交与并(interval,2s,512MB)

【算法分析】

首先证明一个命题：最大价值所包含的区间个数为2。

证明：由题设知$k>1$。假设最大价值包含k个区间$(k>2)$，那么若我们把除最左、最右以外的区间删去，将会得到一个并集不变、交集不减小的新的区间集合，显然这个集合的价值比原集合更优，假设不成立。命题得证。

若选择的最左、最右的区间是同一个区间，由于$k>1$，所以我们需要再选择一个最大的被该区间所包含的区间，尝试更新答案。

若选择的最左、最右区间不是同一个区间，此时选择一个被包含的区间显然不优。因此可以把所有被包含的区间都删去。为方便叙述，我们将剩余的区间从左到右排序，以下所有区间编号都建立在已排好序的基础上。我们称被选中的一对区间中的编号较小区间为编号较大区间的一个匹配。容易发现对于每个区间，它的最优匹配满足决策单调（即第i个区间的最优匹配一定不在第$i-1$个区间的最优匹配的左边），如何证明？

反证法：设第i个区间的最优匹配为第j个区间，第$i-1$个区间的最优匹配为第j'个区间，假设$j<j'$，则：

对于第i个区间，有：

$$(r[i]-l[j])(r[j]-l[i])\geq(r[i]-l[j'])(r[j']-l[i])$$
$$r[i](r[j]-r[j'])+l[i](l[j]-l[j'])-l[j]r[j]+l[j']r[j']\geq 0$$

对于第$i-1$个区间，有：

$$(r[i-1]-l[j'])(r[j']-l[i-1])\geq(r[i-1]-l[j])(r[j]-l[i-1])$$
$$r[i-1](r[j']-r[j])+l[i-1](l[j']-l[j])-l[j']r[j']+l[j]r[j]\geq 0$$

两式相加，得：

$$(r[i]-r[i-1])(r[j]-r[j'])+(l[i]-l[i-1])(l[j]-l[j'])\geq 0$$

又$r[j]<r[j']<r[i-1]<r[i], l[j]<l[j']<l[i-1]<l[i]$，则：

$$(r[i]-r[i-1])(r[j]-r[j'])+(l[i]-l[i-1])(l[j]-l[j'])<0$$

与上式矛盾，假设不成立。命题得证。

于是区间的最优匹配满足决策单调，可用分治求解。我们记参数(l,r,x,y)表示编号为$[l,r]$的区间对应的最优匹配的编号在$[x,y]$内。对于第$mid=\left\lfloor\frac{l+r}{2}\right\rfloor$个区间，找出其对应的最优匹配的编号$pos$，由最优匹配的决策单调性可知，编号为$[l,mid-1]$和编号为$[mid+1,r]$的区间对应的最优匹配的编号分别在$[x,pos]$和$[pos,y]$内，递归求解即可。注意判断无最优匹配的情况。

时间复杂度$O(n\log n)$。

【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;
}

const int MaxN = 1e6 + 5;
int n, m;
s64 ans;
struct seg
{
    int l, r;

    inline void scan()
    {
```

```
        get(l), get(r);
        return;
    }

    inline bool operator < (const seg &rhs) const
    {
        return l < rhs.l || l == rhs.l && r > rhs.r;
    }
} a[MaxN], b[MaxN];

template <class T>
inline void ckmax(T &x, T y)
{
    if (x < y) x = y;
    return;
}

inline s64 calc(seg a, seg b)
{
    return 1LL * (a.r - b.l) * (b.r - a.l);
}

inline void solve(int l, int r, int x, int y) //编号为[l,r]的区间只可能与编号为[x+1,y]的区间匹配
{
    if (l > r)
        return;

    int mid = l + r >> 1;
    if (x >= mid)
        return solve(mid + 1, r, x, y);

    int pos = x;
    s64 res = calc(b[x], b[mid]);
    For(i, x + 1, mid - 1)
    {
        s64 tmp = calc(b[i], b[mid]);
        if (tmp > res)
        {
            res = tmp;
            pos = i;
        }
    }
```

