Algoritmos y Estructuras de Datos III

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Práctico 1 - Pato-lógico

Integrante	LU	Correo electrónico
Ortiz de Zarate, Juan Manuel	403/10	jmanuoz@gmail.com
Martelletti, Pablo	849/11	pmartelletti@gmail.com
Kujawski, Kevin	459/10	kevinkuja@gmail.com
Carreiro, Martin	45/10	carreiromartin@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Introducción

En el siguiente trabajo presentaremos la resolución a tres problemas que se nos dio a resolver. Mostraremos para cada uno el algoritmo que utilizamos para resolverlo describiendo que técnicas de las vistas en clase utilizamos, mostraremos también el análisis de complejidad de cada algoritmo y mostraremos gráficamente los resultados de las mediciones para ver si se cumple el análisis teórico.

2. Problema 1

2.1. Enunciado

Hay que encontrar la longitud de la meseta más grande de la unión ordenada de dos arreglos ordenados crecientemente (no estricto), definiendo a la meseta como el subcojunto de números consecutivos iguales.

2.2. Solución

El problema presenta dos subproblemas a resolver: realizar la unión ordenada y encontrar la longitud de dicha meseta. Para el primer subproblema, utilizaremos el algoritmo de Merge ya conocido por Merge-Sort en el que se irán recorriendo linealmente ambos arreglos, tomando en cuenta que están previamente ordenados y se irán colocando, en un nuevo arreglo de tamaño de la longitud de los arreglos de entrada, los valores de dichos arreglos de forma ordenada. Una vez encontrada la unión ordenada, buscamos la longitud de la meseta más grande. La idea consiste en recorrer linealmente el arreglo obtenido previamente e ir contando todos los valores iguales consecutivos encontrados. En el momento que el siguiente valor al evaluado sea distinto significa que es el inicio de otra meseta y hay que chequear si la longitud de la meseta anterior es la máxima hasta el momento. Para ello, utilizaremos una variable como acumulador que dentro del caso que el siguiente valor a evaluar sea distinto y el contador de la meseta actual sea mayor al anterior (en caso de existir), se acumulará esta nueva longitud en dicha variable. Notese que, al final del algoritmo se devuelve acum + 1, esto es debido a que empezamos a contar a partir de una meseta mayor a dos, anotando en contador solo 1, ya que al saber que la entrada es no vacía, se estaría guardando el valor de la longitud de las mesetas de longitud uno que eso siempre existe al ser no vacía.

2.3. Pseudocódigo

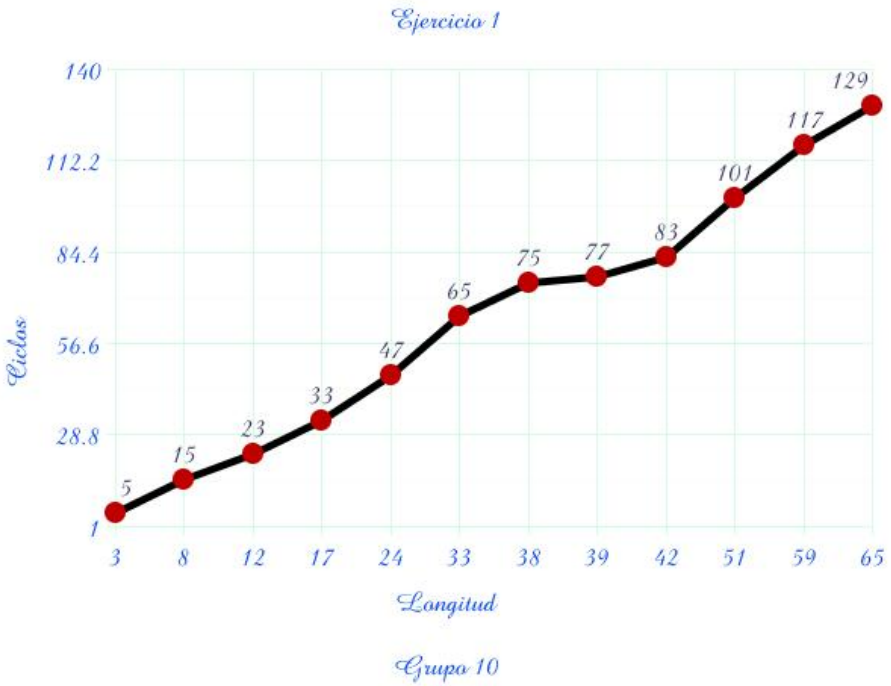
```
arrayLongMeseta (in array1 ; in array2)  ⟶  res : Int
 1  ArrayMerged = MergeOrdenado(Array1,Array2)
 2  cont = 0;
 3  acum = 0;
 4  Mientras( i < |ArrayMerged| - 1)
 5        si (ArrayMerged[i] == ArrayMerged[i+1])
 6              cont ++;                          ▷ Si estoy dentro de la misma meseta, sigo contando
 7        sino
 8              si (cont > acum)
 9                    acum = cont;          ▷ Si estoy fuera de la meseta y el contador previo supera mi acumulador, lo guardo
10              cont = 0;                          ▷ Como estoy fuera de la meseta, reseteo contador
11  si (cont > acum)
12        acum = cont;                                          ▷ Esto es para el último caso
13  devuelvo acum + 1;
```

