INFORME
LAS TORRES DE HANÓI
Práctica 2

Diseño y Análisis de Algoritmos
Universidad de La Laguna

Óscar Darias Plasencia

INTRODUCCIÓN

Las Torres de Hanói es un rompecabezas o juego matemático inventado en 1883 por el matemático francés Édourd Lucas. Consiste en un determinado número de discos de radio creciente, que se insertan en una de tres varillas, ordenadas según el tamaño de su radio. El objetivo consiste en mover la pila completa a otra varilla, siguiendo las siguientes reglas:

- Sólo se puede mover un disco al mismo tiempo.
- Un disco nunca podrá estar encima de otro más pequeño.
- Sólo puede desplazarse el disco que se encuentra en lo más alto de la pila.

Se puede saber el número mínimo de pasos necesarios para solucionar el problema calculando $2^n - 1$, siendo n el número de discos, es decir, el tamaño del problema. Por tanto, podemos adelantar que la complejidad del algoritmo tradicional es de $\Theta(2^n)$, pero ya lo analizaremos más en profundidad.

Hemos implementado un algoritmo para la resolución de este problema que sigue los principios establecidos por la estrategia de programación Divide y Vencerás. La resolución recursiva es un código bastante pequeño pero que alcanza una complejidad bastante elevada. También hemos implementado un algoritmo que resuelve el problema siguiendo las reglas de las Torres de Hanói cíclicas, también recursivo y basado en la estrategia Divide y Vencerás.

ALGORITMO TRADICIONAL

El algoritmo tradicional lleva a cabo las tres operaciones fundamentales de un algoritmo Divide y Vencerás en cada nivel de recursividad:

1. Dividir el problema en subproblemas.
2. Resolver los subproblemas de forma recursiva.
3. Combinar las soluciones de los subproblemas para obtener una solución final.

Veamos a continuación un pseudocódigo del algoritmo:

```
Solve (origin, auxiliar, destination, n)
  If n == 1
    Move from origin to destination
  Else
    Solve (origin, destination, auxiliar, n-1)
    Move from origin to destination
    Solve (auxiliar, origin, destination, n-1)
```

El problema se divide en subproblemas mediante las llamadas recursivas con un número de discos más pequeño. El caso base es aquel en el que el número de discos a mover es 1, donde basta con mover de origen a destino para finalizar la ejecución. En caso de que aún sea mayor, lo resuelve moviendo de origen a auxiliar todos los discos salvo el último, el más grande, que es el que se mueve a destino en la siguiente sentencia. Por último, se mueven los discos restantes que se encuentren en la varilla auxiliar a la varilla destino.

El tiempo de ejecución de este algoritmo, como en cualquier otro algoritmo que siga el paradigma Divide y Vencerás, puede determinarse mediante la fórmula de recurrencia:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c, \\ aT(n/b) + D(n) + C(n) & \text{en otro caso.} \end{cases}$$

En este caso, *c* tiene valor 1, el cual es el caso base de nuestro problema y su complejidad es la más básica, pues solo tiene que realizar un movimiento. Consideraremos que un movimiento tiene coste 1. Para el resto de casos:

- El valor de *a*, que se corresponde con el número de subproblemas que surgen de la división de un problema, es de 2. Para solucionar cada nivel de recursividad, hay que resolver dos subproblemas, que se corresponden con las dos llamadas recursivas.
- El valor de *n / b* es n-1. Se trata del tamaño de cada uno de los subproblemas que surgen. Sabemos que cada subproblema consiste en mover todos los discos a una varilla auxiliar excepto uno, el más grande, que va a moverse al destino. Por tanto, mueve n-1 elementos en cada nivel de recursividad.
- El valor de D(n), el tiempo necesario para dividir un problema en subproblemas, es constante y es 1. Simplemente hace una llamada a una función; no tiene que realizar ningún otro tipo de operación.
- El valor de C(n) es el tiempo necesario para combinar las soluciones de los subproblemas. Podríamos decir que, en el caso de este algoritmo, este valor es 0, ya que cada nivel de recursividad modifica unas mismas "instancias" de los discos y las varillas, sin ser necesario combinar nada.

Finalmente, tenemos que el tiempo de ejecución del algoritmo para cualquier caso menos el caso base es de:

$$T(n) = 2T(n-1) + 1$$

Si tratásemos de aplicar el método maestro para saber la complejidad final del algoritmo, tenemos que el valor de *d* es 1 y el valor de *a* es 2. Sin embargo, el valor de *b* no podemos obtenerlo correctamente, ya que el tamaño de cada subproblema no supone precisamente una división del tamaño del problema original. Por tanto, en lugar de utilizar el método maestro, vamos a razonar un poco.

