ISIS-1105, Diseño y Análisis de Algoritmos
Semestre 2023-20. Proyecto - Parte 1
Sergio Franco Pineda (202116614), Lina María Ojeda (202112324), Daniel Rudas Bohórquez (202112926)

1. Algoritmo de solución

1.1 Explicación el algoritmo elegido

Dado que las ranas en su primer movimiento eligen la dirección (izquierda o derecha) del resto de sus movimientos, nos interesa conocer las diferentes disposiciones de ranas y piedras después de realizar los m movimientos con una configuración de movimientos especifica, por ejemplo:

Para aclarar mejor a lo que nos referimos con configuraciones de movimientos, si tenemos 2 ranas, las configuraciones de movimientos serán las siguientes: $[(I,\ I),\ (I,\ D),\ (D,\ I),\ (D,\ D)]$. En general, tenemos un total de 2^r configuraciones de movimiento.

Ya que diferentes configuraciones de movimientos pueden resultar en disposiciones finales distintas, nuestro algoritmo calcula las disposiciones finales para cada configuración de movimientos

El caso base de nuestro algoritmo es cuando se han realizado 0 movimientos, es decir, en este caso el resultado independientemente de la configuración de movimiento es la disposición inicial, y para realizar el movimiento $n,\ 0 \leq n \leq m$ se necesita el resultado del movimiento $n-1$, y esta es la forma en la que dividimos el problema:

Lo primero que se hizo es identificar el número de ranas, los movimientos a realizar y su posición inicial omitiendo el caso base explicado anteriormente. El código genera todas las combinaciones posibles de movimientos para las ranas utilizando itertools.product y, luego, itera a través de estas combinaciones. Para cada combinación, simula los movimientos en la configuración inicial de las ranas, utilizando la función mover para verificar la viabilidad de cada movimiento y actualizar la configuración. Para evitar cálculos repetidos y mejorar la eficiencia, se emplea la memorización mediante el diccionario memo. Las configuraciones resultantes después de los movimientos se almacenan en conjuntos (resultados) para asegurarse de que no haya duplicados, permitiendo así contar con precisión cuántas configuraciones únicas se pueden obtener después de un número específico de movimientos en el juego de las ranas.

Uno de los aspectos más importantes para solucionar este problema fue el uso de la memoización es decir solucionar el problema con programación dinámica. En nuestro caso esto fue una técnica clave que, ya que al almacenamiento los resultados previamente calculados en un diccionario llamado memo, evitamos la realización de cálculos redundantes, mejorando la eficiencia del programa.
Antes de realizar cualquier cálculo o exploración, el código verifica si los parámetros y condiciones ya han sido calculados y se encuentran en memo. Si se encuentra una coincidencia, el programa recupera el resultado almacenado en lugar de recalcularlo, si no se encuentra se realiza el cálculo y el resultado se almacena en el diccionario memo. La clave para almacenar el resultado en el diccionario generalmente se forma a partir de los parámetros de entrada y las condiciones del cálculo.

Memo guarda información podría decirse de 3 formas distintas:

- Memo en la función mover se utiliza para almacenar resultados booleanos que indican si un movimiento específico es posible en una configuración dada sin violar las reglas del juego. Las claves en memo se forman utilizando la configuración actual de las ranas, la posición, la dirección y la forma de movimiento.
- Memo en la función contar_configuraciones_contador se utiliza para almacenar conjuntos de configuraciones únicas después de ciertos movimientos. Los conjuntos se usan para asegurarse de que no se almacenen configuraciones duplicadas. Las claves en memo representan la combinación de la configuración actual y la forma de movimiento.
- Memo en la función contar_configuraciones se utiliza para almacenar el resultado final, que es el número total de configuraciones únicas que se pueden obtener después de un número específico de movimientos. En este caso, las claves en memo están relacionadas con la cantidad de ranas y el número de movimientos.

1.2 Otras alternativas de implementación

Otras alternativas de implementación considerados fueron programación dinámica dividiendo la disposición inicial en diferentes listas y programación dinámica con backtracking, que es la base de nuestro algoritmo.

1.3 ¿Por qué se escogió la solución implementada y por qué resuelve el problema?

