Clasificatorio Abierto 2025 - Soluciones

Camino

Autor: Pablo Sáez Reyes

Para explicar la solución del problema voy a dar antes varias observaciones:

Observación 1: Podemos descomponer el problema en los ejes Este-Oeste y Norte-Sur, que son independientes. Combinar los resultados para los ejes es sencillo, luego me centraré en el eje Este-Oeste por ejemplo.

Observación 2: El tener que moverse al menos una unidad de distancia cuando desconocemos cuanto nos hemos movido es molesto. Si descomponemos los tramos de longitud desconocida en una parte fija de uno y otra con longitud ≥ 0, entonces es más sencillo de manejar. En adelante supondré que es así.

Observación 3: El órden de los movimientos es irrelevante, supondré que primero muevo las longitudes conocidas y luego las desconocidas.

Usando las observaciones usando los datos puedo obtener la posición en la que me encuentro tras moverme con las cantidades conocidas.

A la hora de ver si puedo regresar con los movimientos de longitud desconocida hay tres casos posibles.

- Me encuentro a la izquierda (oeste) del punto de partida: Podré llegar a él si y solo si tengo al menos un movimiento de longitud desconocida a la derecha (este).
- Me encuentro a la derecha (este) del punto de partida: Podré llegar a él si y solo si tengo al menos un movimiento de longitud desconocida a la izquierda (oeste).
- Me encuentro en el punto de partida: Siempre puedo volver al punto de partida (moviéndome 0 en los tramos restantes).

Si puedo regresar al orígen veo que la distancia mínima que puedo recorrer con la parte no fija es ir directamente desde donde estoy.

Es decir la distancia mínima total es la distancia recorrida fija más la distancia al punto de partida.

Por último, necesito ver cual es la distancia máxima recorrida.

Si la distancia recorrida no es fija (es la misma para todos los caminos posibles que cumplen las condiciones), la distancia adicional recorrida respecto de la mínima viene de ir a la derecha una distancia extra y luego ir a la izquierda esa misma distancia respecto a las longitudes mínimas. Eso implica que hay al menos un movimiento de longitud desconocida a la derecha y otro a la izquierda. Además podemos ver que si existen esos movimientos, los podemos usar para hacer la distancia recorrida arbitrariamente grande.

Por tanto hemos demostrado que la distancia recorrida es fija (y por tanto igual a la mínima) si y solo si no existen dos tramos ilimitados en direcciones opuestas, y en caso contrario la distancia recorrida es ilimitada.

Para programar esto basta recorrer todos los movimientos $O(n)$ y almacenar el desplazamiento fijo y las direcciones en las que me puedo mover ilimitadamente. El resto es comprobar casos en $O(1)$. La complejidad total es $O(n)$.

```cpp
#include <bits/stdc++.h>
#define int long long
using namespace std;

signed main()
{
    int t;
    cin >> t;
    while (t--) {
        int n;
        cin >> n;
        int dx = 0;
        int dy = 0;
        bool hn = false;
        bool hs = false;
        bool he = false;
        bool ho = false;
        int rec = 0;
        for (int i = 0; i < n; ++i) {
            char d;
            int l;
            cin >> d >> l;
            if (d == 'N') {
                if (l == -1) {
                    l = 1;
                    hn = true;
                }
                dx += l;
            }
            if (d == 'S') {
                if (l == -1) {
                    l = 1;
                    hs = true;
                }
                dx -= l;
            }
            if (d == 'E') {
                if (l == -1) {
                    l = 1;
                    he = true;
                }
                dy += l;
            }
            if (d == 'O') {
                if (l == -1) {
                    l = 1;
                    ho = true;
                }
                dy -= l;
            }
            rec += l;
        }
        rec += abs(dx)+abs(dy);
        bool pos = (dx == 0 || (dx > 0 && hs) || (dx < 0 && hn)) &&
                   (dy == 0 || (dy > 0 && ho) || (dy < 0 && he));
        if (!pos) {
            cout << "-1 -1";
        }
        else {
            cout << rec << ' ';
            bool inf = (hn && hs || he && ho);
            if (inf) cout << -1;
            else cout << rec;
        }
        cout << '\n';
    }
}
```

Pequeño contexto del problema:

Originalmente propuesto para la regional de Castilla y León.

