UNIVERSIDAD
DE GRANADA

Algorítmica: un resumen como cualquier otro

Advertencia: este resumen se encuentra en desarrollo.
Bienvenidas son las modificaciones y sugerencias.

Juan Ocaña Valenzuela

7 de julio de 2018

https://github.com/patchispatch

Índice

1. La eficiencia de los algoritmos

1.1. Tema 1: Planteamiento general

En este tema trataremos conceptos generales, definiendo qué son las Ciencias de la Computación, el concepto de algoritmo, y la distinción entre problemas y casos.

1.1.1. Definiciones iniciales

- **Ciencia de la Computación:** La Ciencia de la Computación es el estudio de los Algoritmos, incluyendo sus propiedades, su hardware, sus aspectos lingüísticos y sus aplicaciones.
- **Algoritmo:** un algoritmo es una secuencia finita y ordenada de pasos, exentos de ambigüedad, tal que al llevarse a cabo con fidelidad, dará como resultado que se realice la tarea para la que se ha diseñado, con recursos limitados y tiempo finito.

  Esta definición presenta una serie de problemas, como la definición de fidelidad, o lo que se considera como tiempo finito.

  Definición de Bazaara, Sheraly y Shetty: un algoritmo es un proceso iterativo que genera una sucesión de puntos, conforme a un conjunto dado de instrucciones, y un criterio de parada.

  Existen más definiciones, pero el autor de estos apuntes no considera que resulten relevantes como para ponerlas aquí.

  Nota: *Algoritmo y programa son conceptos diferentes. Un programa es una serie de instrucciones codificadas en un lenguaje de programación, que expresa un algoritmo y que puede ser ejecutado en un computador.*

1.1.2. Características primordiales de un algoritmo

Un algoritmo debe poseer las siguientes cinco características:

1. **Finitud:** ha de terminar después de un tiempo acotado superiormente.
2. **Especificidad:** cada etapa debe estar rigurosamente definida y especificada para cada caso.
3. **Input:** un algoritmo tiene una o más entradas o *inputs*.
4. **Output:** un algoritmo tiene una o más salidas u *outputs*.
5. **Efectividad:** todas las operaciones a realizar deben ser tan básicas como para ser resueltas en un periodo de tiempo finito usando lápiz y papel.

   Nota: *esta última característica pierde el sentido en cuanto el algoritmo se concibe para ser utilizado en computadores. Podría entenderse como «utilizar operaciones que puedan ser resueltas en un tiempo razonable».*

1.1.3. La Algorítmica

Hemos visto qué es un algoritmo y cuáles son sus propiedades, pero, ¿existe alguna rama de las Ciencias de la Computación dedicada a su estudio? Por supuesto; si no, no estarías dedicando tu tiempo a leer este resumen. Como pone encima de este párrafo, la **Algorítmica** o Teoría de Algoritmos es la rama dedicada a las siguientes tareas relacionadas con los algoritmos:

1. La construcción.
2. La expresión.
3. La validación.
4. El análisis.
5. El testeo.

A continuación, explicaremos en qué consiste cada uno de estos pasos.

- **Construcción**

  El área de la construcción de algoritmos engloba el estudio de los métodos que, facilitando esta tarea, se han demostrado más útiles en la práctica.

- **Expresión**

  Los algoritmos han de tener una expresión lo más clara y concisa posible.

- **Validación**

  Una vez construido el algoritmo, se debe demostrar que realiza su tarea correctamente sobre inputs *legales*. Este proceso se conoce como validación, y pretende asegurar que el algoritmo trabajará sin problemas independientemente del lenguaje empleado, o la tecnología usada.

  Una vez validado el algoritmo, puede utilizarse para escribir un programa, por ejemplo.

- **Análisis**

  Es el proceso de determinar cuánto tiempo de cálculo y almacenamiento consumirá un algoritmo. Esto nos permite comparar algoritmos entre sí, y decidir cuál es mejor para realizar una determinada tarea.

