Complejidad algorítmica

Eficiencia de algoritmos y tipos abstractos de datos

Julià Minguillón i Alfonso

P06/75001/00576
Módulo 2

UOC
www.uoc.edu

Índice

Introducción 5

Objetivos 6

1. Introducción a la complejidad algorítmica 7
1.1. Complejidad temporal y complejidad espacial 7
1.2. Complejidad algorítmica de las estructuras de control básicas 9
1.3. Complejidad de la combinación de algoritmos 10
1.4. Mejor caso, peor caso y caso medio 15

2. Notación asintótica de la complejidad algorítmica 17
2.1. Tipo de notación asintótica 17
2.2. Comparación de complejidades 18
2.3. Propiedades de la notación *O* 19

3. Complejidad algorítmica de los tipos abstractos de datos 21
3.1. Complejidad del TAD de los números naturales 21
3.2. Complejidad del TAD *Conjunto* 22

Resumen 23

Ejercicios de autoevaluación 25

Solucionario 26

Glosario 28

Bibliografía 28

Introducción

En este módulo se definen los conceptos básicos relacionados con la complejidad algorítmica. Esta herramienta se usa para comparar la eficiencia de algoritmos. Como es de esperar, un algoritmo se utiliza para resolver un problema que debe ser eficaz y eficiente. Eficaz en tanto que sea capaz de resolver el problema propuesto, y eficiente en tanto que lo resuelva lo más rápidamente posible. Dado que es posible disponer de diferentes algoritmos que resuelvan un mismo problema (todos ellos son igualmente eficaces), se trata de compararlos entre ellos de manera que se elija el más eficiente, que computacionalmente hablando significa que usa menos recursos para encontrar la solución deseada.

La complejidad algorítmica, por lo tanto, está relacionada con la cantidad de recursos que usa un algoritmo para resolver un problema, una cantidad que está relacionada con los parámetros que determinan la medida del problema. Así, por ejemplo, el parámetro que determina la medida del problema en un algoritmo de ordenación sería el número de elementos que se ordenan, mientras que los recursos necesarios pueden ser de dos tipos: cantidad de memoria o almacenamiento requerido, o bien el número de pasos (o tiempo de ejecución en un entorno controlado) que necesita el algoritmo para alcanzar la solución. Normalmente, conocer el tiempo de ejecución de un algoritmo es más útil (por su importancia) que la cantidad de memoria requerida.

En general, la eficiencia de un algoritmo resulta relevante para un problema de grandes dimensiones, es decir, cuando el número de elementos que manipula el algoritmo es elevado. Para medidas pequeñas, existen muchas variables que pueden determinar el tiempo de ejecución (por ejemplo, la velocidad del procesador, cárga del sistema operativo, etc.) y tener una incidencia significativa, mientras que para medidas grandes, el tiempo de ejecución está básicamente determinado por la propia estructura interna del algoritmo (su complejidad algorítmica). Evidentemente, un ordenador más potente resolverá el mismo problema más rápidamente que un ordenador más sencillo; pero si, por ejemplo, la medida del problema se multiplica por dos, el incremento de tiempo que necesitará un ordenador sencillo será equivalente al incremento de tiempo que necesitará un ordenador más potente. La notación asintótica permite dar una aproximación de la complejidad algorítmica y realizar comparaciones para medidas de problema grandes, independientemente de la configuración del ordenador sobre el que se ejecuta el algoritmo.

Objetivos

Los objeivos que se espera que el estudiante satisfaga con los contenidos de este módulo son los siguientes:

1. Tener una visión general de los conceptos de *complejidad algorítmica* y *eficiencia*.

2. Conocer las diferentes notaciones asintóticas para denotar la complejidad algorítmica.

3. Calcular la complejidad algorítmica de las estructuras de control básicas.

4. Calcular la complejidad algorítmica de los tipos abstractos de datos.

1. Introducción a la complejidad algorítmica

La complejidad algorítmica, informalmente, es una medida que permite a los programadores conocer la cantidad de recursos que necesita un algoritmo para resolver un problema en función de su tamaño. El objetivo es comparar la eficiencia de algoritmos a la hora de resolver un problema conocido.

Del mismo modo, en el caso de tipos abstractos (TAD), se trata de medir la idoneidad de una estructura u otra para la representación de un conjunto de información en función de la eficiencia de las operaciones que se deberán realizar. Es evidente que un programa resolverá un problema (por ejemplo, encontrar la posición de un elemento de la lista) en un cierto tiempo, dependiendo de una serie de factores: el ordenador y el sistema operativo sobre los cuales se ejecuta el programa, el lenguaje de programación utilizado y el compilador, la plataforma de desarrollo, y la destreza y la habilidad del programador; pero, sobre todo, dependerá de la dificultad intrínseca y del tamaño del programa.

Es importante destacar que se mide un coste en unidades (de tiempo, pero también de memoria) que un algoritmo necesita para resolver un problema con una entrada en función de su medida. Aunque los algoritmos se acostumbren a describir con lenguajes de alto nivel, por ejemplo Java, la ejecución se realiza en un entorno controlado, como es una CPU, que ejecuta instrucciones muy básicas como mover información, realizar cálculos aritméticos, saltar entre diferentes puntos del programa, etc. Idealmente, los recursos necesarios deberían valorarse en este nivel, pero entonces se pierde la abstracción que proporciona el lenguaje de alto nivel y eso añade una complicación adicional para comparar algoritmos. En consecuencia, es necesario estudiar la complejidad algorítmica en un nivel más elevado, usando las estructuras de control (secuencial, alternativa y alterativa) como herramientas básicas para la construcción de algoritmos y, por lo tanto, también básicas para medir su eficiencia.

1.1. Complejidad temporal y complejidad espacial

La **complejidad algorítmica** puede medir dos conceptos diferentes, que son complementarios entre sí:

1) Complejidad temporal: mide el número de unidades de tiempo que necesita un algoritmo (o una simple sentencia) para resolver un problema con una entrada de tamaño N, y se denota por $T(N)$.

2) Complejidad espacial: mide el número de unidades de memoria que necesita un algoritmo (o una simple sentencia) para resolver un problema con una entrada de tamaño N, y se denota por $S(N)$.

Cuando se habla de unidades de tiempo o de memoria, no se está haciendo siempre referencia a segundos o bytes, respectivamente, sino que es una cantidad constante conocida que permite hacer comparaciones relativas, usando el 1 como la unidad mínima de coste. Por ejemplo, en el caso de unidades de tiempo, en lugar de utilizar un segundo (o un nanosegundo), se utiliza como unidad mínima de coste temporal el coste que corresponde a hacer la operación más sencilla, por ejemplo una asignación. De esta manera, se minimiza la influencia de la arquitectura del ordenador, el sistema operativo, el compilador, etc., en el cálculo de la complejidad. En el caso de unidades de memoria, sí que se acostumbra a usar el byte como unidad mínima, a pesar de que también se pueden utilizar otras longitudes de acuerdo con la arquitectura de los datos que se usan (por ejemplo palabras de 64 bits). Habitualmente, sin embargo, no se trabaja con el coste en unidades mínimas de cada operación, sino que se hace una simplificación y se supone que todas las operaciones sencillas (implementadas directamente por la CPU o la UAL, es decir, las comparaciones, asignaciones, operaciones aritméticas, etc.) tienen el mismo coste.

