区间覆盖(xmasinterval,3s,512MB)

首先将关键点排序、去重，接着对每个区间$[l_i,r_i]$二分查找出u_i,v_i，表示$[l_i,r_i]$能覆盖第u_i个到第v_i个关键点。

我们先考虑能覆盖至少一个关键点的区间。设 f[i][j]表示在前 i 个区间中任意选取若干个区间，至少覆盖前 j 个关键点（可以覆盖 j 后面的关键点）的方案数。

显然有初值：f[0][0]=1。

接着按u_i升序枚举每个区间，考虑枚举到第 i 个区间时如何转移：

1.$0\le j\le v_i$：如果前 i-1 个区间至少覆盖了前u_i-1个关键点，那么选择第 i 个区间之后就能至少覆盖前v_i个关键点。即：$\forall 0\le j\le v_i$，$f[i][j]=f[i-1][j]+f[i-1][u_i-1]$。

2.$v_i<j\le m$：显然第 i 个区间选或不选都不会有任何影响，即：$\forall v_i<j\le m$，$f[i][j]=f[i-1][j]\times 2$。

设能覆盖至少一个关键点的区间有 x 个，显然剩下 n-x 个区间选或不选都不会有任何影响，最后的答案就是$f[x][m]\times 2^{n-x}$。

这样时间复杂度是 O(nm)的，考虑如何优化转移。建立下标为[0..m]的线段树，初始时位置 0 的值为 1，其余位置的值为 0。枚举到第 i 个区间时，在线段树上查询位置u_i-1的值，记为 s，然后把位置$0\sim v_i$的值都加上 s，把位置$v_i+1\sim m$的值都乘 2。最后位置 m 的值就是 f[x][m]。

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

【参考程序】

```
//陈予菲
#include <bits/stdc++.h>

using namespace std;

#define p2 p << 1
#define p3 p << 1 | 1
#define ll long long

template <class t>
inline void read(t & res)
{
    char ch;
    while (ch = getchar(), !isdigit(ch));
    res = ch ^ 48;
    while (ch = getchar(), isdigit(ch))
    res = res * 10 + (ch ^ 48);
}

const int e = 5e5 + 5, mod = 1e9 + 9;
struct point
{
    int l, r;
}a[e];
int n, m, b[e], ans, mul[e * 4], add[e * 4], c[e];
```

```
inline int plu(int x, int y)
{
    x += y;
    if (x >= mod) x -= mod;
    return x;
}

inline void pushdown(int l, int r, int p)
{
    if (mul[p]) //mul[p]表示区间[l,r]的每个位置都乘上 2^mul[p]
    {
        mul[p2] += mul[p];
        mul[p3] += mul[p];
        int x = c[mul[p]];
        add[p2] = (ll)add[p2] * x % mod;
        add[p3] = (ll)add[p3] * x % mod;
        mul[p] = 0;
    }
    if (add[p]) //add[p]表示区间[l,r]的每个位置都加上 add[p]
    {
        int mid = l + r >> 1;
        add[p2] = plu(add[p2], add[p]);
        add[p3] = plu(add[p3], add[p]);
        add[p] = 0;
    }
}

inline void build(int l, int r, int p)
{
    if (l == r)
    {
        if (!l) add[p] = 1; // 初始位置 0 的值为 1
        return;
    }
    int mid = l + r >> 1;
    build(l, mid, p2);
    build(mid + 1, r, p3);
}

inline void update(int l, int r, int s, int t, int p) //区间乘 2
{
    if (l == s && r == t)
    {
```

```
        mul[p]++;
        add[p] = plu(add[p], add[p]);
        return;
    }
    pushdown(l, r, p);
    int mid = l + r >> 1;
    if (t <= mid) update(l, mid, s, t, p2);
    else if (s > mid) update(mid + 1, r, s, t, p3);
    else
    {
        update(l, mid, s, mid, p2);
        update(mid + 1, r, mid + 1, t, p3);
    }
}

inline void modify(int l, int r, int s, int t, int v, int p) //区间加
{
    if (l == s && r == t)
    {
        add[p] = plu(add[p], v);
        return;
    }
    pushdown(l, r, p);
    int mid = l + r >> 1;
    if (t <= mid) modify(l, mid, s, t, v, p2);
    else if (s > mid) modify(mid + 1, r, s, t, v, p3);
    else
    {
        modify(l, mid, s, mid, v, p2);
        modify(mid + 1, r, mid + 1, t, v, p3);
    }
}

inline int query(int l, int r, int s, int p) //查询位置 s 的值
{
    if (l == r) return add[p];
    int mid = l + r >> 1;
    pushdown(l, r, p);
    if (s <= mid) return query(l, mid, s, p2);
    else return query(mid + 1, r, s, p3);
}

inline bool cmp(const point &a, const point &b)
{
```

```
    return a.l < b.l;
}

int main()
{
    freopen("xmasinterval.in", "r", stdin);
    freopen("xmasinterval.out", "w", stdout);
    int i;
    read(n); read(m);
    for (i = 1; i <= n; i++) read(a[i].l), read(a[i].r);
    for (i = 1; i <= m; i++) read(b[i]);
    c[0] = 1;
    for (i = 1; i <= m; i++) c[i] = 2ll * c[i - 1] % mod; //预处理 2 的幂
    sort(b + 1, b + m + 1);
    m = unique(b + 1, b + m + 1) - b - 1;
    for (i = 1; i <= n; i++)
    {
        if (a[i].l <= b[m]) a[i].l = lower_bound(b + 1, b + m + 1, a[i].l)
- b;
        else a[i].l = m + 1;
        if (a[i].r >= b[1]) a[i].r = upper_bound(b + 1, b + m + 1, a[i].r)
- b - 1;
        else a[i].r = 0; //二分查找
    }
    sort(a + 1, a + n + 1, cmp);
    build(0, m, 1);
    for (i = 1; i <= n; i++)
    {
        int l = a[i].l, r = a[i].r;
        if (l > r) ans++;           //跳过覆盖不到任何关键点的区间
        int s = query(0, m, l - 1, 1);
        modify(0, m, 0, r, s, 1);  //位置 0~r 的值都加 s
        if (r != m) update(0, m, r + 1, m, 1); //位置 r+1~m 的值都乘 2
    }
    ans = (ll)c[ans] * query(0, m, m, 1) % mod;
    cout << ans << endl;
    fclose(stdin);
    fclose(stdout);
    return 0;
}
```

