Proyecto final de la asignatura Diseño y Análisis de Algoritmos

Autor :
Amanda Noris Hernández

4to Año Ciencias de la Computación
Universidad de La Habana

September 22, 2024

1 Population Size Problem

Polycarpus desarrolla una interesante teoría sobre la interrelación de las progresiones aritméticas con todo en el mundo. Su idea actual es que la población de la capital de Berland cambia con el tiempo como una progresión aritmética. Bueno, o como múltiples progresiones aritméticas.

Polycarpus cree que si escribe la población de la capital durante varios años consecutivos en la secuencia $a_1, a_2, \ldots, a_n$, entonces es conveniente considerar el arreglo como varias progresiones aritméticas, escritas una tras otra. Por ejemplo, la secuencia $(8, 6, 4, 2, 1, 4, 7, 10, 2)$ puede considerarse como una secuencia de tres progresiones aritméticas $(8, 6, 4, 2)$, $(1, 4, 7, 10)$ y (2), que se escriben una tras otra.

Desafortunadamente, Polycarpus puede no tener todos los datos de los n años consecutivos (después de todo, un censo de la población no ocurre todos los años). Por esta razón, algunos valores de a_i pueden ser desconocidos. Dichos valores están representados por el número -1.

Para una secuencia dada $a = (a_1, a_2, \ldots, a_n)$, que consta de enteros positivos y valores -1, encuentre el número mínimo de progresiones aritméticas que Polycarpus necesita para obtener a. Para obtener a, las progresiones deben anotarse una tras otra. Los valores -1 pueden corresponder a un entero positivo arbitrario y los valores $a_i > 0$ deben ser iguales a los elementos correspondientes del registro consecutivo buscado de las progresiones.

Recordemos que una sucesión finita c se llama progresión aritmética si la diferencia $c_{i+1} - c_i$ de dos elementos consecutivos cualesquiera en ella es constante. Por definición, cualquier secuencia de longitud 1 es una progresión aritmética.

Entrada

La primera línea de la entrada contiene un entero n $(1 \leq n \leq 2 \cdot 10^5)$ — el número de elementos de la secuencia. La segunda línea contiene valores enteros $a_1, a_2, \ldots, a_n$ separados por un espacio $(1 \leq a_i \leq 10^9$ o $a_i = -1)$.

Salida

Imprima el número mínimo de progresiones aritméticas que necesita escribir una tras otra para obtener la secuencia a. Las posiciones marcadas como -1 en a se pueden representar mediante cualquier número entero positivo.

2 Solución de fuerza bruta

La primera idea que viene a la mente es una solución de fuerza bruta para ver como se comporta el porblema con diferentes casos de prueba y luego comparar con otras propuestas más eficientes. Un primer acercamiento sería buscar exhaustivamente todas las formas de cubrir un arreglo dado con subarreglos que son progresiones aritméticas, utilizando recursión y backtracking. Esto puede ser costoso en términos de tiempo de ejecución, especialmente para arreglos más grandes, debido a la naturaleza exponencial de la búsqueda de combinaciones.

2.1 Backtrack

El algoritmo propuesto resuelve el problema planteado sobre las progresiones aritméticas en la secuencia de población de Berland utilizando una búsqueda recursiva y backtracking.

Descomposición del Problema

1. **Identificación de Progresiones Aritméticas**:
    - El objetivo es encontrar el menor número de subarreglos (progresiones aritméticas) que pueden concatenarse para formar la secuencia original. Esto significa que hay que buscar continuamente segmentos de la secuencia que cumplan la condición de ser progresiones aritméticas.
2. **Valores Desconocidos**:
    - La secuencia puede contener valores -1, que representan datos desconocidos. Estos valores pueden ser sustituidos por cualquier entero positivo, lo que aumenta la complejidad del problema, ya que ahora se deben considerar varias posibilidades para los segmentos de la secuencia.

Funcionamiento del Algoritmo

1. **Búsqueda Recursiva**:

- Se recorre la secuencia desde un índice de inicio. Para cada posición, itera hacia adelante para definir un subarreglo (posibles progresiones aritméticas) desde el índice de inicio hasta un índice final.
- La función verifica si el subarreglo puede ser una progresión aritmética a través de la función `IsArithmeticProgression`, que evalúa la constante diferencia entre elementos.

2. **Backtracking**:
   - Si se encuentra una progresión aritmética, se añade a la lista de coberturas actuales, y se llama recursivamente a la misma función para continuar buscando desde el siguiente índice.
   - Después de explorar una opción, el algoritmo realiza un "backtrack", eliminando la última progresión de la cobertura actual para explorar otras posibles combinaciones.