Esta fue la solución implementada porque facilitaba diferenciar las disposiciones ya encontradas o no y resuelve el problema porque se basa en el método de backtracking donde se deben encontrar todas las posibles formas de hacer algo sin el requerimiento de optimizar algún valor, es decir recorre un árbol de decisiones y toma las decisiones que satisfacen el requerimiento.

2. Análisis de complejidad espacial y temporal

2.1 Complejidad espacial

2.1 Complejidad temporal

En la función principal *contar_configuraciones* se encuentran los siguientes bloque de código:

```
formas_movimientos = calcular_combinaciones(ranas_cantidad)
resultados = set()
memo = {}
```

La primera línea tiene una complejidad de $O(2^r)$, es aquí donde se calculan todas las posibles configuraciones de movimientos.

```
identificador = 1
for posicion in range(len(configuracion_inicial)):
    if configuracion_inicial[posicion] == "r":
        configuracion_inicial[posicion] = "r" + str(identificador)
        identificador += 1
```

La complejidad de este bloque es $O(p)$, ya que se recorre cada piedra de la disposición inicial.

```
for forma_movimiento in formas_movimientos:
    configuraciones = {tuple(configuracion_inicial)}
    for _ in range(movimientos_realizar):
        nuevas_configuraciones = set()
        for configuracion in configuraciones:
            nuevas_configuraciones |= contar_configuraciones_contador(list(configuracion), forma_movimiento, memo)
        configuraciones = nuevas_configuraciones
```

La complejidad de este bloque es $O(2^r mp^2 - 2^{r+1}mp + 2^r m)$ (resultado de expandir varias multiplicaciones), ya que se itera sobre cada configuración de movimiento, después sobre cada movimiento y después sobre cada resultado de movimiento un movimiento con una configuración, que es máximo la cantidad de piedras menos uno. Adicionalmente, se llama a la función *contar_configuraciones_contador* que recorre cada configuración valida encontrada anteriormente y tiene una complejidad de $O(p-1)$ máximo, y esta llama a una función mover que tiene complejidad constante.

Por lo tanto, nuestra complejidad es:

$$O(2^r mp^2 - 2^{r+1}mp + 2^r m + p + 2^r) \Rightarrow O(2^r mp^2)$$

3. Respuestas a los escenarios de comprensión de problemas algorítmicos

3.1 Las ranas pueden devolverse

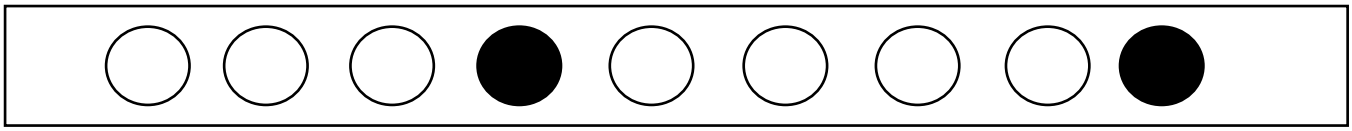
3.1.1 ¿Que nuevos retos presupone este nuevo escenario?

Este escenario añade más posibilidades que considerar para cada movimiento, es decir, una vez una rana se ha movido, están las opciones de seguir con ese moviéndose en la misma dirección o cambiar de dirección, además, este cambio podría ocurrir en durante cualquiera de los m movimientos. El reto es saber si devolverse en un momento dado es la mejor decisión o no.

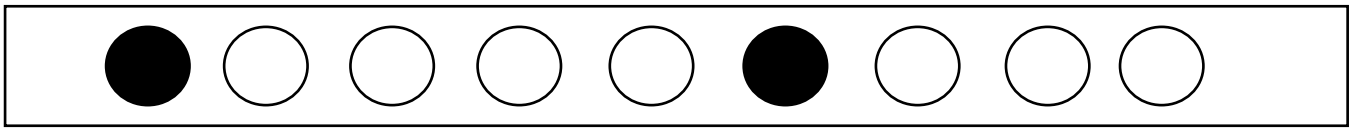
3.1.2 ¿Qué cambios le tendrían que realizar a su solución para que se adapte a este nuevo escenario?

