Capítulo 1

Introducción a la algorítmica

1.1. Competencias

Las competencias a desarrollar en la asignatura son:

- **CB5** Que los estudiantes hayan desarrollado las habilidades de aprendizaje necesarias para emprender estudios posteriores con un alto grado de autonomía.
- **CTEC1** Capacidad para tener un conocimiento profundo de los principios fundamentales y modelos de la computación [1] y saberlos aplicar para interpretar, seleccionar, valorar, modelar, y crear nuevos conceptos, teorías, usos y desarrollos tecnológicos relacionados con la informática.
- **CTEC3** Capacidad para evaluar la complejidad computacional de un problema, conocer estrategias algorítmicas que puedan conducir a su resolución y recomendar, desarrollar e implementar aquella que garantice el mejor rendimiento de acuerdo con los requisitos establecidos.

1.2. Introducción

En las asignaturas relacionadas con la programación que el alumno ha estudiado en cursos anteriores ha adquirido una serie de conocimientos que le capacitan para:

- Codificar cualquier algoritmo que resuelva un problema, utilizando el lenguaje C o el lenguaje C++.
- Seleccionar la estructura o estructuras de datos más eficientes para almacenar los datos que utiliza dicho algoritmo.
- Diseñar e implementar algoritmos que resuelvan una gran parte de los problemas que se le puedan plantear.

Para diseñar un algoritmo que permita resolver un determinado problema, la técnica que el alumno ha utilizado hasta ahora consiste en analizar el problema de una forma profunda y detallada, y a partir

[1] en la teoría de la complejidad computacional, un modelo de computación es la definición un conjunto de operaciones permitibles usadas en el cómputo y sus respectivos costos. Solo asumiendo un cierto modelo de computación es posible analizar los recursos de cómputo requeridos, como el tiempo de ejecución o el espacio de memoria, o discutir las limitaciones de algoritmos o computadores

de las propiedades del mismo y de su analogía con otros problemas ya resueltos, se ha formulado un algoritmo que lo resuelve. Este método es el que se ha utilizado en muchos campos de la Ciencia, como las Matemáticas o la Física, asumiendo que no existe ningún tipo de algoritmo *universal* que permita resolver cualquier problema.

En esta asignatura se estudiarán una serie de métodos generales para afrontar la elaboración de algoritmos que resuelvan un problema dado. Hay que hacer hincapié en la palabra afrontar, ya que es lo único que se puede garantizar, debido a que existen problemas que se podrán resolver con los métodos que se van a estudiar, pero otros no, o para precisar más, problemas que hasta ahora nadie ha sabido resolverlos usando estos métodos.

Por otra parte, es importante resaltar que los métodos que se van a analizar no son mutuamente excluyentes, en ciertas ocasiones habrá problemas en los que varios métodos se complementarán a la hora de resolver un problema, o habrá problemas que se puedan resolver usando más de un método.

Finalmente hay que destacar que estos métodos no proporcionan unas *reglas* o *recetas* para la elaboración *automática* de algoritmos, sino que facilitan ideas para el desarrollo *personal* de dichos algoritmos.

1.3. Definiciones y conceptos

La palabra algoritmo procede del nombre del matemático persa del siglo IX al-Khowârizmî. Un **algoritmo** es un conjunto de reglas que se utilizan para realizar algún cálculo, ya sea de tipo manual o en una máquina. Nosotros nos centraremos en aquellos que realizarán cálculos en un ordenador. A lo largo de nuestra formación y desde que se aprenden las operaciones elementales, se están utilizando constantemente algoritmos. Los métodos que aprendimos para realizar sumas, restas, multiplicaciones y divisiones son algoritmos. El algoritmo más famoso de la historia es el que propuso Euclides para calcular el máximo común divisor entre dos enteros. El algoritmo dice lo siguiente:

Sean a y b dos números enteros, tal que $a > b$ y $b \neq 0$, entonces el $mcd(a,b) = mcd(b,r)$ siendo r el resto de dividir a entre b. Si $b = 0$ entonces $mcd(a,b) = a$.