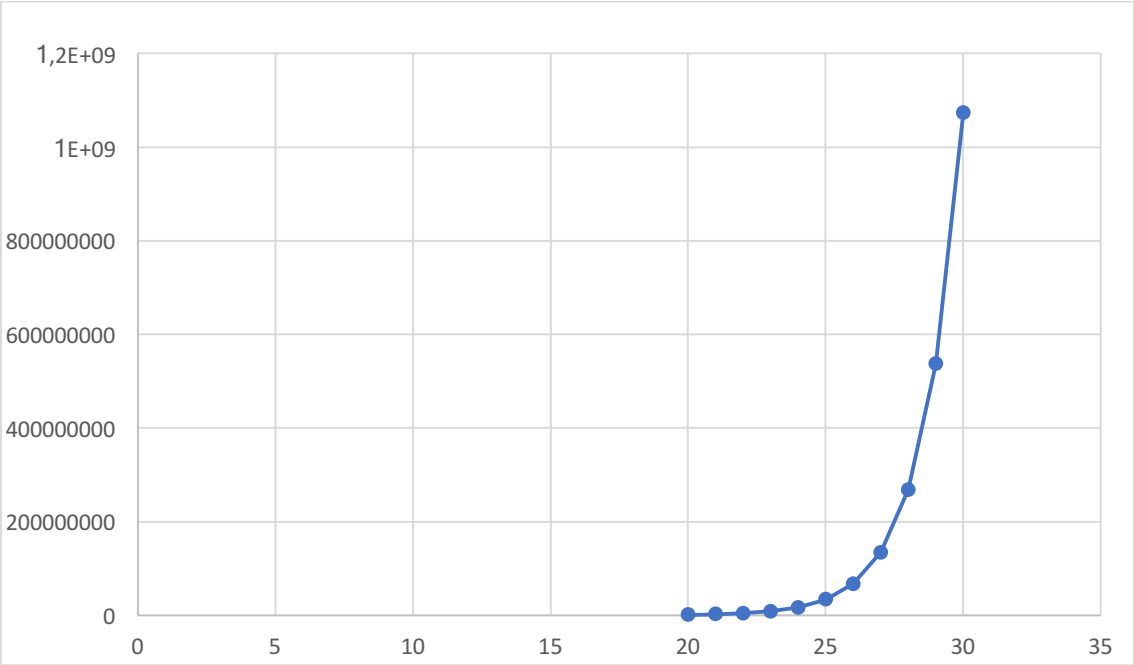
- En un árbol de recurrencia, cada nodo tiene dos subnodos, con un tamaño igual al tamaño del padre menos 1. Por tanto, llegaremos al caso base en el nivel en el que el tamaño del problema sea de n – (n – 1). Sabiendo que el primer nodo tiene tamaño n – 0, tenemos en total n niveles.
- En un nivel i, el tamaño del problema es de n – i. En el último nivel, i tiene un valor igual a n – 1.
- En cada nivel hay el doble de problemas que en el nivel anterior. En el nivel 0, hay 1 nodo; en el nivel 1, dos nodos; en el nivel 2, cuatro nodos… En cada nivel, entonces, hay 2^i subproblemas.
- El tamaño total de cada nivel sería de $2^i * (n – i)$. El sumatorio de esto desde 0 hasta n – 1 es la complejidad del algoritmo. Se trata de dos progresiones, una geométrica y otra aritmética, que al resolverse dan como resultado que efectivamente $T(n) = 2^n – 1$.

La constante 1 que resta es despreciable. Por tanto, la complejidad final es:

$$T(n) \in \Theta(2^n)$$

RENDIMIENTO

Al tratarse de un algoritmo de complejidad exponencial, el incremento del tamaño del problema supone un aumento de la complejidad bastante considerable. Los tamaños con los que podemos llegar a experimentar sin que el programa tarde demasiado tiempo o se cuelgue, son de n = 30 aproximadamente. También comprobaremos que la diferencia entre este tamaño y el tamaño 29 o 31 es bastante grande.



Como podemos ver, el incremento es bastante notable, hasta tal punto que cuando el tamaño del problema cae por debajo de 22 aproximadamente, el tiempo empleado es despreciable con respecto a lo que tarda con tamaños mayores. A partir de 30, el problema no ha podido ser resuelto por la máquina.

Para comprobar que la complejidad de nuestro algoritmo no es mayor de lo que esperábamos, hemos establecido un contador en nuestro programa, que se inicializa a cero y se incrementa con cada movimiento de un disco. Al finalizar, devuelve el valor final del contador, que es recibido por sus niveles superiores de recursividad y lo acumulan.

Finalmente, tenemos que, para el tamaño 30, se ejecutan 1.073.741.823 movimientos. Para 29, se ejecutan 536.870.911; y para 28, 268.435.455. Y así sucesivamente, coincidiendo efectivamente todos los valores con $2^n - 1$, que era la complejidad esperada.

También hemos implementado el algoritmo que resuelve el juego de las Torres de Hanói cíclicas, que resuelve el mismo problema, pero añadiendo la restricción de que los discos no pueden pasar por la misma varilla sin haber pasado antes por las demás. Así, en un

juego con tres varillas A, B y C, siendo A el origen y C el destino, y B una varilla auxiliar, los movimientos de un disco solo pueden ser del tipo A – B – C – A o A – B – C – A.

Este algoritmo aumenta su complejidad a $\Theta(2.733^n)$, lo cual puede parecer una diferencia pequeña pero no lo es en absoluto. De hecho, para problemas de tamaño 22, la máquina no puede resolverlos. Concretamente, a nosotros nos ha devuelto el resultado de que se han ejecutado -1135228929 pasos. Para tamaños de problema menores, el algoritmo resuelve el problema correctamente, pero en un tiempo muchísimo mayor al algoritmo tradicional. Por ejemplo, para un problema de tamaño 21, el algoritmo cíclico tarda más que el algoritmo tradicional para un problema de tamaño 30:

- El tradicional, para n = 30 -----> 1.073.741.823
- El cíclico, para n = 21 ----------> 1.156.544.511

CONCLUSIONES

Como hemos visto, el algoritmo se hace realmente complejo para tamaños de problema que podríamos considerar como “comedidos” (valores entre 20 y 30). Sin embargo, tenemos que tener en cuenta que, matemáticamente, la mejor forma en la que se puede resolver este problema es siempre en $2^n - 1$ pasos. Teniendo esto en cuenta, podemos afirmar que nuestro algoritmo es inmejorable (al margen de mejoras en el lenguaje de programación y otros aspectos que son independientes del algoritmo en sí).

Además, el código es bastante sencillo de entender y requiere de muy pocas líneas. El aprovechamiento del paradigma de programación Divide y Vencerás claramente beneficia su calidad y simpleza.