RMQ

简介

> RMQ 是英文 Range Maximum/Minimum Query 的缩写，表示区间最大（最小）值。

在接下来的描述中，默认初始数组大小为 n，询问次数为 m。

在接下来的描述中，默认时间复杂度标记方式为 $O(A) \sim O(B)$，其中 $O(A)$ 表示预处理时间复杂度，而 $O(B)$ 表示单次询问的时间复杂度。

单调栈

由于 **OI Wiki** 中已有此部分的描述，本文仅给出 链接。这部分不再展开。

时间复杂度 $O(m \log m) \sim O(\log n)$，空间复杂度 $O(n)$。

ST 表

由于 **OI Wiki** 中已有此部分的描述，本文仅给出 链接。这部分不再展开。

时间复杂度 $O(n \log n) \sim O(1)$，空间复杂度 $O(n \log n)$。

线段树

由于 **OI Wiki** 中已有此部分的描述，本文仅给出 链接。这部分不再展开。

时间复杂度 $O(n) \sim O(\log n)$，空间复杂度 $O(n)$。

Four Russian

Four russian 是一个由四位俄罗斯籍的计算机科学家提出来的基于 ST 表的算法。

在 ST 表的基础上 Four russian 算法对其做出的改进是序列分块。

具体来说，我们将原数组——我们将其称之为数组 A——每 S 个分成一块，总共 n/S 块。

对于每一块我们预处理出来块内元素的最小值，建立一个长度为 n/S 的数组 B，并对数组 B 采用 ST 表的方式预处理。

同时，我们对于数组 A 的每一个零散块也建立一个 ST 表。

询问的时候，我们可以将询问区间划分为不超过 1 个数组 B 上的连续块区间和不超过 2 个数组 A 上的整块内的连续区间。显然这些问题我们通过 ST 表上的区间查询解决。

在 $S = \log n$ 时候，预处理复杂度达到最优，为 $O((n/\log n)\log n + (n/\log n) \times \log n \times \log\log n) = O(n\log\log n)$。

时间复杂度 $O(n\log\log n) \sim O(1)$，空间复杂度 $O(n\log\log n)$。

当然询问由于要跑三个 ST 表，该实现方法的常数较大。

一些小小的算法改进

我们发现，在询问的两个端点在数组 A 中属于不同的块的时候，数组 A 中块内的询问是关于每一块前缀或者后缀的询问。

显然这些询问可以通过预处理答案在 $O(n)$ 的时间复杂度内被解决。

这样子我们只需要在询问的时候进行至多一次 ST 表上的查询操作了。

一些玄学的算法改进

由于 Four russian 算法以 ST 表为基础，而算法竞赛一般没有非常高的时间复杂度要求，所以 Four russian 算法一般都可以被 ST 表代替，在算法竞赛中并不实用。这里提供一种在算法竞赛中更加实用的 Four russian 改进算法。

我们将块大小设为 $\sqrt{n}$，然后预处理出每一块内前缀和后缀的 RMQ，再暴力预处理出任意连续的整块之间的 RMQ，时间复杂度为 $O(n)$。

查询时，对于左右端点不在同一块内的询问，我们可以直接 $O(1)$ 得到左端点所在块的后缀 RMQ，左端点和右端点之间的连续整块 RMQ，和右端点所在块的前缀 RMQ，答案即为三者之间的最值。

而对于左右端点在同一块内的询问，我们可以暴力求出两点之间的 RMQ，时间复杂度为 $O(\sqrt{n})$，但是单个询问的左右端点在同一块内的期望为 $O(\frac{\sqrt{n}}{n})$，所以这种方法的时间复杂度为期望 $O(n)$。

而在算法竞赛中，我们并不用非常担心出题人卡掉这种算法，因为我们可以通过在 $\sqrt{n}$ 的基础上随机微调块大小，很大程度上避免算法在根据特定块大小构造的数据中出现最坏情况。并且如果出题人想要卡掉这种方法，则暴力有可能可以通过。

这是一种期望时间复杂度达到下界，并且代码实现难度和算法常数均较小的算法，因此在算法竞赛中比较实用。

以上做法参考了 P3793 由乃救爷爷 中的题解。

加减 1RMQ

若序列满足相邻两元素相差为 1，在这个序列上做 RMQ 可以成为加减 1RMQ，根究这个特性可以改进 Four Russian 算法，做到 $O(n) \sim O(1)$ 的时间复杂度，$O(n)$ 的空间复杂度。

由于 Four russian 算法的瓶颈在于块内 RMQ 问题，我们重点去讨论块内 RMQ 问题的优化。

由于相邻两个数字的差值为 ± 1，所以在固定左端点数字时 长度不超过 $\log n$ 的右侧序列种类数为 $\sum_{i=1}^{\log n} 2^{i-1}$，而这个式子显然不超过 n。

这启示我们可以预处理所有不超过 n 种情况的 最小值 - 第一个元素 的值。

在预处理的时候我们需要去预处理同一块内相邻两个数字之间的差，并且使用二进制将其表示出来。

在询问的时候我们找到询问区间对应的二进制表示，查表得出答案。

这样子 Four russian 预处理的时间复杂度就被优化到了 $O(n)$。

笛卡尔树在 RMQ 上的应用

不了解笛卡尔树的朋友请移步 笛卡尔树。

不难发现，原序列上两个点之间的 min/max，等于笛卡尔树上两个点的 LCA 的权值。根据这一点就可以借助 $O(n) \sim O(1)$ 求解树上两个点之间的 LCA 进而求解 RMQ。$O(n) \sim O(1)$ 树上 LCA 在 LCA - 标准 RMQ 已经有描述，这里不再展开。