Por otro lado, y teniendo en cuenta que cuando se habla de *eficiencia de algoritmos* lo más importante acostumbra a ser la rapidez con que se ejecuta, cuando se habla de *complejidad algorítmica* normalmente nos referimos a la complejidad temporal. Esto es así porque es más fácil incrementar la capacidad de memoria de un ordenador que aumentar la velocidad de proceso y, por lo tanto, es mejor diseñar algoritmos que se ejecuten rápidamente que algoritmos que utilicen menos cantidad de memoria. Eso puede no ser así en otros contextos, en los que la capacidad de memoria es limitada (por ejemplo, en un satélite que envía datos a una estación receptora) y la única opción es usar algoritmos más lentos, pero con unas necesidades de memoria limitadas.

De hecho, las dos complejidades no son independientes, ya que algunos algoritmos pueden mejorar su complejidad temporal a costa de empeorar la espacial y al contrario. Por ejemplo, un algoritmo que necesite realizar muchos cálculos intermedios repetitivos puede reducir el tiempo de cálculo si los almacena para reaprovecharlos. Es decir, si se almacena información redundante aumentado la complejidad espacial, se puede reducir la complejidad temporal, y al contrario.

El concepto de recursividad se desarrolla en el módulo "Árboles".

Reaprovechamiento de cálculos

Esto es evidente en el caso de algoritmos recursivos, como el cálculo de la secuencia de Fibonacci para un conjunto de enteros determinado, en dos niveles: el primero, por el cálculo de cada término $F(n)$, ya que se realiza a partir de los dos anteriores, $F(n-1)$ y $F(n-2)$. Del mismo modo, para calcular $F(n-1)$ se usan $F(n-2)$ y $F(n-3)$, y así sucesivamente. Por lo tanto, si se almacena $F(n-2)$ en algún lugar, se puede reaprovechar y ahorrarse tener que calcularlo de nuevo, y ocurre lo mismo con el resto de términos de las secuencias. En el segundo nivel, por cada elemento del conjunto del que se quiere calcular el número que ocupa la posición en la secuencia de Fibonacci, si los elementos se ordenan de más pequeños a más grandes, será posible reaprovechar todos los cálculos intermedios.

Secuencia de Fibonacci

El término $F(n)$ de la secuencia de Fibonacci se calcula como $F(n) = F(n-1) + F(n-2)$, en el que $F(1) = 1$ y $F(2) = 1$ son los dos primeros términos de la secuencia.

1.2. Complejidad algorítmica de las estructuras de control básicas

Todos los lenguajes procedimentales de alto nivel, como Pascal, C o el mismo Java, comparten las mismas estructuras de control:

- **Secuencia**: las instrucciones se ejecutan una después de la otra.

- **Alternativa**: dependiendo de una condición, se ejecutará un bloque de instrucciones u otro (o bien no se ejecutará ninguno).

- **Repetitiva**: un bloque de instrucciones se repite un cierto número de veces (que puede ser cero o más), dependiendo de una condición.

Por cuestiones de simplicidad, no se diferencian los costes individuales entre cada tipo de operación, ya que se trata de obtener una estimación en función del tamaño del problema, y no tanto del entorno (ordenador, sistema operativo, compilador, etc.) en el que se ejecuta el algoritmo que lo resuelve. Por lo tanto, cada sentencia o instrucción individual, evaluación de una condición o de una expresión aritmética sencilla, etc., se considera que tiene una complejidad temporal $T(N) = 1$. Es decir, que, de hecho, no dependen de N y son constantes. Consecuentemente, el coste estimado depende más del número de operaciones realizadas que realmente de las operaciones en concreto, y sólo dependerá de N cuando el número de operaciones también dependa de aquélla, como repetir un bloque de instrucciones dentro de un bucle que depende de N.

Consideremos el siguiente fragmento de código en Java:

```
int suma = 0;

for (int k = 1; k <= N; k++) {
  if ((k%2)==1) {
    suma+ = k;
  }
}

System.out.println(suma);
```

Observación

Fijaos que la condición del bucle se evalua $N + 1$ veces, ya que la última iteración que devuelve 'falso' y provoca la salida del bucle, también se evalúa.

Las operaciones realizadas son dos asignaciones que se hacen en una sola vez (las variables *suma* y *k*), una comparación (la condición del bucle $k \leq N$) que se realiza $N + 1$ veces, un autoincremento y una evaluación de una condición (que incluye una operación módulo y una comparación) que se efectúan N

ocasiones, una suma que se hace $N/2$ veces, y finalmente una llamada a una función del sistema que se realiza en una ocasión. Por lo tanto:

$$T(N) = 1 + 1 + 1 \cdot (N + 1) + (1 + (1 + 1)) \cdot N + 1 \cdot (N/2) + 1 = 4N + N/2 + 4.$$

Es necesario señalar, sin embargo, que todas las operaciones (asignación, suma, etc.) están contando con el mismo coste (la unidad mínima de coste), mientras que seguramente la operación aritmética de suma tendrá un coste real de ejecución mayor que la comparación. Esta simplificación es necesaria para no complicar excesivamente el análisis de la complejidad de un algoritmo. De hecho, para realizar un análisis exhaustivo de cualquier algoritmo, se debería ir al código máquina generado por el compilador para una plataforma concreta, y conocer el coste de cada operación, lo cual haría que el análisis dejase de ser independiente de la plataforma. Para comparar algoritmos escritos en lenguajes de programación de alto nivel, es preciso efectuar simplificaciones como ésta.

El cálculo de la complejidad puede ser tan sencillo como se ha visto en este ejemplo, pero en la mayoría de casos los programas que se deben analizar serán mucho más complejos, e intentar contar las operaciones realizadas directamente puede ser complicado. Por lo tanto, la complejidad se calcula agrupando las instrucciones en bloques, y utilizando las reglas de cálculo para combinaciones de algoritmos que se describen en el siguiente subapartado.

1.3. Complejidad de la combinación de algoritmos

Habitualmente, los algoritmos se pueden descomponer en varias etapas, cada una de las cuales realiza una tarea diferente para la resolución de un problema. Cada etapa es, de hecho, un subalgoritmo que ejecuta una tarea concreta con su propia complejidad. Por lo tanto, si un algoritmo A se puede descomponer en n subalgoritmos A_1, A_2, ..., A_n, de complejidades conocidas T_1, T_2, ..., T_n, entonces se puede aplicar un conjunto de reglas para calcular la complejidad del algoritmo A.

Observación

Es importante no confundir n con N, ya que son valores diferentes. N hace referencia al tamaño del problema, mientras que n hace referencia al número de subalgoritmos.

- Si dos o más algoritmos se ejecutan secuencialmente

  A_1
  A_2
  ...
  A_n

  la complejidad total será la suma de las complejidades parciales, es decir,

$$T(N) = T_1(N) + T_2(N) + ... + T_n(N)$$

- Si en una estructura alternativa como la siguiente:

  si (*condición*) **entonces** // *condición* se denota por C
  A_1
  sino
  A_2
  fsi

Entre dos subalgoritmos se ejecuta uno u otro, según se cumpla o no una expresión (una condición); la compejidad del algoritmo dependerá de qué rama se ejecute. La rama que se ejecute dependerá de los datos de entrada, pero no de su tamaño. Para diferentes datos se ejecutará una rama u otra. Por lo tanto, en general no podremos calcular el coste temporal exacto del algoritmo para datos de tamaño N. Por lo tanto, lo más apropiado como medida de complejidad sería proporcionar una medida del coste medio para una ejecución del algoritmo con datos de tamaño N, en función de las veces que se ejecuta cada rama del algoritmo en término medio. Así pues, la complejidad total será la suma ponderada de los valores posibles de la función de evaluación para la distribución de probabilidades $\{p, 1 - p\}$ más el coste de evaluar la condición $T_C(N)$,es decir:

$$T(N) = p \cdot T_1(N) + (1 - p) \cdot T_2(N) + T_C(N).$$

Observación

Fijaos en que la evaluación de la condición puede depender del tamaño del problema N, aunque habitualmente será una comparación y, por lo tanto, en general $T_C(N) = 1$.

