d1[u];
}

// 第二遍DFS：计算向上的最长路径
dfs_u(int u, int father) {
    for (int j : children) {
        if (p1[u] == j) u1[j] = max(u1[u], d2[u]) + w;  // j在最长链上,用次长更新
        else u1[j] = max(u1[u], d1[u]) + w;             // 用最长更新
        dfs_u(j, u);
```

```
    }
}

// 节点u的最大距离 = max(d1[u], u1[u])
```

树形背包（选m条边）：

```
void dfs(int u, int father) {
    for (int i = h[u]; ~i; i = ne[i]) {
        int p = e[i];
        if (p == father) continue;
        dfs(p, u);
        // 类似分组背包：子树p分配k条边
        for (int j = m; j >= 0; j--)
            for (int k = 0; k < j; k++)  // k条边给子树p
                f[u][j] = max(f[u][j], f[u][j - k - 1] + f[p][k] + w[i]);
    }
}
```

3.3 进阶技巧与模型变形

- **换根DP**：两次DFS，第一次自底向上，第二次自顶向下
- **树的重心**：以子树大小为状态
- **树的点分治**：处理带权路径问题
- **树上依赖背包**：选子节点必须先选父节点
- **二分 + 树形DP**：如求使最大值最小的边权
- **基环树DP**：断环成树后处理

3.4 ⚠ 细节与致命坑点

坑点	说明
链式前向星初始化	`memset(h, -1, sizeof h)`，`idx = 0`
无向边存两遍	`add(a,b,c)`，`add(b,a,c)`，DFS时注意判父节点
叶子节点判断	`d1[u] == -INF` 表示叶子，需初始化 `d1[u]=d2[u]=0`
背包遍历顺序	容量逆序，类似0/1背包防止重复
数组大小	边数 = 2*(n-1)，数组开 `2*N`
INF设置	树边权可能为负，用 `-INF` 而非 `-1`

---

4. 区间DP

4.1 算法概述与选型指南

- **核心思想**：将大问题分解为小区间，通过合并小区间求解大区间。

算法/模型	适用场景	时间复杂度	典型例题
石子合并	相邻合并，求最小代价	$O(N^3)$	环形石子合并
最优二叉搜索树	构建最优树结构	$O(N^3)$	加分二叉树
凸多边形划分	三角形划分	$O(N^3)$	凸多边形的划分
棋盘分割	二维区间DP	$O(N^5)$	棋盘分割
环形处理	破环成链	$O(N^3)$	能量项链

4.2 核心模板 (Core Templates)

变量定义:

- `f[l][r]`：合并区间 $[l,r]$ 的最小/最大代价
- `s[i]`：前缀和，用于快速计算区间和

石子合并（朴素版）：

```
// f[l][r] = min(f[l][k] + f[k+1][r] + s[r]-s[l-1])
memset(f, 0x3f, sizeof f);
for (int i = 1; i <= n; i++) f[i][i] = 0;

for (int len = 2; len <= n; len++)       // 区间长度
    for (int l = 1; l + len - 1 <= n; l++) {  // 左端点
        int r = l + len - 1;              // 右端点
        for (int k = l; k < r; k++)       // 枚举分割点
            f[l][r] = min(f[l][r], f[l][k] + f[k+1][r] + s[r] - s[l-1]);
    }
```

石子合并（记忆化搜索版）：

```
int dfs(int l, int r) {
    if (f[l][r] != INF) return f[l][r];
    if (l == r) return f[l][r] = 0;
    for (int k = l; k < r; k++)
        f[l][r] = min(f[l][r], dfs(l, k) + dfs(k+1, r) + s[r] - s[l-1]);
    return f[l][r];
}
```

环形石子合并（破环成链）：

```
// 复制一倍: w[i+n] = w[i]
for (int i = 1; i <= n; i++) w[i+n] = w[i];
for (int i = 1; i <= 2*n; i++) s[i] = s[i-1] + w[i];

memset(f, 0x3f, sizeof f);
```

```
memset(g, -0x3f, sizeof g);

