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 $1 \leq i \leq n$, 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], 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 Greedy:
   - 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 solución 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.

Solución Búsqueda Binaria

Vamos a usar búsqueda binaria para la siguiente solución.

Definamos primero dos funciones de utilidad `f(x)` y `score(x)`

Para un valor dado x, definimos la función `f(x)` como el número mínimo de operaciones necesarias para que, después de realizar dichas operaciones, cada elemento en el arreglo es menor que `x` o, lo que significa lo mismo, que todos los valores añadidos al score sean al menos x.

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 $f(x) = \sum_{i=0}^{n} \left\lfloor \frac{a[i]-x}{b[i]} \right\rceil$

Luego tambien definamos la funcion `score(x)`, como la funcion que dado un numero x, devuelve el score total añadido despues de haber realizado todas las operaciones de `f(x)`.

Demostremos ahora que `f(x)` es **no creciente**. O sea que para $x_1 < x_2 \Rightarrow f(x_1) \geq f(x_2)$. Seanse dos números x_1 y x_2 tal que $x_1 < x_2$. Llamemos $y = f(x_2)$. Luego sea el estado del array despues de haber aplicado dichas y operaciones el siguiente $[a_1, a_2, ..., a_i]$.

Ahora, si $\neg\exists(i) : a_i > x_1 \Rightarrow f(x_1) = f(x_2)$, ya que no hay necesidad de realizar operaciones extras para que todos los valores del arreglo sean igual a x_1. Ahora, imaginemos que $\exists(i) : a_i > x_1$, este valor puede existir ya que $x_1 < x_2$. Luego es necesario realizar al menos una operacion extra para reducir ese valor para que $a_i < x_1$, o sea que $f(x_1) > f(x_2)$ en ese caso. Por tanto queda que $\forall x_1 < x_2 \Rightarrow f(x_1) \geq f(x_2)$

Ahora demostremos que sucede lo mismo con `score(x)`. O sea que para $x_1 < x_2 \Rightarrow \text{score}(x_1) \geq \text{score}(x_2)$. Seanse dos números x_1 y x_2 tal que $x_1 < x_2$. Como se demostro anteriormente, en esta situacion $f(x_1) \geq f(x_2)$, luego como ya se demostro en la solucion greedy, este problema posee la propiedad de subestructura optima, o sea que la solucion de mas de operaciones de apoya en la solucion de menos operaciones, y como todos los valores para añadir al score son positivos, menos operaciones implican menos score. Por tanto $\text{score}(x_1) \geq \text{score}(x_2)$, osea que `score(x)` es una funcion no creciente.

Probablemente la primera igual salia por subestructura optima pero aun no me he dado cuenta de como hacerlo

Ahora, como tanto `f(x)` como `score(x)` son funciones monotonas no crecientes dado el mismo parametro, yo puedo decir con seguridad que buscar la `x` que maximiza la cantidad de operaciones mientras las mantiene por debajo de `k` implica a su vez buscar la `x` que maximiza el score. Y como `f(x)` es una funcion monotona, se puede hacer busqueda binaria.

Por tanto vamos a buscar el valor de `x` que maximice el score restringiendo que $f(x) \leq k$. Podemos restringir el espacio de busqueda a $[0, \max(a) + 1]$ por obvias razones.

```
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
```

Ahora depues de encontrar dicho `x` necesitamos resolver el problema, o sea calcular el score que dicho `x` genera.

Para cada elemento `a[i]`, calculamos el número de operaciones t_i necesarias para reducir su valor por debajo de `x`:

$t_i = \left\lfloor \frac{a[i]-x}{b[i]} \right\rceil$

Detalles:

- Si a[i] ≤ x: No se requieren operaciones, es decir, $t_i = 0$.
- Si a[i] > x: Se necesitan t_i operaciones para reducir `a[i]` por debajo de `x`.

Una vez que conocemos cuántas veces se aplica una operación a cada `a[i]`, calculamos cómo contribuye cada uno al score total.

Para un índice i con t_i operaciones, la secuencia de valores de a[i] antes de cada operación es:

$$a[i], a[i] - b[i], a[i] - 2b[i], ..., a[i] - (t_i - 1)b[i]$$

Contribución al Score (S_i):

La suma de estos valores es:

$$S_i = a[i] + (a[i] - b[i]) + (a[i] - 2b[i]) + ... + (a[i] - (t_i - 1)b[i])$$

Esto es una serie aritmética con:

- Primer término (a_1): $a[i]$
- Último término (a_{t_i}): $a[i] - (t_i - 1)b[i]$
- Número de términos (t_i): t_i

La suma de una serie aritmética se calcula mediante la fórmula:

$$S_i = t_i \times \frac{a_1 + a_{t_i}}{2}$$

Sustituyendo:

$$S_i = t_i \times \frac{a[i] + (a[i] - (t_i - 1)b[i])}{2} = t_i \times \left(a[i] - \frac{(t_i - 1)b[i]}{2}\right)$$

Simplificando nos lleva a la formula final para calcular el score contribuido por un solo lugar del arreglo:

$$S_i = t_i \times a[i] - \frac{t_i \times (t_i - 1)}{2} \times b[i]$$

El score total es la suma de las contribuciones individuales de todos los índices:

$$\text{Score} = \sum_{i=1}^{n} S_i = \sum_{i=1}^{n} \left(t_i \cdot a[i] - \frac{t_i \cdot (t_i - 1)}{2} \cdot b[i]\right)$$

Es posible que la suma total de operaciones $S = \sum t_i$ exceda el número permitido de operaciones k porque esta no es la formula exacta.

¿Por Qué Pueden Suceder Operaciones Excedentes?

- Asignación Localmente Óptima: Al calcular t_i para cada `a[i]` de manera independiente, garantizamos que cada operación aplicada a `a[i]` contribuye con al menos x.
- No Consideración del Límite Global: Al hacerlo de manera independiente, no consideramos inmediatamente el límite global de k operaciones, lo que puede resultar en S > k.

En esencia, cuando se hace la operacion `t = (a[i] - x) // b[i] + 1` se pueden cometer errores de −1 ya que el resultado no es exacto asi que se esta añadiendo x extra por cada vez que se haga eso.

En tal caso, necesitamos ajustar el score para reflejar únicamente `k` operaciones. O sea que buscamos la cantidad de operaciones excedentes $S - k$, y cada una de estas operaciones excedentes aporto al menos `x` al score. Asi que al resultado final le restamos $x \cdot (S - k)$

```
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 del problema se divide en lo siguiente:

- Busqueda binaria en x $\rightarrow$ O(log max(a))
- Dentro de la busqueda binaria se realiza el calculo de f(x) $\rightarrow$ O(n)
- Luego para asignar el score se recorre el arregle $\rightarrow$ O(n)

$$\therefore O(n \cdot \log \max(a) + n) = O(n \cdot \log \max(a))$$