OMI 2022 - Solución para *Rango Máximo*

SUBTAREA 1: $p_i = p_j$, $E = 0$. Todos los usuarios inician con el mismo puntaje.

Dado que todos los usuarios inician con el mismo puntaje, todos pueden participar en los mismos exámenes. Por lo tanto la respuesta al problema es el número de usuarios de la plataforma.

SUBTAREA 2: $N, p_i \leq 1000$, $E = 0$.

Los límites de la subtarea te permiten probar todas las opciones posibles de puntaje para un examen. Con cada opción de puntaje puedes validar la cantidad de usuarios que podrían participarl.

```
// VALIDA TODOS LOS POSIBLES PUNTAJES
for (int puntaje = 1; puntaje <= 1000; ++puntaje) {
  // PARA CADA PUNTAJE REVISA CUANTOS ELEMENTOS HAY DENTRO DEL RANGO PERMITIDO
  cnt = 0;
  minimo = puntaje - d;
  maximo = puntaje + d;
  for (int i = 1; i <= n; ++i) {
    // SI EL PUNTAJE ESTA EN EL RANGO, AUMENTA LA CUENTA.
    if (a[i] >= minimo && a[i] <= maximo) ++cnt;
  }

  // QUEDATE CON EL MAXIMO QUE ENCUENTRES.
  res = std::max(res, cnt);
}
```

Para cada puntaje tienes que validar todos los usuarios. La complejidad es $O(N * p_i)$.

SUBTAREA 3: $N \leq 1000$.

En esta subtarea el rango de puntajes se amplia, por lo que ya no puedes seguir probándolos todos.

Observación: Si ordenas los usuarios por puntaje, los usuarios que pueden participar en un examen dado siempre están consecutivos.

Cuando encuentres problemas en los que la solución es un rango de elementos consecutivos de una lista, existe la posibilidad de que el problema se pueda resolver con una *ventana deslizante* (sliding window). En esta técnica el objetivo es ir moviendo un rango (la ventana) de elementos consecutivos por la lista hasta encontrar el que deseas.

Para que la técnica de la *ventana deslizante* pueda aplicarse, se requiere que el *costo* computacional de *recorrer* la *ventana* sea menor que el de revisar la *ventana* completa. Esto es indispensable para que la técnica realmente represente un ahorro de complejidad.

Observa que para este problema puedes usar de *ventana* un rango de elementos consecutivos tales que la diferencia entre el mayor y el menor de ellos sea menor o igual que $2 * d$. El algoritmo para usar la técnica es:

- Ordena los usuarios por su puntaje inicial.
- Inicia con una *ventana* que contenga únicamente al primer elemento (el usuario con el puntaje más bajo).
- Ve agregando usuarios a la *ventana* (de menor a mayor) mientras la distancia entre el mayor puntaje de la *ventana* y el menor puntaje de la *ventana* no exceda $2 * d$.

- Cuando el siguiente elemento a agregar cause que se rompa la restricción de distancia de la *ventana*...
  - Registra el largo de la *ventana* actual, para verificar si es mejor que las que has encontrado.
  - Agrega el nuevo elemento (el que rompe la condición).
  - Comienza a eliminar elementos del inicio de la *ventana* (nuevamente de menor a mayor) hasta que la restricción se cumpla nuevamente. Observa que siempre se va a cumplir, ya que en el peor de los casos, eliminarás todos los elementos salvo el que acabas de agregar. Una *ventana* de un elemento siempre cumple la restricción.

```
rango = 2 * d;  // EL ANCHO DE LA VENTANA ES DE 2d
inicio = fin = 1;
// SE PUEDE PONER UN VALOR MUY GRANDE EN LA POSICION n + 1 PARA EVITAR ESTAR VALIDANDO
a[n + 1] = 4e9;

// PROCESA LA VENTANA HASTA QUE EL INICIO HAYA PASADO POR TODAS LAS POSICIONES
while (inicio <= n) {
  // SI SE PUEDE AUMENTAR EL EXTREMO DERECHO DE LA VENTANA, HAZLO
  if (a[fin] - a[inicio] <= rango)
    ++fin;
  else {
    // SI NO SE PUEDE CRECER, ESTE ES EL LIMITE DE UNA VENTANA HAY QUE REGISTRARLA
    // Y MOVER EL EXTREMO IZQUIERDO HASTA QUE NUEVAMENTE VALIDA
    res = std::max(res, fin - inicio);

    while (a[fin] - a[inicio] > rango)
      ++inicio;  // MUEVE EL INICIO HASTA QUE LA VENTANA SEA VALIDA
  }
}
```

Para procesar las ventanas cada elemento va a entrar y salir de la *ventana* una vez. La complejidad de la solución es $O(N)$

SUBTAREAS 4 Y 5: $E \leq 10$.