// 枚举每个起点,长度为n的区间
int minv = INF, maxv = -INF;
for (int i = 1; i <= n; i++) {
    minv = min(minv, dfs_min(i, i+n-1));
    maxv = max(maxv, dfs_max(i, i+n-1));
}
```

4.3 进阶技巧与模型变形

- **四边形不等式优化**：将 $O(N^3)$ 优化到 $O(N^2)$，适用于满足四边形不等式的DP
- **平行四边形优化**：决策单调性优化
- **破环成链**：环形问题经典处理技巧
- **Garsia-Wadsworth算法**：$O(N^2)$ 解决石子合并（仅需最小值）
- **高精度**：石子合并结果可能很大，需要高精度

4.4 ⚠ 细节与致命坑点

坑点	说明
区间长度从2开始	`len = 1` 时 `f[i][i]=0` 已初始化，从2开始枚举
前缀和下标	`s[r] - s[l-1]` 注意是 `l-1` 不是 `l`
环形数组大小	开 `2*N`，否则访问 `i+n` 越界
同时求最大最小	最大值初始化 `-0x3f`，最小值初始化 `0x3f`
区间DP与矩阵链乘	注意状态定义区别，矩阵链乘是 `f[i][k] + f[k+1][j] + p[i-1]*p[k]*p[j]`

---

5. 数位DP

5.1 算法概述与选型指南

- **核心思想**：按位处理数字，用DFS/DP统计满足条件的数字个数。

算法/模型	适用场景	时间复杂度	典型例题
基础数位DP	统计区间满足条件的数	$O(\log N \times state)$	度的数量
Windy数	相邻位差值限制	$O(\log N \times 10 \times 2)$	Windy数
不要62	特定数字限制	$O(\log N \times 2)$	不要62
数字游戏II	数位和限制	$O(\log N \times 100)$	数字游戏II

5.2 核心模板 (Core Templates)

变量定义：

- `f[i][j]`：从第 i 位开始，前一位/某种状态为 j 的方案数

- `nums`：存储数字各位的vector

通用数位DP框架：

```
int dp(int n) {
    if (!n) return 0;

    vector<int> nums;
    while (n) {
        nums.push_back(n % 10);
        n /= 10;
    }

    int res = 0, last = 0;  // last: 前面已经确定的部分
    for (int i = nums.size() - 1; i >= 0; i--) {
        int x = nums[i];
        // 枚举这一位填 0 ~ x-1 的方案数（直接从f数组获取）
        for (int j = 0; j < x; j++) {
            if (check(last, j))  // 检查是否满足条件
                res += f[i][get_state(last, j)];
        }

        // 如果x本身不满足条件,后面也不用继续了
        if (!check(last, x)) break;

        // 继续处理下一位
        last = update(last, x);

        // 最后一位且全部满足条件
        if (!i && final_check(last)) res++;
    }

    return res;
}

// 答案 = dp(R) - dp(L-1)
```

度的数量（K个1的B进制数）：

```
// f[i][j]: i位中有j个1的方案数（组合数）
void init() {
    for (int i = 0; i < N; i++) f[i][0] = 1;
    for (int i = 1; i < N; i++)
        for (int j = 1; j <= i; j++)
            f[i][j] = f[i-1][j] + f[i-1][j-1];  // C(i,j)
}