区间(interval,1s,256MB)

问题可以转化为：求有多少个数字区间$[l,r](l<r)$，满足该区间内的数字在P中构成的连通块个数为 1 或 2。

算法一

枚举数字区间$[l,r]$，遍历P，求出该区间在P中构成的连通块个数，判断是否满足条件。

时间复杂度$O(n^3)$，期望得分 10 分。

算法二

枚举数字区间的左端点l。考虑从$[l,r-1]$转移到$[l,r]$时如何维护连通块个数（设数字r在P中的位置为x）：

1. 若位置$x-1$和$x+1$均未被标记，则连通块个数$+1$；
2. 若位置$x-1$和$x+1$均被标记，则连通块个数-1；
3. 若位置$x-1$和$x+1$中恰有一个被标记，则连通块个数不变。

那么我们只要在连通块个数为 1 或 2 时将区间计入答案即可。

时间复杂度$O(n^2)$，期望得分 40 分。

算法三

当$P_i=i$时，显然答案为$\frac{n(n-1)}{2}$。

时间复杂度$O(1)$，结合算法二期望得分 50 分。

算法四

设$f[l][r]$表示数字区间$[l,r]$在P中构成的连通块的个数，答案即为：

$$\sum_{1\le l<r\le n}[1\le f[l][r]\le 2]$$

特别地，我们定义$\forall l>r,\ f[l][r]=0$。

考虑倒序枚举数字区间的左端点l，考虑如何从$f[l+1][1...n]$转移到$f[l][1\dots n]$（设now为转移时新加入的数字，显然有$now=l$；L、R分别表示在P中位置与now相邻的两个数，为方便起见，我们令$L<R$）：

