1009 GCD 题解

Gospel_rock

1 题意

给定两个长度为 n 的整数数组 a 和 b。q 次询问，每次给出区间 $[l, r]$，取出子数组 $a' = a[l \ldots r]$，$b' = b[l \ldots r]$（长度 $m = r - l + 1$）。

每次操作可以选取下标 i（$1 \le i \le m$），计算后缀 GCD $g_i = \gcd(a'_i, a'_{i+1}, \ldots, a'_m)$，然后将 b'_j 减去 g_i（对所有 $i \le j \le m$）。每次操作代价为 1。求使 b' 所有元素 ≤ 0 的最小总代价。

2 贪心策略

2.1 关键观察：后缀 GCD 的单调性

后缀 GCD 满足 $g_1 \mid g_2 \mid \cdots \mid g_m$，因此 $g_1 \le g_2 \le \cdots \le g_m$。这意味着越靠右的位置，单次操作的减法力度越大。

2.2 贪心方案

从左到右扫描位置 $k = 1, 2, \ldots, m$，维护一个累积减量 S（初始 $S = 0$）。S 表示之前所有操作对位置 k 产生的总减量。

$$\text{若 } b'_k > S\text{，则在位置 } k \text{ 追加 } c = \left\lceil \frac{b'_k - S}{g_k} \right\rceil \text{ 次操作，随后更新 } S \leftarrow S + c \cdot g_k\text{。}$$

2.3 为什么在当前位置操作？

当扫到位置 k 时，如果 $b'_k > S$，说明之前的操作不够用，必须追加。追加操作可以在 1 到 k 的任何位置执行。但由于 $g_k \ge g_j$（$\forall j \le k$），在位置 k 执行单次操作的减量最大。

同时，位置 1 到 $k-1$ 已经在之前的步骤中被满足了，所以我们不需要利用”更早位置能覆盖更多左侧元素”这一优势——那些左侧元素早已被处理完毕。

综合来看：在位置 k 追加操作，既享受了最大的单次减量 g_k，又不浪费在已满足的左侧位置上。而且这些操作也会对 k 之后的所有位置产生减量，为后续处理提供帮助。

2.4 样例验证

以询问 $[1, 3]$ 为例：$a' = [6, 12, 18]$，$b' = [10, 20, 30]$。

后缀 GCD：$g_1 = \gcd(6, 12, 18) = 6$，$g_2 = \gcd(12, 18) = 6$，$g_3 = 18$。

- $k = 1$：$S = 0 < b'_1 = 10$。追加 $\lceil 10/6 \rceil = 2$ 次。$S \leftarrow 12$。代价 $= 2$。
- $k = 2$：$S = 12 < b'_2 = 20$。追加 $\lceil 8/6 \rceil = 2$ 次。$S \leftarrow 24$。代价 $= 4$。
- $k = 3$：$S = 24 < b'_3 = 30$。追加 $\lceil 6/18 \rceil = 1$ 次。$S \leftarrow 42$。代价 $= 5$。

总代价为 5，与样例吻合。

图 1 展示了上述过程。

3 $O(nq)$ 暴力

上述贪心直接实现即可得到 $O(nq)$ 的暴力解法。对每个查询 $[l, r]$，先从右向左计算后缀 GCD 数组，再从左到右扫描执行贪心：

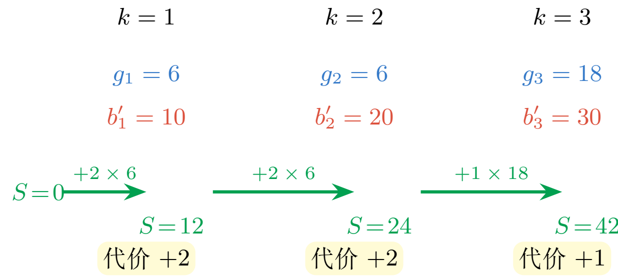

Figure 1: 询问 $[1,3]$ 的贪心过程。S 为累积减量（绿色），每一步在当前位置追加操作使 $S \ge b'_k$。

```
ll reply_to_query(ll l, ll r) {
  gcd_ls[r] = aGlo[r];
  for (int i = r - 1; i >= l; --i)
    gcd_ls[i] = __gcd(gcd_ls[i + 1], aGlo[i]);
  ll acc = 0, ans = 0;
  for (int i = l; i <= r; ++i) {
    if (acc >= bGlo[i]) continue;
    ll rem = bGlo[i] - acc;
    ll add = (rem + gcd_ls[i] - 1) / gcd_ls[i];
    acc += add * gcd_ls[i];
    ans += add;
  }
  return ans;
}
```