2.4. Analisís de complejidad

```
 1  ArrayMerged = MergeOrdenado(Array1,Array2)
    Este algoritmo lo tomaremos como conocido junto a su complejidad O(|Array1| + |Array2|)

    Al no tener llamadas recursivas, el análisis del algoritmo es bastante simple e intuitivo.
 2  Mientras( i < |ArrayMerged| - 1)
    Se recorren todos los elementos del ArrayMerged sin excepción
 3        si (ArrayMerged[i] == ArrayMerged[i+1])                          ▷ O(1)
 4              cont ++;                                                   ▷ O(1)
 5        sino                                                             ▷ O(1)
 6              si (cont > acum)                                           ▷ O(1)
 7                    acum = cont; ▷                                       ▷ O(1)
 8              cont = 0;                                                  ▷ O(1)
 9  si (cont > acum)                                                       ▷ O(1)
10        acum = cont;                                                     ▷ O(1)
11  devuelvo acum + 1;
    Entonces es \sum_{i=0}^{n} O(1), donde n es |Array1| + |Array2|

    Entonces la complejidad es: O(2*(|Array1| + |Array2|))

    Lo que implica: O(|Array1| + |Array2|)
```

2.5. Tests y Gráficos

En el test del problema en cuestión, consideramos que no hay peores ni mejores casos posibles. Por ende, cualquier caso elegido para testear nos representa lo mismo. Por ende, probamos unos casos random para ver la linealidad en base a la longitud de la unión de los arreglos. Para probar hicimos script como test que genera arreglos ordenados aleatorios (tanto a longitud como los valores internos). El test en si, toma estos valores y escribe la cantidad de ciclos obtenidos para cada longitud retornando esos valores en un archivo. El siguiente gráfico está basado en ese resultado.



Como se puede observar, tomando la cantidad de ciclos utilizados en base a la longitud se nota claramente la linealidad del algoritmo.

2.6. Conclusiones

Como pudimos ver en el análisis de complejidad nuestro algoritmo cumple con la complejidad requerida y esta complejidad es igual para cualquier dato de entrada, es decir, que no hay "casos límite", la complejidad es de O(|x|+|y|) siempre. Destacamos en este ejercicio la practicidad de generar tests ya que si en algún momento se decide hacer un cambio, y para un primer intento de demostración de correctitud, correr los tests y ver los gráficos de los mismos, es un buen parámetro para empezar. Esto es debido a que los inputs de los test se generan automáticamente (de forma aleatoria y la cantidad de inputs deseados) y luego sobre cada uno de esos datos de entrada se corre el algoritmo que termina escribiendo el resultado de cada ejecución en el arhcivo .out.

3. Problema 2

3.1. Enunciado

En éste problema se nos pide que dada una matriz de n x m, hallemos la cantidad de elementos de la meseta más extensa. Para ello, definimos como meseta al conjunto de celdas adyacentes sobre sus lados que comparten el mismo valor. Se nos pide, además, que resolvamos el ejercicio en, como máximo, $O((n * m)^2)$.

3.2. Solución

Para la resolucón de éste problema, lo que decidimos hacer fue, para cada celda de la matriz, contabilizar la cantidad de elementos que le son adyacentes y comparten su mismo valor. Para ello, recorremos recursivamente las celdas adyacentes de cada una, hasta que encontremos una cuyo valor sea diferente al que estemos analizando. Una vez encontrado dicho valor, y sabiendo la cantidad de elementos de la meseta para dicha celda, analizamos si es mayor o menos a las otras mesetas encontradas. Si es menor, la descartamos; caso contrario, guardamos la longitud de la misma hasta tanto encontremos una nueva meseta que sea mayor a la recientemente encontrada.
La forma que utilizamos para contar la cantidad de elementos de la meseta, como dijimos antes, es recursiva. Ésta se lleva a cabo contando, para cada celda, las posibles componentes de la meseta en una dirección de las 4 posibles (arriba, abajo, izquierda, derecha). Si en una de esas direcciones encontramos un componente de la meseta (el mismo valor de la que comenzamos) y además, esa celda no fui visitada anteriormente (es decir, no es parte de ninguna meseta ya existente), la recorremos recursivamente de la misma forma que recorrimos la celda original: para cada dirección posible, analizamos si las celdas adyacentes son parte de la meseta y de serlo, analizamos cada una de esas celdas nuevamente, de forma recursiva. Así procederemos hasta encontrar que no hay más celdas adyacentes que pertenezcan a la meseta (o, en el peor caso, que se termine nuestra matriz), y luego contaremos todas las *submesetas* encontradas con el mismo valor, y que sea adyacentes, y las sumaremos entre si para calcular la cantidad de elementos de la meseta estudiada.
Para ir recorriendo las distintas celdas de la matriz, creamos una *matriz auxiliar*, del mismo tamaño que la original, con sus valores todos en 0, y a medida que vayamos visitando las distintas celdas i, j de la matriz original, iremos marcando como visitada (con un número 1) las celdas i, j de la *matriz auxiliar*.
Lo importante para mantener la complejidad lo más baja posible, y evitar visitar varias veces la misma celda, está en utilzar nuestra *matriz auxiliar* para saber si ya pasamos o no por dicha celda, y no vistarla nuevamente. La razón por la que podemos hacer ésto es que si ya visité dich celda en otra ocasión, entonces no puede pertenecer a la meseta que estoy estudiando en éste momento y, por tanto, no debo considerala.