Si no se conocen las probabilidades de cada valor resultado de evaluar la expresión, no podremos calcular el coste medio de una ejecución del algoritmo para datos de tamaño N. En este caso, una buena medida es considerar que la complejidad total es el máximo de las complejidades parciales, es decir:

$$T(N) = \text{máx}(T_1(N), T_2(N)) + T_C(N).$$

Eso corresponde a lo que se conoce como *análisis del peor caso*, que se desarrollará más adelante, mientras que el caso anterior se corresponde a lo que se conoce como *caso medio*.

Del mismo modo, si la estructura alternativa sólo tiene una rama, se puede considerar que, en el peor de los casos (cuando la condición es cierta), la complejidad total es:

$$T(N) = T_1(N) + T_C(N).$$

- Si en una estructura repetitiva como la siguiente:

  mientras (*condición*) { // *condició*n se denota por C
  A
  }

  un algoritmo A con complejidad $T_A(N)$ se ejecuta dentro de un bucle que depende de exactamente del tamaño del problema N, la complejidad resultante se calcula como:

$$T(N) = N \cdot T_A(N) + (N + 1) \cdot T_C(N).$$

Eso es cierto sólo si el algoritmo *A* mantiene una complejidad constante a lo largo de todas las interacciones del bucle. Si no fuera así, y la complejidad dependiera de la interacción, sería necesario hacer una descomposición de las diferentes ejecuciones del algoritmo *A* y hacer un sumatorio de la complejidad en cada ejecución, es decir:

$$T(N) = \sum_{j=1}^{N} T_A(N,j) + (N+1) \cdot T_C(N)$$

Cuando $T_A(N,j) = T_A(N)$ para toda iteración *j* (es decir, la complejidad se mantiene constante), el sumatorio se reduce a $N \cdot T_A(N)$.

Del mismo modo, si el bucle ejecuta un número log(*N*) de veces, la complejidad total se calcularía como:

$$T(N) = \log(N) \cdot T_A(N) + (\log(N) + 1) \cdot T_C(N)$$

Es decir, un bucle que depende del tamaño del problema en función de *G*(*N*), incrementa la complejidad del problema original en *G*(*N*) veces, más el coste de evaluar la condición *G*(*N*) + 1 veces. Esta regla se aplica directamente en caso de dos o más bucles imbricados de la siguiente forma:

mientras (*condición*$_1$) { // este bucle se ejecuta $G_1(N)$ veces
  mientras (*condición*$_2$) { // y este $G_2(N)$ veces
    A
  }
}

y da una complejidad total (en este caso) de:

$$T(N) = G_1(N) \cdot (G_2(N) \cdot T_A(N) + (G_2(N) + 1) \cdot T_{C_2}(N)) + (G_1(N) + 1) \cdot T_{C_1}(N)$$

Si los costes de evaluar las expresiones son constantes, es decir, $T_{C_1}(N) = T_{C_2}(N) = 1$, y se pueden dejar respecto al coste del algoritmo $T_A(N)$, entonces la expresión anterior se puede simplificar en:

$$T(N) = G_1(N) \cdot G_2(N) \cdot T_A(N)$$

Es decir, la complejidad crece de manera multiplicativa por cada nivel de bucle.

- Si en una sentencia *s* (una instrucción o bien la evaluación de una expresión) se hace una llamada a una función o procedimiento que tiene complejidad $T_s(N)$, se considera que la sentencia es un subalgoritmo con complejidad $T_s(N)$ y se aplica la regla de descomposición de un algoritmo en subalgoritmos.

Ejemplo de aplicación

Pensemos en un bingo con *NJ* jugadores, en el que no se canta *línea*, sino sólo *bingo* (cuando un jugador consigue todos los números de su cartón). Antes de comenzar una partida, se genera un cartón aleatorio para cada jugador, que representamos mediante un conjunto de números enteros, con *MC* que indica el tamaño del cartón, que es la misma para todos los jugadores. De eso se encarga el método *generarCarton*(), del que queremos calcular la complejidad algorítmica usando las propiedades que acabamos de describir, sin hacer un análisis exhaustivo de él. Se puede suponer que las llamadas a los métodos para generar la salida por pantalla tienen una complejidad temporal constante $T(N) = 1$, a pesar de que será necesario tener en cuenta los casos en los que se haga uso de *toString*() para imprimir un conjunto de elementos, por ejemplo. La generación de numeros aleatorios se realiza mediante un método del JDK, cuya documentación dice:

> "An instance of this class is used to generate a stream of pseudorandom numbers. The class uses a 48-bit seed, which is modified using a linear congruential formula."
>
> Donald Knuth, *The art of computer programming* (vol. 2, sección 3.2.1)

A partir de aquí, podemos deducir (si analizamos el algoritmo de congruencia lineal) que las llamadas a este método tiene coste constante. Siempre que se hace una llamada a un método del JDK hay que considerar, sin embargo, que puede tener un coste no constante en función del tamaño de la entrada.

Por lo tanto, partimos del siguiente código:

uoc.ei.ejemplos.modulo2.Bingo

```
...
public class Bingo {
   private Jugador[] jugadores;
   private int numeroDeJugadores;
   private int numeroDeBolas;
   private int tamanoCarton;
   private Conjunto<Integer> bolasCantadas;
   private Random generadorNumerosCarton;

   public Bingo(int numeroDeJugadoresMaximo,int numeroDeBolas,int tamanoCarton) {
     jugadores = new Jugador[numeroDeJugadoresMaximo];
     numeroDeJugadores = 0;
     this.numeroDeBolas = numeroDeBolas;
     this.tamanoCarton = tamanoCarton;
     generadorNumerosCarton = new Random();
   }

   public void prepararPartida() {
     System.out.println("Preparando partida...");
```

```
    for (int i = 0;i<numeroDeJugadores;i++) {
      Conjunto<Integer> carton = generarCarton();
      jugadores[i].setCarton(carton);
    }
    bolasCantadas=new ConjuntoVectorImpl<Integer>(numeroDeBolas);
  }

  protected Conjunto<Integer> generarCarton() {
    Conjunto<Integer> carton=
      new ConjuntoVectorImpl<Integer>(tamanoCarton);
    for (int n  =0;n<tamanoCarton;n++)
      carton.añadir(generadorNumerosCarton.nextInt(numeroDeBolas) + 1);
    return carton;
  }
  ...
}
```

uoc.ei.ejemplos.modulo2.Jugador

```
...
public class Jugador {
  private String nombre;
  private Conjunto<Integer> carton;

  public Jugador(String nombre) {
    this.nombre = nombre;
  }

  public void setCarton(Conjunto<Integer> carton) {
    this.carton = carton;
    System.out.println(this.toString());
  }
  ...
}
```

La llamada *generarCarton*() se puede descomponer en otra llamada al constructor *ConjuntoVectorImpl*(), que podemos suponer que tiene coste $T(N) = 1$, un bucle que se ejecuta *MC* veces, y la sentencia *return*, también con complejidad constante $T(N) = 1$. Dentro del bucle se hace una llamada al método *insertar*(); pero hay que tener en cuenta que como parámetro se ejecuta *generadorNumerosCarton.nextInt*() que –como hemos dicho antes– sólo hace una llamada al método del JDK para generar números aleatorios, que podemos suponer que tiene coste $T(N) = 1$ y una suma adicional. Así, el coste de la sentencia que se repite *MC* veces sólo depende realmente del coste del método *insertar*() implementado dentro de *ConjuntoVectorImpl*.