这段代码的瓶颈在于：每次询问都要逐个扫描 $O(m)$ 个位置。能否加速单次查询?

4 加速单次查询：后缀 GCD 分块

暴力中，内层循环逐个扫描 $k = l, l+1, \ldots, r$，而每个位置的 g_k 值可能不同。但仔细观察会发现，g_k 的取值远没有 $O(m)$ 那么多——它最多只有 $O(\log V)$ 种。如果我们按 GCD 值相同的连续段（称为**块**）来处理，单次查询就只需遍历 $O(\log V)$ 个块，而非 $O(m)$ 个位置。

4.1 为什么只有 $O(\log V)$ 个块？

固定右端点 r，考虑后缀 GCD $g_k = \gcd(a_k, a_{k+1}, \ldots, a_r)$，其中 k 从 r 到 1。

- $g_r = a_r$。
- $g_{k-1} = \gcd(a_{k-1}, g_k)$，因此 $g_{k-1} \mid g_k$。
- 若 $g_{k-1} \ne g_k$，则 g_{k-1} 是 g_k 的**真因子**，故 $g_{k-1} \le g_k/2$。

从 $g_r \le V$ 出发，每次变化至少减半，最多减半 $\lfloor \log_2 V \rfloor$ 次就到 1。因此不同的 g 值最多 $O(\log V)$ 个 $(V = \max a_i)$，它们构成连续的若干**块**。

注意这里的界来自”每次变化必须整除且至少减半”，与 n 无关。即使所有 a_i 互素，GCD 也会一步降到 1 并不再变化，块数反而最少。

图 2 以 $a = [6, 12, 18, 40, 50]$（对应下标 $1 \sim 5$，右端点 $r = 5$）为例展示了分块结构。

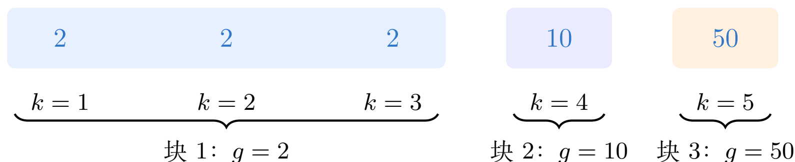

Figure 2: $a = [6, 12, 18, 40, 50]$，右端点 $r = 5$ 时后缀 GCD 的分块结构。5 个位置仅产生 3 个 GCD 块。

4.2 按块做贪心：单次查询 $O(\log V)$

有了分块结构，我们重新审视暴力的内层循环。对同一个 GCD 块（值为 g，覆盖位置 $[lo, hi]$），g 是定值，而累积减量 S 单调递增。所以块内唯一能触发追加的是 b' 的**最大值** M——追加 $\lceil (M-S)/g \rceil$ 次后 $S \geq M$，块内其余位置自然被满足。

因此，对一个块只需查询 $M = \max(b'_{lo}, \ldots, b'_{hi})$：若 $M \leq S$ 则跳过；否则追加并更新 S。单次查询只遍历 $O(\log V)$ 个块，每个块 $O(1)$，总共 $O(\log V)$。

