OMI 2022 - Solución para *Bruxismo*

SUBTAREA 1: $T = 1$. Sólo se requiere evaluar una noche.

Basta que revises para cada diente si la fuerza con que se apretó esa noche es mayor a su dureza y en caso de que sí, revises si el daño que sufre es mayor o igual al daño total que resiste. Esta subtarea es útil para validar que tienes correctos los "*mayor*", "*mayor o igual*" del problema.

📋

```
for (int diente = 1; diente <= n; ++diente) {
  if (fuerza > d[diente] && a[diente] >= m[diente]) std::cout << "1 ";
  else std::cout << "0 ";
}
```

SUBTAREA 2: $N, T \leq 1000$.

Los límites de la subtarea te permiten simular cada noche con todos los dientes e ir marcando aquellos que se caen.

📋

```
// SIMULA CADA DIA PARA MARCAR QUE DIENTES SE CAEN
for (int dia = 1; dia <= T; ++dia) {
  for (int diente = 1; diente <= n; ++diente) {
    // SI ESE DIENTE YA SE CAYO, NO TIENE SENTIDO SEGUIRLO REVISANDO
    if (dano[diente] >= m[diente]) continue;

    // SI EL DIENTE SUFRE DANO ESA NOCHE, ACUMULALO, SI NO RECUPERALO SIN QUE
    // BAJE DE 0 SU DANO TOTAL
    if (t[dia] > d[diente])
      dano[diente] += a[diente];
    else
      dano[diente] = std::max(0, dano[diente] - b[diente]);
  }
}

for (int diente = 1; diente <= n; ++diente) {
  if (dano[diente] >= m[diente])
    std::cout << "1 ";
  else
    std::cout << "0 ";
}
std::cout << "\n";
```

Para cada día debes simular todos los dientes. La complejidad es $O(T \times N)$.

SUBTAREA 3: $D_i = D_j$, $A_i = A_j$, $B_i = B_j$. Todos los dientes tienen la misma dureza, sufren y recuperan el mismo daño.

Todos los dientes son iguales salvo en el máximo daño que soportan. Basta con simular un diente (todos son iguales) y guardar el daño máximo que alcanza, los dientes que soporten más de ese daño permanecerán y los que no en algún momento se caerán.

```cpp
// SIMULA TODOS LOS DIAS
for (int dia = 1; dia <= T; ++dia) {
  // SIMULA UNICAMENTE CON EL DIENTE 1
  if (t[dia] > d[1]) {
    dano[1] += a[1];
    dmaximo = std::max(dmaximo, dano[1]);
  } else
    dano[1] = std::max(0, dano[1] - b[1]);
}

for (int diente = 1; diente <= n; ++diente) {
  if (dmaximo >= m[diente])
    std::cout << "1 ";
  else
    std::cout << "0 ";
}
std::cout << "\n";
```

Se recorren todos los días pero se simula únicamente con un diente. La complejidad de la solución es $O(T)$

SUBTAREAS 4: $A_i = B_i$. El daño que cada diente sufre en un día es igual a su recuperación.

Para resolver esta subtarea necesitas cambiar el orden de procesamiento a algo que te resulte más conveniente. En específico, ordena todas las noches por fuerza de mayor a menor y guárdalas en el conjunto $Noches = f_1, f_2, \ldots, f_T$, ordena los dientes por dureza de mayor a menor y guárdalos en el conjunto $Dientes = d_1, d_2, \ldots, d_N$. Procesas elemento por elemento ambos conjuntos tomando siempre el valor que sea máximo entre los dos elementos más grandes sin procesar, es decir, procesas el siguiente diente o la siguiente noche dependiendo de si la dureza del diente es más grande que la fuerza de la noche o viceversa. Observa que en el momento que proceses un diente, todas las noches que le hacían daño ya fueron procesadas (su fuerza era mayor y por tanto las procesaste antes), de modo que el daño máximo que ese diente pueda tener será el máximo que se pueda alcanzar con las noches que ya fueron procesadas.

Observa además que como $A_i = B_i$ puedes visualizar las noches en que hay daño como un $+1$ y las noches en las que hay recuperación como un -1 (siempre y cuando no baje de 0) y el daño máximo que sufrirá un diente es la suma máxima que se alcance multiplicada por su A_i.

El problema entonces se transforma en ser capaz de ir agregando $+1$ en las noches que se vayan procesando y pudiendo calcular de forma eficiente el máximo a lo largo de las T noches.

La solución oficial funciona como sigue:

Digamos que cada noche puede tener uno de tres valores: * $+1$: Esto significa que en esa noche se causó daño. * -1: Esto significa que en esa noche se tiene recuperación. * 0: Esto significa que en esa noche **NO** hay daño pero el diente ya tenía un daño 0 por lo que no pasa nada.

