Divide and Conquer
Binary Search

Materia: TTPS

Autor: JTP - Matías Fluxa

Divide and Conquer

Definición de Gemini

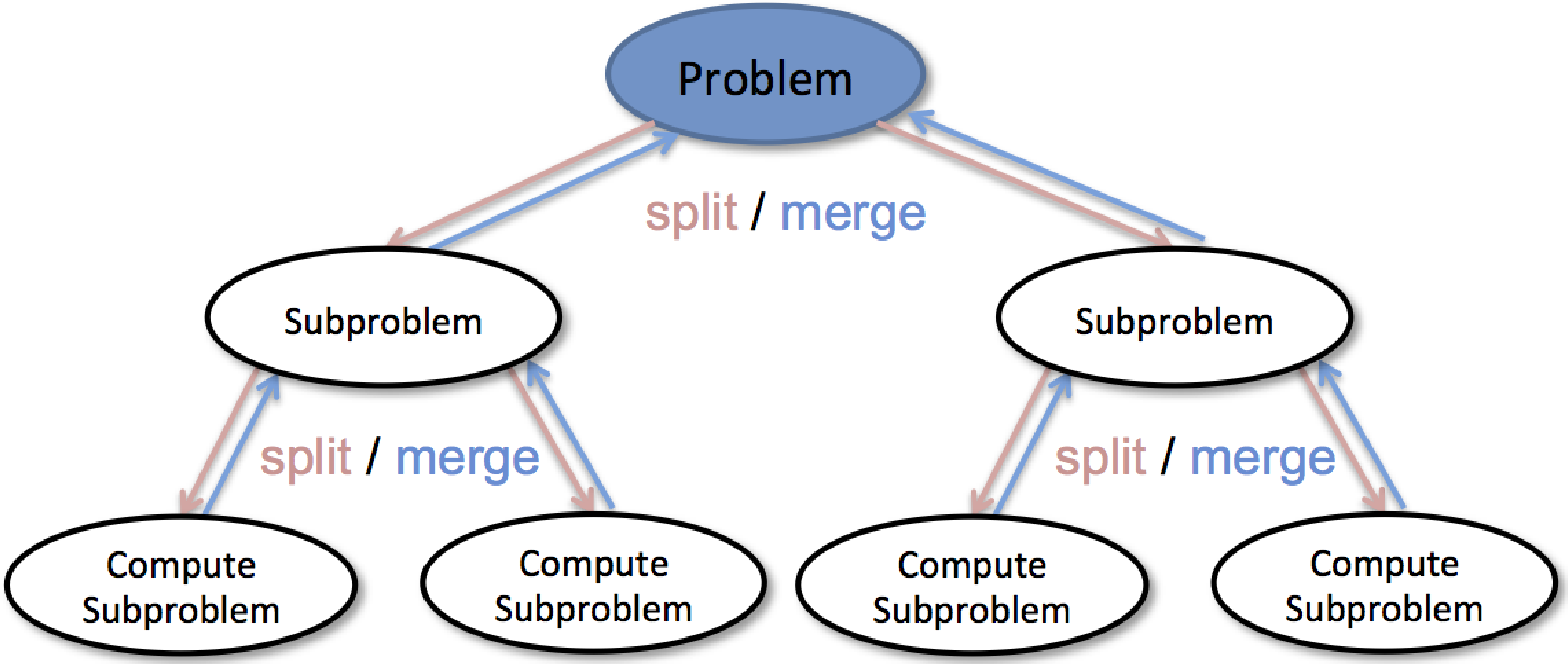
- La frase "*divide y vencerás*" (o *divide et impera* en latín) se atribuye al emperador romano Julio César y resume una estrategia militar y política para mantener el poder. Esta técnica consistía en dividir a los pueblos o grupos de oposición en facciones más pequeñas, a menudo otorgando privilegios desiguales para generar rencores y envidias entre ellos. Al evitar que se unieran contra Roma, era más fácil controlarlos y conquistarlos.

Divide and Conquer

Definición de Gemini

- En informática, "*divide y vencerás*" es un paradigma de diseño de algoritmos que resuelve un problema dividiéndolo recursivamente en subproblemas más pequeños hasta que son lo suficientemente simples para ser resueltos directamente, y luego combina las soluciones para obtener la solución final.
- Esta técnica se compone de tres pasos principales: **Dividir**, **Conquistar** (resolver subproblemas) y **Combinar**.