Al mismo tiempo, el método *insertar*() hace una llamada al método *esta*(), y en el caso de que se cumpla la condición, se ejecutan dos sentencias. Si se aplica la propiedad de la estructura alternativa, el coste del método *insertar*() es, pues, el coste del método *esta*() más dos. Por lo que respecta al método *esta*(), se hace una llamada a *buscarPosicionElemento*(), una comparación y la sentencia *return*.

El método *buscarPosicionElemento*() –que realmente realiza la tarea de buscar un elemento– hace un total de dos asignaciones; un bucle que se repite como mucho *MC* veces, dentro del cual se ejecutan una asignación; una llamada al método *equals*() que equivale a hacer una comparación entre enteros (el tipo base del constructor genérico); y una sentencia *if* que, en caso de ser cierta, hace ejecutar una operación de incremento (que hace un total de cuatro operaciones, como método *buscarPosicionElemento* acaba con la evaluación del operador ternario que supondremos que son dos operaciones básicas (evaluación de la condición y la asignación), y la sentencia *return*. Por lo tanto, el coste de este método es, en el peor caso, $T(MC) = 4 \cdot MC + 5$.

Recuperamos ahora los resultados intermedios que habían quedado pendientes de calcular en cada llamada a un nuevo método. El método *esta*() tiene una complejidad de $T(MC) = 4 \cdot MC + 7$ y, por lo tanto, el método *insertar*() tiene $T(MC) = 4 \cdot MC + 9$. Finalmente, el método *generarCarton*() tiene una complejidad temporal de:

$$T(MC) = MC \cdot (4 \cdot MC + 9 + 2) + 2 = 4 \cdot MC^2 + 11 \cdot MC + 2$$

De manera intuitiva, si nos quedamos sólo con el término de mayor crecimiento, $4 \cdot MC^2$ –que es el que determina realmente los valores de $T(MC)$ para valores grandes de *MC*– podemos ver que al analizar el algoritmo encontraremos dos bucles imbricados (a pesar de que no directamente, sino mediante llamadas a métodos), y observaremos que ambos se ejecutan *MC* veces cada uno. Como veremos, éste es el proceso para obtener medidas sencillas para conocer la complejidad de un algoritmo.

Es interesante observar que la complejidad temporal de este método no depende del número de jugadores, sino sólo de la medida del cartón, un hecho por otro lado coherente con su funcionalidad. Este ejemplo permite ver que el estudio de la complejidad temporal puede llegar a ser muy complicado, incluso con las simplificaciones pertinentes relativas a los bucles y a las llamadas a otros métodos. Además, es necesario hacer suposiciones sobre el comportamiento de los algoritmos en función de la entrada, como es el caso de las sentencias *if* o *while*, donde la condición puede ser cierta o falsa un número desconocido de veces, tal y como se discute a continuación.

1.4. Mejor caso, peor caso y caso medio

En general, se puede considerar que todos los algoritmos tienen un comportamiento determinista, de manera que, por la misma entrada, generan la misma

salida efectuando exactamente las mismas operaciones. Dependiendo de la entrada, los resultados esperados se obtendrán más o menos rápido. Por ejemplo, un algoritmo de búsqueda de un elemento en un vector de N elementos acabará en un solo paso si el elemento buscado se encuentra en la primera posición explorada, o bien necesitará hasta N pasos si el elemento buscado se encuentra en la útima posición explorada. Y en media, si todas las entradas posibles son equiprobables, el algoritmo necesitará $(N + 1)/2$ pasos para encontrar el elemento en cuestión.

Por lo tanto, el mismo algoritmo presenta un comportamiento diferente en función de la entrada concreta, y no únicamente de su tamaño, hecho que complica mucho el análisis a causa de la gran cantidad de entradas posibles. Se habla, entonces, del comportamiento del algoritmo en el mejor caso, en el peor caso, y en el caso medio.

El **comportamiento en el mejor caso** se produce cuando el algoritmo que ha de resolver un problema se encuentra una entrada por la que se llega al resultado esperado, y eso sin ninguna operación o quizá con sólo una comprobación. Por ejemplo, ordenar un vector de N elementos que ya ha sido ordenado previamente. Este caso se considera trivial y normalmente no se usa para comparar la eficiencia de los algoritmos, ya que no aporta información valiosa. Además, la mayoría de algoritmos tienen la misma complejidad en el mejor caso, normalmente $T(N) = 1$. Por este motivo, esta información no sirve para decidir qué algoritmo es el más eficiente.

El **comportamiento en el peor caso** se produce cuando el algoritmo encuentra una entrada que obliga a recorrer todos los datos de entrada. Por ejemplo, buscar un elemento en un vector en el que no se encuentra; eso no se descubre hasta que no se han inspeccionado todos los elementos. A pesar de que también puede parecer un caso trivial, en este caso sí que es importante, porque pueden existir algoritmos parecidos que tengan una complejidad diferente en el peor caso y, por lo tanto, puede ser un motivo para decantarse por un algoritmo o por otro.

Finalmente, el **caso medio** estudia el comportamiento del algoritmo para cualquier entrada posible, y asume que todas las entradas son equiprobables, o bien que siguen una distribución conocida. Es importante destacar que el cálculo en el caso medio puede ser muy complicado, o incluso imposible si la distribución de las entradas posibles es desconocida. Eso hace que a veces se usen simulaciones por ordenador, como el método de Montecarlo, para medir la complejidad en el caso medio. Y eso también conlleva que la complejidad en el peor caso adquiera una especial relevancia, ya que si no es posible comparar por el caso medio, el peor caso puede ser un límite superior válido para realizar comparaciones entre algoritmos.

Web recomendada

Podéis encontrar información del método de Montecarlo en la Wikipedia:

http://en.wikipedia.org/wiki/Monte_Carlo_method

2. Notación asintótica de la complejidad algorítmica

El uso del número de operaciones básicas ejecutadas por un algoritmo, denotado por $T(N)$, puede ser complicado para hacer comparaciones entre algoritmos. Para valores pequeños de N, las constantes que acompañan los términos de $T(N)$ pueden influir muy significativamente en el coste total que se está evaluando, y pueden llevar a conclusiones erróneas respecto a la eficiencia del algoritmo. En cambio, lo que interesa es conocer el comportamiento del algoritmo para valores de N grandes, una situación en la que la elección correcta de un algoritmo puede permitir reducir la complejidad algorítmica necesaria para resolver un problema de manera significativa. Un ejemplo claro son los algoritmos de ordenación, en los que para valores pequeños de N, los algoritmos simples –como el de inserción, el de selección o el de la burbuja– pueden ser más eficientes que otros más complejos –por ejemplo, el *quicksort*, el *shellsort* o el *heapsort*. En cambio, para valores grandes de N, es bien sabido que estos últimos algoritmos son mucho más eficientes a pesar de su complejidad intrínseca.