Este problema está inspirado por los problemas en los que te dan una figura en la que todos sus ángulos son rectos y algunos de los lados (pero no todos), y te piden dar el perímetro. Por ejemplo el problema 5 de la Olimpiada Matemática de 2º de la ESO de 2019.

Esos problemas son un poco más difíciles porque no hay una dirección implícita al ver el dibujo. Recorrerlo en uno de los dos sentidos posibles nos lleva al problema aquí planteado.

Originalmente iba a tratar sobre la longitud de una valla (como el problema mencionado antes), pero la posible autointersección de un recorrido llevó a la interpretación actual.

Una generalización interesante para pensar es: qué pasa si en vez de darme una dirección cardinal me dan un vector cualquiera (incluso en una dimensión cualquiera).

Tareas

Autora: Elena V. R.

Usamos la técnica de programación dinámica. Sea $dp[i][j]$ igual al mínimo tiempo necesario para completar j tareas, teniendo en cuenta sólo las primeras $i-1$ tareas. Tenemos la siguiente relación de recurrencia:

$$dp[i+1][j+1] = \begin{cases} \mathbf{mín}(dp[i][j+1], dp[i][j] + t_i) & \text{si } dp[i][j] \leq p_i \\ dp[i][j+1] & \text{si } dp[i][j] > p_i \end{cases}$$

Usándola, podemos calcular la DP en tiempo $O(n^2)$ (y memoria $O(n)$ si eliminamos la primera dimensión). La respuesta es el mayor valor de j tal que $dp[n+1][j] < \infty$.

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

using namespace std;

const int INF = 1000000001;

