Capítulo 2

Análisis de algoritmos

El *análisis de algoritmos* se erige como uno de los pilares fundamentales en el campo de la algoritmia, proporcionando herramientas para evaluar y comparar algoritmos en condiciones teóricas y prácticas. Esta evaluación es crítica no solo para el desarrollo de software, sino también para la optimización de recursos en sistemas computacionales complejos.

El análisis de algoritmos se centra en medir la eficiencia con la que un algoritmo opera. La eficiencia puede medirse de múltiples formas, pero su objetivo principal es determinar qué tan bien se comporta un algoritmo bajo diversas condiciones con respecto a unos criterios correctamente definidos. Esto incluye identificar si un algoritmo es, en esencia, *eficiente* en términos de los recursos que consume, y si puede considerarse superior a otras alternativas para resolver el mismo problema. Entender cómo y por qué un algoritmo consume recursos específicos es esencial para mejorar y adaptar las soluciones a los requisitos cambiantes de la tecnología y las aplicaciones.

2.1. Complejidad algorítmica

Cuando hablamos de los recursos que consume un algoritmo, nos referimos a su *complejidad*. Tradicionalmente, el estudio de la complejidad algorítmica se centra en dos categorías: complejidad temporal y complejidad espacial, referidas a la cantidad de tiempo y de memoria que el algoritmo requiere, respectivamente. Aun así, según el contexto, puede haber otros recursos relevantes, como el número de accesos a base de datos o la energía que consume ejecutar un algoritmo, entre otros.

La ejecución de un algoritmo y, por ende, su complejidad, pueden verse afectadas por factores tanto externos como internos. Entre los externos,

encontramos variables como la capacidad del hardware, que puede acelerar o retardar la ejecución; el compilador, cuyas optimizaciones pueden reducir significativamente el número de operaciones necesarias; y la naturaleza de los datos de entrada, que puede variar dramáticamente entre ejecuciones. Internamente, la complejidad está influenciada únicamente por el número y tipo de instrucciones del algoritmo.

Siguiendo la idea anterior, vamos a distinguir dos tipos de análisis de complejidad:

- **Análisis empírico**: un análisis empírico consiste en ejecutar el algoritmo para distintos valores de entrada y medir directamente la cantidad de recursos utilizados. Por ejemplo, si nos referimos a recursos temporales, podemos cronometrar el tiempo de ejecución. La ventaja principal es que obtenemos una medida *real* del comportamiento del algoritmo en un entorno concreto. Sin embargo, este análisis puede verse afectado por cuestiones extrínsecas al propio algoritmo.

- **Análisis teórico**: un análisis teórico consiste en obtener una función matemática que represente la complejidad del algoritmo. Tiene la ventaja de que no necesita ejecutar el algoritmo y el resultado depende exclusivamente del diseño del mismo. Sin embargo, es difícil trasladar esta función a términos prácticos de ejecución en un entorno real.

2.2. Complejidad teórica

El análisis de la complejidad teórica de un algoritmo se basa en la formulación de una función matemática que describe cómo varía el consumo de recursos en función del tamaño de la entrada. Esta medida abstracta proporciona una visión independiente del hardware y del entorno de ejecución, enfocándose en el diseño y la estructura del algoritmo.

Comencemos con un ejemplo. Consideremos el siguiente algoritmo:

```
función es_par(número):
    devuelve mód(número,2) = 0
```

Para simplificar, se suele considerar que el coste temporal de las operaciones elementales es unitario (1). Por tanto, la complejidad de este algoritmo es de 3: operador módulo, comparación y retorno.