3.3. Pseudocódigo

RESOLVER (**in** *matriz*) $\longrightarrow$ *mesetaMaxima* : Int

```
1 arrayAux = CrearArrayAux(arrOrig)
2 Para cada celda de la matriz Hacer:
3       mesetaMaximaCelda = Contar(celda)
4       Si mesetaMaximaCelda es mayor a la mesetaMaxima actual Hacer:
5             Reemplazo el valor de mesetaMaxima por el de mesetaMaximaCelda
6 devuelvo el valor de la mesetaMaxima
```

CONTAR (*celda*) $\longrightarrow$ *meseta maxima de la celda* : Int

```
1 largo de la meseta = 1
2 Para cada una de las celdas adyacentes Hacer:
3       sumo al largo de la meseta el tamaño de la meseta de la celda adyacente
4 devuelvo el largo de la meseta
```

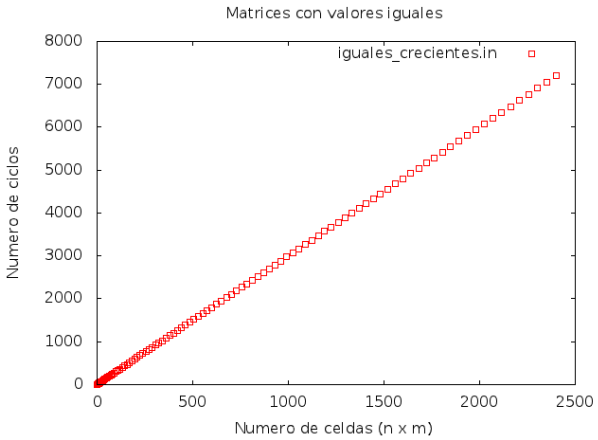
3.4. Analisís de complejidad

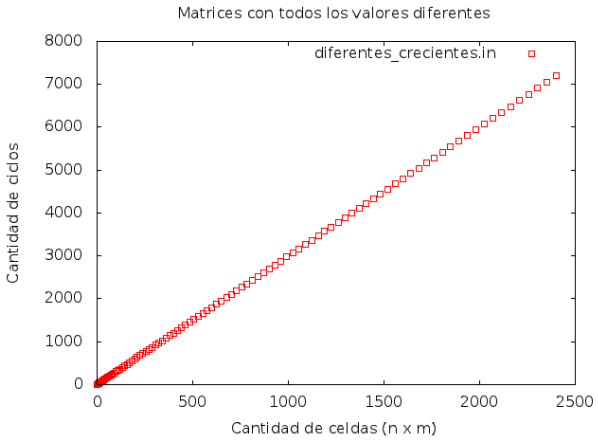
En éste ejercicio, podríamos mencionar como 3 las partes importantes que afectan a la complejidad del mismo: la creacion de la matriz leyendo del archivo de entrada, la creación de una matriz auxiliar y el recorridorecursivo de la matriz para ir sabiendo la cantidad de elementos pertenecientes a una meseta tiene cada celda. El enunciado nos pedía que, como máximo, la complejidad del algoritmo debía ser de $O((n * m)^2)$ y, de acuerdo a nuestro cálculos, nuestro algoritmo baja dicha complejidad notoriamente, ya que realiza todos los pasos (y para todos los casos, como veremos en la parte de Test y Gráficos) en $k(n * m)$, donde $k \approx 3$.
Decimos que k es "aproximadamente 3" ya que no estamos contabilizando, por ejemplo, la complejidad de crear un ArrayList o de agrandarlo una vez que todos sus elementos están llenos, pero éstos se tornan casi despreciables a partir de valores de m y n relativamente pequeños.
Por otro lado, como dijimos, las 3 pasos más importantes del ejercicio, se realizan en la misma cantidad de ciclos, ya que en todos ellos, lo más importante que tiene que hacer el algoritmos es recorrer celda por celda una matriz (fila a fila). Cómo sabemos, la complejidad de recorrer una matriz depende del tamaño de la matriz (n x m), y por tanto podemos decir que la complejidad del ejercicio es $O(m * n)$, ya que el 3 es despreciable para entradas muy grandes.

