Advent of Code 2023

José Antonio González Morente

27 de diciembre de 2023

Índice

1. Día 5: *If You Give A Seed A Fertilizer*

El problema consiste en encontrar el número de ubicación más bajo que corresponde a cualquiera de los números iniciales de semillas, utilizando una serie de mapas proporcionados en un fichero. Estos mapas convierten números de una categoría (como semillas) en números de otra categoría (como suelo, fertilizante, agua, luz, temperatura, humedad y ubicación). Cada mapa contiene rangos de números y muestra cómo convertir un número de una categoría fuente a un número en una categoría destino. El objetivo es seguir las conversiones a través de las categorías para cada número de semilla inicial y determinar cuál es el número de ubicación más bajo correspondiente. En la segunda parte, se introduce una complicación adicional: los números iniciales de semillas representan rangos de números, lo que significa que se deben considerar varios números de semillas en lugar de solo uno.

Para ver el enunciado completo de este problema, consulta el siguiente enlace:

https://adventofcode.com/2023/day/5

Código fuente de la solución en Python:

https://github.com/jgonzalezmorente/advent-of-code/blob/main/AOC2023/aoc2023_5.py

1.1. Parte I

El objetivo de esta parte es determinar la ubicación óptima para cada semilla, partiendo de una lista de semillas proporcionada en el archivo de texto de entrada. Para lograr esto, se implementa un enfoque secuencial y compuesto, donde cada etapa del proceso transforma la información de una forma a otra, comenzando con las semillas y pasando por diferentes fases como suelo, fertilizante, agua, luz, temperatura, humedad, y finalmente, ubicación.

1. **Definición de la clase `Map`:**
   - Se define una clase llamada `Map`, que hereda de `dict`.
   - A la clase `Map` se le añade un atributo `__ranges`. Este atributo es una lista de tuplas, definida como `List[Tuple[range, int]]`, para almacenar los intervalos de mapeo.
   - Cada tupla en `__ranges` contiene dos elementos: un rango de origen (`range`) y un valor inicial del rango destino (`int`).
2. **Redefinición del método `__missing__`:**
   - Se redefine el método especial `__missing__`, que se invoca automáticamente cuando se intenta acceder a una clave ausente en el diccionario.
   - Si la clave k está en un rango r de `__ranges`, se calcula su valor mediante la siguiente expresión: $r[1] + (k - r[0][0])$. Esto es simplemente una traslación de la clave al intervalo destino, ya que es así como nos dicen en el enunciado que funciona el mapeo.
   - Si la clave no se encuentra en `__ranges`, el método devuelve la propia clave. El enunciado del problema nos dice que cuando la clave no pertenece al intervalo, su valor es la propia clave.
3. **Método `add_range`:**
   - Se añade el método `add_range`, que transforma una cadena de texto procedente del fichero, en una tupla (r, s) donde r es el rango origen y s es el valor inicial del rango destino, y lo añade a `__ranges`.
4. **Método `to_fun`:**
   - Este método devuelve una función cuyo propósito es mapear cada clave a su valor correspondiente, actuando efectivamente como una representación funcional del diccionario. Esta función toma una clave como entrada y retorna el valor asociado dentro del diccionario. Si la clave existe en el diccionario, devuelve su valor asociado directamente. En caso contrario, la función aplica la lógica definida en el método `__missing__` para determinar y retornar un valor adecuado. De esta manera, el método convierte el diccionario, que es una estructura de datos estática, en una entidad dinámica y funcional que puede interactuar con otras partes del programa de manera más flexible y versátil.
5. **Definición de la función `set_map`:**

- Esta función recibe dos parámetros principales: un objeto `TextIOWrapper` y un objeto de la clase `Map`. El objeto `TextIOWrapper` se obtiene al abrir un archivo de texto, proporcionando una interfaz para leer su contenido. La función itera a través de cada línea del archivo, donde cada línea representa un intervalo específico. Utiliza el método `add_range` de la clase `Map` para procesar y añadir estos intervalos al mapa. Este proceso implica la transformación de la representación textual de cada intervalo en un objeto `range`, que luego se incorpora a la estructura de datos del mapa para su posterior uso y consulta.