int main() {
    int T;
    cin >> T;
    while (T--) {
        int n;
        cin >> n;
        vector<int> t(n);
        vector<int> p(n);
        for (int i=0; i < n; ++i) cin >> t[i] >> p[i];
        vector<int> dp(n+1, INF);
        dp[0] = 0;
        for (int i=0; i < n; ++i) {
            for (int j=n-1; j > -1; --j) {
                if (dp[j] <= p[i]) {
                    dp[j+1] = min(dp[j+1], dp[j] + t[i]);
                }
            }
        }
        int ans = n;
        while (dp[ans] == INF) ans--;
        cout << ans << endl;
    }
}
```

Codificación

Autor: Félix Moreno Peñarrubia

Solución en $3n + 2$ bits:

La codificación consiste en repetir tres veces el mensaje, poniendo un 0 de separación entre cada repetición.

Para decodificar, recuperamos el valor de n e identificamos qué bits eran originalmente los 0 de separación. Entonces:

- Si los dos bits son ahora 1, el intervalo invertido cubre totalmente la segunda repetición del mensaje, por tanto la respuesta es el segundo bloque invertido.
- Si uno de los dos bits es 0, podemos saber que la primera o la última repetición no ha sido alterada, por lo que damos ese bloque como respuesta.
- Si los dos bits son 0, entonces el intervalo invertido está estrictamente dentro de uno de los bloques, por tanto quedan dos bloques iguales entre sí que serán iguales al mensaje.

Esta solución ya proporciona $99{,}06$ puntos y es la mejor solución "sencilla" que hemos encontrado. Era la principal solución que esperábamos de los concursantes durante el concurso.

Solución en menos bits:

Primero, reducimos el problema a otro problema más conocido: el de codificar una cadena binaria de forma que se puedan corregir un pocos errores en posiciones individuales.

Dada una cadena binaria S de longitud n, podemos considerar su correspondiente *cadena de diferencias consecutivas* $\Delta(S)$ de longitud $n - 1$, donde el i-ésimo bit de $\Delta(S)$ es 1 si el i-ésimo y el $(i + 1)$-ésimo bit de S son diferentes, y 0 si son iguales. Nótese que invertir un intervalo de S equivale a invertir dos bits de $\Delta(S)$. Por tanto, podemos reducir el problema a encontrar una codificación C de S que sea descodificable aunque haya dos errores en posiciones aleatorias (la codificación que enviaremos será C' tal que $\Delta(C') = C$). Una codificación sencilla para este caso sería repetir cada bit 5 veces, lo que nos da una solución de longitud $5n + 1$.

La teoría de los códigos correctores de errores nos da varias opciones muy sofisticadas para construir una codificación C con longitud menor a $2n$ que funciona para cualquier error que invierta dos posiciones. Sin embargo, dado que en este problema sólo es necesario corregir errores en posiciones aleatorias con una probabilidad de acierto razonable, se pueden hacer cosas más "chapuceras". En particular, para n grande podemos dividir el mensaje en trozos pequeños, codificar cada trozo por separado con un código corrector de un error, y suponer que no habrán dos errores en el mismo trozo. Hay construcciones sencillas de códigos correctores de un solo error, pero también podemos hacer una construcción pseudoaleatoria, como se muestra en el código.

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

using namespace std;

string xor_encode(string s) {
    string t;
    t.push_back(s[0]);
    int n = s.length();
    for (int i=0; i < n; ++i) {
        t.push_back(s[i] == t[i] ? '0' : '1');
    }
    return t;
}

string xor_decode(string s) {
    string t;
    int n = s.length();
    for (int i=0; i+1 < n; ++i) {
        t.push_back('0'+((s[i]-'0')^(s[i+1]-'0')));
    }
    return t;
}

string repetition_encode(string s) {
    const int B = 5;
    string t;
    for (char c : s) {
        for (int i=0; i < B; ++i) t.push_back(c);
    }
    return t;
}

string repetition_decode(string s) {
    const int B = 5;
    string t;
```

```
    vector<int> ct(2);
    for (char c : s) {
        ct[(c-'0')]++;
        if (ct[0]+ct[1] == B) {
            t.push_back(ct[0] > ct[1] ? '0' : '1');
            ct = vector<int>(2);
        }
    }
    return t;
}

const int L = 15;
const int B = 8;
vector<string> blocks;
string null_block;

int hamming_distance(string s, string t) {
    int n = s.length();
    assert(n == (int)t.length());
    int d = 0;
    for (int i=0; i < n; ++i) d += s[i] != t[i];
    return d;
}

void random_init(int seed) {
    mt19937 rng(seed);
    null_block = string(L, '0');
    for (int i=0; i < (1<<B); ++i) {
        string s;
        int min_dist;
        do {
            s = string();
            for(int j=0; j < L; ++j) s.push_back('0'+uniform_int_distribution<>(0, 1)(rng));
            min_dist = hamming_distance(s, null_block);
            for (int j=0; j < i; ++j) {
                min_dist = min(min_dist, hamming_distance(s, blocks[j]));
            }
        } while(min_dist < 3);
        blocks.push_back(s);
    }

}

int toint(string s) {
    int r = 0;
    for (int i=0; i < B; ++i) {
        if (s[i] == '1') r |= (1<<i);
    }
    return r;
}

string tostring(int r) {
    string s;
    for (int i=0; i < B; ++i) {
        s.push_back('0'+(r&1));
        r >>= 1;
    }
    return s;
}

string random_encode(string s) {
    int n = s.length();
    string t;
    for (int i=0; (i+1)*B <= n; ++i) {
        string block = s.substr(i*B, B);
        t += blocks[toint(block)];
    }
    t += null_block;
    if (n%B) {
        t += repetition_encode(s.substr(n-n%B));
    }
    return t;
}

string random_decode(string s) {
    int n = s.length();
    string t;
    for (int i=0; (i+1)*L <= n; ++i) {
        string block = s.substr(i*L, L);
        int md = hamming_distance(block, null_block);
        int mb = -1;
        for (int b=0; b < (1<<B); ++b) {
            int d = hamming_distance(block, blocks[b]);
            if (d < md) {
                md = d;
                mb = b;
            }
        }
        if (mb == -1) {
            if ((i+1)*L < n) {
                t += repetition_decode(s.substr((i+1)*L));
            }
            break;
        }
        else {
            t += tostring(mb);
        }
    }
    return t;
}

int main() {
    random_init(1337);
    string name;
    string str;
    cin >> name >> str;
    string out;
    if (name == "ENC") {
        if (str.length() <= 1e3) {
            out = "00000" + repetition_encode(str);
        }
        else {
            out = "11111" + random_encode(str);
        }
        out = xor_encode(out);
    }
    else if (name == "DEC") {
        str = xor_decode(str);
        cerr << str << endl;
        string pref = str.substr(0, 5);
        string suf = str.substr(5);
        int c0 = 0;
        for (char c : pref) c0 += (c == '0');
        if (c0 >= 3) {
            out = repetition_decode(suf);
        }
        else {
            out = random_decode(suf);
        }
    }
    cout << out << endl;
}
```