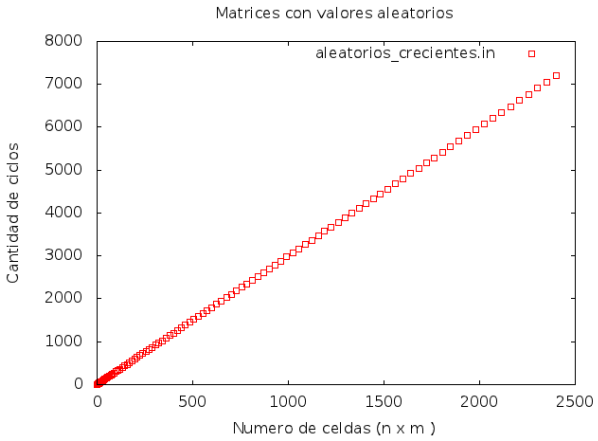
Por último, y con respecto a por qué nuestro algoritmos de *Contar*, a pesar de ser recursivo, es considerado lineal, se debe a que nuestra implementación hace uso de una matriz auxiliar que, si bien crearla es igual de costosa que recorrer toda la matriz una vez, a la larga nos ahorra visitar celdas que ya fueron visitadas y no debería volverse a analizar. Esto es posible ya que el enunciado nos dice que sólo debemos contabilizar los valores iguales que estén en mesetas adyacentes y, por tanto, si ya visitamos una celda adyacente a la que estamos analizando, se debe únicamente a dos situaciones: o bien, ya la contabilizamos en nuestra meseta actual, o bien, ya fue visitada mientras contabilizabamos otra meseta. Cualquiera sea la situación, es claro que en ninguna de las dos deberíamos contabilizar la celda. Por tanto, podemos decir que, a pesar de ser recursivo, nuestro algoritmos, ayudado de su *matrix auxiliar*, se comporta de manera linear.

3.5. Tests y Gráficos

En ésta seccion mostraremos los resultados de nuestras pruebas que utilizamos para comprobar que, en todos los casos posibles, el ejercicio actúa siempre de la misma forma.
Los casos que tuvimos en cuenta fueron 3: aquellos donde los valores de la matriz son todos distintos (y, por tanto, la meseta máxima sería 1), aquellos donde son todos iguales (donde habría una sola meseta, la matriz entera, y su valor sería el de mxn) y, por último, aquellos en que los valores son aleatorios (y de donde no podemos inferir nada acerca del tamaño de la matriz). Todos los casos los analizamos contra la misma variable: cantidad de ciclos realizados por nuestro ejercicio. Para ello, generamos valores de entradas que cumplieran dichas pautas (que los valores de las celdas sean iguales, distintos o aleatorios), y con valores de n y m variables.







Como se puede apreciar en los tres gráficos, el resultado de los test está mostrando claramente que la complejidad de los algoritmos es totalmente independiente de cómo recibimos la entrada (ordenada, desorneda, totalmente aleatoria), sino más bien que es linealmente dependiente de el tamaño de la entrada, en éste caso, n y m.

3.6. Conclusiones

Como pudimos ver en nuestro análisis y en los test, se puede decir que el ejercico desarrollado es muy eficiente, ya que tarda menos de lo que nos pide el enunciado y, en particular, menos de lo que tardaría con otras técnicas algorítmicas (como lo es Programación Dinámica, que en su momento se pensó en utilizarla para resolver éste ejercicio). Por ejemplo, si hubieramos desarrollado con PD, iríamos visitando celda por celda, y para cada celda debería calcular el valor de todas las submesetas existentes, y luego ir sumandolas. De todas formas, hay celdas que contaríamos o visitaríamos más de una vez, y si bien la complejidad no sería mucho peor a la que estamos manejando en nuestra solución, se visitarían más celdas de las necesarias con nuestra implementación.

4. Problema 3

4.1. Enunciado

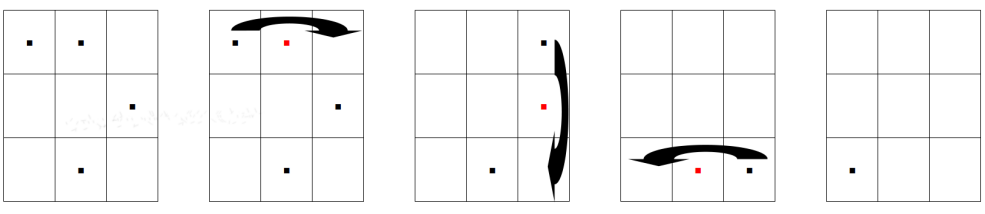
El problema es una generalización del juego "Uno Solo". La idea es mover una pieza por vez "comiendo" mediante un salto una ficha adyascente siempre y cuando este vacío el siguiente casillero al adyacente. El objetivo es eliminar todas las piezas, dejando solo una en el tablero.