6. **Proceso principal:**

   - **Creación de objetos `Map`:** Se instancian varios objetos de la clase `Map` para representar diferentes etapas del proceso. Cada `Map` se asocia con una fase específica, como se muestra a continuación:

     ```
     seed_to_soil = Map()
     soil_to_fertilizer = Map()
     fertilizer_to_water = Map()
     water_to_light = Map()
     light_to_temperature = Map()
     temperature_to_humidity = Map()
     humidity_to_location = Map()
     ```

     Estos mapas se utilizarán para relacionar distintas etapas del proceso, como semillas a suelo, suelo a fertilizante, y así sucesivamente.

   - **Procesamiento del archivo de texto:** Se realiza parseo del archivo de texto para obtener una lista de elementos denominada `seeds`. Por otro lado, utilizando el método `add_range`, se informan los intervalos correspondientes en cada uno de los mapas creados anteriormente. Este proceso implica leer y transformar la información del archivo en una serie de rangos que se incorporan a los mapas, permitiendo así crear una cadena de relaciones entre las diferentes fases del proceso.

   - **Composición de funciones:** Se implementa una función denominada `compose` cuyo propósito es componer dos funciones dadas, f y g. La composición de estas funciones se define matemáticamente como $compose(f, g) = g \circ f$, lo que significa que, para un argumento dado x, la función compuesta $(g \circ f)(x)$ es igual a $g(f(x))$.

     Esta técnica se aplica a los objetos `Map` creados anteriormente. Cada `Map` se asocia con una función específica (método `to_fun`), y estas funciones se almacenan en una lista. Posteriormente, se utiliza la función `reduce` para combinar secuencialmente todas estas funciones en una sola función compuesta, mediante la aplicación reiterada de `compose`.

     La función compuesta final resultante se aplica entonces a la lista de semillas (`seeds`). Este proceso permite transformar cada semilla a través de todas las etapas representadas por los mapas de forma secuencial y eficiente. Como resultado, se puede obtener el valor mínimo de ubicación deseado.

1.2. Parte II

El objetivo sigue siendo encontrar la ubicación óptima para el cultivo, pero ahora con la complejidad añadida de manejar intervalos extensos de semillas. La magnitud de estos intervalos descarta el uso de métodos de fuerza bruta, requiriendo en su lugar un enfoque más eficiente.

Se considera el uso de funciones lineales del tipo $f : \mathbb{R} \longrightarrow \mathbb{R}$, definidas como $f(x) = \alpha + x$, donde α es una constante. Estas funciones tienen la propiedad de transformar intervalos de la forma $[a, b]$ en $[f(a), f(b)]$. Este tipo de transformaciones son las que se están realizando, por lo que podemos aplicar esta propiedad.

La estrategia implica aplicar un proceso de composición de funciones similar al desarrollado en la Parte I, pero adaptado para trabajar con intervalos en lugar de valores específicos de semillas. En este contexto, cada función lineal aplicada a un intervalo de semillas transforma este intervalo en otro, reflejando la acumulación de cambios a lo largo del proceso.

Al final de este proceso compuesto, se obtendrá un conjunto de intervalos transformados. El paso final consiste en identificar el intervalo con el extremo inferior mínimo. Este intervalo representa el rango óptimo de semillas que, después de todas las transformaciones, resulta en la ubicación más favorable.

1. **Método `__project_interval`:**
   Este método toma como entrada un intervalo $[a, b]$ y realiza un proceso de búsqueda y transformación en varias etapas:

   a) *Búsqueda de rangos relevantes:* Inicialmente, el método identifica todos los rangos dentro de `self.__ranges` cuyos extremos se encuentran entre a y b. Esta búsqueda efectiva segmenta el intervalo original $[a, b]$ en varios subintervalos.

   b) *Partición en subintervalos:* A partir de los rangos identificados, se genera una partición de $[a, b]$. Cada subintervalo resultante es una porción del intervalo original que está influenciada por uno o más rangos en `self.__ranges`.

   c) *Transformación de subintervalos:* Debido a la naturaleza lineal y creciente de la transformación, mencionada anteriormente, es suficiente transformar únicamente los extremos de cada subintervalo. Esta transformación se aplica a los puntos extremos, generando nuevos intervalos que son la proyección de los subintervalos originales a través de la función de transformación.