Sumas

Autor: Manuel Torres Cid

Primero de todo, observemos que calcular la suma en las posiciones coprimas es análogo a calcular la suma en las posiciones no coprimas.

Definamos $g(n) := \sum_{n|i} a_i$, la suma de múltiplos de n, y $S_n = \{p : p|n \text{ y } p \text{ primo}\}$, los divisores primos de n. Una primera idea sería creer que la respuesta es $\sum_{i \in S_n} g(i)$, pero habría valores que estaríamos contando más de una vez. Para solucionar esto, usaremos el principio de inclusión-exclusión.

Sea $T_n = \{i : i|n \text{ y } \nexists p \text{ tal que } p^2|i \text{ y } p \text{ primo}\}$. Aplicando inclusión-exclusión obtenemos que la respuesta para la pregunta k es $g(1) - \sum_{i \in T_k} (-1)^{|S_i|} g(i)$.

```cpp
#include <bits/stdc++.h>
using namespace std;
#define int long long

void solve() {
    int n, q;
    cin >> n >> q;
    vector<int> a(n+1);
    for (int i = 1; i <= n; i++)
        cin >> a[i];

    vector<vector<int>> divisores_buenos(n+1);
    vector<vector<int>> divisores_primos(n+1);

    for (int i = 2; i <= n; i++) {
        bool es_divisible_potencia_primo = false;
        for (auto x : divisores_primos[i])
            if ((i/x)%x == 0) es_divisible_potencia_primo = true;
        if (!es_divisible_potencia_primo) {
            for (int j = i; j <= n; j += i) {
                divisores_buenos[j].push_back(i);
            }
        }

        if (divisores_primos[i].empty()) { // Es primo.
            for (int j = i; j <= n; j += i) {
                divisores_primos[j].push_back(i);
            }
        }
    }

    vector<int> sumas(n+1);
    for (int i = 1; i <= n; i++) {
        for (int j = i; j <= n; j += i)
            sumas[i] += a[j];
    }

    while (q--) {
        int k; cin >> k;
        int ans = sumas[1];
        for (auto x : divisores_buenos[k]) {
            int mul = divisores_primos[x].size() % 2 == 0 ? 1 : -1;
            ans += sumas[x] * mul;
        }
        cout << ans << " ";
    }
    cout << "\n";
}

signed main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int T;
    cin >> T;
    while (T--) solve();
}
```

Pizarra

Autores: Darío Martínez Ramírez y Elena V. R.

El enunciado nos da dos listas de números a_i, b_i y nos pide que convirtamos a en una lista que contenga todos los números de b. Supongamos que ambas listas están ordenadas.

La primera observación es que si $a_i \leq b_j \leq a_{i+1}$, siempre es óptimo conseguir b_j a partir de a_i o a_{i+1}, ya que si lo consiguiéramos a partir de otro número de a, estaríamos gastando dinero para construir a_i o a_{i+1} como paso intermedio, cuando duplicarlos al principio es gratis.

