算法学习笔记(56): 康托展开

Pecco
可能算ACMer

康托展开用于求一个排列在所有 $1\sim n$ 的排列间的字典序排名。

其实康托展开的原理很简单。设有排列 $p=a_1a_2\dots a_n$ ，那么对任意字典序比 p 小的排列，一定存在 i ，使得其前 $i-1$ $(1\le i<n)$ 位与 p 对应位相同，第 i 位比 p_i 小，后续位随意。于是对于任意 i ，满足条件的排列数就是从后 $n-i+1$ 位中选一个比 a_i 小的数、并将剩下 $n-i$ 个数任意排列的方案数，即为 $A_i\cdot(n-i)!$ （ A_i 表示 a_i 后面比 a_i 小的数的个数）。遍历 i 即得总方案数 $\sum_{i=1}^{n-1}A_i\cdot(n-i)!$ ，再加1即为排名。

例如若 $p=4\ 1\ 3\ 2$ ，可以求得 $A=[3,0,1,0]$ ，第一位比 p_1 小的排列数为 $3\times3!=18$ ；第一位与 p_1 相等，第二位比 p_2 小的排列没有；第一、二位分别等于 p_1 、 p_2 ，第三位比 p_3 小的排列数为 $1\times1!=1$ 。所以 p 的排名是 $18+1+1=20$ 。

那么问题就转换成了如何求 A_i ，显然可以 $O(n^2)$ 地求：

```
ll fact[MAXN] = {1}, P[MAXN], A[MAXN]; // fact需要在外部初始化
ll cantor(int P[], int n) // 这里传入的P是1-index数组
{
    ll ans = 1;
    for (int i = 1; i <= n; i++)
        for (int j = i + 1; j <= n; j++)
            if (P[j] < P[i])
                A[i]++;
    for (int i = 1; i < n; i++)
        ans += A[i] * fact[n - i];
    return ans;
}
```

这个复杂度其实很够了，毕竟 $21!$ 就已经超过 long long 范围了，很多时候康托展开也就是在这个规模下进行。如果再大，就需要上高精度或者需要取模了。当然，优化也不难，很容易想到用树

```
ll query(ll x)
{
    ll ans = 0;
    for (int i = x; i >= 1; i -= lowbit(i))
        ans += tree[i];
    return ans;
}
void update(ll x, ll d)
{
    for (int i = x; i < MAXN; i += lowbit(i))
        tree[i] += d;
}
ll cantor(int P[], int n)
{
    ll ans = 1;
    for (int i = n; i >= 1; i--)
    {
        A[i] = query(P[i]);
        update(P[i], 1);
    }
    for (int i = 1; i < n; i++)
        ans = (ans + A[i] * fact[n - i]) % MOD;
    return ans;
}
```

与康托展开相对应的是**逆康托展开**，即求指定排名的排列。原理也很简单，注意到 $n!=n(n-1)!=(n-1)\cdot(n-1)!+(n-1)!$ ，继续展开得 $n!=\sum_{i=1}^{n-1}i\cdot i!$ ，而 $A_i\le n-i$ ，所以 $\sum_{i=j}^{n-1}A_i\cdot(n-i)!\le\sum_{i=j}^{n-1}(n-i)\cdot(n-i)!=\sum_{i=1}^{n-j}i\cdot i!=(n-j+1)!$ 。

这意味着对于这个和式而言，每一项的 $(n-i)!$ 都比后面所有项的总和还大。于是可以用类似进制转换的方法，不断地模、除，来得到 A 的每一项。

得到 A 后，我们已经知道每一项之后有多少个比该项小的数，也就是说 p_i 就是剩余未用的数中第 A_i+1 小的。可以朴素地实现：

```
{
    x--;
    vector<int> rest(n, 0);
    iota(rest.begin(), rest.end(), 1); // 将rest初始化为1,2,...,n
    for (int i = 1; i <= n; ++i)
    {
        A[i] = x / fact[n - i];
        x %= fact[n - i];
    }
    for (int i = 1; i <= n; ++i)
    {
        P[i] = rest[A[i]];
        rest.erase(lower_bound(rest.begin(), rest.end(), P[i]));
    }
}
```

当然，也可以使用各种平衡树来优化到 $O(n\log n)$ ：

```
ll fact[MAXN] = {1}, P[MAXN], A[MAXN]; // fact需要在外部初始化
void decanter(ll x, int n)             // x为排列的排名，n为排列的长度
{
    x--;
    for (int i = 1; i <= n; ++i)
        insert(i);
    for (int i = 1; i <= n; ++i)
    {
        A[i] = x / fact[n - i];
        x %= fact[n - i];
    }
    for (int i = 1; i <= n; ++i)
    {
        P[i] = kth(A[i] + 1);
        remove(P[i]);
    }
}
```

其中 `insert` 、 `kth` 、 `remove` 函数分别表示插入、查找第k小、删除元素。（参加二叉搜索树的笔记）