Estas subtareas son las primeras en las que hay cambios de puntaje.

La cantidad de exámenes aún es pequeña, lo que te permite resolver examen por examen actualizando los puntajes de los usuarios. En cada examen puedes actualizar los puntajes de cada usuario, ordenar nuevamente los puntajes y ejecutar la *ventana deslizante* de la subtarea anterior.

Por cada examen tendrás que volver a ordenar a los N usuarios y posteriormente recorrerlos con la *ventana deslizante* por lo tanto la complejidad es de $O(E \times NlogN + E \times N)$

El siguiente código junto con la *ventana deslizante* de la subtarea anterior puede resolver las subtareas de la 1 a la $5%.

```
for (int examen = 0; examen <= e; ++examen) {
  res = 0;  // INICIALIZA EL OPTIMO PARA ESTE EXAMEN
  cnt = 0;

  // LA PRIMERA VEZ NO LO HAGAS, YA QUE ES EL ARREGLO ORIGINAL
  if (examen) {
    // LEE LOS PUNTAJES QUE CAMBIARON Y ACTUALIZALOS EN EL ARREGLO TESTIGO
    std::cin >> c;
    while (c--) {
      std::cin >> x >> punt;
      p[x] = punt;
    }
  }

  // COPIA LOS PUNTAJES AL ARREGLO DE REVISION Y ORDENALO
  for(int i = 1; i <= n; ++i) nuevo[i] = p[i];
  std::sort(nuevo + 1, nuevo + 1 + n);
  nuevo[n + 1] = 2e9;

  // AQUI HAZ LA VENTANA DESLIZANTE
}
```

SUBTAREA 6: Sin restricciones adicionales.

Para resolver esta subtarea resulta conveniente *voltear los ejes* del problema. Es decir, la forma de verlo que se usó en las subtareas anteriores es: "*Para cada puntaje de examen, ¿cuántos usuarios pueden presentarlo?*". La forma *volteada* de verlo es: *Por cada usuario ¿en qué rango de puntajes de examen puede participar?*".

Analizando el problema desde el nuevo punto de vista puedes ver que cada usuario suma 1 a un intervalo de puntajes (los puntajes en los que puede participar). Y resolver el problema se vuelve:

- Para contestar el *rango máximo* requieres encontrar el máximo de entre todos los puntajes. Recuerda que cada usuario le suma 1 a los puntajes en los que puede participar, de modo que el máximo entre todos los puntajes será la respuesta del *rango máximo*.
- Cuando un usuario cambia su puntaje se deben hacer dos cosas:
  - Restarle 1 a todos los puntajes en los que podía participar según su antiguo puntaje.
  - Sumarle 1 a todos los puntajes en los que puede participar con el nuevo puntaje.

De modo que requieres realizar de forma eficiente tres operaciones:

- Encontrar el máximo entre todos los elementos.
- Restarle 1 a un rango de elementos.
- Sumarle 1 a un rango de elementos.

Existe una estructura de datos, quizá la más utilizada actualmente en programación competitiva, que puede realizar de manera eficiente todas esas preguntas. Se conoce como *árbol de segmentos* (segment tree) y si nunca la has oído te recomiendo ampliamente que busques información sobre ella.

La solución oficial utiliza un *segmente tree* sobre el rango de puntajes. El segment tree permite sumar o restar algún valor a todos los nodos de un rango y además puede contestar el máximo en un rango. Para el caso específico de este problema, siempre se requiere contestar el máximo de entre todos los puntajes, así que basta con ver el máximo en la raíz para conocer la respuesta.

El siguiente es el código de la solución oficial.