int dp(int n) {
    if (!n) return 0;
    vector<int> nums;
```

```
    while (n) { nums.push_back(n % B); n /= B; }

    int res = 0, cnt = 0;  // cnt:已经用了多少个1
    for (int i = nums.size() - 1; i >= 0; i--) {
        int x = nums[i];
        if (x) {
            res += f[i][K - cnt];       // 这一位填0
            if (x > 1) {
                if (K - cnt - 1 >= 0)
                    res += f[i][K - cnt - 1];  // 这一位填1
                break;  // 填>1的数,后面无论怎么填都不合法
            } else {
                cnt++;  // x==1,继续下一位
                if (cnt > K) break;
            }
        }
        if (!i && cnt == K) res++;  // 到最后一位且恰好K个1
    }
    return res;
}
```

5.3 进阶技巧与模型变形

- **记忆化搜索写法**：更通用，易于处理复杂限制
- **前导零处理**：单独标记 `lead` 状态
- **上一位限制**：如Windy数需要知道上一位数字
- **数位和限制**：增加一维记录数位和模值
- **连续段限制**：如不能有连续3个相同数字
- **区间两端点**：答案 = `dp(R) - dp(L-1)`，注意 `L=0` 的情况

5.4 ⚠ 细节与致命坑点

坑点	说明
L=0的情况	`dp(L-1)` 当 `L=0` 时需特判，避免负数
前导零	如统计0的个数时，前导零不应计入
枚举上限	非最高位时上限是9，最高位是 `nums[i]`
状态设计	状态数不能太大，否则记忆化会MLE/TLE
** long long**	答案可能很大，用 `long long`
记忆化初始化	`memset(f, -1, sizeof f)`，区分是否计算过

---

6. 状态压缩DP

6.1 算法概述与选型指南

- **核心思想**：用二进制数表示集合状态，状态转移通过位运算实现。

算法/模型	适用场景	时间复杂度	典型例题
棋盘放置	N皇后变种、不相邻放置	$O(N \times 3^N)$	小国王、玉米田
三行限制	当前行状态与上两行相关	$O(N \times 4^N)$	炮兵阵地
集合覆盖	最小覆盖所有点	$O(N \times 2^N)$	愤怒的小鸟
旅行商问题 (TSP)	最短哈密顿路径	$O(N^2 \times 2^N)$	宝藏

6.2 核心模板 (Core Templates)

变量定义：

- `state`：用 `vector<int>` 存储所有合法状态
- `f[i][j]`：第 i 行状态为 `state[j]` 时的最优值

基础模板（当前行只与上一行相关）：

```
const int N = 12, M = 1 << 10;
vector<int> state;
int cnt[M];
vector<int> head[M];

// 检查状态是否合法：不存在相邻的1
bool check(int state) {
    for (int i = 0; i < n; i++)
        if ((state >> i & 1) && (state >> (i+1) & 1))
            return false;
    return true;
}

// 统计状态中1的个数
int count(int state) {
    int res = 0;
    for (int i = 0; i < n; i++) res += state >> i & 1;
    return res;
}

int main() {
    // 预处理所有合法状态
    for (int i = 0; i < (1 << n); i++)
        if (check(i)) {
            state.push_back(i);
            cnt[i] = count(i);
        }

    // 预处理状态间转移：a可以转移到b
    for (int i = 0; i < state.size(); i++)
        for (int j = 0; j < state.size(); j++) {
            int a = state[i], b = state[j];
            if ((a & b) == 0 && check(a | b))  // 不相交且无相邻
                head[i].push_back(j);
```

```
        }

    // DP转移
    f[0][0] = 1;
    for (int i = 1; i <= n + 1; i++)
        for (int j = 0; j <= m; j++)
            for (int a = 0; a < state.size(); a++)
                for (int b : head[a]) {
                    int c = cnt[state[a]];
                    if (j >= c)
                        f[i][j][a] += f[i-1][j-c][b];
                }
}
```

炮兵阵地（与上两行相关，滚动数组优化）：

```
const int N = 110, M = 1 << 10;
int f[2][M][M];  // 只保留两行状态

// 三行检查：a(上上行), b(上一行), c(当前行)
if ((a & b) | (a & c) | (b & c)) continue;
if (g[i] & b) continue;  // 山地不能放

f[i & 1][j][k] = max(f[i & 1][j][k],
                     f[(i-1) & 1][u][j] + s[b]);
```

6.3 进阶技巧与模型变形

- **预处理合法状态**：避免运行时判断，加快速度
- **滚动数组**：空间优化，如 `f[2][M]` 代替 `f[N][M]`
- **位运算技巧**：
  - `state >> i & 1`：取第 i 位
  - `state | (1 << i)`：第 i 位置1
  - `state & ~(1 << i)`：第 i 位置0
  - `state & -state`：取最低位的1
  - `__builtin_popcount(x)`：统计1的个数（GCC内置）
- **轮廓线DP**：处理二维网格的另一种状态压缩方式
- **子集枚举**：`for (int s = state; s; s = (s-1) & state)`

6.4 ⚠ 细节与致命坑点

坑点	说明
状态数限制	$N \leq 20$ 时 $2^{20} \approx 10^6$ 可接受，再大需优化
数组大小	`f[N][K][M]` 可能MLE，注意用滚动数组或 `vector`
状态预处理	`check(a | b)` 检查两行合并后是否合法

坑点	说明
位运算优先级	`state >> i + 1` 是错误的，应写 `(state >> i) + 1` 或 `state >> (i+1)`
地图限制	如山地不能放置，用 `g[i] & state` 判断
初始化	`f[0][0] = 1` 或 `0`，视题目而定

---

7. 最长上升子序列模型 (LIS)

7.1 算法概述与选型指南

- **核心思想**：求序列中最长的严格/非严格递增子序列。

算法/模型	适用场景	时间复杂度	典型例题
朴素LIS	$N \leq 10^3$	$O(N^2)$	拦截导弹（入门）
二分优化LIS	$N \leq 10^5$	$O(N \log N)$	拦截导弹、友好城市
最长下降子序列 (LDS)	改为 `a[i] >= a[j]`	$O(N \log N)$	登山
双向LIS	先增后减、山峰型	$O(N \log N)$	合唱队形、怪盗基德
最长公共上升子序列 (LCIS)	两个序列的公共上升	$O(N^2)$	最长公共上升子序列
拦截导弹（贪心动规）	求最少下降序列数 = LIS长度	$O(N \log N)$	Dilworth定理应用

7.2 核心模板 (Core Templates)

变量定义：

- `f[i]`：以 i 结尾的LIS长度
- `q[i]`：长度为 i 的LIS的末尾最小值

朴素 $O(N^2)$ 做法：