4.2. Solución

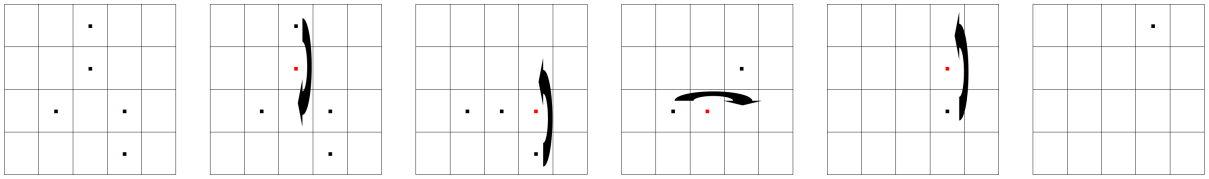
El algoritmo presenta no solamente la existencia de un resultado, si no también el resultado del mismo (en caso de haber más se devuelve el primero encontrado). La solución presentada utiliza el método de backtracking. La idea del algoritmo es ir recorriendo todas las fichas, corroborar que puedan realizar algún movimiento posible (dadas las reglas), realizarlo y ver si en esta instancia del tablero tiene solución. En caso de no tener solución, vuelve a la instancia anterior deshaciendo el movimiento antes mencionado y prueba con el moviemiento de esa ficha, si es que tiene, o la próxima ficha. Ahora explicaremos más en detalle y la correctitud del algoritmo:
El caso base es cuando el tablero tiene una sola ficha, lo que significa que ya está resuelto. Por lo que devuelve true. El caso recursivo recorre la lista de fichas , si la ficha está activa, es decir no fue comida, se prueban los cuatro moviemientos posibles, en caso de ser legal, realiza el movimiento, se apila el moviemiento, y se realiza la recursión sobre el tablero con el movimiento hecho. Para mover la ficha, ideamos una clase tablero que contiene operaciónes y una estructura para facilitar y abstraer la matriz (el tablero) del algoritmo principal. Una de estas operaciones es la de mover, que básicamente cambia de valor en la matriz ,basándose en el movimiento previamente elegido, la posición de la ficha otorgada por parámetro. Queremos aclarar que la ficha obtenida no es una posición si no que es del tipo movimiento. Este tipo movimiento nos permite almacenar la posición de la ficha y el movimiento (arriba, abajo, izquierda, derecha) a realizar. Volviendo al mover, esta operación, a partir del "movimiento"dado, calcula el lugar donde va a quedar la ficha, la ficha qué va a comer y preserva la posición anterior de la ficha movida. Para ello, en la matriz mantenemos los índices de la lista de fichas. Primero, recupero el índice de la ficha guardada en dicha matriz según su posición actual. Con este índice, actualizo en la lista de fichas, la nueva posición considerando el salto para çomerz apilo el índice de la que comí en caso de que el recorrido elegido no sea el verdadero y tengamos que deshacer el movimiento mencionado. Por último, obtengo de vuelta el índice de la ficha comida, para poder marcarla como çomidaz en la matriz invalidar el índice, simulando la desaparición de la ficha. Volviendo al algoritmo principal, después del movimiento, aplicamos recursivamente la solución en base a esta nueva versión del tablero. Una vez que retorna un valor, true o false, hay que separar en casos. En caso de que el valor sea veradero, implica que se encontró un resultado. Lo que implica el final del algoritmo. Recordemos que todos los movimientos fueron apilados de forma tal que solo resta devolver en orden esta pila. En caso de que el valor sea falso, implica que no se encontró una solución para esta instancia de tablero. Por lo tanto, deshacemos dicho movimiento, borramos el movimiento fallido antes apilado y recorremos el siguietne moviemiento, ya se de esta ficha o la siguiente. Para deshacer el movimiento, pasamos por parámetro el movimiento y el índie de la ficha movida, calculamos la posiciones para restaurar el movimiento. Esto es con el índice, actualizo en la lista de fichas la nueva posición, previamente calculando según la dirección. Actualizo los índice de la matriz, ya que ahora volvemos a tener las fichas que previamente fueron movidas y comidas. Y por último, marco como activa, es decir en juego, la última que comió.
Dado el algoritmo queremos demostrar la correctitud (informalmente) diciendo que ya que recorre las fichas, y sabemos que son finitas por precondición el algoritmo termina. Por otro lado, sabemos que recorremos todos los posibles movimientos. Con esto quremos decir que no hay posibilidad no recorrida, así que si tiene solución, esta existe dentro de todas estas posibilidades. Sin emargo, no es fuerza bruta ya que mide en tiempo constante, por cada instancia de tablero, aquellos movimientos que son posibles y reales (legales) podando ramas del árbol de posibilidades que sabemos que no pueden ser parte de la solución.

Para un mejor entendimiento incluímos ejemplos de cómo resuelve el algoritmo:

Caso 3x3

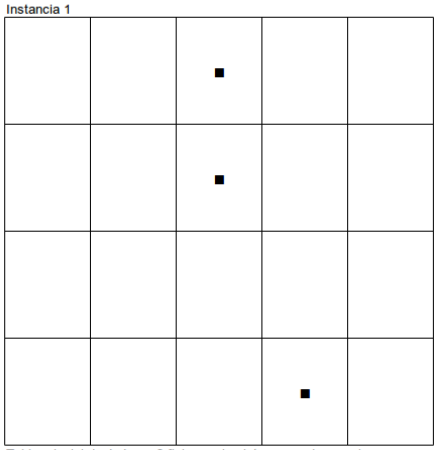


Caso 4x4



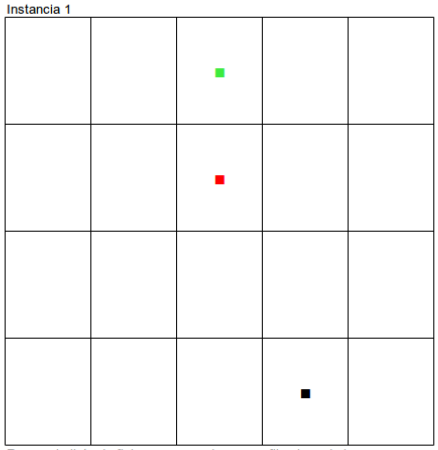
Caso 4x4 - Sin Solucion

Instancia 1



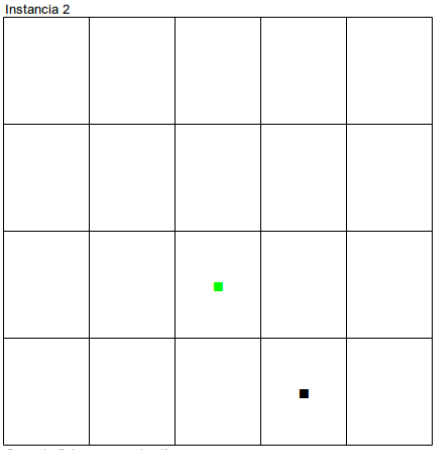
Tablero incial de 4x4 con 3 fichas, a la vista se puede apreciar que no tiene solución. Veamos como lo resuelve el algoritmo.