Podemos por tanto resolver el problema independientemente para cada a_i, a_{i+1} y los números de b que hay en el medio, y luego sumar las respuestas. Sean estos $a_i \leq b_l \leq b_{l+1} \leq \ldots \leq b_r \leq a_{i+1}$.

Claramente la forma óptima de generar $b_l, \ldots, b_r$ es elegir algún $l \leq x \leq r+1$ y generar $b_l, \ldots, b_{x-1}$ a partir de a_i y el resto a partir de a_{i+1} (ya que de nuevo, hacer lo contrario significaría que estás generando como paso intermedio el mismo número desde a_i y a_{i+1}, que es una pérdida de dinero respecto a generarla desde un solo lado y luego duplicarlo).

Para generar $b_l, \ldots, b_{x-1}$, primero duplicamos a_i gratis, luego generamos b_l a partir de la copia (con coste $r(b_l - a_i)$), luego lo duplicamos, ahora generamos b_{l+1} a partir de la copia (con coste $r(b_{l+1} - b_l)$), luego de duplicamos, etc.

El coste de todo esto será por tanto $r((b_l - a_i) + (b_{l+1} - b_l) + \ldots + (b_{x-1} - b_{x-2})) = r(b_{x-1} - a_i)$ euros (si $x = l$ el coste será 0). Similarmente, el coste de generar $b_x, \ldots, b_r$ será $l(a_{i+1} - b_x)$ euros (si $x = r+1$ el coste será 0).

Por tanto, podemos simplemente iterar por todos los valores de x, calcular su coste, y elegir el mejor.

Hacer esto para todos los índices i nos da la respuesta correcta (también hay que añadir el coste de generar los valores de b_i más pequeños que a_1 y más grandes que a_n. La forma es básicamente la misma, pero estás obligado a generarlos todos desde el mismo lado).

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

using namespace std;

#define int long long
#define vi vector<int>
#define vvi vector<vi>
#define pb push_back

#define sz(x) (int)(x).size()

signed main(){
    int t;
    cin >> t;
    while (t--){
        int n,m,l,r;
        cin >> n >> m >> l >> r;
        vi a(n), b(m);
        for (int &x:a) cin >> x;
        for (int &x:b) cin >> x;
        sort(a.begin(),a.end());
        sort(b.begin(),b.end());

        int ans=0;
        vvi intervals(n-1);
        int ac=0;
        if (b[0]<a[0]) ans+=l*(a[0]-b[0]);
        if (b.back()>a.back()) ans += r*(b.back()-a.back());
        for (int x:b){
            while (ac<n && x>=a[ac]) ac++; // ac is first bigger
            if (ac!=0 && ac!=n) intervals[ac-1].pb(x);
        }
        for (int i=0;i<n-1;i++) if (sz(intervals[i])){
            int k = sz(intervals[i]);
            int interans = min(l*(a[i+1]-intervals[i][0]),r*(intervals[i].back()-a[i]));
            for (int j=0;j<k-1;j++) interans = min(interans, r*(intervals[i][j]-a[i])+l*(a[i+1]-intervals[i][j+1]));
            ans+=interans;
        }