```
    ckmax(ans, res); //用区间 pos 与区间 mid 匹配的价值更新答案，并递归求编号为[l,mid-1]、[mid+!,r]的区间的匹配
    solve(l, mid - 1, x, pos);
    solve(mid + 1, r, pos, y);
    return;
}

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

    get(n);
    For(i, 1, n)
        a[i].scan();
    std::sort(a + 1, a + n + 1);

    b[m = 1] = a[1];
    For(i, 2, n) //删去被包含的区间
    if (b[m].r >= a[i].r) //被包含的区间只有与包含它的区间组合时，才有可能产生答案
        ckmax(ans, 1LL * (b[m].r - b[m].l) * (a[i].r - a[i].l));
    else
        b[++m] = a[i];

    solve(1, m, 1, m); //分治求互不包含的 m 个区间相互匹配的答案
    put(ans, '\n');

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

共享单车(bike,1s,128MB)

【算法分析】

算法一

很容易想到设 f[i][j]表示已经考虑了前 j 个部分的居民且第 i 个存车点设置在第 j 个部分的麻烦度之和，转移显然为 $f[i][j]=\min_{1\le k<j}\{f[i-1][k]+Cost(k,j)\}$。

即枚举第 i-1 个存车点放置的位置 k，其中 Cost(k,j)表示第 k 个部分到第 j 个部分的居民的麻烦度之和。显然第 k 个部分到第 j 个部分中存在一个分界点，使得这个分界点以及分界点左边的居民距离第 i-1 个存车点最近，分界点右边的居民距离第 i 个存车点最近，并且随着我们枚举 k 的增大，这个分界点只可能向右移动。我们记录一个指针表示分界点，预处理相关前缀和即可 $O(1)$ 计算 Cost(k,j)。

初值为 $f[1][i]=Cost(1,i)$，最后的答案为 $\min_{K\le j\le n}\{f[K][j]+Cost(j,n)\}$（显然设置的存车点越多答案只可能越优）。

时间复杂度 $O(n^2K)$，期望得分 60 分。

算法二

不难发现上述转移满足决策单调。我们在最外层枚举 i，对于 j 的转移维护一个单调队列，单调队列中的元素是一个三元组(l,r,k)，表示当前[l,r]内的最优转移为 k，每次把队首 r<j 的元素弹出，得到的即为对于当前 j 的最优转移。在队尾插入一个新的元素时，可能会有若干个队尾元素被弹出以及一个队尾元素的[l,r]被修改。修改的部分需要通过二分来找到新的区间范围，并且计算一种决策的贡献也需要通过二分来找到分界点。

时间复杂度 $O(nK\log^2 n)$，期望得分 80 分。

算法三

注意到如果我们把两个存车点之间的分界点也看做一个转移点，那么我们在转移的时候就不需要考虑每个部分的居民是否都被恰好分配到距离最近的存车点。

考虑这种做法的正确性，如果最后得到的方案中每个居民不是都被恰好分配到距离最近的存车点，那么我们显然可以通过位移分界点来使答案变得更优，这与原来的方案是最优方案矛盾。

于是另外设 g[i][j]表示已经考虑了前 j 个部分的居民并且第 i 个分界点在第 j 个部分的麻烦度之和。考虑 f 和 g 的转移，我们预处理三个数组：

$$c[i]=\sum_{j=1}^{i}a_j,d[i]=\sum_{j=1}^{i-1}l_j,s[i]=\sum_{j=2}^{i}a_jl_{j-1}$$