1. 若$L<R<now$，则对于数字区间$[now,now\dots n]$，now在P中只能自成一个连通块，即$\forall now\le j\le n,\ f[now][j]=f[now+1][j]+1$。
2. 若$L<now<R$，则：
   a) 对于数字区间$[now,now\dots R-1]$，now在P中同样只能自成一个连通块，即$\forall now\le j\le R-1,\ f[now][j]=f[now+1][j]+1$；
   b) 对于数字区间$[now,R\dots n]$，now在P中可并入R所在的联通块，即$\forall R\le j\le n$, $f[now][j]=f[now+1][j]$。
3. 若$now<L<R$，则：
   a) 对于数字区间$[now,now\dots L-1]$，now自成一个连通块，即$\forall now\le j\le L-1$, $f[now][j]=f[now+1][j]+1$；
   b) 对于数字区间$[now,L\dots R-1]$，now可并入L所在的联通块，即$\forall L\le j\le R-1$, $f[now][j]=f[now+1][j]$；
   c) 对于数字区间$[now,R\dots n]$，now可与L、R所在的连通块合并，即$\forall R\le j\le n$, $f[now][j]=f[now+1][j]-1$。

以上转移均类似于针对数组f的第二维的区间修改操作，且对于每个now，显然有$f[now][now+1\dots n]>0$，于是我们可用线段树维护$f[now][1\dots n]$值。线段树上对应着区间$[l,r]$的结点记录$f[now][l\dots r]$中的最小值、次小值及其出现次数，答案即为：

$$\sum_{1 \le now < n} \text{线段树上区间}[now+1, n]\text{中 1 和 2 出现的总次数}$$

时间复杂度$O(n\log n)$，期望得分 100 分。

【参考程序】//潘恩宁

```
#include <bits/stdc++.h>
#define For(i, a, b) for (int i = a, bb = b; i <= bb; ++i)
#define Rof(i, a, b) for (int i = a, bb = b; i >= bb; --i)
#define s64 long long

template <class T>
inline void get(T &res)
{
    char ch;
    bool bo = false;
    while ((ch = getchar()) < '0' || ch > '9')
        if (ch == '-') bo = true;

    res = ch - '0';
    while ((ch = getchar()) >= '0' && ch <= '9')
        res = (res << 1) + (res << 3) + ch - '0';

    if (bo) res = ~ res + 1;
    return;
}

template <class T>
inline void _put(T x)
{
    if (x > 9) _put(x / 10);
    putchar(x % 10 + '0');
    return;
}

template <class T>
inline void put(T x, char ch)
{
    if (x < 0)
    {
        putchar('-');
        x = ~ x + 1;
    }
    _put(x);
    putchar(ch);
```

```
    return;
}

#define k2 k << 1
#define k3 k << 1 | 1

const int MaxN = 3e5 + 5, INF = 0x3f3f3f3f;
int a[MaxN], pos[MaxN], tag[MaxN << 2], n;
s64 ans;
struct tree //线段树记录区间最小值、区间次小值以及出现次数
{
    int val1, cnt1, val2, cnt2;

    inline tree operator + (tree rhs)
    {
        tree lhs = *this;
        if (lhs.val1 > rhs.val1) lhs = rhs, rhs = *this;
        if (lhs.val2 < rhs.val1) return lhs;
        if (rhs.val1 == lhs.val2)
        {
            lhs.cnt2 += rhs.cnt1;
            return lhs;
        }
        else if (rhs.val1 == lhs.val1)
        {
            lhs.cnt1 += rhs.cnt1;
            if (rhs.val2 < lhs.val2)
                return tree{lhs.val1, lhs.cnt1, rhs.val2, rhs.cnt2};
            else
            {
                if (rhs.val2 == lhs.val2)
                    lhs.cnt2 += rhs.cnt2;
                return lhs;
            }
        }
        else
            return tree {lhs.val1, lhs.cnt1, rhs.val1, rhs.cnt1};
    }
} tr[MaxN << 2];

inline void build(int k, int l, int r)
{
    if (l == r)
    {
```

```
        tr[k] = tree {0, 1, INF, 1};
        return;
    }

    int mid = l + r >> 1;
    build(k2, l, mid);
    build(k3, mid + 1, r);
    tr[k] = tr[k2] + tr[k3];
    return;
}

inline void marktag(int k, int v)
{
    tag[k] += v;
    tr[k].val1 += v, tr[k].val2 += v;
    return;
}

inline void downtag(int k)
{
    if (!tag[k]) return;
    marktag(k2, tag[k]);
    marktag(k3, tag[k]);
    tag[k] = 0;
    return;
}

inline void modify(int k, int l, int r, int x, int y, int v)
{
    if (x <= l && r <= y)
        return marktag(k, v);

    downtag(k);
    int mid = l + r >> 1;
    if (x <= mid) modify(k2, l, mid, x, y, v);
    if (y > mid) modify(k3, mid + 1, r, x, y, v);
    tr[k] = tr[k2] + tr[k3];
    return;
}

inline tree query(int k, int l, int r, int x) //询问区间[x,n]的答案
{
    if (x <= l)
        return tr[k];
```

```
    downtag(k);
    int mid = l + r >> 1;
    return x <= mid ? query(k2, l, mid, x) + tr[k3] : query(k3, mid + 1,
r, x);
}

int main()
{
    freopen("interval.in", "r", stdin);
    freopen("interval.out", "w", stdout);

    get(n);
    For(i, 1, n)
        get(a[i]);
    For(i, 1, n)
        pos[a[i]] = i;
    build(1, 1, n);

    Rof(now, n, 1)
    {
        int L = a[pos[now] - 1], R = a[pos[now] + 1];
        if (L > R) std::swap(L, R);
        //L、R 分别表示在 P 中位置与 now 相邻的两个数

        if (R < now) //分类讨论加入数字 now 对连通块个数的贡献
            modify(1, 1, n, now, n, 1);
        else if (L < now)
            modify(1, 1, n, now, R - 1, 1);
        else
            modify(1, 1, n, now, L - 1, 1),
            modify(1, 1, n, R, n, -1);

        if (now == n)
            continue;
        tree x = query(1, 1, n, now + 1);
        if (x.val1 <= 2) //组成区间[now,now+1...n]的方案数计入答案
        {
            ans += x.cnt1;
            if (x.val2 == 2) ans += x.cnt2;
        }
    }
    put(ans, '\n');
```

```
    fclose(stdin), fclose(stdout);
    return 0;
}
```

