Soluciones entrenos OIE
Shoes

Solución

Primero, resolveremos el problema en el caso específico en el que todos los zapatos son de tamaños distintos y además el zapato izquierdo siempre está a la derecha del zapato derecho. En este caso, lo que queremos hacer es juntar cada zapato con su pareja. Por cada par de zapatos del mismo tamaño p, consideremos la distancia $d(p)$ que hay entre los dos zapatos (el número de intercambios mínimos para juntarlos), y denotemos por $\sum d(p)$ la suma de todas las distancias para todos los pares. Entonces, la cantidad $\sum d(p)$ puede reducirse como máximo en 2 por cada intercambio que hagamos (en un intercambio, podemos acercar cada uno de los zapatos que movemos hacia su pareja en 1 unidad). Por tanto, como la suma de distancias tiene que ser 0 al final,

$$S_1 = \frac{\sum d(p)}{2}$$

es una cota inferior del número de intercambios que tenemos que hacer. Pero hay casos en los que tenemos que hacer más intercambios, por ejemplo $[-2, -1, 1, 2]$, donde tenemos que hacer dos intercambios y no uno. Vemos que si hay un par de zapatos del mismo tamaño que esté entre otro par de zapatos del mismo tamaño, necesariamente en algún momento tenemos que hacer un intercambio que no acerque a los dos zapatos que intercambiamos, y por tanto necesitaremos más de S_1 intercambios. Ahora, para un par de zapatos del mismo tamaño p sea $e(p)$ el número de pares de zapatos del mismo tamaño que están colocados entre los dos zapatos de p. Consideremos la cantidad:

$$S_2 = \frac{\sum d(p)}{2} + \sum e(p)$$

Esta cantidad baja como mucho en 1 en cada intercambio: si los dos zapatos que intercambiamos tienen su pareja correspondiente en direcciones opuestos, entnces $\sum d(p)$ sube o baja en 2 y $\sum e(p)$ se queda igual, si tienen su pareja en la misma dirección, $\sum d(p)$ se queda igual y $\sum e(p)$ sube o baja en 1. Así que esta cantidad es una cota inferior más refinada del número de intercambios, pero también es una cota superior porque siempre podemos hacer que baje en 1 (si no es igual a 0): si movemos un zapato en la dirección hacia su pareja, podemos comprobar que siempre baja en 1. Por tanto, la respuesta que buscamos es S_2.

Ahora consideremos el caso general. En primer lugar, vemos qué pasa si el zapato izquierdo está a la derecha del derecho. Necesitaremos un intercambio adicional para poner bien el orden los dos zapatos. Este intercambio es independiente de los otros, así que simplemente tenemos que sumar 1 a la cantidad S_2 por cada par de zapatos en orden cambiado. En segundo lugar, si hay más de un par de zapatos con el mismo tamaño, notemos que siempre es óptimo emparejar el primer zapato izquierdo (en cuanto a posición) con el primer zapato derecho, el segundo con el segundo, etcétera. Si no, en algún momento tendríamos que intercambiar dos zapatos iguales, cosa que claramente es un intercambio inútil. Por tanto, lo que podemos hacer es para cada tamaño emparejar el i-ésimo zapato izquierdo con el i-ésimo derecho

y hacer como si cada par de estos fueran de tamaño distinto. Por tanto, podemos reducir el caso general al caso específico que hemos resuelto.

Ahora vemos cómo calcular S_2 eficientemente. $d(p)$ lo podemos calcular fácilmente por cada par de zapatos almacenando el índice del primer y del segundo zapato del par. $e(p)$ es más complicado: hay que contar cuántas parejas de zapatos hay en el intervalo de posiciones entre los dos zapatos. Podemos ir actualizando en un *segment tree* cada vez que completamos una pareja sumando un 1 en la posición del zapato más a la izquierda y entonces calcular $e(p)$ es hacer la suma de los 1s que hemos puesto en el intervalo entre los dos zapatos, para más detalles, ver el código. Existe otra estructura de datos llamada *Fenwick tree* que también permite también hacer cálculos de este tipo pero con una implementación más sencilla.