```
for (int i = 1; i <= n; i++) {
    f[i] = 1;  // 至少包含自己
    for (int j = 1; j < i; j++)
        if (a[i] > a[j])  // 严格递增
            f[i] = max(f[i], f[j] + 1);
    res = max(res, f[i]);
}
```

二分优化 $O(N \log N)$：

```
int cnt = 0;
for (int i = 1; i <= n; i++) {
    // 找第一个 >= a[i] 的位置（严格递增用lower_bound）
    int k = lower_bound(q, q + cnt, a[i]) - q;
```

```
    if (k == cnt) q[cnt++] = a[i];  // 可以接在后面
    else q[k] = a[i];                // 替换,使末尾更小
}
// 答案 = cnt

// 非严格递增(允许相等): 用 upper_bound
int k = upper_bound(q, q + cnt, a[i]) - q;
```

最长公共上升子序列 (LCIS):

```
// f[i][j]: a的前i个和b的前j个,且以b[j]结尾的LCIS
for (int i = 1; i <= n; i++) {
    int maxv = 1;  // 记录满足b[j] < a[i]的最大f[i-1][j]+1
    for (int j = 1; j <= n; j++) {
        f[i][j] = f[i-1][j];  // 不包含a[i]
        if (a[i] == b[j]) f[i][j] = max(f[i][j], maxv);
        if (b[j] < a[i]) maxv = max(maxv, f[i][j] + 1);
    }
}
```

双向LIS（先增后减）：

```
// f[i]: 以i结尾的LIS长度
// g[i]: 以i开头的LDS长度(倒序求LIS)
for (int i = 1; i <= n; i++)
    ans = max(ans, f[i] + g[i] - 1);  // -1是因为i被算了两次
```

7.3 进阶技巧与模型变形

- **输出具体方案**：记录每个位置的前驱，倒序回溯
- **LIS计数**：记录每个长度的方案数
- **二维偏序LIS**：先按一维排序，再对第二维求LIS
- **Dilworth定理**：最小链覆盖 = 最长反链长度；最少下降序列数 = LIS长度
- **树状数组优化LIS**：处理带修改或在线查询
- **CDQ分治优化**：三维偏序问题

7.4 ⚠ 细节与致命坑点

坑点	说明
严格 vs 非严格	`lower_bound` (>=) 用于严格递增，`upper_bound` (>) 用于非严格
最长 vs 最短	最少下降序列数 = 最长上升子序列长度 (Dilworth定理)
双向LIS边界	`f[i] + g[i] - 1`，注意峰值被计算两次
LCIS优化	朴素 $O(N^3)$ 可用 `maxv` 优化到 $O(N^2)$

坑点	说明
二分查找范围	`lower_bound(q, q+cnt, a[i])`，注意是 `q+cnt` 不是 `q+n`

---

8. 单调队列优化DP

8.1 算法概述与选型指南

- **核心思想**：维护一个单调的 deque，将 $O(N^2)$ 的区间最值查询优化到 $O(N)$。

算法/模型	适用场景	时间复杂度	典型例题
滑动窗口最值	固定长度窗口的最大/最小值	$O(N)$	滑动窗口
最大子序和	长度不超过M的最大连续子数组和	$O(N)$	最大子序和
多重背包优化	同余类内的单调队列	$O(NM)$	多重背包 III
修剪草坪	不能连续选超过K个	$O(N)$	修剪草坪
二维单调队列	二维滑动窗口	$O(NM)$	理想的正方形

8.2 核心模板 (Core Templates)

变量定义：

- `q[N]`：单调队列，存储下标
- `hh`：队头，`tt`：队尾
- `s[N]`：前缀和数组

滑动窗口最大值：