序列（sequence,3s,256MB）

首先，我们发现条件一等价于：若$i<j$且$b_i \le a_j$，则i与j在一段中。

这样我们就可以将满足条件的每段区间先并成一个数对，因为它们一定是在一段区间的。我们枚举区间的起点i，我们现在试图不断扩大区间，得到区间的终点ed。

假设目前我们的区间得到是$[i,j]$。

若$\exists x \in [i,j], y \in [j+1,n], b_x \le a_y$，则$x,y$在一个区间，又因为分的段是连续的，所以我们就能将区间扩展到$[i,y]$。否则扩展结束，可知$ed=j$。

而这个判断条件也可以简化成$\min\limits_{i \le x \le j}\{b_x\} \le \max\limits_{j<y \le n}\{a_y\}$，我们只要通过预处理，从而快速求出$b$的区间最小值和$a$的后缀最大值即可。

这个并起来的数对的a就等效于区间内所有a的最大值，数对的b就等效于区间内所有b的和。

现在我们的目的就是要将新得到的序列进行分组。

因为要最小化每一段b的和的最大值，所以很容易想到二分每一段b的和的最大值mid，限制每一段b的和不能超过最大值mid。然后我们在这个限制条件下，通过 DP 求出每一段的a的最大值之和的最小值，并通过判断这个值是否不超过m来判断二分到的答案是否合法。

定义状态$f[i]$表示把前i个数分成若干段，在每一段的b的和都不超过mid前提下，每一段的最大的a和最小值是多少。

令$s_i=\sum\limits_{j=1}^{i} b_j$，即表示$b$的前缀和。

很容易得到状态转移方程：

$$f[i]=\min_{0 \le j<i \text{且} s_i-s_j \le mid}\{f[j]+\max_{j<k \le i}\{a_k\}\}$$