   El método concluye devolviendo una lista de estos subintervalos transformados. Cada subintervalo transformado refleja una parte del intervalo original $[a, b]$ después de haber sido procesado a través de las transformaciones definidas.

2. **Método `__project_intervals`:**
   Este método toma una lista de intervalos, aplica `__project_interval` a cada uno de ellos, y devuelve una lista de todos los intervalos transformados. Este método sería útil en situaciones donde se necesita aplicar la misma transformación a múltiples intervalos y recopilar todos los resultados en una única lista.

3. **Proceso principal**:

   - **Procesamiento inicial de los intervalos de semillas:** En la primera fase, se extraen los intervalos de semillas del archivo de texto. Cada par de números en el archivo representa un intervalo de semillas, donde el primer número indica el extremo inferior del intervalo y el segundo número representa la longitud del mismo. De esta manera, se convierte la fila de semillas del archivo en una serie de intervalos concretos que serán el punto de partida para el proceso.

- **Transformación de intervalos:** Una vez que se han identificado todos los intervalos de semillas, el proceso procede de manera similar a lo descrito en la Parte I. Se emplea la función `compose` para combinar de manera sucesiva las aplicaciones del método `project_intervals` a través de las distintas fases del proceso.

  Cada fase implica la transformación de los intervalos de semillas, adaptándolos y modificándolos según las especificaciones de cada etapa del proceso. La composición final de estas transformaciones, realizada mediante la función `compose`, permite proyectar los intervalos iniciales de semillas a través de todas las etapas, resultando en intervalos transformados que reflejan el efecto acumulativo de todo el proceso.
- **Resultado final:** El resultado de esta secuencia de transformaciones es un conjunto de intervalos que han sido modificados sucesivamente en cada fase. Estos intervalos finales proporcionan una representación detallada y completa de cómo cada intervalo de semillas se ha transformado a lo largo del proceso. Basta calcular el mínimo de los extremos inferiores de estos intervalos.

2. Día 6: *Wait For It*

- Enunciado:
  https://adventofcode.com/2023/day/6
- Solución:
  https://github.com/jgonzalezmorente/advent-of-code/blob/main/AOC2023/aoc2023_6.py

2.1. Parte I

El objetivo de esta parte es determinar de cuantas formas podemos batir el récord de cada carrera de barcos de juguete. Las carreras nos las proporcionan en un fichero con dos registros: el primero contiene los tiempos de duración de cada carrera y el segundo las distancias recorridas. Por ejemplo,

Tiempo empleado	7	15	30
Distancia recorrida	9	40	200

El barco tiene un botón de forma que si se deja pulsado t milisegundos, éste tomará una velocidad de t milímetros por segundo. Supongamos que la distancia récord para una carrera dada es de d milímetros y que el tiempo empleado en esa carrera fue de s milisegundos, para batir este récord deberá cumplirse:

$$t(s-t) > d \iff ts - t^2 > d \iff t^2 - ts + d < 0$$

Para que la inecuación anterior tenga soluciones reales, deberá cumplirse que el discriminante de la ecuación cuadrática, $s^2 - 4d > 0$. Sean t_1 y t_2 con $t_1 < t_2$ las raíces del polinomio $P(t) = t^2 - ts + d$

$$t_1 = \frac{s - \sqrt{s^2 - 4d}}{2}, \quad t_2 = \frac{s + \sqrt{s^2 - 4d}}{2}$$

entonces deberá cumplirse que $(t - t_1)(t - t_2) < 0$ y puesto que $t_1 < t_2$ esto ocurre si y sólo si $t \in (t_1, t_2)$. Como además t debe ser un número entero positivo y menor a s.

$$t \in [\text{máx}(1, \text{floor}(t_1) + 1), \text{mín}(s - 1, \text{ceil}(t_2) - 1)]$$