则转移即为：

$$g[i][j]=\min_{1\le k\le j}\{f[i][k]-(c[j]-c[k])\times d[k]+s[j]-s[k]\}$$

$$f[i][j]=\min_{1\le k\le j}\{g[i-1][k]+(c[j]-c[k])\times d[j]-s[j]+s[k]\}$$

注意存车点和分界点可以重合，所以这里的 j,k 可以相等。

f 和 g 的转移中同时与 j,k 有关的项都只有一项，且是两个下标为 j,k 的量相乘的形式，显然可以转化成斜率优化的模型，用单调队列维护凸壳来优化转移。

为方便起见，我们在第 K 个存车点之后再设置一个恰好在第 n 个部分的分界点，最后的答案即为 g[K][n]。

时间复杂度$O(nK)$，期望得分 100 分。

【参考程序】

```
//陈贤
#include <bits/stdc++.h>

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;
}

typedef long long ll;
typedef long double ld;
const int N = 5e4 + 5;
const int M = 22;
ll f[M][N], g[M][N], s[N], Y[N];
int n, K, t, w, h[N];
int X[N], a[N], c[N], d[N];

inline bool check1(int i, int j, ll k)
{
    return (Y[j] - Y[i]) <= k * (X[j] - X[i]);
}

inline bool check2(int i, int j, int k)
{
    return (Y[j] - Y[i]) / (ld)(X[j] - X[i])
        >= (Y[k] - Y[j]) / (ld)(X[k] - X[j]);
}

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

    read(n); read(K);
    for (int i = 1; i <= n; ++i)
```

```
    read(a[i]);
for (int i = 1; i <= n; ++i)
    c[i] = c[i - 1] + a[i];
for (int i = 2; i <= n; ++i)
    read(d[i]), d[i] += d[i - 1];
for (int i = 1; i <= n; ++i)
    s[i] = s[i - 1] + 1ll * a[i] * d[i];

for (int j = 1; j <= n; ++j)
    f[1][j] = 1ll * c[j] * d[j] - s[j];
for (int i = 1; i <= K; ++i)
{
    if (i != 1)
    {
        int t = 1, w = 0;
        for (int j = 1; j <= n; ++j)
        {
            X[j] = c[j];
            Y[j] = g[i - 1][j] + s[j];
            while (t < w && check2(h[w - 1], h[w], j))
                --w; //维护凸壳斜率递增
            h[++w] = j;
            while (t < w && check1(h[t], h[t + 1], d[j]))
                ++t; //每次通过凸壳上相邻点的斜率找到最优转移
           f[i][j]=g[i-1][h[t]]+1ll*(c[j]-c[h[t]])*d[j]-s[j]+s[h[t]];
        }
    }
    t = 1, w = 0;
    for (int j = 1; j <= n; ++j)
    {
        X[j] = d[j];
        Y[j] = f[i][j] + 1ll * c[j] * d[j] - s[j];
        while (t < w && check2(h[w - 1], h[w], j))
            --w;
        h[++w] = j;
        while (t < w && check1(h[t], h[t + 1], c[j]))
            ++t;
        g[i][j] = f[i][h[t]]+s[j]-s[h[t]]-1ll*(c[j]-c[h[t]])*d[h[t]];
    }
}
std::cout << g[K][n] << std::endl;

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

```
}
```

01 背包(jewelry,3.5s,512MB)

【算法分析】

观察数据，发现不同的体积最多 300 种，考虑把体积相同的放一起处理，然后按价值从大到小取，即按体积从小到大排序，体积相同的从大到小排序。

排序之后，取出每段体积相同的物品。考虑利用之前的状态，再取若干个这样的物品来更新一些状态。

设 f[s][h]表示，目前取到的最大物品体积为 s，背包容量为 h 的最大价值，记上一段物品的体积为 t。显然 f[s][h]可以直接从 f[t][h]转移过来，也可以从 f[s][i]转移过来。显然耗时最多的是第 2 个转移。

考虑如何优化第 2 个转移：如果 i<j，显然有 f[s][i]≤f[s][j]。决策 i 表示从 f[s][i]转移过来，决策 j 表示从 f[s][j]转移过来。假设此时的 h 很接近 j，体积为 s 的物品中，价值最大的为 v1，第 2 大的为 v2。如果选决策 i，h-i 的容量能放一个体积为 s 的且价值最大的物品。但是如果选决策 j，h-j 的容量不够放这个物品，然后两个决策的价值差为：f[s][i]+v1-f[s][j]。

如果以后 h 变成了 h+s。那么此时从 f[s][i]转移过来能放两个体积为 s 的物品，从 f[s][j]转移过来能放一个体积为 s 的物品。那么两个决策的价值差为：f[s][i]+v1+v2-(f[s][j]+v1)。化简即 f[s][i]+v2-f[s][j]。

于是可以发现：随着 h 的增大，决策的价值差是不上升的，也就是说如果刚开始决策 i 比决策 j 优，h 变大之后，两个决策的价值大小关系可能会反转。

那么考虑用一个单调队列来维护并选取最优决策，当决策 i,j 的大小关系反转时将 i 移出队列。它们关系反转的时间可以利用二分查找。

如果队首元素和下一个元素的反转时间到了，那么弹出队首元素。

如果队尾的 2 个决策点为 a,b，准备新加入一个决策点 c(a<b<c)。a,b 的反转时间比 b,c 要晚，那么 b 永远不会作为最优决策，因为如果 a,b 反转之后，a 移出了队列，但是这时候 b,c 的反转时间早就到了，马上轮到 c 作为最优决策。所以在这种情况下，先把 b 从队尾移出，再加入 c。

可以发现这个单调队列的单调性体现在 f[s][i]关于 i 单调不下降，相邻两个决策的反转时间关于 i 单调不下降。当然，只有在 i 对 s 取模的值相同的情况下，这个单调性才成立。所以每次把容量对 s 取模的值相同的放一起处理。

记 p 为体积种数，则时间复杂度 O(pm+n log n)。

【参考程序】//陈贤