直接转移的 DP 时间复杂度是$O(n^2)$的，显然不足以通过。

而$\max\limits_{j<k \le i}\{a_k\}$难以表示成简单多项式，我们不容易发现其单调性。为了实现转移的优化，我们考虑哪些决策可能成为最优决策。

引理：

若一个决策$j(0 \le j<i \text{且} s_i-s_j \le mid)$是最优决策，则$j$一定满足以下**两个条件之一**：

1．$a_j=\max\limits_{j \le k \le i}\{a_k\}$

2．$s_i-s_{j-1}>mid$（即j是满足$s_i-s_j \le mid$的最小的j）

证明：

反证法。假设两个条件都不满足，即 $a_j < \max_{j \le k \le i}\{a_k\}$ 且 $s_i - s_{j-1} \le mid$ 。

则 $\max_{j \le k \le i}\{a_k\} = \max_{j < k \le i}\{a_k\}$ 且 $j-1$ 是一个合法决策，又因为 $f[j-1] \le f[j]$，所以有：

$$f[j-1] + \max_{j \le k \le i}\{a_k\} \le f[j] + \max_{j < k \le i}\{a_k\}$$

即决策 $j-1$ 比 j 优，与 j 是最优决策矛盾，故假设不成立，原命题得证。

证毕。

得到这个引理之后，我们就可以对满足两个条件的决策分别维护，最后取最优决策即可。

对于条件二，我们只要维护一个指针，使得这个指针始终指向满足 $s_i - s_j \le mid$ 的最小的 j，并随着 i 的增加，将这个指针也更新右移即可。

对于条件一，情况就比较复杂。

我们可以发现，若我们将满足条件一的决策 j 构成一个队列，使得**决策点 j 单调递增**，则根据引理，a_j 应当**单调不增**。若我们想要在队尾加入一个新的决策点 j_0，对于队尾满足 $a_j < a_{j_0}$ 的决策 j，决策 j 不满足条件一，不可能成为最优决策，则不断将队尾这样的不合法决策弹出，最后将决策点 j_0 加入队尾，这样就维护了队列中 j 单调递增，a_j 单调不增的单调性。

但是这个队列只维护了满足条件一的所有决策，并没有维护 $f[j] + \max_{j<k\le i}\{a_k\}$ 的单调性。

我们假设在得到 $f[i]$ 之前，先将决策 i 插入队尾。

不难发现，对于不在队尾的一个决策 j，$\max_{j<k\le i}\{a_k\}$ 其实就等于它在队列中的下一个元素值。所以在将决策 i 插入队尾前，我们应当用 i 维护原来在队尾的决策 j 的 $\max_{j<k\le i}\{a_k\}$。

我们可以建立一个数据结构，维护队列中非队尾决策的 $f[j] + \max_{j<k\le i}\{a_k\}$，支持查询最小值，并且随着队列的变化，支持弹出插入固定元素。

每次我们只要在数据结构中查询 $f[j] + \max_{j<k\le i}\{a_k\}$ 的最大值，来更新 $f[i]$。

我们可以用堆、线段树、平衡树等数据结构实现。

下面的参考程序使用 STL 的 multiset 来实现，插入删除查询的单次时间复杂度都是 $O(\log n)$，所以二分判断的时间复杂度就是 $O(n\log n)$ 的，总时间复杂度就是 $O(n\log^2 n)$，期望得分 100 分。