Código

C++

```cpp
#include "shoes.h"

using namespace std;

typedef long long ll;
typedef vector<int> vi;
typedef vector<vi> vvi;


//Funcion que transforma el vector que nos dan
//emparejando los zapatos del mismo tamano y asignando
//un nuevo tamano para cada uno de ellos para que solo haya
//un par de cada tamano.
vi transform(vi s) {
        int n = s.size();

        //v sera el nuevo vector
        vi v(n);

        //Para cada tamano guardamos cuantos izquierdos y cuantos derechos
        //nos hemos encontrado y que nuevos tamanos hemos asignado.
        //Aqui usamos n como cota superior de numero de tamanos.
        vi lc(n);
        vi rc(n);
        vvi transf(n);

        int tam = 1;

        for(int i=0; i < n; ++i) {
                if(s[i] < 0) {
                        int x = -s[i];
                        //Si este zapato corresponde a una nueva pareja, le
                        ↪  asignamos un nuevo
                        //tamano. Si no, le asignamos el de su pareja.
                        if(lc[x] >= rc[x]) {
                                transf[x].push_back(tam);
                                v[i] = -tam;
                                tam++;
                        }
```

```
                        else {
                                v[i] = -transf[x][lc[x]];
                        }
                        lc[x]++;
                }
                else {
                        int x = s[i];
                        if(rc[x] >= lc[x]) {
                                transf[x].push_back(tam);
                                v[i] = tam;
                                tam++;
                        }
                        else {
                                v[i] = transf[x][rc[x]];
                        }
                        rc[x]++;
                }
        }

        return v;
}

//Funciones del segment tree

vi st;

void st_update(int p, int l, int r, int i) {
        if(l > i || r < i) return;
        if(l == i && r == i) {
                st[p] += 1;
        }
        else {
                st_update(2*p, l, (l+r)/2, i);
                st_update(2*p+1, (l+r)/2+1, r, i);
                st[p] = st[2*p]+st[2*p+1];
        }
}

int st_query(int p, int l, int r, int i, int j) {
        if(l > j || r < i) return 0;
        if(l >= i && r <= j) {
                return st[p];
        }
        else {
                return st_query(2*p, l, (l+r)/2, i, j)+st_query(2*p+1, (l+r)/2+1,
                 ↪  r, i, j);
        }
}


//Funcion que cuenta la respuesta, asumiendo que ahora solo hay un par de cada
 ↪  tamano
//(tambien se podria haber implementado de forma que no requiriera la funcion
 ↪  transform previa)
ll count(vi s) {
```

```
91
92          int n = s.size();
93
94          st = vi(4*n, 0);
95
96          //Calcularemos el doble de la respuesta y al final dividiremos entre dos
97          //para no tener que hacer divisiones entre dos por el medio
98          ll ans = 0;
99
100         //Aqui guardamos la posicion del primer zapato de la pareja
101         vi prev_pos(n, -1);
102
103         for(int i=0; i < n; ++i) {
104                 int x = s[i];
105                 if(x < 0) x = -x;
106                 if(prev_pos[x] == -1) {
107                         prev_pos[x] = i;
108                 }
109                 else {
110                         //Si estamos en el segundo zapato de la pareja, sumamos a
                            ↪  la respuesta d(p) y 2*e(p)
111                         ans += (i-prev_pos[x]-1)+2*st_query(1, 0, n-1,
                            ↪  prev_pos[x], i);
112                         //Si la pareja esta cambiada de orden hay que sumar un
                            ↪  (multiplicado por 2) intercambio extra
113                         if(s[i] < 0) ans+=2;
114                         //Actualizamos el segment tree para que se tenga en cuenta
                            ↪  esta pareja al calcular el e(p) de parejas posteriores
115                         st_update(1, 0, n-1, prev_pos[x]);
116                 }
117         }
118
119         ans /= 2;
120
121         return ans;
122
123 }
124
125 ll count_swaps(vi s) {
126         return count(transform(s));
127 }
```