Divide and Conquer



Divide and Conquer

Algunos ejemplos de aplicación

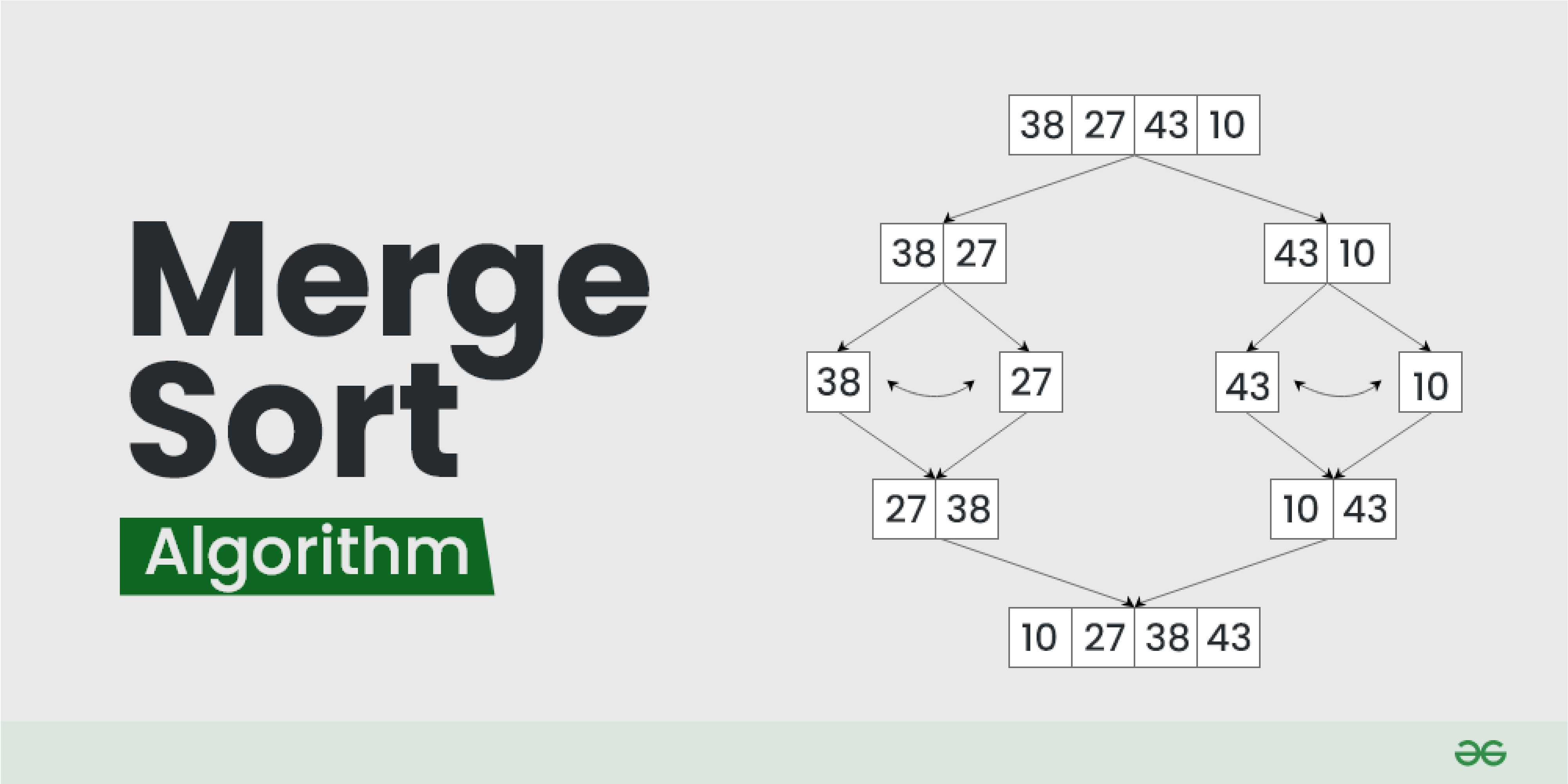
- **Merge Sort**
- **Binary Exponentiation**
- **Binary Search**
- **Segment Tree**

Merge Sort

¿Qué es el algoritmo de Merge Sort?

- **Merge Sort** es un algoritmo de ordenamiento, capaz de ordenar un conjunto de elementos eficientemente en tiempo y en memoria, sin importar la configuración inicial.
- El algoritmo está basado en la técnica de **Divide and Conquer**.

Merge Sort



Merge
Sort
Algorithm
38 27 43 10
38 27
43 10
38
27
43
10
27 38
10 43
10 27 38 43

Merge Sort – Implementación en Python

Parte I – Dividir el problema

Caso base: si el tamaño de la lista es 1, no hay nada que hacer, retornamos la lista.

