线性动态规划

dbywsc

2025/8

目录

先来看一个最简单的问题：
求斐波那契数列的第 $n(1 \leq 10^7)$ 项对 998244353 取模的结果。
众所周之，这个问题有很多种求解方法。

我们发现，求第 i 项，在它之前需要求 $i-1$ 项和 $i-2$ 项，可以很容易地画出递归树并写出代码：

```
int f(int x) {
    if(x <= 2) return 1;
    return f(x - 1) + f(x - 2);
}
```

然而，每次计算 $f(i)$，都会重复的计算 $f(i-1)$ 和 $f(i-2)$，因此时间复杂度非常高，在本题的数据规模下，它无法解决这个问题。

进一步地，既然每次计算都要用到 $f(i-1)$ 和 $f(i-2)$，那么我们可以在第一次计算到这个数的时候就将它们存下来，等到需要使用的时候直接拿出来调用，这样的操作叫做 **记忆化搜索** 。

```
i64 f(int x) {
    if(res[x]) return res[x] % 998244353;
    if(x < 2) {
        res[x] = 1;
        return res[x];
    }
    res[x] = (f(x - 1) + f(x - 2)) % 998244353;
    return res[x];
}
```

有了记忆化数组，每一项相当于只被计算了一次，因此时间复杂度被优化到了线性。

事实上，我们可以总结出斐波那契数列的递推公式，即 $f_i = f_{i-1} + f_{i-2}$ ，因此可以把记忆化搜索改写为一个循环的方式：

```
f[1] = f[2] = 1;
for(int i = 3; i <= n; i++) f[i] = (f[i - 1] + f[i - 2]) % 998244353;
```

显然，这样的做法也是线性的。

我们发现，解决这个问题的关键，就是如何快速的通过 f_{i-1} 和 f_{i-2} 计算出 f_i ，最快的方式显然是第三种方法，通过递推的关系计算出新的项。因此我们称，f_{i-1} 和 f_{i-2} 是已经得出的旧状态，f_i 是需要求解的新状态，由旧状态得出新状态的过程，就叫做 **状态转移** ，这种解决问题的思想，我们称为 **动态规划** 。

形式化的说：动态规划是一种，通过把原问题分解为相对简单的子问题的方式求解复杂问题的方法。

要想用动态规划解决问题，需要满足三个条件：最优子结构，无后效性和子问题重叠。

通过求斐波那契额数列的问题来解释一下：

最优子结构：想要求解 f_n ，就必须先求 f_{n-1} 和 f_{n-2} ，想要求 f_{n-2} 就要求 f_{n-3} 和 f_{n-4} …… 因此我们把问题变成了通过递推的方式，从前往后求每一项 f_i 。由于每一项 f_i 都是用这种方式求出来的，因此我们可以说每一步都是最优解。

无后效性：作为从前往后的递推，我们可以用 f_{i-1} 得到 f_i ，但是绝对不会用 f_{i+1} 得到 f_i 。

子问题重叠：每次求解 f_i 后，我们都将其存储到数组 $f[1..n]$ 中，避免重复的计算。

可以发现，我们从递归到记忆化，再到递推到过程，就是在逐步的满足这三个要求。事实上，由递归的从上而下，转变为从下而上的递推的过程，就是一种求解动态规划问题的方法。

解决动态规划问题，一般采用如下思路解决：

1. 将问题分解为若干个子问题。
2. 寻找不同状态之间的转移方式，即 **状态转移方程** 。
3. 按顺序求解每一个子问题。

由于斐波那契的递归公式非常简单且广为人知，因此它是一种常见的简单动态规划模型。

P1255 数楼梯

由于一次可以上一步或者上两步楼梯，因此，对于走到第 i 阶楼梯而言，它可以从前 $i-1$ 阶楼梯走上来，也可以从前 $i-2$ 阶楼梯走上来，那么走上来的方案数应该是走到 $i-1$ 阶的方案数加上走到 $i-2$ 阶楼梯的方案数，因此状态转移方程即为：

$$f_i = f_{i-1} + f_{i-2}$$

当然，我们还需要处理边界情况，在本题中，只有 $i=1$ 和 $i=2$ 时不能使用这个方程，当 $i=1$ 时，我们只有一种方案，当 $i=2$ 时，同样只有一种方案，所以边界设置为 $f_1 = f_2 = 1$ 。