```
#include <bits/stdc++.h>

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;
}

template <class T>
```

```
inline void put(T x)
{
    if (x > 9) put(x / 10);
    putchar(x % 10 + 48);
}

typedef long long ll;
const int Maxn = 0x3f3f3f3f;
const int N = 1e6 + 5, M = 1e5 + 5, L = 305;
ll f[L][M], sum[N]; int h[M], n, m, tot;

template <class T>
inline void CkMax(T &x, T y) {if (x < y) x = y;}
template <class T>
inline T Min(T x, T y) {return x < y ? x : y;}

struct thing
{
    int s, v;

    inline void scan()
    {
        read(s); read(v);
    }

    inline bool operator < (const thing &a) const
    { //体积相同的物品相邻，并且按价值降序排
        return s < a.s || s == a.s && v > a.v;
    }
}p[N];

inline ll calc(int s1, int i, int t)
{ //计算从容量为 i 的状态转移到容量为 i+t*p[i].s 的状态的价值
    return f[s1][i] + sum[t];
}

inline int find_time(int s1, int s2, int i, int j)
{ //二分计算决策 i,j 的反转时间
    int tmp = (j - i) / s2, l = 0,
        r = Min(tot, (m - j) / s2), res = -1;
    //这里先二分的是选取物品的个数，最后再计算反转时间，减小复杂度
    while (l <= r)
    {
        int mid = l + r >> 1;
```

```
        if (calc(s1, i, mid + tmp) <= calc(s1, j, mid))
            res = mid, r = mid - 1;
        else
            l = mid + 1;
    }
    return res == -1 ? Maxn : j + res * s2;
}

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

    read(n); read(m);
    for (int i = 1; i <= n; ++i)
        p[i].scan();

    std::sort(p + 1, p + n + 1);
    for (int i = 1; i <= n; ++i)
        if (p[i].s != p[i - 1].s) //将体积相同的放在一起处理
        {
            tot = 0;
            for (int j = i; j <= n && p[j].s == p[i].s; ++j)
                ++tot, sum[tot] = sum[tot - 1] + p[j].v;
            for (int j = 0; j <= m; ++j)
                f[p[i].s][j] = f[p[i - 1].s][j];
            for (int j = 0; j < p[i].s; ++j) //将模 a[i].s 同余的容量放在一
起处理
            {
                int t = 1, w = 1; h[1] = j;
                for (int k = j + p[i].s; k <= m; k += p[i].s)
                {//用单调队列维护决策单调性
                    while (t < w && find_time(p[i - 1].s, p[i].s, h[t], h[t
+ 1]) <= k) ++t;
                    CkMax(f[p[i].s][k], calc(p[i - 1].s, h[t], Min(tot, (k
- h[t]) / p[i].s)));
                    while (t < w && find_time(p[i - 1].s, p[i].s, h[w - 1],
h[w])
                                >= find_time(p[i - 1].s, p[i].s, h[w], k))
--w;
                    h[++w] = k;
                }
            }
    }
```

```
    for (int i = 1; i <= m; ++i)
        put(f[p[n].s][i]), putchar(' ');

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