# Problema 1

## Orden

### Link
https://codeforces.com/problemset/problem/1996/F

### Definicion
Sparkly tiene pensado detonar una bomba nuclear pero te invita a jugar un juego antes de hacerlo. Te da dos listas de enteros `a` y `b`, ambas de tamaño `n` y un número total de operaciones `k`. Realizar una operación implica escoger un numero `i` tal que `1 <= i <= n`, sumar `a_i` a tu puntaje y restar `b_i` a `a_i`. El objetivo es maximizar tu puntaje antes de que Sparkly pueda detonar la bomba.

## Solucion Greedy
Enfoque:

1. Configuración del Heap :
   - Usar un heap de máximo (cola de prioridad) para seleccionar siempre el mayor a[i] disponible en cada operación.
   - Inicialmente, empuja todos los elementos (a[i], i) en el heap, donde a[i] es el valor, e i es el índice.
2. Ejecución de k Operaciones:
   - Para cada una de las k operaciones:
     - Extrae el elemento máximo (a[i], i) del heap.
     - Agrega a[i] a la puntuación.
     - Actualiza a[i] = max(0, a[i] - b[i]) .
     - Vuelve a insertar el (a[i], b[i], i) actualizado en el heap si a[i] > 0 .

Esta estrategia greedy asegura que en cada paso, la operación se aplique al elemento que proporciona la mayor ganancia inmediata en la puntuación. Luego de realizar k operaciones, se obtiene la puntuación máxima posible.

Por qué funciona?

Para demostrar que la elección greedy de siempre seleccionar el mayor a[i] en cada paso es óptima, necesitamos demostrar que este enfoque tiene la propiedad de elección greedy. Esto significa que una solución óptima global se puede lograr tomando una serie de decisiones localmente óptimas.

1. Asume la Solución Óptima:
   - Supongamos que existe una secuencia óptima de k operaciones que maximiza la puntuación, pero que no selecciona el mayor a[i] en algún paso.
2. Considera la Primera Desviación:
   - Considera la primera operación donde esta secuencia óptima no selecciona el mayor a[i] . En su lugar, selecciona algún a[j] donde a[j] < a[i] .
   - Al seleccionar a[j] en lugar de a[i] , la contribución inmediata a la puntuación es a[j] , que es menor que a[i] .
3. Reemplazo con la Elección Avara:
   - Si reemplazamos a[j] con a[i] para esa operación, la puntuación en ese paso aumenta. Además, el resto de las operaciones permanecen sin afectar o se pueden ajustar reinsertando el a[i] actualizado de vuelta en el heap, lo que podría dar los mismos o incluso mejores resultados.
4. Puntuación Resultante:

- Dado que reemplazar a[j] con a[i] lleva a un aumento en la puntuación, esto implica que la elección greedy (seleccionar el mayor a[i] en cada paso) no puede dar un resultado peor que cualquier otra secuencia.
- Por lo tanto, tomar la elección greedy en cada paso debe llevar a una solución óptima.

```
heap = []
for i in range(n):
    heapq.heappush(heap, (-a[i], i))

# Se usa -a[i] porque la implementacion de heap de python es un min heap, - lo vuelve
max heap

score = 0
for _ in range(k):
    poppedA, i = heapq.heappop(heap)

    # Add the value to the score
    score -= poppedA

    # Update a[i]
    a[i] = max(0, a[i] - b[i])

    # Update the heap
    heapq.heappush(heap, (-a[i], i))

return score
```

La complejidad de esta solucion es O(nlogn + klogn) donde n es el tamaño de la lista a y b y k es el numero de operaciones a realizar. Es un poco fuerza bruta ya que se puede mejorar.

## Solucion Busqueda Binaria

Vamos a usar busqueda binaria para la siguiente solucion.

Definamos primero la funcion de optimizacion `f(x)`:

- `f(x)`: es el numero de operaciones necesarias para que todos los valores añadidos al score en alguna operacion sean al menos x, y que todos los valores restantes en `a` sean a lo sumo x - 1.

  Para un solo valor de i, que a[i] < x, se necesita reducir a[i] una cierta cantidad de b[i], o sea

  $$a[i] - b[i] * t < x$$

  , luego despuejando `t` queda que

  $$t > \frac{a[i] - x}{b[i]}$$

  , por lo tanto, el numero de operaciones necesarias para que a[i] sea al menos x es

  $$\left\lceil \frac{a[i] - x}{b[i]} \right\rceil$$

  .

  , especificamente para todos los valores de a queda $f(x) = \sum_{i=0}^{n} \left\lfloor \frac{a_i - x}{b_i} \right\rceil$

Usemos primero busqueda binaria para encontrar el valor de `x` que maximiza el score, podemos usar busqueda binaria en el rango [0, max(a) + 1], restringiendo que f(x) <= k.

```
low, top = 0, max(a) + 1
while top - low > 1:
    mid = (low + top) // 2

    sum_val = 0
    for i in range(n):
        if a[i] >= mid:
            sum_val += (a[i] - mid) // b[i] + 1

    if sum_val >= k:
        low = mid
    else:
        top = mid

x = low
```

Luego de encontrar el valor de `x`, tenemos que calcular el score. Cada indice `i` tal que `a[i] >= x` contribuye con `x` al score, y los indices `i` tal que `a[i] < x` contribuyen con `a[i] - b[i] * t` al score, donde `t` es el numero de operaciones necesarias para que `a[i]` sea al menos `x`. Luego para cada indice `i`, se puede decir que contribuye a la suma total una serie de crecimiento discreto (en concreto de saltos de tamaños b[i]).

Analicemos cada indice `i` por separado. Como se demostro antes en la formula de `f(x)`, el numero de operaciones necesarias para que `a[i]` sea al menos `x` es $t = \left\lceil \frac{a[i]-x}{b[i]} \right\rceil$. La secuencia de esos números tiene como primer elemento a a[i], ya que fue el primero en ser añadido al score, y el último es $a_1 - b_1 * (t-1)$. Como estamos tratando con una serie de crecimiento discreto la suma de un fragmento de ella se resume a la formula de `S=t*(primer_term + ultimo_term)/2`. Sustituyendo y calculando queda de la siguiente forma

$$S = t * a[i] - \frac{t * (t-1)}{2} * b[i]$$

Asi que el score total es la suma de todos los `S` para cada `i` tal que `a[i] < x` y `a[i] >= x`. Puede darse el caso que esta ultima formula se usen mas operaciones de las que se tienen, cada una aportando `x` a la solucion general, en ese caso se debe ajustar la solucion quitando las operaciones extras y su contribucion.

```
ans = 0
s = 0

for i in range(n):
    if a[i] >= x:
        t = (a[i] - x) // b[i] + 1
        ans += t * a[i] - t * (t - 1) // 2 * b[i]
        s += t

ans -= x * (s - k)

return ans
```

La complejidad de esta solucion es O(n*log(f)) donde f es max(a)