El algoritmo *heapsort* se explica con detalle en el módulo "Colas con prioridad" de esta asignatura.

Por lo tanto, es necesario disponer de una notación que permita comparar algoritmos directamente, sin haber de preocuparse por los casos particulares que aparecen cuando se tienen en cuenta las constantes de las diferentes funciones $T(N)$. Por eso, se usa lo que se conoce como *notación asintótica*, que permite hacerse una idea de la complejidad de un algoritmo cuando el tamaño de la entrada de un problema se hace muy grande.

2.1. Tipos de notación asintótica

Dependiendo del tipo de análisis que se realice, podemos encontrar hasta cinco notaciones asintóticas diferentes. Todas ellas están relacionadas, y algunas son más útiles que otras según el uso que se les quiera dar. La notación más usada es la llamada O grande, y se denota por O.

Las cinco notaciones son o, O, $\sim$, Ω y Θ.

> Se dice que un algoritmo F tiene una complejidad $O(G(N))$ si existen dos constantes C y N_0 para las que se cumpla $|F(N)| < C \cdot G(N)$ para todo $N > N_0$.

Bibliografía recomendada

Podéis encontrar más información de las otras notaciones en la obra siguiente:

Herbert S. Wilf. *Algorithms and complexity*. Disponible en línea en: <http://www.math.upenn.edu/~wilf/AlgoComp.pdf>

Es decir, informalmente, un algoritmo F tiene complejidad $O(G)$ si el número de operaciones necesarias queda fijado por el comportamiento de G para valores grandes de N. Normalmente, las funciones G (órdenes de complejidad) son sencillas, sin constantes. En realidad, $O(G)$ corresponde a un conjunto de funciones, y con la notación O se consigue poner en un mismo conjunto todas las funciones con costes "comparables", es decir, equivalentes con indepen-

dencia de factores externos (hardware, sistema operativo, lenguaje de programación, etc.). Respecto a la constante C, su presencia en la definición de la notación O hace que, en una función G con complejidad $O(G)$, todas las funciones de la forma $C \cdot G$ (en las que C es una constante) pertenecen también a $O(G)$. Por lo tanto, se habla de complejidad $O(N)$, por ejemplo, pero no de complejidad $O(2 \cdot N)$, ya que asintóticamente son equivalentes según la definición. Así, por ejemplo, N pertenece a $O(N)$ y $2 \cdot N$ también pertenece a $O(N)$.

La tabla 1 muestra las complejidades algorítmicas más importantes.

Tabla 1

Notación	Nombre	Ejemplo de algoritmo
$O(1)$	Constante	Acceso a un elemento de un vector
$O(\log N)$	Logarítmica	Búsqueda binaria
$O(N)$	Lineal	Búsqueda secuencial
$O(N \log N)$	Lineal-logarítmica	Algoritmo de ordenación *quicksort*
$O(N^2)$	Cuadrática	Algoritmos de ordenación simples
$O(N^3)$	Cúbica	Multiplicación de matriz
$O(2^N)$	Exponencial	Partición de conjuntos

Es importante destacar que la notación O nos permite establecer un hito superior del comportamiento de un algoritmo F. Es evidente que se pueden encontrar infinitas funciones G de manera que se cumpla que F es de orden $O(G)$. Por ejemplo, si un algoritmo tiene una complejidad temporal $T(N) = 2N^2 + N$, se puede decir que $T(N)$ es de orden $O(N^2)$, pero también $O(N^3)$. Sin embargo, el hecho de decir que $T(N)$ es de orden $O(N^2)$ aporta mucha más información que $O(N^3)$. Las complejidades $T(N)$ que son combinaciones de potencias de N, como la cuadrática o la cúbica, se denominan *polinómicas*, y son el umbral de lo que hoy se considera computacionalmente tratable.

2.2. Comparación de complejidades

Es posible hacerse una idea del coste real que representa tener o no disponible un algoritmo con una complejidad limitada a medida que la dimensión del problema va creciendo. La tabla 2 muestra el tiempo proporcional (por ejemplo, en segundos) que necesita un algoritmo según la medida de la entrada y de su complejidad.

Tabla 2

N vs $O(N)$	$\log_2 N$	N	$N \log_2 N$	N^2	N^3	2^N
100	6,64	100	664	10^4	10^6	10^{30}
1.000	9,97	1.000	9.970	10^6	10^9	10^{301}
10.000	13,29	10.000	13.290	10^8	10^{12}	$10^{3.010}$
100.000	16,61	100.000	166.100	10^{10}	10^{15}	$10^{30.103}$
1.000.000	19,93	1.000.000	19.930.000	10^{12}	10^{18}	$10^{301.030}$

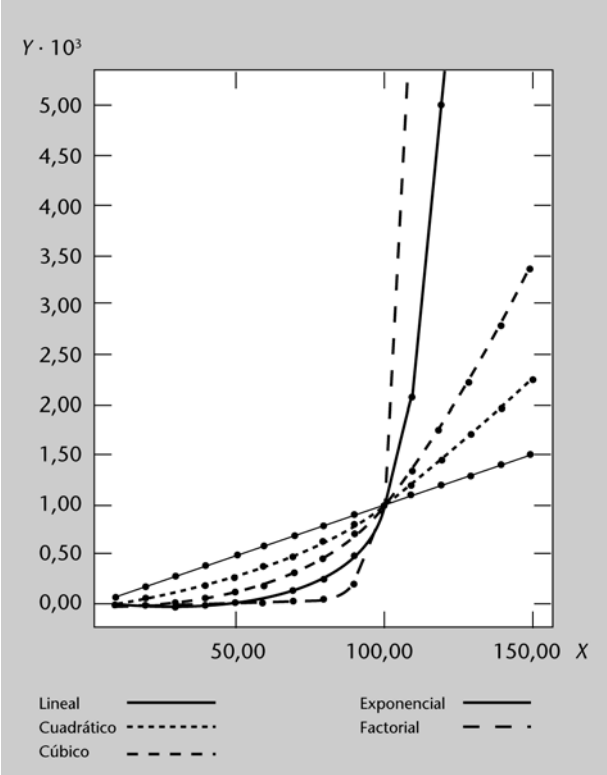
Multiplicar la medida del problema por un orden de magnitud tiene un incremento muy diferente dependiendo de la complejidad del algoritmo. Con la potencia de los superordenadores actuales –que pueden ejecutar casi 10^{14} operaciones en un segundo, incluso con una medida de sólo $N = 100$–, un algoritmo con complejidad exponencial no se puede resolver en un tiempo razonable, ya que se necesitarían 10^{16} segundos o, lo que es lo mismo, 317 millones de años.

El BlueGene/L

El superordenador BlueGene/L de IBM puede efectuar un máximo de 91,75 teraflops (el prefijo *tera*– equivale a un billón de veces, es decir, 10^{12} operaciones en coma flotante por segundo).

En general, los algoritmos con complejidad exponencial son computacionalmente intratables según los estándares actuales, y no se pueden tratar directamente a partir de mecanismos basados en la fuerza bruta que prueban todas las combinaciones posibles para llegar a la solución deseada, sino que es necesario utilizar algoritmos que, a partir de una solución aproximada, obtienen una solución más buena (o incluso óptima) a partir de técnicas aproximadas o heurísticas.