Correctitud del Algoritmo

- **Exploración Exhaustiva**: El algoritmo explora todas las combinaciones posibles de subarreglos, lo que es necesario dado que hay que considerar los valores -1, que pueden ser reemplazados por varios enteros positivos.
- **Identificación de Progresiones**: Cada vez que se forma un subarreglo, se comprueba si es una progresión aritmética, garantizando que solo se añaden subarreglos válidos a la lista de resultados.
- **Solución Óptima**: La búsqueda se detiene cuando se ha examinado toda la secuencia, lo que asegura que se encuentra la combinación mínima de progresiones aritméticas necesarias para cubrirla.

2.2 Complejidad temporal

Función `FindCoveringsRecursive`:

- Esta función es la clave en el análisis. Tiene un bucle anidado:
  - El bucle exterior (controlado por `startIndex`) se ejecuta hasta n (donde n es la longitud del array).

  - El bucle interior (controlado por `endIndex`) se ejecuta desde `startIndex` hasta n, lo que en el peor de los casos puede ser $n - $ `startIndex`.

- Además, para cada par de índices `startIndex` y `endIndex`, se crea un subarreglo y se llama a `IsArithmeticProgression`.

Función `IsArithmeticProgression`:

- La función `IsArithmeticProgression` recorre el subarreglo para comprobar si es una progresión aritmética. En el peor de los casos, esta función tiene una complejidad de $O(m)$, donde m es el tamaño del subarreglo (que puede ser hasta n).

Dado que `FindCoveringsRecursive` se llama recursivamente para cada combinación de índices, podemos considerar que la complejidad general de esta función es exponencial. Cada posible subsecuencia del array se considera, y hay 2^n formas de dividir el array en subsecuencias, ya que para cada elemento se puede decidir si incluirlo en una nueva subsecuencia o no.

- Para cada llamada a `FindCoveringsRecursive`, el bucle interior examina subarreglos y verifica si son progresiones aritméticas. Esto da lugar a una complejidad en el peor de los casos de $O(n^2)$ (debido a la creación del subarreglo y a la verificación).

Combinando todo esto, la complejidad temporal total del programa puede considerarse:

$$O(2^n \cdot n)$$

Esto significa que el tiempo de ejecución del programa crecerá exponencialmente con respecto al número de elementos n en la entrada, ya que se generan muchas combinaciones de subsecuencias y se verifica cada una de ellas.

3 Solución greedy

Analizando el comportamiento del algoritmo anterior, podemos percatarnos de que al encontrar una progresión es mejor (al menos no peor) incluir tantos elementos a la derecha del mismo como sea posible, o sea, a través de una

estrategia greedy tratar de ampliar la progresión que se tiene lo más posible. La solución entonces sería: encontrar el número más a la izquierda que no esté cubierto por una progresión, comenzar una nueva progresión con ese número (el intervalo cubierto por esa progresión será de tamaño 1) y luego intentar extender este intervalo hacia la derecha tanto como sea posible. Luego se repite este paso hasta que se cubran todos los números.

Debido a que el problema tiene una particularidad con los años en los que no se tienen datos sobre la población, se deben tener en cuenta unos pequeños detalles:

- Si no hay números fijos en un intervalo (todos son -1), podemos cubrirlo con una progresión.
- Si solo hay un número no fijo en un intervalo (todos los elementos son -1 excepto 1), entonces podemos cubrir este intervalo con una progresión.
- Si hay más de un número no fijo en el intervalo, entonces podemos calcular los parámetros de la progresión (valor inicial y diferencia). Todos los números no fijos (-1) deben coincidir con esos números.

3.1 Demostración de correctitud de la solución

Definir el criterio greedy

El criterio greedy que se utiliza en este algoritmo es el siguiente:

Criterio: En cada paso, el algoritmo selecciona el número más a la izquierda que aún no está cubierto por ninguna progresión y comienza una nueva progresión aritmética con ese número como el primer elemento.

Extensión greedy: Luego, se extiende esta progresión hacia la derecha tanto como sea posible, es decir, mientras los números adyacentes en la secuencia mantengan la misma diferencia, se siguen incluyendo en la progresión.

Demostración de la propiedad de subestructura óptima

Después de que el algoritmo greedy ha construido una progresión aritmética que cubre un conjunto de números, el problema restante consiste en cubrir los números no incluidos en esa progresión.

Este subproblema tiene la misma estructura que el problema original: hay una secuencia de números no cubiertos, y el objetivo es cubrirlos con el menor número posible de progresiones aritméticas.