对比暴力代码，优化后的单次查询逻辑几乎一一对应：

```
// 暴力：逐个位置扫描
for (int i = l; i <= r; ++i) {



    if (acc >= bGlo[i]) continue;
    ll rem = bGlo[i] - acc;
    ll add = (rem + gcd_ls[i] - 1)
             / gcd_ls[i];
    acc += add * gcd_ls[i];
    ans += add;
}
```

```
// 优化：逐个 GCD 块扫描
for (auto &blk : blocks) {
    int lo = max(blk.lo, l);
    int hi = blk.hi;
    if (lo > hi) continue;
    ll mx = queryMax(lo, hi);
    if (mx > acc) {
        ll c = (mx - acc + blk.g - 1)
               / blk.g;
        acc += c * blk.g;
        cost += c;
    }
}
```

左边用 b_k 判定，右边用块内 $\max b$ 判定；左边用 g_k，右边用块的统一 GCD 值。逻辑完全一致，只是粒度从单个位置变成了一整个块。

4.3 $O(1)$ 区间最大值：ST 表

按块做贪心需要快速查询任意区间 $[lo, hi]$ 内 b 的最大值。对数组 b 预处理一个 **ST 表**即可实现 $O(1)$ 查询：

```
void buildST() {
  LOG = 1;
  while ((1 << LOG) <= N) LOG++;
  st.assign(LOG, vector<ll>(N + 1));
  for (int i = 1; i <= N; i++) st[0][i] = b[i];
  for (int j = 1; j < LOG; j++)
    for (int i = 1; i + (1 << j) - 1 <= N; i++)
      st[j][i] = max(st[j - 1][i], st[j - 1][i + (1 << (j - 1))]);
}

ll queryMax(int l, int r) {
  int k = 31 - __builtin_clz(r - l + 1);
  return max(st[k][l], st[k][r - (1 << k) + 1]);
}
```

4.4 高效维护块列表：扫描线

至此，单次查询已经是 $O(\log V)$ 了，但还剩一个问题：对每个查询 $[l, r]$，我们需要拿到右端点为 r 时的 GCD 块列表。如果每次都从头算，又退化回 $O(n)$。

解决办法是将所有询问离线，按右端点 r 分组，然后从 $r = 1$ 到 n 扫描维护块列表。当右端点从 $r-1$ 扩展到 r 时，每个块的 GCD 值与 a_r 取 gcd，相邻同值块合并，末尾追加新块 (a_r, r, r)。任何时刻块数不超过 $O(\log V)$，单步更新代价 $O(\log V)$。

```
vector<Block> blocks;
for (int r = 1; r <= N; r++) {
  vector<Block> nb;
  for (auto &blk : blocks) {
    ll ng = __gcd(blk.g, a[r]);
    if (!nb.empty() && nb.back().g == ng)
      nb.back().hi = blk.hi;
    else
```

```cpp
      nb.push_back({ng, blk.lo, blk.hi});
  }
  if (!nb.empty() && nb.back().g == a[r])
    nb.back().hi = r;
  else
    nb.push_back({a[r], r, r});
  blocks = move(nb);

  // 回答所有右端点为 r 的查询
  for (auto [l, id] : queries[r]) {
    ll acc = 0, cost = 0;
    for (auto &blk : blocks) {
      int lo = max(blk.lo, l);
      int hi = blk.hi;
      if (lo > hi) continue;
      ll mx = queryMax(lo, hi);
      if (mx > acc) {
        ll c = (mx - acc + blk.g - 1) / blk.g;
        cost += c;
        acc += c * blk.g;
      }
    }
    ans[id] = cost;
  }
}
```

5 复杂度分析

- **ST 表预处理**：$O(n \log n)$。
- **扫描线维护 GCD 块**：每步更新 $O(\log V)$ 个块，总计 $O(n \log V)$。
- **回答查询**：每次查询遍历 $O(\log V)$ 个块，每块 $O(1)$ 查询区间最大值，总计 $O(q \log V)$。
- **总时间复杂度**：$O(n \log n + (n+q) \log V)$，其中 $V = \max a_i \le 10^9$，$\log V \approx 30$。
- **空间复杂度**：$O(n \log n)$（ST 表）。