Figura 1



2.3. Propiedades de la notación *O*

La primera propiedad, ya comentada, es que el orden de cualquier función de la forma $O(k \cdot g)$ en la que k es una constante es $O(g)$; es decir, sea cual sea la función de crecimiento según el parámetro de entrada, su complejidad crece proporcionalmente a g.

Usando la definición de la notación O, vemos también que $O(g_1 + g_2)$ es $O(\text{máx}(g_1,g_2))$. Es decir, si un algoritmo G tiene dos o más subalgoritmos que se ejecutan secuencialmente sobre el mismo conjunto de entrada, la comple-

jidad (temporal o espacial) de G será la misma que la del subalgoritmo que presente la complejidad (o función g) más elevada. En otras palabras, si un algoritmo realiza dos tareas, la tarea que tenga un comportamiento más costoso es la que determinará la complejidad total del algoritmo. El mismo criterio se puede aplicar al caso de la estructura de control alternativa, tal y como se ha comentado.

Eso significa que, en general, si un algoritmo se puede descomponer en varias partes, y una de ellas tiene una complejidad algorítmica superior al resto, la complejidad del algoritmo quedará determinada por esta parte. Por ejemplo, si un algoritmo de búsqueda intenta ordenar los datos de entrada con un algoritmo $O(N^2)$ o $O(N \log N)$ para aplicarle después una búsqueda dicotómica –que es más eficiente que la búsqueda secuencial ($O(\log N)$ de la búsqueda dicotómica delante de $O(N)$ de la búsqueda secuencial)–, el resultado final es peor que si se hiciese la búsqueda secuencial directamente, ya que la complejidad del algoritmo de ordenación es superior.

Del mismo modo –y hablando de la complejidad algorítmica de los tipos abstractos de datos–, la complejidad de un TAD dependerá de las operaciones disponibles, su implementación y la complejidad resultante; por lo cual, un TAD que sea eficiente para unas operaciones puede no serlo para las otras. Por lo tanto, dependiendo del tipo de operaciones que sea necesario ejecutar más a menudo, habrá que escoger los algoritmos y las implementaciones de cada una de las operaciones en función de las necesidades. Así, en el ejemplo anterior, si sobre un conjunto grande de datos se realizan muchas búsquedas, sí que puede ser interesante ordenarlo una vez, aunque sea costoso, y reducir el coste global de las operaciones de búsqueda.

3. Complejidad algorítmica de los tipos abstractos de datos

En el caso concreto de los tipos abstractos de datos, cuando se habla de complejidad algorítmica es interesante relacionar los conceptos de *implementación*, *eficiencia* y *complejidad*. El objetivo es ser capaces de elegir el tipo de contenedor más adecuado para una cierta colección de objetos en función de ciertas restricciones temporales y espaciales.

La idea básica es que cada TAD tiene una complejidad conocida en función de su implementación. De hecho, se indica la complejidad de cada una de las operaciones que ofrece el TAD. Así, por ejemplo, en la implementación de los números naturales usando el tipo *int* de Java que ya conocemos, las operaciones de predecesor y sucesor de un número natural son ambas de orden $O(1)$; es decir, que se hacen en tiempo constante independientemente del número natural que se manipula. Eso puede parecer muy eficiente (y, de hecho, lo es), pero es necesario tener en cuenta que la representación de los números naturales utilizando el tipo entero de Java limita el rango posible de números representados, lo que puede no ser válido para resolver algún problema en el que haga falta manipular números naturales muy grandes. Por lo tanto, es posible que la representación interna más eficiente no sea siempre válida o esté limitada a un cierto tamaño de problema.

Podéis ver el tipo *int* de Java presentado en el módulo "Tipos abstractos de datos" de esta asignatura.

3.1. Complejidad del TAD de los números naturales

Para demostrar este hecho, podemos utilizar el ejemplo del TAD que implementa los números naturales, donde se pueden encontrar dos implementaciones diferentes: una primera que hace servir el tipo *int* de Java –en el que el número natural se almacena directamente por su valor–, y una segunda implementación que usa un vector de elementos booleanos –en los que cada número natural N se representa por un conjunto de elementos en los que los N primeros elementos son 'cierto' y el resto son 'falso'. A pesar de que este elemento está lejos de ser realista, permite una primera comparación de las complejidades de las dos implementaciones.

En principio, la implementación utilizando un *int* resulta ideal: la complejidad espacial es $O(1)$, y la complejidad temporal de todos los métodos es también $O(1)$; ya que se ejecutan en un tiempo constante, independientemente del tamaño (la magnitud, en este caso) del número natural representado. Podríamos decir que la eficiencia de esta implementación es perfecta.

El TAD de los números naturales en Java

El lenguaje Java proporciona una implementación de los números enteros muy eficiente en tiempo constante, y la implementación del TAD se aprovecha de ello.

En cambio, la representación basada en un vector de booleanos presenta una eficiencia menor: la complejidad espacial es $O(N)$ –ya que el espacio necesario

crece linealmente con la magnitud del número natural representado– y, por lo que respecta a la complejidad temporal, los métodos *pred*() y *succ*() tienen una complejidad *O*(*N*), –ya que es necesario recorrer todos los elementos del vector hasta encontrar la posición de lo que se representa. Lo mismo sucede en el resto de métodos: la causa de esta complejidad temporal es la necesidad de ejecutar un bucle dentro del método *buscarUltimaPosicion*() y *duplicarVector*().

3.2. Complejidad del TAD *Conjunto*

En este caso, un conjunto de *N* elementos se representa mediante un vector de al menos *N* posiciones de tipo *Objeto*; por lo tanto, su complejidad espacial será *O*(*N*). Por lo que respecta a la complejidad temporal, la tabla 3 muestra el resultado a partir de la implementación de cada una de las operaciones disponibles.

Tabla 3

Operación	Complejidad	Razonamiento
constructor	*O*(1)	La reserva de memoria por parte del sistema operativo y la máquina virtual es una operación que no depende de la cantidad de memoria solicitada.
insertar()	*O*(*N*)	El método *buscarPosicionElemento*(), al que denominan los tres métodos, debe recorrer secuencialmente todo el vector de *N* elementos.
esta()	*O*(*N*)	
borrar()	*O*(*N*)	

Resumen

La complejidad algorítmica permite medir la eficiencia de un algoritmo para resolver un problema de tamaño conocido. Esta medida permite hacer comparaciones entre algoritmos diferentes con el objetivo de elegir lo más eficiente para resolver el problema. Del mismo modo, es posible elegir el tipo abstracto de datos más adecuado para cada problema, dependiendo de las restricciones temporales o espaciales impuestas. La eficiencia se puede medir respecto a la cantidad de operaciones necesarias para resolver el problema –en este caso se habla de *complejidad temporal*– o bien respecto a la cantidad de memoria necesaria –en este caso, hablaríamos de *complejidad espacial*. Como es más fácil añadir recursos de memoria que incrementar la velocidad de un ordenador, habitualmente, el concepto de *eficiencia* se refiere a la complejidad temporal, es decir, al número de operaciones necesarias para resolver un problema.

Por otro lado, aunque es posible medir la complejidad algorítmica en función de las operaciones que realiza un algoritmo, es mejor utilizar alguna notación que permita efectuar comparaciones sencillas para valores de N grandes, del tamaño del problema por resolver. Una de estas notaciones se conoce como *notación asintótica O*, que permite hacer comparaciones entre la eficiencia de los algoritmos. La idea de la notación O es tener un límite superior del orden de crecimiento de la complejidad en función del tamaño del problema. Así, un algoritmo que resuelva un problema con complejidad cuadrática $O(N^2)$ es mejor que un algoritmo que lo haga con complejidad cúbica $O(N^3)$, por ejemplo.