Todos los dientes inician con un valor 0. Observa que el único cambio posible para un diente es pasar a $+1$, cada que se procesa una noche es porque va a dañar a todos los dientes que aún no han sido procesados. Entonces los cambios posibles son:

- Pasar de 0 a $+1$: En este caso a partir de la noche que cambia todos los valores aumentan en 1. Cuando se llegue al primer 0 a la derecha de esa noche, ese 0 se convertirá en -1. Es decir, como todo después de esa noche subió, el primer 0 sube a 1 y puede usarse como noche de recuperación para regresar a 0.
- Pasar de -1 a $+1$: En este caso a partir de esa noche todos los valores aumentan en 2 (de -1 a 1), por lo tanto los siguientes dos 0s deben cambiar a -1.

De modo que para poder actualizar rápidamente los estados de cada noche, requieres saber de forma eficiente dada una posición, cuál es el siguiente 0. Esto puede hacerse en tipo $logT$ si todas las noches con 0 se mantienen en un `set` que contenga las posiciones.

Queda por resolver eficientemente el problema del máximo. Requieres alguna estructura de datos a la que puedas actualizarle valores $+1$ y -1 en una posición y sea capaz de decirte el valor máximo al que llega la suma a lo largo de las noches. Una estructura que puede hacer esto de forma sencilla es un *segment tree* (Si no conoces *segment tree* es muy recomendable que lo investigues ya que es una de las estructuras de datos más usadas en la programación competitiva).

La solución oficial usa un *segment tree* en el que almancena para cada rango el valor máximo al que llega (con respecto a su valor de inicio) y el valor en el que termina (con respecto a su valor de inicio).

```cpp
int n, a[MAX + 2], b[MAX + 2], d[MAX + 2], m[MAX + 2], res[MAX + 2];
int T, t[MAXD + 2], valordia[MAXD + 2];
int offset;
std::set<int> diasEnCero;
std::vector<std::pair<int, int> > eventos;
std::pair<int, int> stmax[MAX * 4 + 2];

void actualiza(int nodo, int val) {
  nodo = offset + nodo;  // PONLE EL OFFSET DEL SEGMENT TREE
  stmax[nodo].fin = val;
  stmax[nodo].maximo = std::max(0, val);
  nodo >>= 1;
  while (nodo) {
    int hi = nodo * 2;
    int hd = hi ^ 1;
    stmax[nodo].fin = stmax[hi].fin + stmax[hd].fin;
    stmax[nodo].maximo =
        std::max(stmax[hi].maximo, stmax[hd].maximo + stmax[hi].fin);
    nodo >>= 1;
  }
}

int main() {
  std::cin >> n >> T;
  for (int i = 1; i <= n; ++i) std::cin >> d[i] >> a[i] >> b[i] >> m[i];
  for (int i = 1; i <= T; ++i) std::cin >> t[i];

  offset = 1;
  while (offset < T) offset <<= 1;
  --offset;

  // ORDENA POR FUERZA, DUREZA E INICIALIZA LAS NOCHES A 0
  for (int i = 1; i <= n; ++i) eventos.emplace_back(d[i], i);
  for (int i = 1; i <= T; ++i) {
    eventos.emplace_back(t[i], -i);
    diasEnCero.emplace_hint(diasEnCero.end(), i);
  }
  std::sort(eventos.begin(), eventos.end());
  std::reverse(eventos.begin(), eventos.end());

  for (auto ev : eventos) {
    if (ev.id < 0) {  // ES DE TIPO NOCHE
      auto it = diasEnCero.lower_bound(-ev.id);
      int afectados;
      if (it != diasEnCero.end() && *it == -ev.id) {
        // ESA NOCHE TENIA VALOR 0, SOLO HAY QUE ACTUALIZAR UN 0 A NEGATIVO
        afectados = 1;
        it = diasEnCero.erase(it);  // BORRALO Y APUNTA AL SIGUIENTE
      } else
        afectados = 2;  // LA NOCHE ERA NEGATIVA, HAY QUE AFECTAR DOS CEROS.

      actualiza(-ev.id, 1);
      while (it != diasEnCero.end() && afectados) {
        actualiza(*it, -1);  // ACTUALIZA ESTE DIA PARA QUE SEA UN NEGATIVO
        it = diasEnCero.erase(it);  // BORRALO DE LOS CEROS
        --afectados;
      }
    } else {
      // ES UN EVENTO DE DIENTE, HAY QUE VER EL MAXIMO EN ESE MOMENTO,
      // MULTIPLICARLO POR EL DANIO ESPECIFICO DE ESE DIENTE Y VER SI SE CAE
      if (a[ev.id] * stmax[1].maximo >= m[ev.id]) res[ev.id] = 1;
```

```
    }
  }

  for (int i = 1; i <= n; ++i) std::cout << res[i] << " ";
  std::cout << "\n";

  return 0;
}
```