P12833 [蓝桥杯 2025 国 B] 斐波那契字符串

记第 i 项中的逆序对数量为 f_i，观察可以发现：

第三项为 '01'，$f_3 = 0$；第四项为 '101'，$f_4 = 1$；

第五项为 '01101'，$f_5 = 2$；

第六项为 '10101101'，$f_6 = 7$ .....

对于第 i 项，它的逆序对数量一定包括 $i-1$ 项和 $i-2$ 项的数量，在 $i-2$ 和 $i-1$ 拼成新的字符串后，还会产生新的逆序对，具体来说，新的逆序对是由第 $i-2$ 项中 1 的数量和 $i-1$ 项中 0 的数量相乘得到的。因此还需要额外维护两个数组，即第 i 项中 0 的数量和 1 的数量，记做 $cnt0[1...n]$ 和 $cnt1[1...n]$，因此可以得到下列状态转移方程：

$$cnt0_i = cnt0_{i-1} + cnt0_{i-2} \qquad cnt1_i = cnt1_{i-1} + cnt1_{i-2}$$

$$f_i = f_{i-1} + f_{i-2} + cnt1_{i-2} \times cnt0_{i-1}$$

最后定义边界，显然有

$f_1 = f_2 = 0,\ cnt0_1 = 1,\ cnt0_2 = 0,\ cnt1_1 = 0,\ cnt1_2 = 1$

```
cnt0[1] = 1, cnt1[1] = 0;
cnt0[2] = 0, cnt1[2] = 1;
for(int i = 3; i <= N; i++) {
    f[i] = (f[i - 1] + f[i - 2] + cnt1[i - 2] * cnt0[i - 1]) % P;
    cnt0[i] = (cnt0[i - 1] + cnt0[i - 2]) % P;
    cnt1[i] = (cnt1[i - 1] + cnt1[i - 2]) % P;
}
```

P1115 最大子段和

由于子段必须是连续的一段，所以每次进行比较最大和时，要么当前的数字自成一段，要么当前的数字和它之前的一些连续的数字连起来作为一个子段，可以发现对于任何一个数字 i，只会有这两种情况，因此每次只需要取最大值就可以了。

设 f_i 为第 i 项的最大子段和，那么 f_i 应该是第 i 项的数字本身 a_i，和 f_i 前一项的最大子段和 $f_{i-1}+a_i$ 之间的最大值。因此状态转移方程为：

$$f_i = \max(f_{i-1} + a_i, a_i)$$

由于我们要取最大的一段子段和作为答案，那么答案应为

$$ans = \max_{i=1}^{n} f_i$$

对于边界，我们在求最大子段和时，可以做一次从前往后的遍历，因此边界设置为 $f_1 = a_1$ 即可。

```
f[1] = a[1];
for(int i = 2; i <= n; i++) {
    f[i] = std::max(a[i], f[i - 1] + a[i]);
    ans = std::max(ans, f[i]);
}
```

P2642 最大双子段和

相比起最大子段和，本题要求我们求两段不相交的最大子段和之和，并且两段子段之间至少相隔 1 个数。

对于这道题，直接求解比较困难，但是我们可以从前往后遍历一遍，维护一个最大前缀子段和 $pre[1...n]$ ，再从后往前遍历一遍，维护一个最大后缀子段和 $suf[1...n]$ 。最后再枚举分界点，分界点两边的前缀和后缀之和去最大值就为答案。