        cout << ans << '\n';
    }
}
```

Globos

Autor: Darío Martínez Ramírez

El primer paso es çomprimir"los segmentos consecutivos de globos del mismo color. Creamos un nuevo vector donde cada elemento representa un segmento y contiene un entero identificando el color y otro diciendo cuantos globos contiene (para hacer la última subtask, comprimir no es necesario, pues todos los segmentos son de tamaño 1).

Vamos a crear un vector $ans[i]$ que nos diga la respuesta pinchando el color i.

Primero miramos cuáles son los segmentos más grandes que hay antes de pìnchar globos. Si el segmento más grande es del color a y longitud l_1, y el segmento más grande de otro color es de color b y longitud l_2, inicializaremos $ans[i] = l_1$ para $i \neq a$ y $ans[a] = l_2$.

Ahora miramos los segmentos que resultan de juntarse varios al pinchar globos. Nótese que para que al pinchar el color a se junten todos los segmentos en el intervalo $[l, r]$ de color b en uno solo, los colores de la array de segmentos comprimidos en $[l, r]$ tendrán que ser $baba \ldots bab$, es decir, que solo hay esos dos colores alternándose.

Para calcular esto iteraremos por los segmentos comprimidos de izquierda a derecha. Mantenemos 4 variables:

1. c_1: El color del último segmento visto
2. c_2: El color del penúltimo segmento visto
3. s_1: La suma de los tamaños de los segmentos de color c_1 desde que c_1 y c_2 comenzaron a alternarse
4. s_1: La suma de los tamaños de los segmentos de color c_1 desde que c_1 y c_2 comenzaron a alternarse

Claramente podemos mantener estas cuatro variables. Después de actualizarlas, actualizaremos el vector respuesta con $ans[c_2] = \text{máx}(ans[c_2], s_1)$.

Por tanto, lo primero que hacemos se encarga de los segmentos que no se agrandan al pinchar globos, y la segunda cosa que hacemos se encarga de los que sí, resolviendo el problema.

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

using namespace std;

#define int long long
#define vi vector<int>
#define vvi vector<vi>
#define vvvi vector<vvi>
#define pii pair<int,int>
#define vii vector<pii>
#define vvii vector<vii>
#define fi first
#define se second
#define pb push_back

#define sz(x) (int)(x).size()

#define piiii pair<pii,pii>

signed main(){
    int t;
    cin >> t;
    while (t--){
        int n,k;
        cin >> n >> k;
        vi v(n);
        for (int &x:v) cin >> x;
        for (int &x:v) x--;

        vii seginfo(n); //iniciseg, finalseg
```

```
        int l=0;
        while (l<n){
            int r=l;
            while (r<n-1 && v[r+1]==v[l]) r++;
            for (int i=l;i<=r;i++) seginfo[i]={l,r};
            l=r+1;
        }

        vi ans(k,0);

        /// INTERSECTING

        piiii col1, col2; //firstpos, lastpos, col, number
                        // dos ultimos colores
        col1=piiii{{0,-1},{-1,-1e9}};
        col2=piiii{seginfo[0],{v[0],seginfo[0].se-seginfo[0].fi+1}};
        int ac=seginfo[0].se+1;
        while (ac<n){
            if (v[ac]==col1.se.fi){
                col1.fi.se=seginfo[ac].se;
                col1.se.se+=seginfo[ac].se-seginfo[ac].fi+1;
                swap(col1,col2);
            }
            else if (v[ac]==col2.se.fi){
                col2.fi.se=seginfo[ac].se;
                col2.se.se+=seginfo[ac].se-seginfo[ac].fi+1;
            }
            else{
                if (col1.se.fi!=-1) ans[col1.se.fi] = max(ans[col1.se.fi],col2.se.se);
                ans[col2.se.fi] = max(ans[col2.se.fi],col1.se.se);

                col2.fi.fi = col1.fi.se+1;
                col2.se.se = col2.fi.se-col2.fi.fi+1;

                col1=col2;
                col2={seginfo[ac],{v[ac],seginfo[ac].se-seginfo[ac].fi+1}};
            }
            ac=seginfo[ac].se+1;
        }
        ans[col1.se.fi] = max(ans[col1.se.fi],col2.se.se);
        ans[col2.se.fi] = max(ans[col2.se.fi],col1.se.se);

        //// NON-INTERSECTING

        vi longestseg(k);
        for (int i=0;i<n;i++) longestseg[v[i]]=max(longestseg[v[i]],seginfo[i].se-seginfo[i].fi+1);

        int xd=0;
        for (int i=0;i<k;i++){
            ans[i]=max(ans[i],xd);
            xd=max(xd,longestseg[i]);
        }

        xd=0;
        for (int i=k-1;i>=0;i--){
            ans[i]=max(ans[i],xd);
            xd=max(xd,longestseg[i]);
        }

        ///// PRINT

        for (int i=0;i<k;i++) cout << ans[i] << ' ';
        cout << endl;
    }
}
```