Por lo que el número de formas del batir el récord debe ser:

$$\text{mín}(s-1, \text{ceil}(t_2)-1) - \text{máx}(1, \text{floor}(t_1)+1)+1$$

Esto permite definir una función f que calcula el número de formas de batir el récord al par (s,d) donde s es el tiempo empleado y d la distancia recorrida, mediante la siguiente expresión:

$$f(s,d) = \text{mín}\left(s-1, \text{ceil}\left(\frac{s+\sqrt{s^2-4d}}{2}\right)-1\right) - \text{máx}\left(1, \text{floor}\left(\frac{s-\sqrt{s^2-4d}}{2}\right)+1\right)+1$$

Para calcular el resultado final, basta mapear esta función sobre la lista de pares (s,d) y plegar la lista resultante mediante la función producto de dos números.

2.2. Parte II

El objetivo de esta parte es el mismo que en la parte anterior, pero suponiendo que solo hay una carrera. Esta carrera se forma concatenando las cifras del tiempo empleado y distancia recorrida respectivamente. Para el ejemplo anterior, ahora solo tendremos que considerar esta carrera,

- Tiempo empleado: 71530
- Distancia recorrida: 940200

Por la forma en la que se ha resuelto la parte I, no hay que hacer nada adicional, el mismo método funciona, ya que no se ha empleado fuerza bruta.

3. Día 7: *Camel Cards*

- Enunciado:
  https://adventofcode.com/2023/day/7
- Solución:
  https://github.com/jgonzalezmorente/advent-of-code/blob/main/AOC2023/aoc2023_7.py

4. Día 8: *Haunted Wasteland*

- Enunciado:
  https://adventofcode.com/2023/day/8
- Solución:
  https://github.com/jgonzalezmorente/advent-of-code/blob/main/AOC2023/aoc2023_8.py

Nos proporcionan una red formada por dos elementos:

- Un conjunto de instrucciones para moverse por la red, formado por una sucesión finita de los símbolos L y R. Esta sucesión se extienda de manera cíclica. Esto es si la sucesión original es

  $$I_0, I_1, \ldots, I_k, \quad \text{donde } I_i \in \{L, R\}$$

  entonces $I_j = I_i$ con $i \equiv j \mod k$ para $j > k$.
- Un conjunto de nodos, de la forma $n_1n_2n_3 = (l_1l_2l_3, r_1r_2r_3)$ donde cada n_i, l_i, r_i son letras mayúsculas, por ejemplo, $AAA = (BBB, CCC)$

Para movernos por el mapa a partir de un nodo de inicio, basta seguir la sucesión de instrucciones y elegir el nodo correspondiente a L (izquierda) o R (derecha).

4.1. Parte I

En esta parte, nos piden que calculemos el número de pasos necesarios para ir desde el nodo AAA al nodo ZZZ. Para ello solo debemos movernos por el mapa hasta encontrar el nodo deseado. Esto se hace con la función `navigate_network`.

4.2. Parte II

Este parte requiere calcular la cantidad de pasos necesarios para moverse desde todos los nodos que terminan en A hasta llegar a un conjunto de nodos que finalicen en Z. A diferencia de la parte anterior, no es viable utilizar un enfoque de fuerza bruta para este propósito, ya que moverse por la red y verificar en cada paso si todos los nodos resultantes terminan en Z no es una estrategia práctica.

Para abordar este desafío, adoptamos un método más estratégico. Comenzamos desde un nodo específico que termine en A y navegamos a través de la red hasta encontrar un nodo que termine en Z. Al alcanzarlo, registramos este nodo junto con el número de pasos que se necesitaron para llegar a él. Luego, continuamos navengdo por la red hasta que nos encontramos de nuevo con el mismo nodo. Si el número de pasos actuales para llegar a este nodo es congruente con el número inicial de pasos (módulo la longitud de las instrucciones), esto indica que hemos completado un ciclo. En este punto, podemos determinar con precisión las posiciones en las que encontraremos nodos que terminan en Z, partiendo del nodo inicial.