Instancia 1



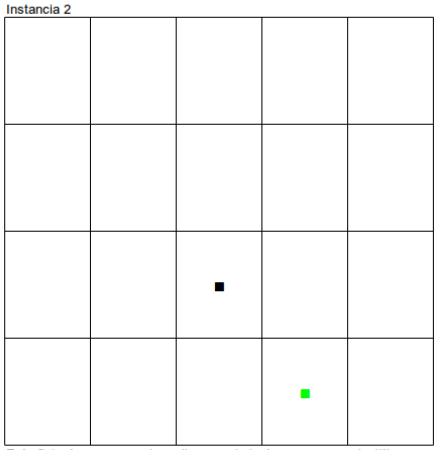
Recorre la lista de fichas, y por cada una se fija si puede hacer un movimiento legal.
En este caso encontró un movimiento de la ficha para comerse a la de abajo

Instancia 2



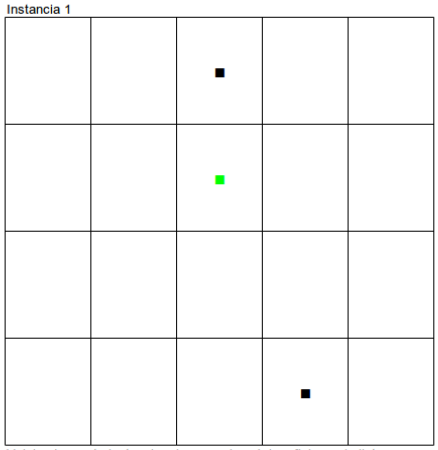
Come la ficha y pega el salto
Luego vuelve a recorrer la lista. En este caso no puede realizar ningún movimiento, entonces sigue a la próxima ficha.

Instancia 2



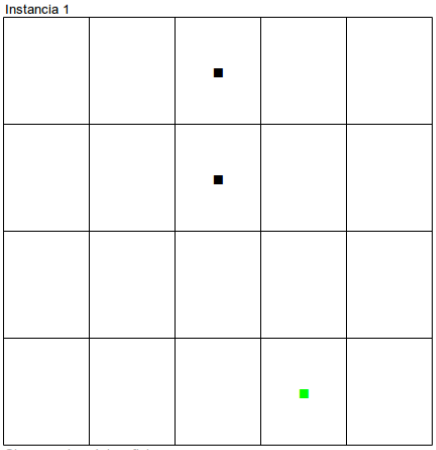
Esta ficha tampoco puede realizar movimientos, y como es la última en la lista tiene que volver a la instancia anterior ya que en esta no encontró solución.

Instancia 1



Volviendo a esta instancia, sigue con la próxima ficha en la lista.
No tiene movimientos para realizar.

Instancia 1



Sigue con la próxima ficha.
No tiene movimientos para realizar y como es la última en la lista tiene que volver a la instancia anterior.
En este caso, como es la instancia "raíz" termina el algoritmo sin encontrar solución.

4.3. Pseudocódigo

SOLUCION (**in** *tablero*, **in/out** *pila*) $\longrightarrow$ *encontrado* : Bool

```
1  Si cantidad de fichas == 1 Hacer:
2        return TRUE
3  Si no
4        Para cada ficha f en el tablero
5              Para cada movimiento m
6                    Si es m movimiento legal
7                          mov = crearMov(f,m)
8                          tablero = moverficha(mov)
9                          pila = apilarMovimiento(mov)
10                         encontrado = Solucion(tablero,pila)
11                         Si encontrado
12                               return TRUE
13                         tablero = deshacerMovimiento(mov,f)
14                         pila = desapilarMovimiento(mov)
15       return FALSE
```

Queremos volver a aclarar que el algoritmo presenta no solo si existe una solución si no también el camino, que tenemos guardado en la pila pasada por parámetro, solo faltaría imprimirlo.
Esta función de imprimir existe en el código pero para el contexto de esta sección decidimos no explicarlo ya que no hace a la solución del enunciado.