```
int hh = 0, tt = -1;
for (int i = 0; i < n; i++) {
    // 出队:窗口左边界为i-k+1,超出范围的出队
    if (hh <= tt && q[hh] < i - k + 1) hh++;

    // 维护单调递减队列(队头最大)
    while (hh <= tt && a[q[tt]] <= a[i]) tt--;

    q[++tt] = i;

    // 窗口形成后输出答案
    if (i >= k - 1) printf("%d ", a[q[hh]]);
}
```

最大子序和（长度不超过M）：

```
// 求 s[i] - min(s[j]) , i-j <= m
int hh = 0, tt = 0;  // 前缀和数组,维护单调递增队列(队头最小)
```

```
q[0] = 0;  // s[0]入队

for (int i = 1; i <= n; i++) {
    if (i - q[hh] > m) hh++;  // 超出长度限制
    res = max(res, s[i] - s[q[hh]]);  // 当前最大 = s[i] - 最小前缀和
    while (hh <= tt && s[i] <= s[q[tt]]) tt--;  // 维护递增
    q[++tt] = i;
}
```

修剪草坪（不能连续选超过K个）：

```
// f[i] = max(f[i-1], f[j] + s[i] - s[j+1]) , i-j-1 <= m
// 即 f[i] = max(f[i-1], s[i] + max(f[j] - s[j+1]))
int hh = 0, tt = 0;
q[0] = 0;  // 0号位置入队
f[0] = 0;

for (int i = 1; i <= n; i++) {
    if (q[hh] < i - m - 1) hh++;  // 出队
    f[i] = max(f[i-1], f[q[hh]] + s[i] - s[q[hh] + 1]);
    // 维护单调递减队列
    while (hh <= tt && f[q[tt]] - s[q[tt] + 1] <= f[i] - s[i + 1]) tt--;
    q[++tt] = i;
}
```

理想的正方形（二维单调队列）：

```
// 先对每行做滑动窗口,再对每列做滑动窗口
// 或两次单调队列:先水平后垂直
```

8.3 进阶技巧与模型变形

- **单调队列 vs 单调栈**：队列两端都可操作，栈只能在一端
- **双端队列维护**：`hh` 维护队头，`tt` 维护队尾
- **多重背包的同余类**：按 `j % v` 分组，每组独立做单调队列
- **斜率优化的预处理**：某些斜率优化可转化为单调队列
- **二维单调队列**：先行后列或先列后行两次处理

8.4 ⚠ 细节与致命坑点

坑点	说明
队列初始化	`hh = 0, tt = -1` 表示空队列；`hh = 0, tt = 0` 表示有一个元素
出队条件	是 `<` 还是 `<=`，是 `i-k` 还是 `i-k+1` 需仔细分析
入队时机	先计算答案再入队 或 先入队再计算，视题目而定

坑点	说明
前缀和起点	`s[0] = 0` 别忘了入队
队列为空判断	访问 `q[hh]` 前确保 `hh <= tt`
答案区间	修剪草坪类问题答案可能在 `f[n-m...n]` 中取最小值

---

9. 斜率优化DP

9.1 算法概述与选型指南

- **核心思想**：将DP转移方程转化为直线形式，用凸包/单调队列维护决策点。

算法/模型	适用场景	时间复杂度	典型例题
基础斜率优化	转移方程含 $i \times j$ 项	$O(N)$ 或 $O(N \log N)$	任务安排1/2/3
运输小猫	带权距离和最小化	$O(N)$	运输小猫
单调队列维护凸包	斜率单调时	$O(N)$	任务安排2
二分/李超树	斜率不单调	$O(N \log N)$	任务安排3

9.2 核心模板 (Core Templates)

变量定义：

- `f[i]`：前 i 个任务的最小费用
- `s[i]`, `c[i]`：时间和费用的前缀和
- `q[N]`：单调队列，存储决策点下标

任务安排（基础版 $O(N^2)$）：