- **Test**

  El test se realiza sobre un programa que implementa un algoritmo. Supone la corrección de errores detectados, y la comprobación de tiempo y espacio para la ejecución del mismo.

1.1.4. Elección de un algoritmo

Ahora que disponemos de nuestros algoritmos diseñados, validados y analizados, es hora de utilizarlos para resolver nuestro problema. Pero, ¿cuál utilizamos? ¿Qué algoritmo podrá garantizarnos un mejor resultado? A veces es muy fácil, pues sólo hay una opción posible, pero no suele ser el caso.
Lo que debemos hacer es determinar ciertas características de cada algoritmo para evaluar su rendimiento. Algunos posibles criterios son los siguientes:

- Adaptabilidad del algoritmo a computadores.
- Simplicidad y elegancia.
- Coste económico de su implementación.
- Tiempo de ejecución del algoritmo.

Dependiendo de las características del problema a resolver, compararemos los diferentes algoritmos según las características que más nos interesen.

1.1.5. Problemas y casos

Es muy sencillo entender la diferencia entre **problema** y **caso** con un ejemplo:

Problema: la multiplicación de dos números.
Caso del problema: multiplicar 14 por 25.

Un algoritmo debe funcionar correctamente en todos sus posibles casos. En caso de que tan sólo uno de ellos fuese incorrecto, el algoritmo no sería válido.

El **tamaño de un caso** es cualquier entero que, de un modo u otro, nos indique el número de componentes del caso.

Ejemplos:
Ordenación de un array: tamaño del array.
Operaciones con matrices: número de filas y columnas.

El tiempo consumido por el mismo algoritmo puede ser muy diferente entre dos casos diferentes del mismo tamaño. Usemos como ejemplo la inserción de un elemento en un vector de forma ordenada.

Si tenemos un caso **U** en el que el vector ya se encuentra ordenado ascendentemente, y un caso **V** en el que lo está pero de forma descendente, la inserción de un valor en U será mucho más rápida que en V para un tamaño de, por ejemplo, 5000 elementos.

V describe el **peor caso** de este algoritmo, mientras que U es el **mejor caso**.

El **peor caso** es útil cuando necesitamos una garantía total acerca del tiempo de ejecución de un programa. **Generalmente utilizaremos este.**

El **mejor caso** es el caso en el que el algoritmo tarda menos tiempo.

También podemos considerar el **caso promedio**, aunque no suele tenerse en cuenta. Se trata del número medio de etapas en cualquier caso de tamaño *n*.

1.2. Tema 2: Tiempo de ejecución. Notaciones para la eficiencia de los algoritmos

En este tema vamos a ver cómo calcular y expresar la eficiencia de un algoritmo, y las diferentes notaciones que podemos utilizar para ello.

1.2.1. La eficiencia de los algoritmos

¿Cómo podríamos expresar la eficiencia de un algoritmo? ¿En qué unidad? Antes de responder a estas cuestiones, necesitamos calcularla. Para ello, disponemos de tres métodos:

- **El enfoque empírico (a posteriori):** probar el algoritmo y medir sus resultados. Depende del agente tecnológico que usemos.
- **El enfoque teórico (a priori):** no depende del agente tecnológico, sino de cálculos matemáticos.
- **El enfoque híbrido:** la forma de la función que describe el algoritmo se calcula de forma teórica, y cualquier parámetro numérico que se necesite se determinará de forma empírica sobre un programa y una máquina en concreto.

Para seleccionar la unidad con la que medir la eficiencia de un algoritmo, debemos tener en cuenta el **Principio de Invarianza:**

Dos implementaciones diferentes de un mismo algoritmo no difieren en eficiencia más que, a lo sumo, en una constante multiplicativa.

Por tanto, si dos implementaciones consumen $t_1(n)$ y $t_2(n)$ unidades de tiempo, respectivamente, siempre existe una constante positiva c tal que $t_1(n) \leq t_2(n)$.
Este principio es independiente del agente tecnológico usado.