Expresado en forma de algoritmo quedaría como sigue:

Algoritmo $mcd(a,b)$
  inicio
    si $b = 0$ **entonces**
      devolver a
    sino
      $r \leftarrow a \ \ mod \ \ b$
      devolver $mcd(b,r)$
    finsi
  fin

Las reglas del algoritmo han de ser precisas y no pueden plantear ninguna ambigüedad que implique una decisión subjetiva de la máquina o persona que lo esté ejecutando. Por ejemplo, si consideramos una receta de cocina como un algoritmo, no se pueden emplear reglas del tipo *emplee una cantidad de azucar a su gusto*, ya que eso implicaría ambigüedad. Aunque hay que resaltar que esta rigidez se puede violar en algoritmos probabilistas, ya que en ellos se utilizan procedimientos que efectúan elecciones aleatorias, con unas probabilidades conocidas y predeterminadas.

Cuando aplicamos un algoritmo, se supone que de la aplicación de sus reglas obtendremos una respuesta correcta a un problema determinado, aunque puede ocurrir que esta respuesta no sea tan precisa como nosotros queremos. Por ejemplo, el resultado del algoritmo es un número real y no

podemos obtener el número de decimales que necesitamos. En este caso tendremos un algoritmo aproximado, en el cual se podría especificar el error que podemos aceptar.

A veces puede ocurrir que por el tamaño del problema no existen algoritmos prácticos y cualquiera de los que empleemos para encontrar una solución exacta necesitaría un tiempo excesivo (años e incluso siglos). En este caso es preciso buscar un conjunto de reglas que conformen un algoritmo que proporcione una solución aproximada al problema y demostrar que dicha solución no es *excesivamente* errónea. A veces ni siquiera esto es posible y solamente nos podemos fiar de nuestra buena suerte y usar un algoritmo con una pequeña base teórica y una *esperanzadora* intuición. Dichos algoritmos se denominan **algoritmos heurísticos** o simplemente una **heurística**. En estos algoritmos no podemos controlar el error, pero a veces es posible estimar su magnitud.

La **algorítmica** se puede definir como el estudio de los algoritmos desde el punto de vista de su eficiencia. Para resolver un problema pueden existir muchos algoritmos disponibles y es importante el realizar una eleccion correcta del algoritmo a adecuado. Esta elección puede depender de:

- Los límites de memoria.
- La velocidad del equipo disponible.
- El tiempo empleado por el algoritmo.
- Facilidad de implementación del algoritmo.
- Tamaño del ejemplar del problema que queramos resolver.

La algorítmica nos permite evaluar el efecto de todos estos factores sobre los algoritmos disponibles y seleccionar el más adecuado en cada momento.

Un *problema* normalmente puede presentar infinidad de casos o **ejemplares**. Por ejemplo el producto $15 * 289$ es un ejemplar del problema de multiplicar dos enteros positivos, y para este problema se pueden presentar infinitos ejemplares. Un algoritmo debe funcionar correctamente en todos los ejemplares o casos del problema en cuestión. Si se encuentra un solo ejemplar donde el algoritmo no funciona, el algoritmo es incorrecto. La demostración de que un algoritmo es correcto puede llegar a ser una tarea muy compleja, y para conseguirlo es muy importante definir el **dominio de definición** del problema, o lo que es lo mismo, el conjunto de casos a considerar. Por ejemplo, hay algoritmos de multiplicación que no funcionan con números negativos o fraccionarios, pero esto no implica que el algoritmo no sea correcto. En este caso diremos que los números negativos o fraccionarios no pertenecen al dominio de definición del problema.

1.4. Eficiencia o complejidad de un algoritmo

Como se ha mencionado en el apartado anterior, para resolver un problema pueden existir muchos algoritmos disponibles y es importante el realizar una eleccion correcta del más adecuado. A partir de aquí surge la duda de la forma de decidir cual es el algoritmo apropiado. Si por ejemplo queremos ordenar, según un determinado criterio, un conjunto de algunos cientos de elementos puede que no importe demasiado el algoritmo de ordenación utilizado (de hecho no importa), y en este caso se puede usar el algoritmo más fácil de programar. Sin embargo, si tenemos que ordenar muchos conjuntos y con tamaños que sobrepasen las decenas o centenas de miles de elementos, ya hay que seleccionar un método de ordenación eficiente, que permita resolver el problema en un mínimo de tiempo. En la elección del algoritmo se evalúan los recursos necesarios para su ejecución. A este proceso de evaluación de los recursos se le denomina *Cálculo de la eficiencia o complejidad de un algoritmo*. Se dice que un algoritmo es más eficiente o de menor complejidad que otro cuando utiliza menos recursos.