```
// f[i] = min{f[j] + (c[i]-c[j])*t[i] + s*(c[n]-c[j])}
// 展开: f[i] = min{f[j] - (s+t[i])*c[j]} + c[i]*t[i] + s*c[n]
// 令 yj = f[j], xj = c[j]
// 即最小化: yj - k * xj, 其中 k = s + t[i]

memset(f, 0x3f, sizeof f);
f[0] = 0;
for (int i = 1; i <= n; i++)
    for (int j = 0; j < i; j++)
        f[i] = min(f[i], f[j] + (c[i]-c[j])*t[i] + s*(c[n]-c[j]));
```

斜率优化（单调队列 $O(N)$）：

```
const int N = 3e5 + 10;
typedef long long ll;
ll f[N], st[N], sc[N];
```

```
int q[N], s, n;

// 判断队头是否最优: (y2-y1)/(x2-x1) <= k
// 即 (y2-y1) <= k*(x2-x1)
int main() {
    scanf("%d%d", &n, &s);
    for (int i = 1; i <= n; i++) {
        scanf("%lld%lld", &st[i], &sc[i]);
        st[i] += st[i-1];
        sc[i] += sc[i-1];
    }

    int hh = 0, tt = 0;
    q[0] = 0;

    for (int i = 1; i <= n; i++) {
        ll k = s + st[i];  // 当前斜率

        // 队头出队: 下一条直线更优
        while (hh < tt) {
            ll x1 = sc[q[hh]], y1 = f[q[hh]];
            ll x2 = sc[q[hh+1]], y2 = f[q[hh+1]];
            if ((__int128)(y2-y1) <= (__int128)(x2-x1)*k) hh++;
            else break;
        }

        int j = q[hh];
        f[i] = f[j] + s*(sc[n]-sc[j]) + (sc[i]-sc[j])*st[i];

        // 新点入队: 维护下凸包
        ll x = sc[i], y = f[i];
        while (hh < tt) {
            ll x1 = sc[q[tt]], y1 = f[q[tt]];
            ll x2 = sc[q[tt-1]], y2 = f[q[tt-1]];
            // 检查新点是否使q[tt]成为冗余点
            if ((y-y1)*(x1-x2) <= (__int128)(y1-y2)*(x-x1)) tt--;
            else break;
        }
        q[++tt] = i;
    }

    printf("%lld", f[n]);
}
```

关键推导:

```
f[i] = min{f[j] + a[i]*b[j] + c[i] + d[j]}
     = min{f[j] + d[j] - a[i]*(-b[j])} + c[i]

令: yj = f[j] + d[j]
    xj = -b[j]
    k = a[i]
```

```
则: f[i] = min{yj - k*xj} + c[i]

即找一点 (xj, yj) 使 yj - k*xj 最小
= 过 (xj, yj) 斜率为 k 的直线的 y轴截距最小
```

9.3 进阶技巧与模型变形

- **斜率单调性判断**:
  - `st[i]` 递增 → 斜率 `k` 递增 → 单调队列
  - `st[i]` 不单调 → 需要二分或李超树
- **上凸包 vs 下凸包**:
  - 最小化问题维护下凸包
  - 最大化问题维护上凸包
- **__int128防溢出**: 乘法可能超 `long long`，比较时用 `__int128`
- **李超线段树**: 斜率不单调时的通用解法 $O(N \log N)$
- **CDQ分治**: 三维偏序中的斜率优化

9.4 ⚠ 细节与致命坑点

坑点	说明
__int128使用	比较 `(y2-y1)*(x3-x2) <= (y3-y2)*(x2-x1)` 时，乘积可能溢出，用 `__int128`
斜率相等处理	叉积相等时，保留更优的那个点（通常是最新或最旧的）
初始化队列	`q[0] = 0` 表示0号决策点入队，不能忘
队头出队条件	是 `<=` 还是 `<` 取决于是否允许斜率相等时出队
横坐标相等	`x[i]` 可能相等，此时按 `y` 排序，避免除0
long long	前缀和、DP值都要用 `long long`

---

10. 数字三角形模型

10.1 算法概述与选型指南

- **核心思想**: 在二维网格上从起点到终点，每次只能向下/右移动，求最大/最小路径和。

算法/模型	适用场景	时间复杂度	典型例题
基础数字三角形	只能向下/右下走	$O(N^2)$	摘花生
方格取数	两条路径同时走	$O(N^3)$	方格取数、传纸条
滚动数组优化	空间优化	$O(N)$	最低通行费
多源汇最短路	变种问题	$O(NM)$	复杂变种

10.2 核心模板 (Core Templates)

变量定义:

- `f[i][j]`: 到达 (i,j) 的最大/最小值
- `w[i][j]`: 格子 (i,j) 的权值

基础数字三角形（摘花生）:

```
const int N = 110;
int f[N], w[N][N];

for (int i = 1; i <= r; i++)
    for (int j = 1; j <= c; j++)
        scanf("%d", &w[i][j]);

// f[j] = max(从左边来, 从上边来) + w[i][j]
// 注意j要正序遍历(从左边来需要当前行已更新)
for (int i = 1; i <= r; i++)
    for (int j = 1; j <= c; j++)
        f[j] = max(f[j], f[j-1]) + w[i][j];

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

方格取数（两条路径）:

```
// f[k][i1][i2]: 走了k步,第一条在(i1, k-i1),第二条在(i2, k-i2)
// 状态数 O((n+m)*n^2) = O(N^3)

for (int k = 2; k <= n + m; k++)  // 总步数
    for (int i1 = max(1, k-m); i1 <= min(n, k-1); i1++)
        for (int i2 = max(1, k-m); i2 <= min(n, k-1); i2++) {
            int j1 = k - i1, j2 = k - i2;
            int t = w[i1][j1];
            if (i1 != i2) t += w[i2][j2];  // 不在同一格

            // 四种转移: 下下, 下右, 右下, 右右
            int &x = f[k][i1][i2];
            x = max(x, f[k-1][i1-1][i2-1] + t);  // 右右
            x = max(x, f[k-1][i1-1][i2] + t);    // 右下
            x = max(x, f[k-1][i1][i2-1] + t);    // 下右
            x = max(x, f[k-1][i1][i2] + t);      // 下下
        }
```

滚动数组空间优化:

```
// 只保留上一行
for (int i = 1; i <= r; i++)
    for (int j = 1; j <= c; j++)
```

```
        f[j] = max(f[j], f[j-1]) + w[i][j];
// 注意：j要正序,因为f[j-1]是当前行刚更新的
```

10.3 进阶技巧与模型变形

- **路径还原**：记录每个状态从哪转移来，倒序回溯
- **最小路径和**：`max` 改 `min`，初始化 `INF`
- **方格取数优化**：注意到 `k = i + j`，可降维
- **网格图最短路**：带权边可用Dijkstra代替DP
- **多路径扩展**：3条、4条路径，状态维度指数增长

10.4 ⚠ 细节与致命坑点

坑点	说明
遍历顺序	一维优化时 `j` 要正序（从左边转移需要当前行已更新）
数组初始化	`f[0] = 0`，其余 `f[j] = -INF`（最大值）或 `INF`（最小值）
边界判断	方格取数注意 `i1, i2` 的范围 `max(1, k-m)` 到 `min(n, k-1)`
同一格判断	`i1 == i2` 时只加一次权值
滚动数组覆盖	注意上一行数据在更新前需要先保存

附录：DP问题通用解题框架

1. 状态设计

- **第一步**：确定DP的维度（几维数组）
- **第二步**：明确定义 `f[i][j][...]` 的含义
- **第三步**：确定状态表示的**最后一步**是什么

2. 状态转移

- **最后一步的选择**：有哪些选择？
- **状态转移方程**：数学表达式
- **边界条件**：初始状态是什么？

3. 实现方式

- **递推（推荐）**：自底向上，避免递归栈溢出
- **记忆化搜索**：自顶向下，代码直观，适合复杂转移

4. 优化方向

- **空间优化**：滚动数组、状态压缩
- **时间优化**：单调队列、斜率优化、四边形不等式
- **剪枝**：无效状态不计算

5. 常见初始化模式

问题类型	初始化方式
最大值	`memset(f, -0x3f, sizeof f)`，需要的位置设为0
最小值	`memset(f, 0x3f, sizeof f)`，需要的位置设为0
计数	`f[0] = 1` 或 `f[0][0] = 1`，其余为0
可行性	`f[0] = true`，其余为 `false`

EOF