Formalmente un ciclo es un diccionario cuyas claves son tuplas de la forma (N_Z, s) donde N_Z es un nodo terminado en Z y s es el número de pasos hasta llegar a N_Z partiendo desde el nodo de origen N_A. El valor asociado a la clave (N_Z, s) inicialmente es cero y cuando nos encontremos nuevamente con una clave en la que coincide el nodo y los valores de s son congruentes módulo la longitud de las instrucciones, le asignamos como valor la diferencia de pasos.

Ahora calculamos los ciclos de cada uno de los nodos que empiezan en A finalmente basta calcular la intersección de estos conjuntos:

Si los conjuntos resultantes son de la forma $(m_1), (m_2), \ldots, (m_k)$, donde (m_i) denota el conjunto de todos los múltiplos de m_i. Resulta que:

$$(m_1) \cap (m_2) \cap \cdots \cap (m_k) = (\text{mcm}\,(m_1, m_2, \ldots, m_k))$$

Es decir la solución, en este caso, sería el $\text{mcm}\,(m_1, m_2, \ldots, m_k)$.

5. Día 10: *Pipe Maze*

- Enunciado:
  https://adventofcode.com/2023/day/10
- Solución:
  https://github.com/jgonzalezmorente/advent-of-code/blob/main/AOC2023/aoc2023_10.py

Nos dan una matriz formada por los símbolos |, -, L, J, 7, F, y . Cada uno de estos símbolos representa un trozo de una tubería.

- | es una tubería vertical que conecta el norte y el sur.
- – es una tubería horizontal que conecta el este y el oeste.
- L es una curva de 90 grados que conecta el norte y el este.

- J es una curva de 90 grados que conecta el norte y el oeste.
- 7 es una curva de 90 grados que conecta el sur y el oeste.
- F es una curva de 90 grados que conecta el sur y el este.
- . es tierra; no hay ninguna tubería en este punto.
- S es la posición inicial del animal; hay una tubería en este mosaico, pero su boceto no muestra qué forma tiene.

5.1. Parte I

En la primera parte del problema hay que determinar el circuito que comienza y acaba en S y calcular el número de pasos que se necesitan para llegar desde la posición inical hasta el punto más alejado.

La idea para encontrar el circuito es la siguiente. Suponiendo que estamos en el punto $p_k \in \mathbb{Z}^2$, con vector dirección $\mathbf{v_k}$. Se determina el vector dirección $\mathbf{v_{k+1}}$ en función de $\mathbf{v_k}$ y del tipo de tubería situada en el punto p_k.

Obsérvese que para todo k, las componentes del vector $\mathbf{v_k} = (v_{k,0}, v_{k,1})$ pertenecen a $\{-1, 0, 1\}$.

- Si la tubería en p_k es de tipo |, entonces

$$\mathbf{v_k} = (0, v_{k-1,1})$$

  ya que nos movemos en el eje y en la misma dirección con la que llegamos al punto p_k.

- Si la tubería en p_k es de tipo –, entonces

$$\mathbf{v_k} = (v_{k-1,0}, 0)$$

  ya que nos movemos en el eje x en la misma dirección con la que llegamos al punto p_k.

- Si la tubería en p_k es de tipo L, entonces

$$\mathbf{v_k} = \begin{cases} (1,0) & \text{si } v_{k-1,1} = 1 \quad \text{(cuando llegamos a L por el norte)} \\ (0,-1) & \text{si } v_{k-1,0} = -1 \quad \text{(cuando llegamos a L por el este)} \end{cases}$$

- Si la tubería en p_k es de tipo J, entonces

$$\mathbf{v_k} = \begin{cases} (-1,0) & \text{si } v_{k-1,1} = 1 \quad \text{(cuando llegamos a J por el norte)} \\ (0,-1) & \text{si } v_{k-1,0} = 1 \quad \text{(cuando llegamos a J por el oeste)} \end{cases}$$

- Si la tubería en p_k es de tipo 7, entonces

$$\mathbf{v_k} = \begin{cases} (0,1) & \text{si } v_{k-1,1} = 1 \quad \text{(cuando llegamos a 7 por el oeste)} \\ (-1,0) & \text{si } v_{k-1,0} = -1 \quad \text{(cuando llegamos a 7 por el sur)} \end{cases}$$