Dividimos el problema en dos mitades, y llamamos recursivamente a cada mitad

```python
def sort(self, lista) -> None:
    if len(lista) <= 1:
        return

    mid = len(lista) // 2
    left = lista[:mid]
    right = lista[mid:]
    self.sort(left)
    self.sort(right)
```

Merge Sort – Implementación en Python

Parte II – Combinar

Sabiendo que las listas de la izquierda y la derecha están ordenadas, con dos punteros vamos eligiendo el menor elemento entre las dos listas.

```python
while l < len(left) and r < len(right):
    if left[l] <= right[r]:
        lista[idx] = left[l]
        l += 1
    else:
        lista[idx] = right[r]
        r += 1
        self.inv += len(left) - l
    idx += 1
```

Alguna de las dos listas quedará con algunos elementos remanentes. Solo se ejecutará uno de los ciclos while.

```python
while l < len(left):
    lista[idx] = left[l]
    l += 1
    idx += 1

while r < len(right):
    lista[idx] = right[r]
    r += 1
    idx += 1
```

Merge Sort

Complejidad

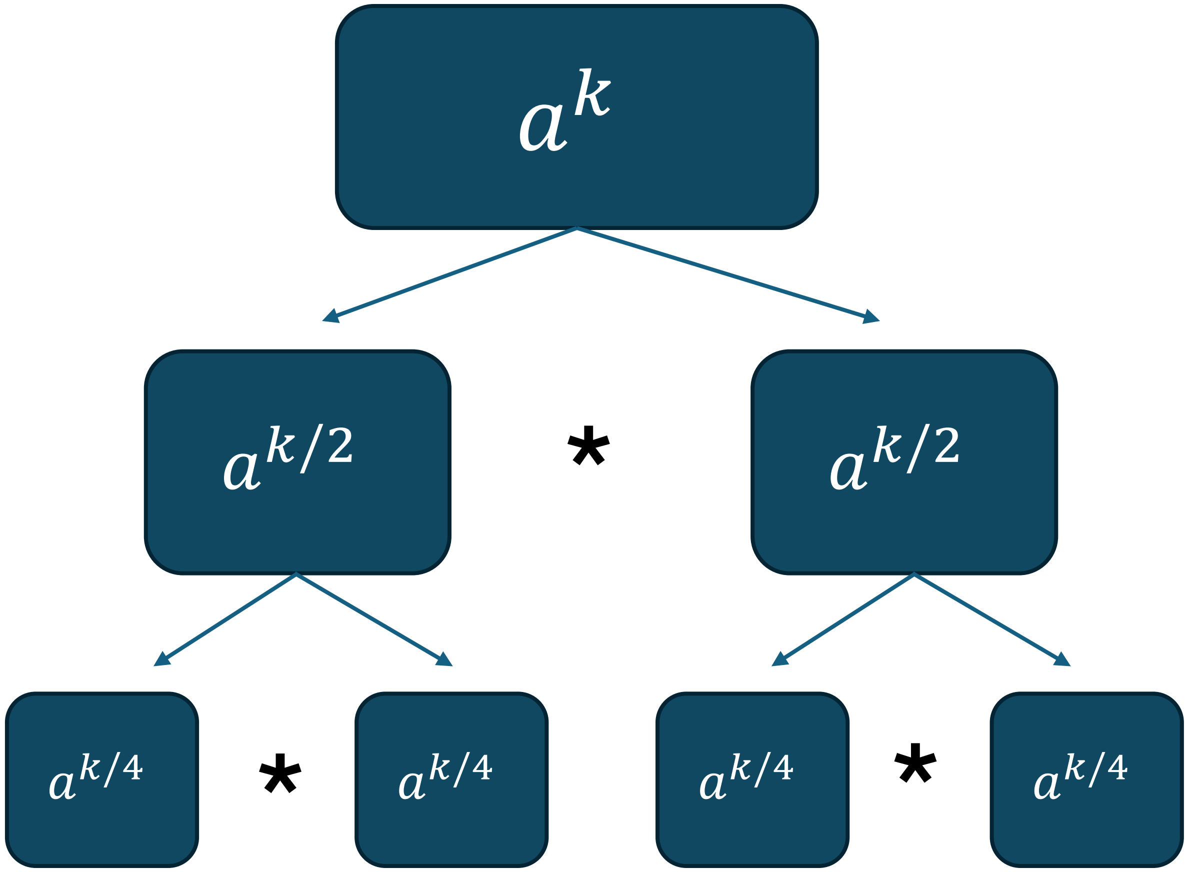
- En tiempo de ejecución: $O(n \cdot \log(n))$
- En memoria: $O(n)$

Binary Exponentiation

¿Qué es Binary Exponentiation?