Teniendo esto en cuenta, parece oportuno referirnos a la eficiencia de un algoritmo en términos de tiempo.
El tiempo de ejecución de un programa depende de:

- El input.
- La calidad del código generado por el compilador.
- La naturaleza y velocidad de las instrucciones utilizadas.
- La complejidad en tiempo del algoritmo utilizado.

$T(n)$ notará el tiempo de ejecución de un programa para un input de tamaño n, y también el del algoritmo que implementa.

Expresaremos el tiempo que consume un algoritmo como $t(n)$ si existe una constante positiva c y una implementación del algoritmo cuyo tiempo de ejecución esté acotado superiormente por $ct(n)$ segundos.
En la constante c acumularemos todos los factores relativos a la máquina utilizada. La llamaremos **constante oculta**.

1.2.2. Notación asintótica O, Ω, Θ

Aún así, necesitamos una notación que nos permita comparar algoritmos en términos iguales. Supongamos un algoritmo con dos implementaciones, una que tarde n^2 días en resolver un caso de tamaño n, y otra que lo haga en n^3 segundos. Para casos muy grandes, el algoritmo cuadrático puede ser más rápido, pero en general, el cúbico nos proporcionará mejores resultados. Algo falla, ¿no?

Para poder realizar esta comparación, se utiliza la **notación asintótica**, que refleja la conducta de las funciones para valores altos de x.

Notación O (la importante)

Una función $f(n)$ está asintóticamente dominada por $g(n)$ si al multiplicar una constante por $g(n)$ obtenemos valores mayores que $f(n)$ para valores de n mayores que una constante k.
O con otras palabras, f es de orden g, $O(g(n))$, si pasado un valor positivo k, f es menor o igual que un múltiplo de g.

La notación O supone usar el término más dominante, por ejemplo:

$$3x^3 + 2x^2 + 4 = O(x^3)$$

Aunque el polinomio también esté acotado superiormente por n^4 y superiores, nos quedamos con $O(n^3)$.

Uso de límites en notación O

Para demostrar que $f(n)$ es de orden $O(g(n))$, tan sólo debemos comprobar si existe límite cuando $n \to \infty$ de $|f(n)/g(n)|$

Notación Ω

Ω es justo lo contrario de O:

$$f(n) = \Omega(g(n)) \leftrightarrow g(n) = O(f(n))$$

Notación Θ

El *orden exacto* o Θ indica que las funciones se dominan asintóticamente entre sí, es decir, son equivalentes:

$$f(n) = \Theta(g(n)) \leftrightarrow f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$$

Tasa de crecimiento

Si un algoritmo está acotado por $O(f(n))$, f(n) es su **tasa de crecimiento**. Teóricamente, un algoritmo con una tasa cuadrática es mejor que otro con una tasa cúbica, pero es posible que en la implementación, con ciertas combinaciones, el segundo algoritmo produzca mejores resultados de tiempo. Esto depende del tamaño de los inputs.

1.2.3. Notación asintótica de Brassard y Bratley

Sea $f : N \to R*$ una función arbitraria.

Notación 0

$$O(f(n)) = t : N \to R* / \exists c \in R^+, \exists n_0 \in N : \forall n \geq n_0 \Rightarrow t(n) \leq cf(n)$$

O lo que es lo mismo, una función es de orden f(n) si existe una constante real positiva c y un valor natural n_0 que cumplan que para todo valor mayor a n_0, $t(n) \leq f(n)$.

Notación Ω

$$\Omega(f(n)) = t : N \to R* / \exists c \in R^+, \exists n_0 \in N : \forall n \geq n_0 \Rightarrow t(n) \geq cf(n)$$

Una función es de orden f(n) si existe una constante real positiva c y un valor natural n_0 que cumplan que para todo valor mayor a n_0, $t(n) \geq f(n)$. Lo contrario a la notación O, como vimos antes.

Notación Θ

$$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$$

Con también mencionamos antes, el orden exacto se cumple cuando ambas funciones se acotan superiormente entre sí, y por tanto, son equivalentes.

Nota: *la condición* $\exists n_0 \in N : \forall n \geq n_0$ *puede evitarse.*