因此可以写出状态转移方程：

$$pre_i = \max(a_i, pre_{i-1} + a_i) \quad suf_i = \max(a_i, suf_{i+1} + a_i)$$

$$pre_i = \max(pre_i, pre_{i-1}) \quad suf_i = \max(suf_i, suf_{i+1})$$

$$ans = \max_{i=2}^{n-1}(pre_{i-1} + suf_{i+1})$$

关于边界，由于前缀是从前往后遍历，所以定义 $pre_1 = a_1$ ，相对的，由于后缀是从后往前遍历，所以定义 $suf_n = a_n$ 。

```
pre[1] = a[1], suf[n] = a[n];
for(int i = 2; i <= n; i++) pre[i] = std::max(pre[i - 1] + a[i], a[i]);
for(int i = 2; i <= n; i++) pre[i] = std::max(pre[i - 1], pre[i]);
for(int i = n - 1; i; i--) suf[i] = std::max(suf[i + 1] + a[i], a[i]);
for(int i = n - 1; i; i--) suf[i] = std::max(suf[i + 1], suf[i]);
for(int i = 2; i < n; i++) ans = std::max(ans, pre[i - 1] + suf[i + 1]);
```

B3637 最长上升子序列

与最大子段和不同的是，子段必须是连续的一段，而本题的子序列只要求对于原数组，相对位置不变即可。因此我们无法只用一次遍历得到结果。

设 f_i 为以 i 结尾的 LIS 的长度，考虑对于每一个 i，与它前面所有的项 $j = 1 \to i-1$ 进行比较，如果 $a_j < a_i$ 的话，就可以把 a_i 接到以 j 结尾的 LIS 的后面。接下来考虑边界设置，最短的情况下，每个数字自己都可以构成一个长度为 1 的上升子序列，因此初始时设置 $f_i = 1$。因此，状态转移方程应为：

$$f_i = \max_{1 \le j < i \text{ and } a_j < a_i} f_j + 1$$

由于最后要取所有上升子序列的最大长度，所以答案为：

$$ans = \max_{i=1}^{n} f_i$$

总时间复杂度 $O(n^2)$。

```cpp
for(int i = 1; i <= n; i++) {
    f[i] = 1;
    for(int j = 1; j < i; j++) {
        if(a[i] > a[j]) f[i] = std::max(f[i], f[j] + 1);
    }
    ans = std::max(ans, f[i]);
}
```

P2782 友好城市

画图后可以发现，本题实际上是对所有的城市按横坐标升序排序后求纵坐标的 LIS。

本题中 $1 \leq n \leq 2 \times 10^5$ ，由于朴素的 LIS 做法是 $O(n)$ 的，因此无法通过这道题。

考虑用贪心的做法优化。设 f 为数组 $a[1..n]$ 的 LIS，接下来，我们要尝试把 a 中的数字尽可能多的插入到 f 中。

显然，f 应该保持单调性，因此每次进行下面的操作：

如果当前的 a_i 比 f 的结尾更大，就将其插入 f 的末尾，序列长度 $+1$ ；

否则用它替换掉数列中比他大但是最小的数，单调性不变。

对于替换操作，可以使用 lower_bound() 将这个数查出来然后替换，因此总时间复杂度为 $O(n \log n)$ 。

```cpp
int n; std::cin >> n;
std::vector<PII> a(n);
std::vector<int> f;
for(int i = 0; i < n; i++) std::cin >> a[i].x >> a[i].y;
sort(a.begin(), a.end());
for(int i = 0; i < n; i++) {
    auto it = std::lower_bound(f.begin(), f.end(), a[i].y);
    if(it == f.end()) f.push_back(a[i].y);
    else *it = a[i].y;
}
std::cout << f.size();
```

P1439 【模板】最长公共子序列

本题类似于 LIS 问题，只不过是要求两个序列共有的子序列。此时仅设一维的状态已经无法满足题目的要求。不妨设 $f_{i,j}$ 为序列 a 的前 i 个，序列 b 的前 j 个中，最长公共子序列的长度。此时更新 $f_{i,j}$ 会有以下情况：