4.4. Análisis de complejidad

El algoritmo recorre una lista de fichas de longitud t. Al recorrerla, si la ficha esta activa y el movimiento es legal (de los cuatro posibles). se realiza el movimiento y se tiene una ficha marcada como inactiva. En este momento, se intentara buscar una solucion a esta nueva instancia, que, suponiendo el peor caso, por cada ficha mueve el tablero e intenta buscar una solución a este recursivamente. Esta recursión vuelve a recorrer todas las fichas (discriminando entre las activas) y todos sus movimientos posibles hasta llegar al caso base de una sola ficha, siendo la profundidad de t (quiere decir que maximo por cada instancia se llamara la solucion hasta agotar todas las fichas). Entonces, queda:

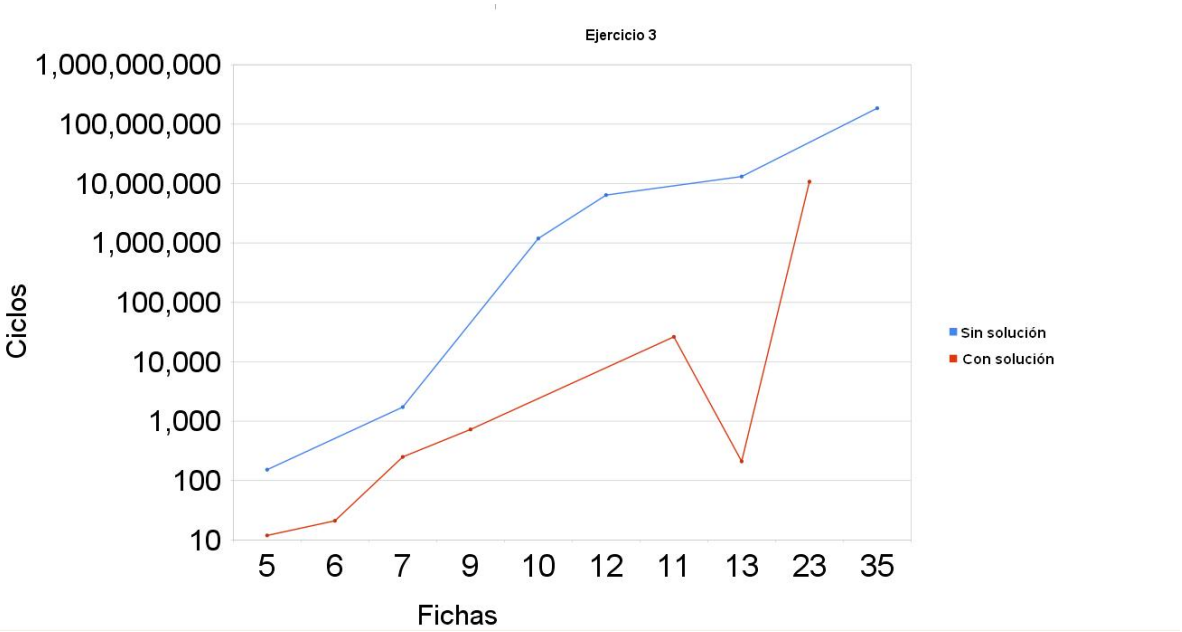
1 -Caso Base
2 -Ciclo fichas ▷ O(t)
3 Ciclo movimientos ▷ O(4)
4 Paso Recursivo ▷ O($(4t)^{t-1}$) Resto de las operaciones son O(1).

Entonces si la complejidad de los ciclos de recorrido de fichas y movimientos es de O(t) * O(4) = O(4t), y a su vez esta complejidad se efectua en cada paso recursivo, que como maximo tiene una profundidad de t, la complejidad total maxima quedara puesta en O(4t) 't' veces.
En otras palabras, como maximo, la lista de fichas y movimientos (O(4t)) se realizara hasta "comer" todas las t fichas, dejando el siguiente producto:

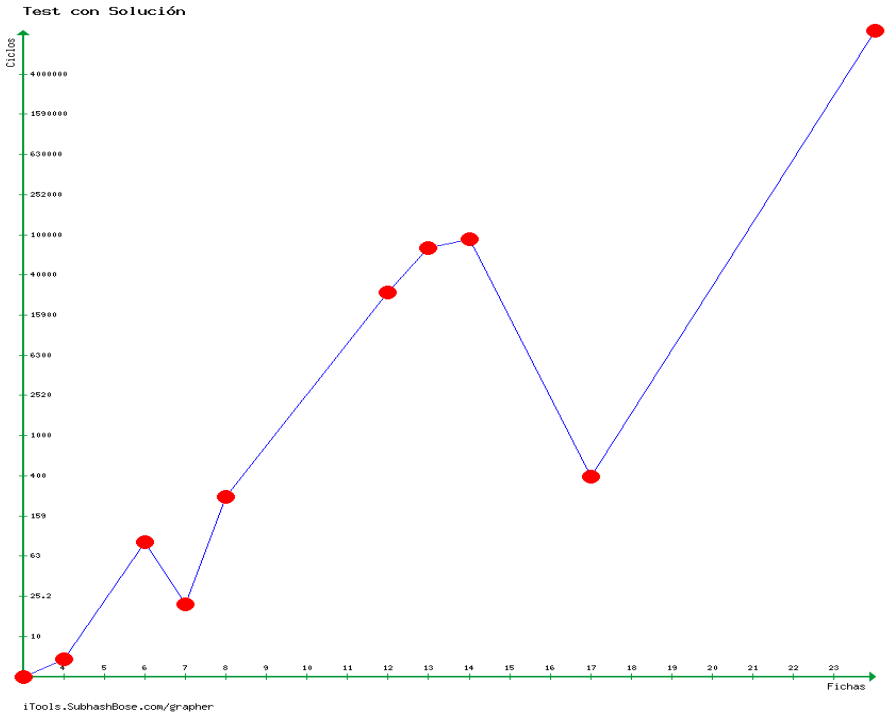
$$\prod_{i=0}^{t} O(4t) = \mathrm{O}((4t)^t)$$