【参考程序】

//陈煜翔

```cpp
#include <set>
#include <cmath>
#include <cstdio>
#include <cctype>
#include <cstdlib>
#include <cstring>
#include <iostream>
#include <algorithm>

typedef long long s64;

template <class T>
inline void read(T &x)
{
    static char ch;
    static bool opt;
    while (!isdigit(ch = getchar()) && ch != '-');
    x = (opt = ch == '-') ? 0 : ch - '0';
    while (isdigit(ch = getchar()))
        x = x * 10 + ch - '0';
    if (opt) x = ~x + 1;
}

const int MaxN = 1e5 + 5;
const int MaxLog = 18;
const int INF = 0x3f3f3f3f;

std::multiset<s64> min_set;

int n, tot;
int logval[MaxN], maxa[MaxN], minb[MaxLog + 1][MaxN];
//maxa[i]记录的是 a 的后缀最大值的序号
//minb[k][i]记录的是[i,i+2^k-1]b 的最大值
s64 m, a[MaxN], b[MaxN], s[MaxN], f[MaxN];

inline int maxpos(const int &x, const int &y)
{
    return a[x] > a[y] ? x : y;
}

inline int getminb(const int &l, const int &r)
{
    int k = logval[r - l + 1];
    return std::min(minb[k][l], minb[k][r - (1 << k) + 1]);
```

```
}

inline bool check(const s64 &mid)
{
    min_set.clear();
    memset(f, 0x3f, sizeof(f));

    static int q[MaxN], head, tail;
    q[head = tail = 1] = 0;
    f[0] = 0;

    int le = 0;
    //le 指向第一个满足 s[i]-s[j]<=mid 的 j
    for (int i = 1; i <= n; ++i)
    {
        while (head <= tail && s[i] - s[q[head]] > mid)
        {   //弹出队首不满足 s[i]-s[j]<=mid 的决策
            if (head < tail)
                min_set.erase(min_set.find(f[q[head]] + a[q[head + 1]]));
            ++head;
        }
        while (head <= tail && a[q[tail]] < a[i])   //维护队列 a 的不增性
        {
            if (head < tail)
                min_set.erase(min_set.find(f[q[tail - 1]] + a[q[tail]]));
            --tail;
        }
        if (head <= tail)
            min_set.insert(f[q[tail]]+a[i]); //维护原队尾的决策转移值
        q[++tail] = i;

        while (le < i && s[i] - s[le] > mid)
            ++le;    //维护 le 指针，用满足条件二的决策更新 f[i]
        if (le == q[head])                  //若 le 指向队首元素
            f[i] = f[le] + a[q[head + 1]];  //则最大值应取队首后一个元素
        else
            f[i] = f[le] + a[q[head]];      //否则最大值取队首元素
        if (head < tail)  //查询满足条件一的最优决策更新 f[i]
            f[i] = std::min(f[i], *min_set.begin());
    }
    return f[n] <= m;
}

int main()
```

```
{
    freopen("sequence.in", "r", stdin);
    freopen("sequence.out", "w", stdout);

    read(n), read(m);
    logval[0] = -1;
    for (int i = 1; i <= n; ++i)
    {
        read(a[i]), read(b[i]);
        logval[i] = logval[i >> 1] + 1;
        minb[0][i] = b[i];
    }
    maxa[n] = n;
    for (int i = n - 1; i >= 0; --i)
        maxa[i] = maxpos(maxa[i + 1], i);
    //a 的后缀最大值预处理
    for (int j = 1; j <= MaxLog; ++j)
        for (int i = 1; i + (1 << j) - 1 <= n; ++i)
            minb[j][i] = std::min(minb[j-1][i], minb[j-1][i + (1 << j -
1)]);
    //b 的 RMQ 预处理
    for (int i = 1; i <= n;)
    {
        int j;
        for (j = i; j < n && a[maxa[j+1]] >= getminb(i,j); j = maxa[j+1]);

        a[++tot] = a[i];
        b[tot] = b[i];
        for (i = i + 1; i <= j; ++i)
        {
            a[tot] = std::max(a[tot], a[i]);
            b[tot] += b[i];
        }
    }
    n = tot;
    //将必定在同一段的每个区间缩成数对
    s64 l = 0, r = 0, mid, res = 0;
    for (int i = 1; i <= n; ++i)
    {
        s[i] = s[i - 1] + b[i];
        l = std::max(l, b[i]);
        r += b[i];
    }
    while (l <= r)
```

```
    {
        mid = l + r >> 1;
        if (check(mid))
            r = mid - 1, res = mid;
        else
            l = mid + 1;
    }
    //二分答案
    std::cout << res << std::endl;

    fclose(stdin);
    fclose(stdout);
    return 0;
}
```