El algoritmo aplica el mismo criterio greedy para resolver este subproblema, comenzando una nueva progresión con el siguiente número más a la izquierda que no esté cubierto y expandiendo esa progresión lo más posible. Así, el subproblema se resuelve óptimamente con la misma estrategia, lo que implica que la solución completa es una combinación de soluciones óptimas a estos subproblemas.

Demostración por absurdo de la optimalidad

Supongamos que el algoritmo greedy no produce una solución óptima. Esto significa que existe una solución óptima O que cubre la secuencia con menos progresiones aritméticas que la solución generada por el algoritmo greedy S.

Examinemos el primer paso en el que S y O difieren. Esto significa que, en algún momento, el algoritmo greedy seleccionó un número a_i y comenzó una progresión, mientras que O eligió otro número o progresión en su lugar.

Si O comienza su progresión en algún número a la derecha de a_i, esto deja a_i sin cubrir hasta que O comience otra progresión, lo que implica que O necesitará más progresiones para cubrir todos los números, ya que S ya ha cubierto a_i y algunos números consecutivos.

Si O comienza en a_i, entonces S y O coinciden en este paso, lo que significa que no habría diferencias hasta un paso posterior.

En ambos casos, no es posible que O cubra la secuencia con menos progresiones que S, lo que lleva a una contradicción con la suposición inicial de que O es mejor que S.

Por lo tanto, el algoritmo greedy genera una solución óptima, es decir, $S = O$.

Otro enfoque

Supongamos que la solución O utiliza al menos una progresión aritmética menos que S para expresar la secuencia de entrada.

Dado que todos los elementos de la secuencia pertenecen, por construcción, a al menos una progresión aritmética, esto implica que en la solución S existen al menos dos progresiones aritméticas I y J distintas que pueden ser combinadas o modificadas de forma que sus elementos puedan pertenecer a

una única progresión aritmética. En otras palabras, la diferencia entre elementos consecutivos (a_i (último elemento de I) y a_j (primer elemento de J)) de ambas progresiones es la misma, lo que permitiría su fusión en una sola progresión aritmética, reduciendo así el número total de progresiones utilizadas.

Sin embargo, la estrategia greedy utilizada garantiza que el primer elemento a_j a la derecha de una progresión aritmética no pertenece a la progresión si la diferencia entre a_j y a_i no es la misma que entre a_i y a_{i-1}. Esto asegura que la progresión se ha expandido de manera máxima sin dejar de cumplir las propiedades de una progresión aritmética.

Por lo tanto, si a_j pudiera pertenecer tanto a I como a J, entonces la diferencia entre a_i y a_j sería igual a la diferencia entre a_i y a_{i-1}, lo que implicaría que a_j también pertenece a I y por cosniguiente todos los elementos de J. Esto contradice la hipótesis de que I y J son dos progresiones aritméticas separadas en S.

Dado que este escenario lleva a una contradicción, concluimos que la suposición inicial de que O es mejor que S es falsa. Por lo tanto, el algoritmo greedy produce una solución óptima, es decir, $S = O$.

3.2 Complejidad temporal

La forma de leer el input es un bucle que itera n veces para leer los elementos. Por lo tanto, la complejidad de esta parte es $O(n)$.

El bucle `for` en `Main` tiene una variable l que se incrementa en cada iteración, y dentro de este bucle hay otros bucles anidados:

- Los bucles `while` que incrementan i_1 e i_2 iteran a través de los elementos en el arreglo. En el peor de los casos, estos bucles pueden recorrer todos los elementos de a, así que pueden tomar un tiempo $O(n)$ en total.
- En el caso de que haya elementos consecutivos que sean -1, ambos bucles avanzarán a través de esos elementos hasta encontrar un número fijo.

Dentro del bucle principal, también hay un bucle `while` que incrementa r, que también puede iterar a través de los elementos del arreglo en el peor de los casos. Este bucle puede ser llamado en cada iteración del bucle principal.

Cada elemento en el arreglo puede ser visitado varias veces, pero en total, no más de n veces en cada bucle anidado, por lo que la complejidad total del bucle principal será $O(n)$.

Dado que la lectura del vector es $O(n)$ y el bucle principal también es $O(n)$, la complejidad temporal total del programa es:

$$O(n)$$

3.3 Implementación

Las implementaciones tanto del algoritmo de fuerza bruta como del algoritmo greedy se encuentran en los archivos adjuntos. Los casos de prueba con los que se comprobó la correctitud de este código son los proporcionados por la plataforma Codeforces.