- Si la tubería en p_k es de tipo F, entonces

$$\mathbf{v_k} = \begin{cases} (0,1) & \text{si } v_{k-1,1} = -1 \quad \text{(cuando llegamos a F por el este)} \\ (1,0) & \text{si } v_{k-1,0} = -1 \quad \text{(cuando llegamos a F por el sur)} \end{cases}$$

Una vez determinado $\mathbf{v_k}$, el punto p_{k+1} es,

$$p_{k+1} = p_k + \mathbf{v_k}$$

Para iniciar el proceso anterior necesitamos conocer el vector $\mathbf{v_1}$ el cual se obtiene dependiendo de los tipos de tubería de los puntos situados alrededor de $p_0 = (x_0, y_0) = S$.

- Si p_1 está situado al norte de S, esto es $p_1 = (x_0, y_0 - 1)$, entonces los posibles tipos de tubería que puede tener p_1 son |, 7 ó F y

$$\mathbf{v_1} = \begin{cases} (0,-1) & \text{si el tipo de tubería de } p_1 \text{ es } | \\ (-1,-1) & \text{si el tipo de tubería de } p_1 \text{ es 7} \\ (1,-1) & \text{si el tipo de tubería de } p_1 \text{ es F} \end{cases}$$

- Si p_1 está situado al sur de S, esto es $p_1 = (x_0, y_0 + 1)$, entonces los posibles tipos de tubería que puede tener p_1 son |, L ó J y

$$\mathbf{v_1} = \begin{cases} (0,1) & \text{si el tipo de tubería de } p_1 \text{ es } | \\ (1,1) & \text{si el tipo de tubería de } p_1 \text{ es L} \\ (-1,1) & \text{si el tipo de tubería de } p_1 \text{ es J} \end{cases}$$

- Si p_1 está situado al este de S, esto es $p_1 = (x_0 + 1, y_0)$, entonces los posibles tipos de tubería que puede tener p_1 son –, J ó 7 y

$$\mathbf{v_1} = \begin{cases} (1,0) & \text{si el tipo de tubería de } p_1 \text{ es } – \\ (1,-1) & \text{si el tipo de tubería de } p_1 \text{ es J} \\ (1,1) & \text{si el tipo de tubería de } p_1 \text{ es 7} \end{cases}$$

- Si p_1 está situado al oeste de S, esto es $p_1 = (x_0 - 1, y_0)$, entonces los posibles tipos de tubería que puede tener p_1 son –, L ó F y

$$\mathbf{v_1} = \begin{cases} (-1,0) & \text{si el tipo de tubería de } p_1 \text{ es } – \\ (-1,-1) & \text{si el tipo de tubería de } p_1 \text{ es L} \\ (-1,1) & \text{si el tipo de tubería de } p_1 \text{ es F} \end{cases}$$

Por tanto, para resolver esta parte del problema basta examinar los puntos de alrededor de S y determinar, en base al criterio anterior, cuales es el posible punto p_1 y vector $\mathbf{v_1}$. A partir de aquí se itera mediante la expresión $p_{k+1} = p_k + \mathbf{v_k}$ hasta encontrar el primer n tal que $p_n = S$, es decir, se cierra el circuito. El número de pasos m para llegar al punto más alejado es la mitad del número de puntos del circuito,

$$m = \frac{n-1}{2}$$

5.2. Parte II

En esta parte del problema nos piden que calculemos el número de puntos interiores al circuito.

En geometría, la triangulación de un polígono o área poligonal es una partición de dicha área en un conjunto de triángulos por un conjunto máximal de diagonales que no se cruzan.

Definición 5.1. Se llama polígono simple al polígono cuyos lados no contiguos no se intersecan, es decir, que dicho polígono es la frontera de la respectiva área poligonal.

Un polígono simple divide al plano que lo contiene en dos conjuntos de puntos: interior de la región poligonal y exterior de la región poligonal. El interior se caracteriza porque no puede contener una recta; el exterior si puede contener una recta. Un polígono que no es simple se denomina polígono complejo.