Los principales recursos que suelen tenerse en cuenta a la hora de evaluar la eficiencia de un algoritmo son:

- Espacio de memoria que necesita: *la eficiencia en memoria o complejidad espacial* de un algoritmo indica la cantidad de almacenamiento necesario para ejecutar el algoritmo, es decir, el espacio de memoria que ocupan todas las variables utilizadas por el algoritmo. Suele ser fácil calcular la memoria estática de un algoritmo, ya que sólo hay que sumar la memoria ocupada por todas las variables declaradas en el algoritmo. Sin embargo, el cálculo de la memoria dinámica no es tan fácil, debido a que ésta depende de cada ejecución concreta del mismo.

- Tiempo que tarda en ejecutarse: *la eficiencia en tiempo o complejidad temporal* de un algoritmo indica el tiempo que requiere para su ejecución.

Desgraciadamente la eficiencia en tiempo y en espacio son, en la mayoría de las ocasiones, objetivos contrapuestos, ya que la optimización de uno ellos, generalmente, siempre es a costa de la del otro. El programador es quien debe establecer la relación adecuada entre espacio y tiempo, para lo cual ha de tener en cuenta el uso que se le va dar del algoritmo y los recursos disponibles.

En esta asignatura nos centraremos en el estudio de la eficiencia en tiempo o complejidad temporal, ya que normalmente es el más importante. Solo compararemos los algoritmos en base a sus tiempos de ejecución, y cuando se hable de eficiencia nos referiremos al tiempo que consume dicho algoritmo.

El incesante aumento de capacidad que poseen los nuevos computadores, tanto en velocidad de cómputo, como en almacenamiento de memoria, puede hacernos creer que es absurdo el estudio de la complejidad temporal de los algoritmos, ya que se podría pensar que cualquier algoritmo por muy complejo o poco eficiente que sea, se ejecutará en un tiempo mínimamente razonable. Nada más lejos de la realidad. Ello es debido a que un mal algoritmo ejecutado en un computador muy potente podría tardar un tiempo excesivo y muy superior al que tardaría otro algoritmo mejor que se ejecute en otra máquina menos potente. Por ejemplo, un ordenador personal puede tardar pocos segundos en ordenar crecientemente un vector de 10.000.000 enteros utilizando un buen algoritmo (*quicksort*), mientras que un computador VAC (100 veces más rápido que un PC) requiere del orden de varios minutos en ordenar el mismo vector si utiliza un mal algoritmo (*método de la burbuja*). También podríamos citar otros ejemplos relevantes:

- **Cálculo de determinantes.** Para calcular un determinante se podría utilizar un algoritmo recursivo, aplicando la definición del cálculo de un determinante o el algoritmo de eliminación de Gauss-Jordan, basado en la triangularización del determinante. El primero de ellos, debido a la repetición de llamadas recursivas podría emplear del orden de años para resolver un determinante de 50x50, si es que antes no se desborda la pila, mientras que el segundo tardaría una fracción de segundo.

- **Cálculo del término n-ésimo de la sucesión de Fibonacci.** Para resolver este problema también se puede emplear un algoritmo recursivo basado en la definición del término n-ésimo de la sucesión, o un algoritmo iterativo. El algoritmo recursivo, si no controla la repetición de las llamadas recursivas, podría tardar del orden de días para calcular el término 60 de la sucesión. Sin embargo, el algoritmo iterativo emplearía una fracción de microsegundo para términos del mismo orden.

1.5. Factores que influyen en el cálculo de la complejidad temporal

Hay tres factores que influyen en el cálculo de la complejidad temporal:

- El tamaño de los datos de entrada o tamaño del ejemplar.
- El contenido de los datos de entrada.
- El computador y el código generado por el compilador.