En el caso particular de los tipos abstractos de datos, es importante conocer la implementación, que es la que realmente determina la complejidad algorítmica de las operaciones que se realizan internamente. El tipo de almacenamiento utilizado para gestionar los datos y las operaciones disponibles unidas a este almacenamiento condicionan la complejidad de cada operación determinada. Así, por ejemplo, el hecho de que unos elementos se almacenen ordenados permite conseguir un tipo de almacenamiento más eficiente que reduce la complejidad de $O(N)$ a $O(\log N)$, mientras que la operación de inserción seguramente tendrá una complejidad más elevada a causa de la necesidad de mantener estos elementos ordenados.

En el resto de módulos de este material, en los que se describen los diferentes tipos abstractos de datos y sus distintas implementaciones, se detallará el cálculo de su complejidad espacial y temporal a partir de la representación interna y de las operaciones disponibles.

Ejercicios de autoevaluación

1. Calculad la complejidad temporal a partir del código Java del siguiente programa, que calcula el producto de dos matrices de $M \times N$ elementos.

package uoc.ei.ejemplos.modulo2.Matriz

```
...
public class Matriz {
    private double[][] elementos;
    private int m,n;

    public Matriz (int m, int n) {
 this.m = m;
 this.n = n;
 elementos = new double[m][n];
 }

 public void set(int i, int j, double s) {
 elementos[i][j] = s;
 }

 public double get(int i, int j) {
 return elementos[i][j];
 }

 public Matriz multiplicarPor(Matriz B) {
    Matriz X = new Matriz(m,B.n);
    for (int i = 0; i < m; i++) {
      for (int j = 0; j < B.n; j++) {
        double s = 0;
        for (int k = 0; k < n; k++) {
          s += get(i,k)*B.get(k,j);
        }
        X.set(i,j,s);
      }
    }
    return X;
 }
```

2. Calculad la complejidad temporal a partir del código del método denominado *prepararSalida*() en el ejemplo del juego del bingo desarrollado en el subapartado 1.3 de este módulo.

Solucionario

1. Comenzaremos calculando la complejidad temporal $T(N)$ de cada método. En este caso, sin embargo, hablaremos de $T(M,N)$ o de $T(M,N,P)$, ya que los parámetros de entrada (el tamaño de una matriz que determina el coste del algoritmo) son dos (y en el caso de multiplicar dos matrices, tres, al tener que coincidir el número de columnas de la primera matriz con el número de filas de la segunda), es decir, multipliquemos una matriz de $M \times N$ elementos por una de $N \times P$.

Método *Matriz* (constructor)

Se hacen dos asignaciones y una llamada al operador *new* para generar la matriz bidimensional. Si suponemos que la reserva de memoria necesaria para la matriz es independiente de su tamaño, tenemos que $T(M,N) = 3$ y, por lo tanto, si aplicamos la definición de complejidad algorítmica y las propiedades que se derivan de ello, $O(M,N) = 1$, es decir, se ejecuta en tiempo constante independientemente del tamaño de la matriz. De hecho, no hace falta aplicar la definición: se busca el término que acompaña cualquier función de M o de N y se elige el que tiene el crecimiento asintótico más grande. En este ejemplo, en el que no aparece ninguno de los parámetros, la complejidad es constante.

Podéis ver la definición y las propiedades de la complejidad algorítmica en el subapartado 2.3 de este módulo

Métodos *set* y *get*

Ambos métodos hacen un sólo acceso a un elemento de la matriz, por lo que $T(M,N) = 1$ y $O(M,N) = 1$; ya que en Java se usa acceso directo para acceder a un elemento de un *array*. Es importante conocer, sin embargo, cómo se almacenan los *arrays* en Java, dado que si hubiese una estructura especial (por ejemplo, para tratar eficientemente matrices muy grandes con muchos elementos vacíos, usando un TAD diseñado para este objetivo), el hecho de que $T(M,N)$ sea constante podría no ser cierto.

Método *multiplicarPor*

En este caso, hay una llamada al constructor de *Matriz*, dos bucles imbricados que se ejecutan M (el número de filas de primera matriz) y P (el número de columnas de la segunda matriz) veces respectivamente. Dentro de estos bucles existe una asignación, una llamada al método *set* y un tercer bucle que se ejecuta N veces (el número de columnas de la primera matriz, idéntico al número de filas de la segunda). Dentro de este bucle más interior se hacen dos llamadas al método *get*, una multiplicación y una operación combinada de sumar y asignar, que podemos suponer que sólo es una. Es necesario recordar que cada bucle incluye una asignación, una comparación (que se ejecuta una vez más) y un incremento. Por lo tanto, se va del bucle más interno hacia fuera:

a) El bucle siguiente tiene una complejidad $T(M,N,P) = 4$, o también $O(1)$:

```
s += get(i,k)*B.get(k,j);
```

b) Este otro bucle tiene la siguiente complejidad:

$T(M,N,P) = 1 + 1 + (N + 1) + N \cdot 4 + N + 1 = 6 \cdot N + 4$, o también: $O(N)$

```
double s = 0;
for (int k = 0; k < n; k++) {// aquí n es N
  s += get(i,k)*B.get(k,j);
}
X.set(i,j,s);
```

c) El bucle siguiente tiene la complejidad indicada a continuación:

$T(M,N,P) = 1 + (P + 1) + P \cdot (6 \cdot N + 4) + P = 6 \cdot P \cdot N + 6 \cdot P + 2$

```
for (int j = 0; j < B.n; j++) { // aquí B.n es P
 ...
}
```

d) Finalmente, el bucle siguiente tiene una complejidad:

$$T(M,N,P) = 1 + (M + 1) + M \cdot (6 \cdot P \cdot N + 6 \cdot P + 2) + M = \\ = 6 \cdot M \cdot N \cdot P + 6 \cdot M \cdot P + 4 \cdot M + 2$$

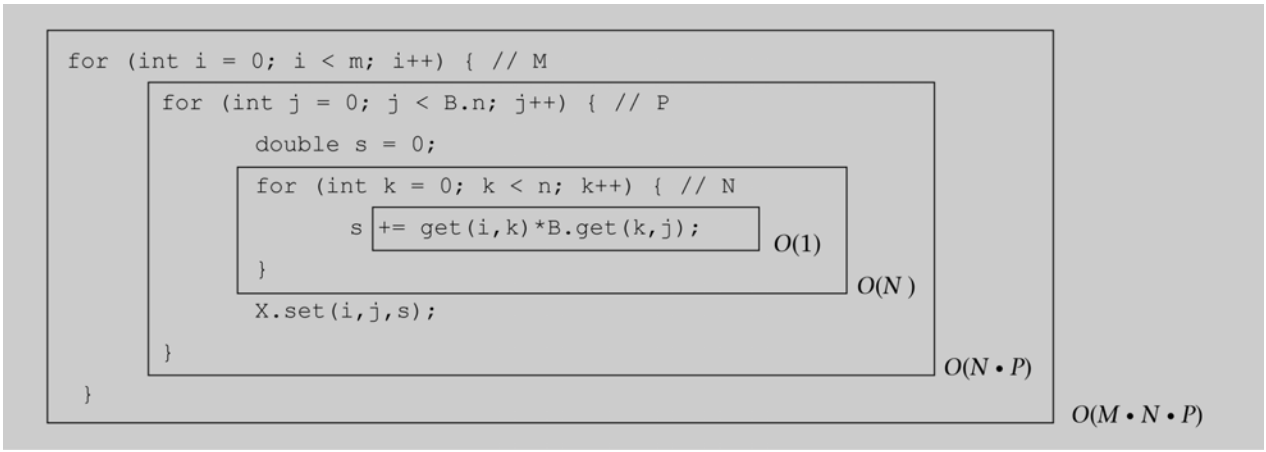
```
for (int i = 0; i < m; i++) { // aquí m es M
 ...
}
```