Definición 5.2. Una triangulación es una división del área en un conjunto de triángulos que cumplen las condiciones siguientes:

- La unión de todos los triángulos es igual al polígono original.
- Los vértices de los triángulos son vértices del polígono original.
- Cualquier pareja de triángulos es disjunta o comparte unicamente un vértice o un lado.

Lema 5.1. *Todo polígono simple de n vértices es triangulizable.*

Demostración. Razonamos por inducción sobre el número de vértices del polígono. Para $n = 3$, el polígono es un triángulo por lo que no hay nada que demostrar.

Sea $n > 3$ y supongamos que el resultado es cierto para todo $m < n$ (inducción fuerte) y veamos que es cierto para n.

Veamos en primer lugar que existe una diagonal. Dibujemos el polígono P de n lados. Sea v el vértice (o uno de los vértices) de más hacía la izquierda P. Sean u, w los dos vértices vecinos de v en la frontera de P. Distinguimos dos casos:

- Si el segmento $\overline{uw}$ cae en el interior de P, entonces hemos encontrado una diagonal (obsérvese que esto no es necesario pues el polígono no tiene por qué ser convexo).
- El segmento $\overline{uw}$ no cae completamente en el interior de P. Esto quiere decir que existe algún vértice en el interior del triángulo $\triangle uvw$ o sobre la diagonal $\overline{uw}$. De entre estos vértices, elijamos el (o uno de los) más alejado(s) de la diagonal $\overline{uw}$ y llamémosle v'. El segmento $\overline{vv'}$ no puede ser cruzado por ningún lado de P, porque ese lado tendría un extremo en el interior del triángulo $\triangle uvw$ más alejado de $\overline{uw}$ que v', lo que entra en contradición con la definición de este último. Por lo tanto, $\overline{vv'}$ es una diagonal de P.

En cualquiera de los casos, hemos encontrado una diagonal. Esta diagonal divide a P en dos subpolígonos de un número de vértices estrictamente menor que n. Por hipótesis de inducción, cada uno de esos subpolígonos puede ser triangulado y estas dos triangulaciones, junto con la diagonal encontrada, inducen una triangulación de P. □

Teorema 5.1 (de Pick). *Supongamos que tenemos una cuadrícula en la que cada vértice corresponde a un punto del plano cuyas coordenadas son números enteros y sea P un polígono simple que cumple que todos su vértices están situados sobre vértices de cuadrícula. Sea i el número de vértices de la la cuadrícula que quedan dentro del polígono y sea f el número de vértices de la cuadrícula que están en algún lado del polígono, es decir, los puntos frontera que tienen sus dos coordenadas enteras. Entonces el área del polígono puede calcularse de la siguiente forma*

$$A_P = i + \frac{f}{2} - 1$$

Demostración. Vamos a demostrar el resultado por inducción. Sea P un polígono simple y T un triángulo con un lado común con P. Asumimos que el teorema es cierto para P y para T de forma separada y demostraremos que también es cierto para el polígono PT formado a partir de P añadiendo T. Como P y T comparten un lado, todos los puntos frontera a lo largo del lado común, excepto los puntos extremos del lado, se convierten en puntos interiores de PT. Por tanto, llamando c al número de puntos frontera en común, tenemos que

$$i_{PT} = (i_P + i_T) + (c-2), \quad f_{PT} = (f_P + f_T) - 2(c-2) - 2$$

Por tanto,

$$i_P + i_T = i_{PT} - (c-2), \quad f_P + f_T = f_{PT} + 2(c-2) + 2$$

Como asumimos que el teorema es cierto para P y T de forma separada:

$$\begin{aligned} A_{PT} = A_P + A_T &= \left(i_P + \frac{f_P}{2} - 1\right) + \left(i_T + \frac{f_T}{2} - 1\right) = \\ &= i_{PT} - (c-2) + \frac{f_{PT} + 2(c-2) + 2}{2} - 2 = i_{PT} + \frac{f_{PT}}{2} - 1 \end{aligned}$$

Por lo tanto, el polígono A_{PT} cumple el teorema.