En términos de una solución, las configuraciones de movimientos siguen siendo las mismas, pero al contrario del caso original del enunciado y de nuestra solución propuesta que cuando no podíamos realizar más movimientos para llegar a los m movimientos simplemente pasamos a la siguiente forma de movernos, en este caso y con este escenario es cuando podemos empezar a movernos en reversa para llegar a los m movimientos y no solo cuando pasa eso sino también cuando no, esto debe modelarse en nuestra función *contar_configuraciones_contador*. Por ejemplo, para una configuración de movimiento, se podría en principio repartir equitativamente la cantidad de movimientos entre las ranas y cuando se llegue a la cantidad deseada, reducir el movimiento de una rana (mover en el sentido contrario) para añadirle ese movimiento a otra rana y encontrar otra posible disposición, como se muestra a continuación:

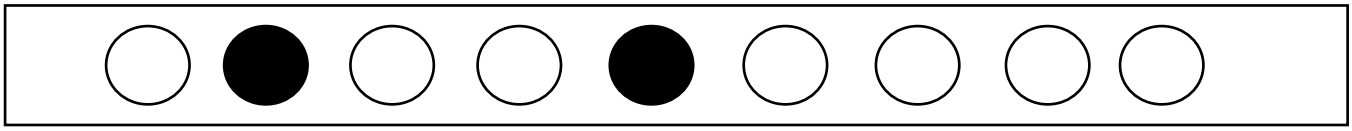
$$m = 6$$



3 movimientos para la primera rana y 3 para la segunda



2 movimientos para la primera rana y 4 para la segunda



Y así sucesivamente.

3.2 Todas las ranas deben ir en la misma dirección

3.2.1 ¿Que nuevos retos presupone este nuevo escenario?

A diferencia del caso anterior, este escenario reduce posibilidades y por lo tanto filas en nuestra matriz, todo lo demás sigue siendo igual.

3.2.2 ¿Qué cambios le tendrían que realizar a su solución para que se adapte a este nuevo escenario?

Si todas las ranas estuvieran inicialmente en la misma dirección en lugar de estar mezcladas, la implementación del código simplificaría significativamente. A pesar de esta simplificación, aún sería necesario verificar que no haya ranas en el lugar de destino al moverse, ya que las ranas solo pueden moverse a piedras vacías y no pueden ocupar el mismo espacio. Sin embargo, la simplificación principal radicaría en la eliminación de la complejidad asociada con las restricciones de movimiento en múltiples direcciones (izquierda y derecha) y la generación de combinaciones de movimientos. La verificación se centraría únicamente en la dirección en la que todas las ranas están inicialmente colocadas, lo que simplificaría significativamente la lógica del código en comparación con el caso en el que las ranas se mueven en direcciones opuestas. Por lo tanto, aunque la verificación de las posiciones de destino de las ranas aún sería esencial, la implementación en general sería más sencilla y directa, lo que facilitaría la comprensión y el mantenimiento del programa.

dada nuestra implementacion nuestro codigo cambiaria de la sigueinte manera:
contar_configuraciones_contador, la lógica de verificación de la disponibilidad de una piedra vacía en el destino de una rana todavía sería esencial para garantizar que las configuraciones se cuenten de manera precisa y de acuerdo con las reglas del juego. Esta verificación es crucial incluso cuando todas las ranas están en la misma dirección, ya que asegura que las ranas solo se muevan a piedras vacías y no violen las restricciones del juego.

función mover: En este escenario, la función mover se volvería innecesaria, ya que no habría movimientos de ranas en direcciones opuestas. Por lo tanto, se podría eliminar por completo. Esta función originalmente se encargaba de verificar si un movimiento era posible y realizarlo, lo cual ya no sería relevante.

Función calcular_combinaciones: La función calcular_combinaciones generaba todas las combinaciones posibles de movimientos ('I' y 'D') para las ranas. En este nuevo contexto, dado que las ranas se mueven en la misma dirección, no sería necesario generar estas combinaciones. La función podría simplificarse o eliminarse por completo.

función contar_configuraciones: La lógica principal de esta función se simplificaría sustancialmente. Ya no sería necesario iterar a través de múltiples combinaciones de movimientos y configuraciones de ranas en direcciones opuestas. En su lugar, simplemente contarías las configuraciones únicas de ranas en la dirección en la que están inicialmente colocadas, lo que resultaría en una función más directa y fácil de entender.