El factor más importante para medir la eficiencia de un algoritmo es el tamaño de los datos de entrada. El tamaño de un ejemplar x se corresponde formalmente con el número de dígitos binarios necesarios para representarlo en el computador, aunque normalmente se suele identificar el tamaño con el número de elementos o items informativos contenidos en el ejemplar. Por ejemplo, un problema consistente en ordenar n enteros o de ordenar n registros de cualquiera tamaño será considerado de tamaño n, aunque se necesite en la práctica más de un dígito binario para representar cada entero o cada registro. Cuando se trate de problemas numéricos, como el cálculo del factorial de un número, la eficiencia se expresará en función del *valor* del dato considerado y no en función de su tamaño (que sería la longitud de la representación binaria de dicho valor). De forma similar, cuando hablemos de operaciones sobre matrices cuadradas, se medirá el tamaño de un ejemplar por el orden de la matriz, y la eficiencia del algoritmo se dará en función de ese orden.

El contenido de los datos de entrada, hace referencia a la disposición o distribución de estos datos. Por ejemplo, a la hora de ordenar un conjunto no es lo mismo que éste esté casi ordenado a que esté totalmente desordenado. Para evaluar su influencia hay dos estrategias posibles:

- Estudio de la eficiencia *en el caso peor*: fijado un tamaño del problema, se analiza la eficiencia del algoritmo en aquellas situaciones en las que emplea más tiempo. El objetivo es obtener una cota superior del tiempo de ejecución del algoritmo.
- Estudio de la eficiencia *en el caso medio*: es necesario conocer el tiempo de ejecución del algoritmo en todas las situaciones y la frecuencia con que éstas se presentan, es decir, su distribución de probabilidades. Esta distribución suele ser difícil de conocer, por lo que se han de realizar hipótesis cuya fiabilidad es dicutible. Además, las operaciones matemáticas necesarias para realizar el cálculo puede llegar a hacer éste impracticable en muchos casos. Por todo ello *el caso medio* se estudia mucho menos que el análisis en *el caso peor*. En lo que sigue, salvo que se indique lo contrario, siempre se realizará el análisis de la eficiencia en el peor de los casos.

El tercer factor, el computador y el código generado por el compilador, no se suele tener en cuenta a la hora de calcular la eficiencia de un algoritmo debido a las siguientes razones:

- Se pretende analizar la eficiencia de un algoritmo de un modo totalmente independiente de las máquinas y lenguajes existentes.
- *El principio de invariancia*: los tiempos de ejecución de las diferentes implementaciones de un mismo algoritmo diferirán a lo sumo en *una constante multiplicativa* positiva, para tamaños del problema suficientemente grandes. Más precisamente, si $t_1(n)$ y $t_2(n)$ son los tiempos de ejecución de dos implementaciones, siempre existe un número real positivo c y un número entero n_0 tales que $\forall n \geq n_0 \ (t_1(n) \leq ct_2(n))$. Por ejemplo, un factor constante de 10, 100 ó 1000 en los tiempos de ejecución no se considera en general importante frente a una diferencia en la dependencia del tamaño n del problema, ya que, para tamaños suficientemente grandes, es dicha dependencia quien establece la diferencia real. Sin embargo, a veces esta constante

puede jugarnos malas pasadas. Por ejemplo si tenemos dos algoritmos cuyas implementaciones en una cierta máquina requieren tiempos de n^2 días y n^3 segundos, para un ejemplar de tamaño n, evidentemente la segunda implementación requiere un tiempo de orden superior, pero la constante multiplicativa es muy pequeña en el segundo caso. Se puede demostrar que solo en casos en los que se requieran más de 20 millones de años para resolverlos, el algoritmo cuadrático es más eficiente que el cúbico, aunque asintóticamente (para n infinito) el cuadrático es mejor que el cúbico.

1.6. Enfoques en la evaluación de la eficiencia de los algoritmos

A la hora de evaluar la eficiencia de los algoritmos existen tres enfoques:

- Empírico o a posteriori: los algoritmos se implementan en un computador y se comparan mediante la realización de pruebas con datos del problema de distinto tamaño. Por ejemplo, implementamos dos métodos de ordenación y obtenemos los tiempos que se emplean para ordenar conjuntos de distinto tamaño.