Por el lema anterior cualquier polígono simple puede ser triangulado. Por tanto, lo que hemos obtenido es que si el teorema es cierto para un triángulo T y para un polígono formado por n triángulos también lo es para un polígono formado por $n+1$ triángulos.

El último paso de la demostración es comprobar el resultado para cualquier triángulo. Es fácil ver que el teorema es cierto para cualquier cuadrado de lado 1. De aquí se deduce que es también es cierto para cualquier rectángulo con sus lados paralelos a los ejes. A partir de esto deducimos que la fórmula es cierta para triángulos rectángulos obtenidos a partir de un rectángulo mediante un corte por una de sus diagonales. Finalmente cualquier triángulo puede particionarse en triángulos rectángulos. □

Por último observemos que si el circuito del problema es $\{p_k\}_{k=0}^n$, podemos calcular el área encerrada, mediante

$$A = \left| \sum_{k=0}^{n} (x_{k+1} - x_k) y_k \right|, \quad p_k = (x_k, y_k)$$

la expresión anterior representa la suma de las áreas rectangulares orientadas entre cada dos puntos. Esto es un caso particular de la fórmula de Gauss para calcular el área de polígonos simples. También es conocido como el algoritmo de los cordones, debido al consante cruce de productos de las correspondientes coordenadas de cada par de vértices, similar a atar los cordones.

Teorema 5.2 (Fórmula de Gauss para el área de polígonos simples)**.** *Dado un polígono simple de vértices $p_0, p_1, \ldots, p_n$ con $p_i = (x_i, y_i)$, y $p_0 = p_n$ se tiene que el área A de dicho polígono se puede calcular mediante la expresión*

$$A = \frac{1}{2} \left| \sum_{k=0}^{n} x_k y_{k+1} - x_{k+1} y_k \right| = \frac{1}{2} \sum_{k=0}^{n} \det \begin{pmatrix} x_k & x_{k+1} \\ y_k & y_{k+1} \end{pmatrix}$$

Demostración. El área del polígono es la suma de todas las áreas orientadas de los trapezoides delimitados por cada dos vértices y el eje horizontal, es decir,

$$A = \left| \sum_{k=0}^{n} (x_{k+1} - x_k) \frac{y_k + y_{k+1}}{2} \right|$$

Desarrollando la expresión anterior y teniendo en cuenta la suma telescópica

$$\sum_{k=0}^{n}(x_{k+1}y_{k+1} - x_k y_k) = x_0 y_0 - x_n y_n = 0$$

obtenemos el resultado deseado. □

Por lo tanto, teniendo en cuenta ambos teoremas, concluimos que el número de puntos interiores al circuito del problema es

$$i = \left|\sum_{k=0}^{n}(x_{k+1} - x_k)y_k\right| - \frac{n-1}{2} + 1$$

Obsérvese que $(n-1)/2$ es la solución de la primera parte.

6. Día 12: *Hot Springs*

- Enunciado:
  https://adventofcode.com/2023/day/12
- Solución:
  https://github.com/jgonzalezmorente/advent-of-code/blob/main/AOC2023/aoc2023_12.py

6.1. Parte I

En este problema nos dan una sucesión de registros proporcionados en un fichero de texto. Cada registro consta de dos partes:

- Una máscara representada como sucesión de símbolos `.` `#` `?`. Los `.` denotan resortes operativos, los `#` resortes defectuosos y los `?` pueden ser `.` o `#`. Por ejemplo, `.??..??...?##.`
- Una sucesión de números. Cada número representa un grupo de resortes defectuosos juntos. Cada grupo está separado por un `.` y deben aparecer en ese orden en la máscara. Por ejemplo, $1, 1, 3$ indica que en la máscara debe aparecer un grupo de un primer grupo de un solo `#`, un segundo grupo de un solo `#` y un tercer y último grupo formado por `###`.

El objetivo es contar a partir de cada máscara y sucesión de grupos, cuantas formas posibles hay. Por ejemplo, para `.??..??...?##.` *`1,1,3`* hay cuatro formas posibles:

(1) `.#...#....###.`

(2) `.#....#...###.`

(3) `..#..#....###.`

(4) `..#...#...###.`