- Es una técnica basada en la idea de Divide and Conquer que se utiliza para calcular rápidamente la potencia de un elemento.
- Como se vio en la clase de teoría de números, podemos obtener rápidamente la potencia de un número en tiempo logarítmico del exponente.
- No solo podemos elevar números, sino también que matrices, polinomios, etc.

Binary Exponentiation



Binary Search

- Es un algoritmo de búsqueda eficiente en un arreglo o lista ordenada.
- El concepto se basa en partir el rango de búsqueda siempre a la mitad, y elaborar un test de decisión eficiente capaz de determinar en cuáles de las dos mitades se encuentra el valor objetivo.
- Dado que en cada iteración dividimos el rango de búsqueda a la mitad, la complejidad será $O(\log(n) \cdot O(Test))$.

Binary Search - Aplicaciones

Encontrar un elemento en un arreglo

Dado un arreglo ordenado de forma no decreciente, encontrar la posición donde aparece el elemento x.

Binary Search - Aplicaciones

Encontrar la posición donde aparece por primera vez el 10

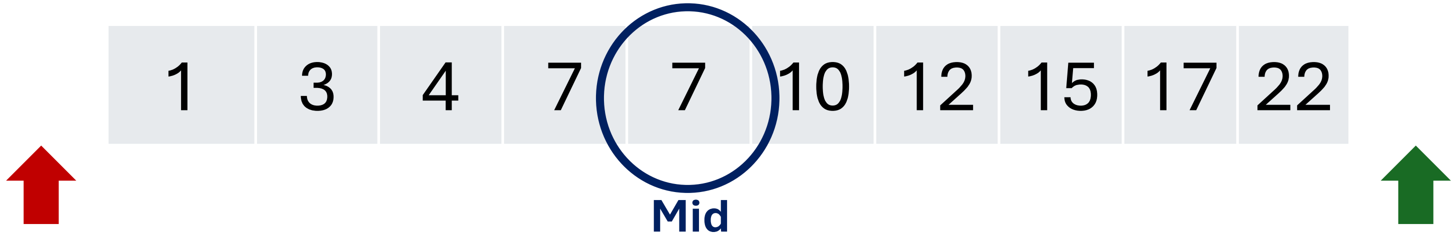
1	3	4	7	7	10	12	15	17	22

La flecha **ROJA**, representa el último número que es menor a 10, siempre la inicializamos en la posición "-1", suponiendo que allí existe el $-\infty$.

La flecha **VERDE**, representa el primer número que es mayor o igual a 10, siempre la inicializamos en la posición "N", suponiendo que allí existe el ∞ .

Binary Search - Aplicaciones

Encontrar la posición donde aparece por primera vez el 10



En cada iteración marcamos el punto medio **“Mid”**. Este se puede hallar como el promedio entre los índices de la flecha **ROJA (L)** y la flecha **VERDE (R)**.

$$Mid = \frac{L + R}{2}$$

Binary Search - Aplicaciones

Encontrar la posición donde aparece por primera vez el 10

1	3	4	7	7	10	12	15	17	22

Como $\boldsymbol{Valor[Mid] < 10}$, entonces movemos el puntero **ROJO** a la posición de Mid.

Volvemos a calcular la nueva posición de Mid y volvemos a iterar.

Binary Search - Aplicaciones

Encontrar la posición donde aparece por primera vez el 10

1	3	4	7	7	10	12	15	17	22

Como $\boldsymbol{Valor[Mid] \geq 10}$, entonces movemos el puntero **VERDE** a la posición de Mid.

Volvemos a calcular la nueva posición de Mid y volvemos a iterar.

Binary Search - Aplicaciones

Encontrar la posición donde aparece por primera vez el 10

1	3	4	7	7	10	12	15	17	22

Como $\boldsymbol{Valor[Mid] \geq 10}$, entonces movemos el puntero **VERDE** a la posición de Mid.

De esta manera logramos juntar los punteros. El puntero **ROJO** representa el máximo valor que es menor estricto de 10, y el puntero **VERDE** el menor valor mayor o igual a 10. En ese caso, como el valor de puntero equivale a 10, entonces podemos responder que 10 está en el arreglo y su primera aparición en la posición del **VERDE**.

