**Estaciones de Recarga**

**Descripción del Problema**

Un vehículo eléctrico con una batería de alcance limitado debe viajar a lo largo de una carretera recta.
Hay M segmentos de carretera donde es seguro estacionar y recargar la batería.
Debes ubicar N estaciones de recarga en puntos enteros distintos dentro de estos segmentos seguros para maximizar la distancia mínima (D) entre cualquier par de estaciones.

Las estaciones deben estar lo más separadas posible.

**Detalles y Restricciones**

- Entrada:
    - La primera línea contiene N (número de estaciones) y M (número de segmentos).
      Los valores cumplen $2 \leq N, M \leq 100000$.
    - Las siguientes M líneas describen los segmentos seguros con dos enteros a y b, donde $0 \leq a \leq b \leq 10^{18}$.
      Los segmentos son mutuamente disjuntos y no se tocan en sus puntos finales.
- Salida:
    - Imprime el mayor valor posible de D tal que todas las N estaciones puedan colocarse con al menos D unidades de distancia entre ellas.

**Ejemplo**

| **Input** | **Output** |
|---|---|
| 4 2<br>0 5<br>10 15 | 5 |

**Explicación del Ejemplo:**

- Segmento 1: de 0 a 5
- Segmento 2: de 10 a 15
- Para N = 4 estaciones, la distancia máxima D es 5.
- Una colocación válida: 0, 5, 10, 15 (todas separadas por 5).
- Si D fuera 6, no sería posible colocarlas.

**CÓDIGO**

```
#include <iostream>
#include <algorithm>
#include <utility>
#include <climits>
using namespace std;

int main() {

  int N, M;
  if (!(cin >> N >> M)) {
    // entrada inválida
    return 0;
  }

  if (N <= 0 || M <= 0) {
    cout << 0;
    return 0;
  }

  long long* A = new long long[M];
  long long* B = new long long[M];
  for (int i = 0; i < M; i++) {
    if (!(cin >> *(A + i) >> *(B + i))) {
      // entrada incompleta
      cout << 0;
      delete[] A;
      delete[] B;
      return 0;
    }
  }
```

```
  // Construir arreglo de segmentos dinámico (sin usar [])
  pair<long long, long long>* seg = new pair<long long, long long>[M];
  for (int i = 0; i < M; i++) {
    *(seg + i) = make_pair(*(A + i), *(B + i));
  }

  // Ordenar por inicio (luego por fin)
  sort(seg, seg + M, [](const pair<long long,long long>& x, const pair<long long,long long>& y){
    if (x.first != y.first) return x.first < y.first;
    return x.second < y.second;
  });

  auto canPlace = [&](long long D) -> bool {
    long long placed = 0;
    long long last = 0; // última posición colocada

    for (int i = 0; i < M && placed < N; i++) {
      long long inicio = (*(seg + i)).first;
      long long fin = (*(seg + i)).second;

      if (placed == 0) {
        // colocar la primera estación en el inicio del primer segmento disponible
        last = inicio;
        placed = 1;
        if (placed >= N) return true;
      }

      // intentar colocar más estaciones en este segmento
      while (true) {
```

```
            long long need = last + D;
            if (need > fin) break;
            long long cand = need;
            if (cand < inicio) cand = inicio;
            last = cand;
            placed++;
            if (placed >= N) return true;
        }
    }

    return placed >= N;
};

// Búsqueda binaria para la máxima distancia mínima
long long lo = 0, hi = 1;
// aumentar hi hasta que no sea posible o sea suficientemente grande
while (hi < LLONG_MAX / 2 && canPlace(hi)) hi <<= 1;
if (hi > LLONG_MAX / 4) hi = LLONG_MAX / 4;

while (lo < hi) {
    long long mid = lo + (hi - lo + 1) / 2;
    if (canPlace(mid)) lo = mid;
    else hi = mid - 1;
}

cout << lo;

delete[] seg;
delete[] A;
delete[] B;
return 0;
```

```
}
```