6 AC 代码

```cpp
// teamname: Gospel_rock
/**
 * Problem: GCD
 * Contest:
 * Judge: HDOJ
 * URL: https://acm.hdu.edu.cn/contest/problem?cid=1200&pid=1009
 * Created: 2026-04-14 11:18:13
 * Author: Gospel_rock
 * My blog: https://znzryb.com/
 *
 * Powered by AutoCp https://github.com/Pushpavel/AutoCp
 */

#include <bits/stdc++.h>
#define all(vec) vec.begin(),vec.end()
#define lson(o) (o<<1)
#define rson(o) (o<<1|1)
#define SZ(a) ((long long) a.size())
#define debug(var) cerr << #var <<" = ["<<var<<"]"<<"\n";
#define debug1d(a)     \
cerr << #a << " = [";   \
for (int i = 0; i < (int)(a).size(); i++) \
```

```cpp
cerr << (i ? ", " : "") << a[i]; \
cerr << "]\n";
#define debug2d(a)  \
cerr << #a << " = [\n";  \
for (int i = 0; i < (int)(a).size(); i++)  \
{   \
cerr << "  [";  \
for (int j = 0; j < (int)(a[i]).size(); j++) \
cerr << (j ? ", " : "") << a[i][j];   \
cerr << "]\n";   \
}  \
cerr << "]\n";
#define cend cerr<<"\n-----------\n"
#define fsp(x) fixed<<setprecision(x)

using namespace std;

using ll = long long;
using ull = unsigned long long;
using DB = double;
using i128 = __int128;
using CD = complex<double>;

static constexpr ll MAXN = (ll) 1e6 + 10, INF = (1ll << 61) - 1;
static constexpr ll mod = 998244353; // (ll)1e9+7;
static constexpr double eps = 1e-8;
const double PI = acos(-1.0);

ll lT, testcase;

/*
 *
 */

/*
 *2025/12/10 AC https://codeforces.com/contest/2175/submission/352780861
 *2025/3/2 实现了这个 op 函数的传入 (可以传入一个 lambda 函数)
 *AC https://codeforces.com/contest/2205/submission/365029208
 *
 */
template<class T, class F>
struct ST {
  ll N;
  vector<vector<T> > dp;
  F op;
  // static T op(const T &a,const T &b) {
  //     return max(a,b);
  // }
  ST(ll N, const vector<T> &a, F fun) : N(N), dp(__lg(N) + 1, vector<T>(N + 5)), op(fun) {
    for (int i = 1; i <= N; ++i) {
      dp[0][i] = a[i];
    }
    for (int lay = 1; lay <= __lg(N); ++lay) {
      for (int i = 1; i <= N - (1 << (lay - 1)); ++i) {
        dp[lay][i] = op(dp[lay - 1][i], dp[lay - 1][i + (1 << (lay - 1))]);
      }
    }
  }

  T query(ll l, ll r) {
    ll lay = __lg(r - l + 1);
    return op(dp[lay][l], dp[lay][r - (1 << lay) + 1]);
  }
```

```
};

struct GCDop {
  ll operator()(ll a, ll b) {
    return __gcd(a, b);
  }
};

struct Solve {
  ll N, Q;
  vector<ll> aGlo;
  vector<ll> bGlo;
  vector<array<ll, 2> > queryLs;
  optional<ST<ll, GCDop> > st_gcd;

  Solve() {
    cin >> N >> Q;
    aGlo.resize(N + 2);
    bGlo.resize(N + 2);
    queryLs.resize(Q);

    for (int i = 1; i <= N; ++i) {
      cin >> aGlo[i];
    }
    for (int i = 1; i <= N; ++i) {
      cin >> bGlo[i];
    }
    for (int i = 0; i < Q; ++i) {
      cin >> queryLs[i][0] >> queryLs[i][1];
    }
    st_gcd.emplace(N, aGlo, GCDop{});

  }
};

signed main() {
  ios::sync_with_stdio(false);
  cin.tie(nullptr);
  cout.tie(nullptr);
#ifdef LOCAL
  cout.setf(ios::unitbuf); // 无缓冲流，方便我们调试
#endif

  cin >> lT;
  for (testcase = 1; testcase <= lT; ++testcase)
    Solve solve;
  return 0;
}

/*

*/
```