Binary Search - Aplicaciones

Binary Search en la respuesta

En muchos problemas de **optimización**, se vuelve muy difícil encontrar una estrategia ganadora que nos de el resultado óptimo que queremos. Es decir, no existe una idea “**greedy**” que nos asegura lo mejor.

Sin embargo, en muchas ocasiones es posible determinar si con cierto costo podemos o no podemos encontrar una solución válida. De esta manera podemos transformar un problema de optimización en un **problema de decisión**.

Binary Search - Aplicaciones

En general, los problemas de decisión los podemos representar como una función "escalón".

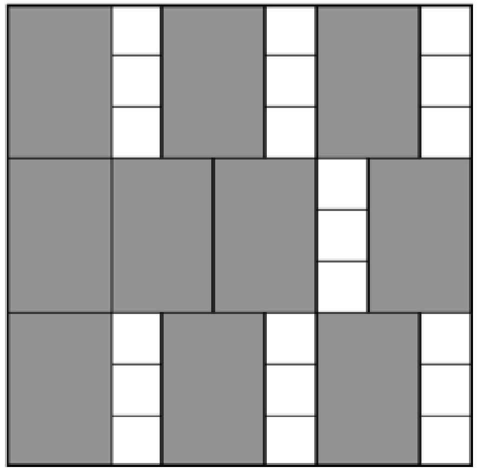
Binary Search - Aplicaciones

A. Packing Rectangles

time limit per test: 2 seconds

memory limit per test: 512 megabytes

There are n rectangles of the same size: w in width and h in length. It is required to find a square of the smallest size into which these rectangles can be packed. Rectangles cannot be rotated.



Input

The input contains three integers w, h, n ($1 \le w, h, n \le 10^9$).

The input contains three integers w, h, n ($1 \le w, h, n \le 10^9$).

Output

Output the minimum length of a side of a square, into which the given rectangles can be packed.

Example

input	Copy
2 3 10	

output	Copy
9	

Binary Search - Aplicaciones

Notemos que este problema lo podemos pensar como un problema de decisión. - -

Supongamos que existe una solución para un cuadrado de lado **L**.

Es simple observar que para otro lado **L' > L** también vamos a poder.

También es fácil observar que, si para un lado **X** no puedo, entonces para un lado **X' < X** tampoco voy a poder.

Binary Search - Aplicaciones

Entonces podemos proponer que el lado del cuadrado sea **M**. Ahora deberíamos ver cuántos rectángulos entran en el cuadrado sabiendo que son de ancho **w** y alto **h**.

¡Y ahora este es un problema conocido! Es el problema **A4. Theatre Square**, del **TPN1** de la materia.

Pueden entrar: $\mathrm{k} = \left\lfloor \frac{\boldsymbol{M}}{\boldsymbol{h}} \right\rfloor \cdot \left\lfloor \frac{\boldsymbol{M}}{\boldsymbol{w}} \right\rfloor$

Si $k \geq N$, entonces sabemos que para **M** se puede. En caso contrario, sabremos que no se puede.

Binary Search - Aplicaciones

Problemas Min-Max

Son una variante a los problemas de Binary Search en la respuesta. Estos problemas se basan es **minimizar un máximo** o **maximizar un mínimo**.

Binary Search - Aplicaciones

C. Cows in Stalls

time limit per test: 2 seconds

memory limit per test: 256 megabytes

Stalls are located on a straight line, your task is to arrange the cows to stalls so that the minimum distance between the cows is as large as possible.

Input

The first line contains numbers n ($2 \le n \le 10^4$), the number of stalls and k ($2 \le k \le n$), the number of cows. The second line contains n integer numbers in ascending order, the coordinates of the stalls (coordinates are in the range from 0 to 10^9, inclusive).

Output

Print one number, the largest possible minimum distance between two cows.

Example

input Copy

```
6 3
2 5 7 11 15 20
```

output Copy

```
9
```

Binary Search - Aplicaciones

Problemas Min-Max

Notemos que es un problema del tipo Min-Max, porque queremos maximizar la mínima distancia entre los establos de las vacas.

Lo podemos llevar a un problema de decisión, es claro que si a una distancia de al menos **x** la podemos ubicar, entonces también las podemos ubicar más cercas.

En cambio, si a una distancia **y** no podemos, con menos razón las podremos ubicar a mayor distancia.

¿Dónde practicar BS?

Cursos EDU de Codeforces:

https://codeforces.com/edu/course/2/lesson/6

Guía Sorting and Searching de CSES:

https://cses.fi/problemset/