Al añadir el constructor y la sentencia *return* final, obtenemos:

$$T(M,N,P) = 6 \cdot M \cdot N \cdot P + 6 \cdot M \cdot P + 4 \cdot M + 6$$

Por lo que respecta a su complejidad algorítmica, eliminando las constantes irrelevantes queda $O(M \cdot N \cdot P)$, que es el término que presenta un mayor crecimiento. En general, calcular la complejidad de manera sistemática es una tarea complicada y tediosa, mientras que aplicando simplificaciones es posible llegar al mismo resultado de manera mucho más inmediata. En este ejemplo, el hecho de tener tres bucles imbricados que envuelven una sentencia con complejidad $O(1)$, hace que aplicando la propiedad multiplicativa de los bucles, se pueda llegar directamente al mismo resultado, tal como muestra la figura 2.

Figura 2



Por lo tanto, suponiendo que las matrices son de medidas similares (no son vectores, por ejemplo) –es decir, de $N \times N$ elementos–, el caso $T(N,N,N)$ se puede simplificar en $T(N) = 6 \cdot N^3 + 6 \cdot N^2 + 4 \cdot N + 6$. Aplicando la definición de complejidad algorítmica, podemos decir que este algoritmo para multiplicar dos matrices tiene una complejidad $O(M \cdot N \cdot P)$ o, en general, $O(N^3)$, es decir, cúbica. Siempre que un algoritme dependa de una entrada determinada por dos o más dimensiones, es interesante hacer el ejercicio de planterarse qué sucede cuando las dimensiones son comparables o, al contrario, cuando se pueden considerar constantes. En este ejemplo, si $M = 1$ y $P = 1$ –las matrices son vectores unidimensionales–, el resultado es $T(1,N,1) = 6 \cdot N + 12$, o lo que es lo mismo, $O(N)$; es decir, multiplicar vectores tiene una complejidad lineal respecto a la longitud de los vectores.

2. Queremos analizar el método *prepararSalida*() en función de las medidas que intervienen en el problema: número de jugadores (lo denotaremos por *NJ*), tamaño del cartón (denotado por *MC*) y número de bolas (denotado por *NB*); por lo tanto hablaremos de $T(NJ,MC,NB)$. Si observamos el código, veremos que estos parámetros determinan el coste de las operaciones (estructuras iterativas, llamadas a métodos, etc.). Así pues, podemos proceder, primero, aplicando la primera propiedad (descomposición):
1) La sentencia con la llamada a *println*() tiene un coste $T(NJ,MC,NB) = 1$.
2) Hay un bucle que se ejecuta *NJ* veces. Por lo tanto, si aplicamos la propiedad multiplicativa de los bucles y las simplificaciones pertinentes, la complejidad temporal será $NJ \cdot T_A(NJ,MC,NB)$, en la que *A* es el algoritmo formado por las llamadas a *generarCarton*(), que tiene una complejidad $T(MC) = 4 \cdot MC^2 + 11 \cdot MC + 2$ y *setCarton*(), que desarrollaremos con posterioridad.
3) Finalmente, se hace una llamada al constructor *ConjuntoVectorImpl*(), el cual tiene un coste constante $T(N) = 1$ en hacer sólo una llamada al constructor *Objeto*() genérico. Observad que este constructor sólo depende de un parámetro genérico, y no de tres como el método *prepararSalida*().

El método *setCarton*() hace una asignación y una llamada al método *println*(), pero es necesario tener en cuenta que también hace una llamada a *toString*() y, en este caso, el hecho de que se trate de un conjunto de enteros hace que no se suponga que tiene un coste constante. En la implementación de *ConjuntoVectorImpl*, se puede ver que el método *toString*() hace básicamente un recorrido por todos los elementos; así pues, simplificando, podemos suponer que tiene un coste $T(MC) = MC$. Por tanto, el método *setCarton*() tiene una complejidad $T(MC) = MC + 2$. Por lo tanto, el coste de *generarSalida*() es:

$$T(NJ, MC, NB) = NJ \cdot (4 \cdot MC^2 + 11 \cdot MC + 2 + MC + 2) + 2 = \\ = 4 \cdot NJ \cdot MC^2 + 12 \cdot NJ \cdot MC + 4 \cdot NJ + 2.$$

Si prescindimos de las constantes, y elegimos sólo los términos de más crecimiento, podemos decir que este algoritmo tiene una complejidad $O(NJ \cdot MC^2)$. En este caso, como *NJ* y *MC* son magnitudes completamente diferentes, podemos estudiar qué pasa cuando una de las dos es mucho más grande que la otra. En el caso habitual de que los cartones tengan un tamaño nivelado, el parámetro que puede crecer mucho es *NJ*, el número de jugadores. Por lo tanto, si consideramos MC^2 constante, el algoritmo tiene una complejidad $O(NJ)$, es decir, de crecimiento lineal con el número de jugadores. Notamos, también, que *generarSalida*() no depende del parámetro *NB*, ya que de hecho este parámetro no interviene hasta el momento de jugar la partida.

Glosario

algoritmo heurístico *m* Algoritmo que soluciona un problema sin asegurar que la solución encontrada sea la óptima, pero que generalmente obtiene una buena solución parcial, próxima a la óptima. Los problemas computacionalmente intratables se acostumbran a resolver usando algoritmos heurísticos.

complejidad espacial *f* Número de unidades de memoria que necesita un algoritmo *A* para resolver un problema con una entrada de tamaño *N*, y se denota por $S_A(N)$.

complejidad polinómica *f* Complejidad que se puede expresar mediante un polinomio y que, en general, quiere decir que el algoritmo en cuestión es computacionalmente tratable.

complejidad temporal *f* Número de unidades de tiempo que necesita un algoritmo *A* para resolver un problema con una entrada de tamaño *N*, y se denota por $T_A(N)$.

computacionalmente intratable *adj* Dicho de aquel problema de una complejidad algorítmica exponencial, cuya resolución tiene un coste que crece exponencialmente con el tamaño de la entrada, como por ejemplo, generar todos los subconjuntos posibles de un conjunto.

notación asintótica *f* Notación que permite simplificar todas las constantes que acompañan a las funciones $T_A(N)$ o $S_A(N)$ en el análisis de la complejidad algorítmica, y hacer comparaciones para valores grandes de *N*. La notación más usada es la *O*, pero hay hasta cinco diferentes, dependiendo de su uso.

Bibliografía

Aho, A.; Hopcroft, J.; Ullman, J. (1998). *Estructuras de datos y algoritmos*. Wilmington: Addison-Wesley Iberoamericana.

Biggs, N. L. (1994). *Matemática discreta*. Barcelona: Vicens Vives.