总结一下，笛卡尔树在 RMQ 上的应用，就是通过将普通 RMQ 问题转化为 LCA 问题，进而转化为加减 1 RMQ 问题进行求解，时间复杂度为 $O(n) \sim O(1)$。当然由于转化步数较多，$O(n) \sim O(1)$ RMQ 常数较大。

如果数据随机，还可以暴力在笛卡尔树上查找。此时的时间复杂度为期望 $O(n) \sim O(\log n)$，并且实际使用时这种算法的常数往往很小。

例题 Luogu P3865【模板】ST 表

基于状压的线性 RMQ 算法

隐性要求

- 序列的长度 n 满足 $\log_2 n \leq 64$。

前置知识

- Sparse Table
- 基本位运算
- 前后缀极值

算法原理

将原序列 $A[1\cdots n]$ 分成每块长度为 $O(\log_2 n)$ 的 $O(\frac{n}{\log_2 n})$ 块。

> 听说令块长为 $1.5\times\log_2 n$ 时常数较小。

记录每块的最大值，并用 ST 表维护块间最大值，复杂度 $O(n)$。

记录块中每个位置的前、后缀最大值 $Pre[1\cdots n], Sub[1\cdots n]$（$Pre[i]$ 即 $A[i]$ 到其所在块的块首的最大值），复杂度 $O(n)$。

若查询的 l,r 在两个不同块上，分别记为第 bl,br 块，则最大值为 $[bl+1,br-1]$ 块间的最大值，以及 $Sub[l]$ 和 $Pre[r]$ 这三个数的较大值。

现在的问题在于若 l,r 在同一块中怎么办。

将 $A[1\cdots r]$ 依次插入单调栈中，记录下标和值，满足值从栈底到栈顶递减，则 $A[l,r]$ 中的最大值为从栈底往上，单调栈中第一个满足其下标 $p\geq l$ 的值。

由于 $A[p]$ 是 $A[l,r]$ 中的最大值，因而在插入 $A[p]$ 时，$A[l\cdots p-1]$ 都被弹出，且在插入 $A[p+1\cdots r]$ 时不可能将 $A[p]$ 弹出。

而如果用 $0/1$ 表示每个数是否在栈中，就可以用整数状压，则 p 为第 l 位后的第一个 1 的位置。

由于块大小为 $O(\log_2 n)$，因而最多不超过 64 位，可以用一个整数存下（即隐性条件的原因）。

参考代码

```cpp
#include <algorithm>
#include <cmath>
#include <cstdio>

constexpr int MAXN = 1e5 + 5;
constexpr int MAXM = 20;

struct RMQ {
  int N, A[MAXN];
  int blockSize;
  int S[MAXN][MAXM], Pow[MAXM], Log[MAXN];
  int Belong[MAXN], Pos[MAXN];
  int Pre[MAXN], Sub[MAXN];
  int F[MAXN];

  void buildST() {
    int cur = 0, id = 1;
    Pos[0] = -1;
    for (int i = 1; i <= N; ++i) {
      S[id][0] = std::max(S[id][0], A[i]);
      Belong[i] = id;
      if (Belong[i - 1] != Belong[i])
        Pos[i] = 0;
      else
        Pos[i] = Pos[i - 1] + 1;
      if (++cur == blockSize) {
        cur = 0;
        ++id;
      }
    }
    if (N % blockSize == 0) --id;
    Pow[0] = 1;
    for (int i = 1; i < MAXM; ++i) Pow[i] = Pow[i - 1] * 2;
    for (int i = 2; i <= id; ++i) Log[i] = Log[i / 2] + 1;
    for (int i = 1; i <= Log[id]; ++i) {
      for (int j = 1; j + Pow[i] - 1 <= id; ++j) {
        S[j][i] = std::max(S[j][i - 1], S[j + Pow[i - 1]][i - 1]);
      }
    }
  }

  void buildSubPre() {
    for (int i = 1; i <= N; ++i) {
      if (Belong[i] != Belong[i - 1])
        Pre[i] = A[i];
      else
        Pre[i] = std::max(Pre[i - 1], A[i]);
    }
```

```
    for (int i = N; i >= 1; --i) {
      if (Belong[i] != Belong[i + 1])
        Sub[i] = A[i];
      else
        Sub[i] = std::max(Sub[i + 1], A[i]);
    }
  }

  void buildBlock() {
    static int S[MAXN], top;
    for (int i = 1; i <= N; ++i) {
      if (Belong[i] != Belong[i - 1])
        top = 0;
      else
        F[i] = F[i - 1];
      while (top > 0 && A[S[top]] <= A[i]) F[i] &= ~(1 <<
Pos[S[top--]]);
      S[++top] = i;
      F[i] |= (1 << Pos[i]);
    }
  }

  void init() {
    for (int i = 1; i <= N; ++i) scanf("%d", &A[i]);
    blockSize = log2(N) * 1.5;
    buildST();
    buildSubPre();
    buildBlock();
  }

  int queryMax(int l, int r) {
    int bl = Belong[l], br = Belong[r];
    if (bl != br) {
      int ans1 = 0;
      if (br - bl > 1) {
        int p = Log[br - bl - 1];
        ans1 = std::max(S[bl + 1][p], S[br - Pow[p]][p]);
      }
      int ans2 = std::max(Sub[l], Pre[r]);
      return std::max(ans1, ans2);
    } else {
      return A[l + __builtin_ctz(F[r] >> Pos[l])];
    }
  }
} R;

int M;

int main() {
  scanf("%d%d", &R.N, &M);
  R.init();
  for (int i = 0, l, r; i < M; ++i) {
```

```
    scanf("%d%d", &l, &r);
    printf("%d\n", R.queryMax(l, r));
  }
  return 0;
}
```

习题

[BJOI 2020] 封印：SAM+RMQ