如果 a_i 和 a_j 相同，此时前 $i-1, j-1$ 构成的公共子序列就可以增加一位了，于是 $f_{i,j} = \max(f_{i,j}, f_{i-1,j-1}+1)$；

否则，考虑从 a 的前 $i-1$ 位，b 的前 j 位，或者 a 的前 i 位，b 的前 $j-1$ 位中去一个最大的转移到 $f_{i,j}$ 中，于是 $f_{i,j} = \max(f_{i-1,j}, f_{i,j-1})$。因此本题的状态转移方程为：

$$fi,j = \begin{cases} \max(f_{i,j}, f_{i-1,j-1}+1) & \text{if } a_i = b_j \\ \max(f_{i-1,j}, f_{i,j-1}) & \text{else} \end{cases}$$

由于每个 i 和 j 都需要一一对应，所以这种做法的时间复杂度是 $O(n^2)$。

```
for(int i = 1; i <= n; i++) {
    for(int j = 1; j <= n; j++) {
        if(a[i] == b[j]) f[i][j] = std::max(f[i][j], f[i - 1][j - 1] + 1);
        else f[i][j] = std::max(f[i - 1][j], f[i][j - 1]);
    }
}
std::cout << f[n][n];
```

在本题的数据规模中，上面的写法显然只能通过 50% 的数据。

考虑优化：
注意到输入的 a 和 b 是一个**排列**，因此这些数一定是 1 n 的数字。
最优情况下，$a = b$，LCS 就是一个 1 n 的排列，因此可以做一个 1 n 对 a_1 a_n 的映射。之后根据 b 中每个数字的出现的顺序构造出序列 $seq[1...n]$，使得 seq_i 为 b_i 在 a 中的位置。
此时，对 seq 求 LIS，LIS 的长度即为要求的 LCS 的长度。
由于将问题转化为了求解 LIS，因此状态可以仅设置为一维，同时可以使用 $n \log n$ 求 LIS 的方法，所以总时间复杂度也为 $O(n \log n)$。

正确性证明:
由于 a 与 b 是排列，每个元素出现次数唯一，所以 a 与 b 的 LCS 一定是他们中某些元素的一种顺序排列。
若有 LCS $x_{i_1}, x_{i_2}, ..., x_{i_k}$ ，在 a 中它的下标递增，对应到 seq 中，这些数出现的顺序也是递增的。此时 $seq_{i_1}, seq_{i_2}, ..., seq_{i_k}$ 刚好是一个 LCS 。
反过来说，若 $seq[1...n]$ 中存在一个 LCS $seq_{i_1} < seq_{i_2} < ... < seq_{i_k}$ ，那么对应到 a 中，$x_{i_1}, x_{i_2}, ...x_{i_k}$ 出现的顺序也是递增的。同时，在 b 中，$i_1, i_2, ..., i_k$ 也是递增的，因此它们相对位置不变，是一段 LCS 。

```cpp
int main(void) {
    int n; std::cin >> n;
    std::vector<int> a(n + 1), b(n + 1), seq(n + 1);
    for(int i = 1; i <= n; i++) {
        std::cin >> a[i];
        seq[a[i]] = i;
    }
    for(int i = 1; i <= n; i++) std::cin >> b[i];
    std::vector<int> f;
    for(int i = 1; i <= n; i++) {
        auto it = std::lower_bound(f.begin(), f.end(), seq[b[i]]);
        if(it == f.end()) f.push_back(seq[b[i]]);
        else *it = seq[b[i]];
    }
    std::cout << f.size();
    return 0;
}
```

在上面讲的所有的例子中，可以发现状态转移往往伴随着“递推的性质”，也就是说可以从前面的项得到后面的项。这种具有线性关系的动态规划统称为“线性状态动态规划”。
然而，动态规划往往没有完全相同的模版，而是需要通过自己观察和总结出状态转移方程。上面的几个模型属于线性动态规划的经典模型，接下来介绍一些别的题目。