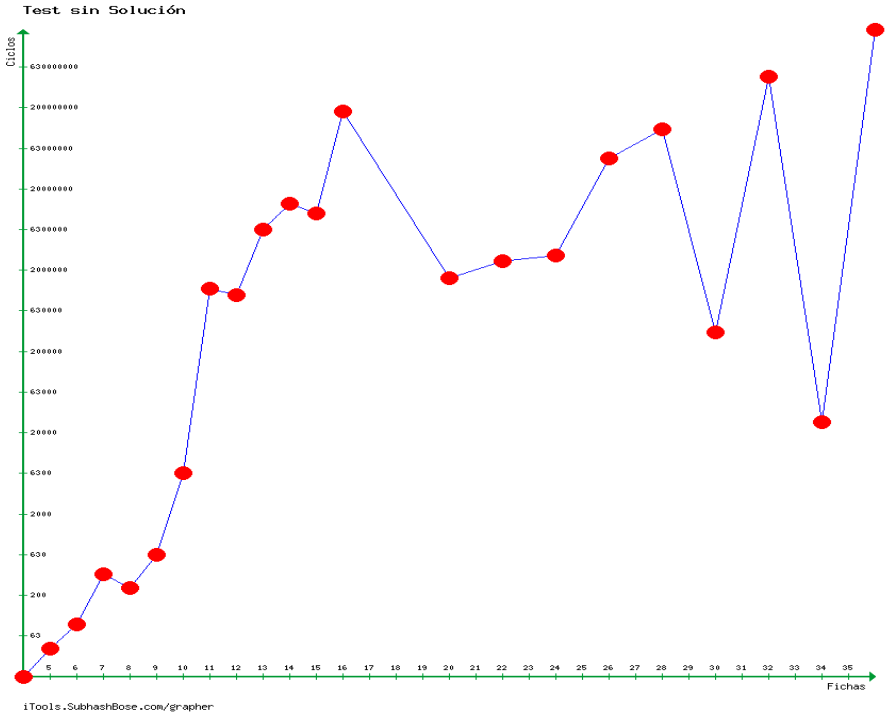
4.5. Tests y Gráficos

En los siguientes gráficos, podemos apreciar que la complejidad del algoritmo varía notoriamente entre los tableros con solucion y aquellos que no tienen solución .



Como podemos ver, aquellos que no tienen solución son de orden exponencial en la cantidad de fichas, ya que recorrerá todas ellas, y todos los movimientos posibles de ellas, sin poder ir descartando o *podando* parte de la solucion. Esto es porque, si no tiene solucion, es que cada ficha o bien esta aislada o no puede llegar a comer a las otras, y el algoritmo no puede darse cuenta de eso, por lo que las analiza para encontrar soluciones, que nunca encontrará.
Lo contrario pasa con los tableros que si tienen solucion. Con un poco de azar (al comenzar a simular con la ficha correcta), podemos encontrar una solución en tiempo lineal, ya que si escogemos a la primera vez la que resuelve el problema, sólo tendremos que simular hasta que termine. En caso de que no encontremos a la primera vez la ficha indicada, la complejidad se torna un poco mayor, pero no alcanza el tiempo exponencial de los casos sin solución ya que, a diferencia de ellos, en éstos se van *podando* soluciones, de forma tal que NO se recorren todas las fichas y sus movimientos posibles.
A continuación, mostramos las pruebas realizadas, donde sólo tomamos en cuenta tableros con hasta 35 fichas, ya que con más fichas, la simulación tarda demasiado en terminar (de hecho, los tableros más grandes tardaron bastante en finalizar), pero suponemos en forma teórica que el crecimiento seguirá de la misma forma, ya que no podemos analizarlo en forma práctica.





Con estos gráficos podemos notar que no solo la cantidad de fichas afecta al tiempo, si no también la posición de las mismas. Con esto queremos decir, que existen casos en donde si bien no tiene solución, podemos darnos cuenta más rápido. Los casos son aquellos en donde no existe ningún movimiento para realizar de entrada (ejemplo: todo el tablero lleno de fichas), y esto aplicado recursivamente.
Podemos llegar a predecir que la complejidad mínima, sin contar el caso de que haya una ficha o ninguna, es O(t), siendo t: la cantidad de fichas.

4.6. Conclusiones

Concluimos que era necesario utilizar una metodología como la de backtracking para la realización de este algoritmo, ya que, para encontrarle una solución a las diferentes entradas se necesitaba ir recorriendo todos los movimientos posibles de las fichas y podando casos que no sirvan para la resolución. Al ir cualculando siempre instancias diferentes del tablero a solucionar, resolverlo con programación dinamica no hubiese sido lo mejor ya que no habia resultados que guardar para no volver a procesarlos. Si bien el algoritmo de backtracking puede llegar a ser muy ineficiente (ver Complejidad), eso solo ocurre cuando no puedo descartar ninguna de las ramas, caso bastante patológico e improbable de ocurrir, que se suele dar cuando la cantidad de fichas es mayor a la cantidad de espacios vacios. También cabe destacar que si bien la complejidad es alta (es exponencial), para los casos donde las fichas son menos que los espacios vacios y tiene solución, la complejidad esta muy por debajo de la cota máxima.