Por cada noche se tiene que buscar en el `set` y hacer actualizaciones en el mismo, además se deben hacer actualizaciones en el *segment tree*, por cada diente es necesario obtener el máximo del *segment tree* pero esto se hace en tiempo constante. La complejidad total del problema es: $O(T \times (logT + logN) + N)$

SUBTAREA 5: Sin restricciones adicionales.

Para resolver la última subtarea se requiere de la siguiente observación:

Observación: Si para dos dientes i y j se cumple que $fracA_iB_i = fracA_jB_j$ entonces sus curvas de daño serán semejantes, es decir, los máximos del daño y los regresos a 0 sucederán en las mismas noches y sólo variarán en magnitud según la relación que haya entre A_i y A_j.

Esta observación te permite concluir que dos dientes que tengan la misma fracción $fracAB$ pueden considerarse equivalentes. Dado que el rango de valores de A y B es de 15 se pueden calcular todas las fracciones irreducibles formables por los números del $[1, 15]$ (en total 143) y crear una copia del `set` y del `segment tree` de la subtarea anterior para procesarlos. La solución entonces es utilizar el algoritmo de la subtarea anterior 143 veces.

Observa que como A puede ser distinto de B los valores ya no son sólo $+1$ y -1 y es necesario llevar un control de cuántos 0s deben actualizarse. Igualmente, un 0 puede actualizarse únicamente de manera parcial.

El codigo completo es muy similar a la subtarea anterior y sólo se agregan aquí las partes faltantes.

Código para calcular fracciones irreducibles:

```
f = 1;
for (int numerador = 1; numerador <= MAXA; ++numerador) {
  for (int denominador = 1; denominador <= MAXA; ++denominador) {
    if (fracciones[numerador][denominador]) continue;
    int mult = 1;
    int num = numerador;
    int den = denominador;
    while (num <= MAXA && den <= MAXA) {
      fracciones[num][den] = f;
      mult++;
      num = numerador * mult;
      den = denominador * mult;
    }
    factor[f] = {numerador, denominador};
    ++f;
  }
}

std::cin >> n >> T;
for (int i = 1; i <= n; ++i) {
  std::cin >> d[i] >> a[i] >> b[i] >> m[i];
  frac[i] =
      fracciones[a[i]][b[i]];  // LA FRACCION A LA QUE PERTENECE ESTE DIENTE
}
```

Código para decidir cuántos 0s se deben afectar:

```
    if (ev.id < 0) {
      // ES DE TIPO NOCHE
      for (int fr = 1; fr < f; ++fr) {
        auto it = diasEnCero[fr].lower_bound({-ev.id, 0});
        int afectacion;
        if (it != diasEnCero[fr].end() && (*it).first == -ev.id) {
          // ESE DIA AUN NO TENIA SU MAXIMO NEGATIVO, HAY QUE COMPENSAR EL VALOR
          afectacion = factor[fr].first + (*it).second;
          it = diasEnCero[fr].erase(it);  // BORRALO Y APUNTA AL SIGUIENTE

        } else {
          afectacion = factor[fr].first +
                       factor[fr].second;  // EL DIA YA ERA NEGATIVO HAY QUE
                                           // COMPENSARLO TAMBIEN.
        }

        actualiza(fr, -ev.id, factor[fr].first);  // PON EN POSITIVO EL ACTUAL

        while (it != diasEnCero[fr].end() && afectacion) {
          int actual = factor[fr].second - (*it).second;
          if (afectacion >= actual) {
            actualiza(fr, (*it).first, -factor[fr].second);

            it = diasEnCero[fr].erase(it);
            afectacion -= actual;
          } else {
            actual = (*it).second + afectacion;
            int dia = (*it).first;

            it = diasEnCero[fr].erase(it);
            diasEnCero[fr].emplace_hint(it, dia, actual);
            afectacion = 0;

            actualiza(fr, dia, -actual);
          }
        }
      }
    } else {
      // ES UN EVENTO DE DIENTE, CALCULA EL MAXIMO SEGUN SU FRACCION
      int maximo = stmax[frac[ev.id]][1].maximo * a[ev.id] / factor[frac[ev.id]].first;
      if (maximo >= m[ev.id]) res[ev.id] = 1;
    }
```

La complejidad de la solución es $O(\mathbf{143} \times T \times (logT + logN))$