- Teórico o a priori: se determina matemáticamente la cantidad de recursos necesitados por cada algoritmo como una función cuya variable independiente es el tamaño de los datos del problema.

- Híbrido: se determina teóricamente la función que describe la eficiencia de un algoritmo y luego se calculan empíricamente los parámetros numéricos requeridos por un programa y un computador concretos. Este enfoque permite hacer predicciones, mediante extrapolación, sobre el tiempo que tardaría en ejecutarse el algoritmo en una implementación concreta con unos datos mucho más grandes que los usados en las pruebas. Por ejemplo, se sabe que un método de ordenación es de orden $O(n^2)$ y empíricamente, mediante un análisis de regresión, podemos obtener los términos del polinomio de de segundo grado que mejor se ajusta al tiempo de ejecución para un ordenador dado.

El segundo enfoque es el más interesante, porque ofrece una medida independiente del computador, el lenguaje de programación y la capacidad del programador. Ahorra, por un lado, el tiempo que se tardaría en programar sin necesidad un algoritmo ineficiente y, por otro lado, el tiempo del computador que se desperdiciaría probándolo. Además, permite estudiar la eficiencia de un algoritmo cuando es usado con datos de cualquier tamaño, mientras que con los enfoques empírico o híbrido, las consideraciones prácticas pueden forzar a probar los algoritmos sólo con datos de tamaño moderado. Este punto es muy importante, ya que muchos algoritmos comienzan a ser eficientes en comparación con otros cuando se usan datos de gran tamaño. Debido a lo anterior, el enfoque teórico o a priori es el que se va a considerar en la asignatura y el principal factor que se utilizará será el tamaño de los datos de entrada, pero también se tendrá en cuenta el factor relativo al contenido de los datos de entrada cuando éstos reflejen el peor de los casos posibles.

1.7. Operaciones elementales

Una **operación elemental** es aquella cuyo tiempo de ejecución se puede acotar superiormente por una constante que solo depende de la implementación particular usada (máquina, lenguaje de programación, etc.). Las operaciones elementales y su tiempo de ejecución definirán un **modelo de computación**. El número de operaciones elementales será el que importe en el análisis y no el

tiempo exacto requerido en cada una de ellas, ya que esto se reflejará en una constante multiplicativa y no en el orden del algoritmo. Esto se puede apreciar en el siguiente ejemplo:

Supongamos que un algoritmo efectúa s sumas y cada una consume a lo sumo t_s microsegundos, m multiplicaciones y cada una consume a lo sumo t_m microsegundos, y a asignaciones que consume cada una a lo sumo t_a microsegundos. Estos tiempos son constantes que dependen de la máquina usada, por tanto la suma, la multiplicación y la asignación se pueden considerar como operaciones elementales.

El tiempo total empleado por el algoritmo estará acotado por:

$$t \leq s * t_s + m * t_m + a * t_a \leq max(t_s, t_m, t_a) * (s + m + a) \tag{1.1}$$

En definitiva, t está acotado por un múltiplo del número de operaciones elementales que emplea el algoritmo. Como el tiempo exacto que requiere cada operación elemental no tiene mucha trascendencia, se considera que las operaciones elementales emplean un tiempo unitario. En definitiva el tiempo total empleado estará acotado por el número de operaciones elementales que emplea el algoritmo.

Aunque se ha considerado que la suma y la multiplicación son operaciones elementales de coste unitario, en teoría estas operaciones no son elementales de coste unitario ya que el tiempo que necesitan aumenta con el tamaño de los operandos. En la práctica se consideran como elementales si los operandos implicados en ellas son de un tamaño razonable en los ejemplares que nos vamos a encontrar en el problema. Por ejemplo si calculamos el factorial de un número n, el hecho de tomar un valor de n relativamente grande puede implicar que la multiplicación no se realice en un tiempo unitario y además se produzcan problemas de desbordamiento aritmético (se sobrepasa el tamaño destinado a un entero). Por ello, el análisis de un algoritmo que implique el cálculo de un factorial debe depender del dominio de la aplicación para el cual está destinado. En concreto, si los operandos son de gran magnitud, un algoritmo que a priori puede parecer lineal, puede requerir de un tiempo cuadrático.