P2842 纸币问题 1

设 f_i 为凑出金额 i 最少需要的纸币的数量。由于本题数据规模很小，我们可以枚举每一种面额，如果当钱的面额 $a_j \leq i$，那么 i 就可以通过凑一张当前面额转移过来，状态转移方程即为：

$$f_i = \min_{1 \leq j \leq n,\ a_j \leq i} (f_i, f_{i-a_j} + 1)$$

接下来考虑边界问题，由于每次要取最小值，所以 f_i 一开始要设定一个极大值，同时，金额 0 不需要任何纸币就可以凑出来，因此再设置 $f_0 = 0$。

```cpp
int main(void) {
    int n, w; std::cin >> n >> w;
    std::vector<int> a(n + 1), f(w + 1, 1e9);
    f[0] = 0;
    for(int i = 1; i <= n; i++) std::cin >> a[i];
    for(int i = 1; i <= w; i++) {
        for(int j = 1; j <= n; j++) {
            if(i >= a[j]) {
                f[i] = std::min(f[i], f[i - a[j]] + 1);
            }
        }
    }
    std::cout << f[w];
    return 0;
}
```

P3842 [TJOI2007] 线段

设 $f_{i,0}$ 为走完第 i 行最后留在左端点的最小步数；$f_{i,1}$ 为走完第 i 行最后留在右端点的最小步数。由于本题要求只能往下、往左、往右走，并且只有走完一条线段才能走下一条。因此走下一条线段时一定是从上一条的左端点或者右端点转移过来的。因此每次转移，应该取从上一条线的左端点或者右端点减去当前端点的距离，并且加上这一条线距离的最小值。状态转移方程为：

$$f_{i,0} = \min(f_{i-1,0} + abs(l_{i-1} - r_i),\ f_{i-1,1} + abs(r_{i-1} - r_i)) + r_i - l_i + 1$$

$$f_{i,1} = \min(f_{i-1,0} + abs(l_{i-1} - l_i),\ f_{i-1,1} + abs(r_{i-1} - l_i)) + r_i - l_i + 1$$

考虑边界，由于从 $(1,1)$ 出发，所以有：

$$f_{1,0} = r_1 + r_1 - l_1 - r_1 - 1$$

$$f_{1,1} = r_1 - 1$$

最后的答案就为处在最后一行时，在左端点和在右端点的距离的最小值。

```cpp
const int N = 2e4 + 10;
int main(void) {
    int n; std::cin >> n;
    std::vector<int> l(n + 1), r(n + 1);
    int f[N][2];
    for(int i = 1; i <= n; i++)  std::cin >> l[i] >> r[i];
    f[1][0] = r[1] + r[1] - l[1] - 1;
    f[1][1] = r[1] - 1;
    for(int i = 2; i <= n; i++) {
        f[i][0] = std::min(f[i - 1][0] + abs(l[i - 1] - r[i]),
                    f[i - 1][1] + abs(r[i - 1] - r[i])) + r[i] - l[i] + 1;
        f[i][1] = std::min(f[i - 1][0] + abs(l[i - 1] - l[i]),
                    f[i - 1][1] + abs(r[i - 1] - l[i])) + r[i] - l[i] + 1;
    }
    std::cout << std::min(f[n][0] + n - l[n], f[n][1] + n - r[n]);
    return 0;
}
```

动态规划题型多变，并且对于初学者来说往往晦涩难懂，想要学会动态规划唯一的方法就是大量的做题。

P1216 [IOI 1994] 数字三角形 Number Triangles
P2840 纸币问题 2
P2834 纸币问题 3
P1802 5 倍经验日
P1091 [NOIP 2004 提高组] 合唱队形
P1020 [NOIP 1999 提高组] 导弹拦截
P1544 三倍经验
P1004 [NOIP 2000 提高组] 方格取数
P1435 [IOI 2000] 回文字串