📋

```
#include <algorithm>
#include <iostream>
#include <set>

using namespace std;

#define debugsl(x) std::cout << #x << " = " << x << ", "
#define debug(x) debugsl(x) << "\n";

#define MAX 100000
#define LOG 32

typedef struct NODO {
  int lazy;
  int maximo;
  int hi, hd;
};

int n, d, e, p[MAX + 2];
NODO st[MAX * LOG * 5];
long long c, x, punt, cont;

void propagaActualizacion(int nodo, int inicioNodo, int finNodo) {
  // SI EL NODO TIENE ALGO QUE PROPAGAR
  if (st[nodo].lazy) {
    // ACTUALIZA EL VALOR DEL MAXIMO DEL RANGO
    st[nodo].maximo += st[nodo].lazy;

    // PROPAGA LA ACTUALIZACION A SUS HIJOS SI LOS TIENE
    if (inicioNodo < finNodo) {
      // SI NO TIENE HIJOS, CREALOS
      if (!st[nodo].hi) {
        st[nodo].hi = cont++;
        st[nodo].hd = cont++;
      }
      st[st[nodo].hi].lazy += st[nodo].lazy;
      st[st[nodo].hd].lazy += st[nodo].lazy;
    }

    // LIMPIA EL VALOR DE PROPAGACION
    st[nodo].lazy = 0;
  }
}

void actualizaSegmentTree(int nodo, int inicioNodo, int finNodo,
                          int inicioIntervalo, int finIntervalo, int valor) {
  // PROPAGA LA ACTUALIZACION, SI HAY ALGUNA
  propagaActualizacion(nodo, inicioNodo, finNodo);

  if (inicioNodo > finNodo) return;  // SI EL NODO QUE SE BUSCA ES INVALIDO SAL.

  // SI EL NODO TIENE UN INTERVALO QUE NO INTERSECTA CON EL QUE SE BUSCA,
  // ENTONCES SAL
  if (inicioNodo > finIntervalo || finNodo < inicioIntervalo) return;

  // SI EL INTERVALO DEL NODO ESTA DENTRO DEL DE INTERES, ACTUALIZALO
  if (inicioNodo >= inicioIntervalo && finNodo <= finIntervalo) {
    st[nodo].lazy += valor;
    propagaActualizacion(nodo, inicioNodo, finNodo);
    return;
  }
```

```
  // SI NO TIENE HIJOS, CREALOS
  if (!st[nodo].hi) {
    st[nodo].hi = cont++;
    st[nodo].hd = cont++;
  }

  // SI EL INTERVALO DEL NODO SE INTERSECTA PARCIALMENTE CON EL DE INTERES, BAJA
  // RECURSIVAMENTE
  int mitad = (inicioNodo + finNodo) / 2;
  actualizaSegmentTree(st[nodo].hi, inicioNodo, mitad, inicioIntervalo,
                       finIntervalo, valor);
  actualizaSegmentTree(st[nodo].hd, mitad + 1, finNodo, inicioIntervalo,
                       finIntervalo, valor);
  st[nodo].maximo = max(st[st[nodo].hi].maximo, st[st[nodo].hd].maximo);
}

int main() {
  cont = 2;  // PRIMER NODO DISPONIBLE EN EL SEGMENT TREE

  std::cin >> n >> d;
  for (int i = 1; i <= n; ++i) {
    std::cin >> p[i];
    actualizaSegmentTree(1, 1, 1e9, max(1, p[i] - d), min((int)1e9, p[i] + d),
                         1);
  }

  // IMPRIME EL MAXIMO DEL RANGO TOTAL
  std::cout << st[1].maximo << "\n";

  // PROCESA EXAMEN POR EXAMEN
  std::cin >> e;
  while (e--) {
    // LEE LOS PUNTAJES QUE CAMBIARON Y ACTUALIZALOS EN EL ARREGLO TESTIGO
    std::cin >> c;
    while (c--) {
      std::cin >> x >> punt;
      // ELIMINA EL INTERVALO DE AFECTACION Y AGREGA EL NUEVO
      actualizaSegmentTree(1, 1, 1e9, max(1, p[x] - d), min((int)1e9, p[x] + d),
                           -1);
      p[x] = punt;
      actualizaSegmentTree(1, 1, 1e9, max(1, p[x] - d), min((int)1e9, p[x] + d),
                           +1);
    }
    std::cout << st[1].maximo << "\n";
  }

  return 0;
}
```

Las operaciones del *segment tree* son todas logarítmicas de modo que la complejidad de la solución es: $O((N+E)logmax(p_i))$