En realidad, el ejemplo anterior no es muy interesante. Normalmente, la complejidad de un algoritmo se calcula en función de su *talla*. La talla de un algoritmo se refiere generalmente a la cantidad de datos de entrada que el

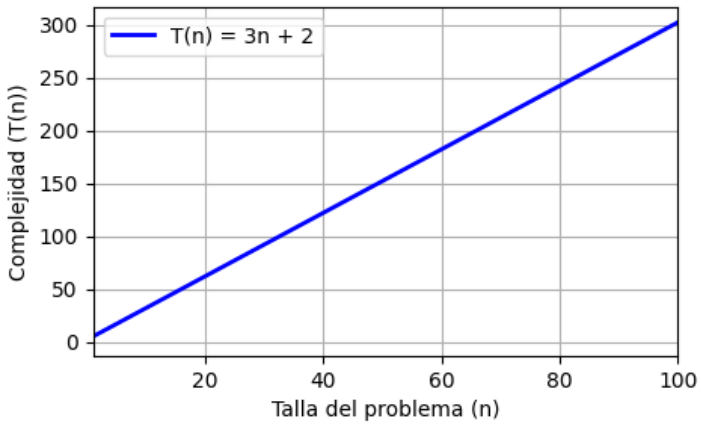

Figura 2.1: Complejidad de `acumulado` en función de la talla del problema.

algoritmo debe procesar, la cual puede medirse en términos del número de elementos en una estructura de datos, como un vector o una lista, o bien en términos de algún otro parámetro relevante para el problema en cuestión. En general, es interesante calcular cuánto varía la complejidad de un algoritmo en función de su talla, lo que permite saber cómo escala a medida que se enfrenta a entradas de mayor tamaño. Por ejemplo:

```
función acumulado(vec)
    suma := 0
    para i := 1 hasta |vec|
        suma += vec[i]
    devuelve suma
```

En este caso, la talla del problema es el tamaño del vector `vec`. Por convención, usaremos la letra n para referirnos a la talla del problema y T para referirnos a la función que representa la función de complejidad del algoritmo. Por tanto, la complejidad de este algoritmo `acumulado` se representaría como $T(n) = 1 + n(3) + 1 = 3n + 2$ (ver Figura 2.1).

Cuando utilicemos los cálculos de complejidad para comparar algoritmos que resuelvan el mismo problema, es importante que en ambos casos la talla se refiera exactamente a la misma magnitud.

2.2.1. Cotas de complejidad

En ocasiones, un algoritmo conlleva una complejidad diferente en función de variables que no dependen del diseño del algoritmo en sí sino de los

elementos que haya en la entrada. Consideremos el siguiente algoritmo, que busca un valor (`z`) en un vector (`v`) y devuelve su posición:

```
función buscar(v, z)
    para i desde 1 hasta |v|
        si v[i] = z
            devuelve i
    devuelve NO_ENCONTRADO
```

En función de los valores concretos de `v` y `z`, podemos definir diferentes complejidades. Analicemos algunas posibilidades:

- El valor `z` es el primer elemento del vector: el interior del bucle solo se ejecutaría una vez. Por tanto, su complejidad podría definirse como $T(n) = 3$ (contando una iteración del bucle, la condición y el retorno).
- El valor `z` no se encuentra: el interior del bucle se ejecutaría n veces (siendo n el número de elementos del vector). Su complejidad se definiría como $T(n) = n(2) + 1 = 2n + 1$.

Por tanto, ¿cuál es la complejidad de este algoritmo? Ante este tipo de situaciones, se estiman las llamadas *cotas de complejidad*, es decir, los valores extremos. Por un lado, tendríamos la complejidad en el *mejor caso* (la complejidad del algoritmo cuando se dan las condiciones más favorables) y la complejidad en el *peor caso* (la complejidad del algoritmo cuando se dan las condiciones más desfavorables. En lugar de dar un único valor, el análisis del algoritmo indiciaría en qué rangos puede encontrarse la complejidad del mismo. En resumen, diríamos que el algoritmo `buscar` tiene una complejidad entre $T(n) = 3$ y $T(n) = 2n + 1$ (ver Figura 2.2).

2.3. Notación asintótica

Para describir la complejidad de un algoritmo, especialmente en términos de su tiempo de ejecución, se utiliza frecuentemente la notación asintótica, conocida popularmente como "Big O" ($\mathcal{O}$). Esta notación proporciona un medio para representar los límites superiores de la complejidad de un algoritmo, enfocándose en cómo se comporta a medida que el tamaño de entrada crece hacia el infinito e ignorando costes superfluos. Este análisis es esencial para entender el comportamiento del algoritmo en escenarios de gran escala, que es donde las diferencias en complejidad se vuelven más relevantes.

Formalmente, decimos que:

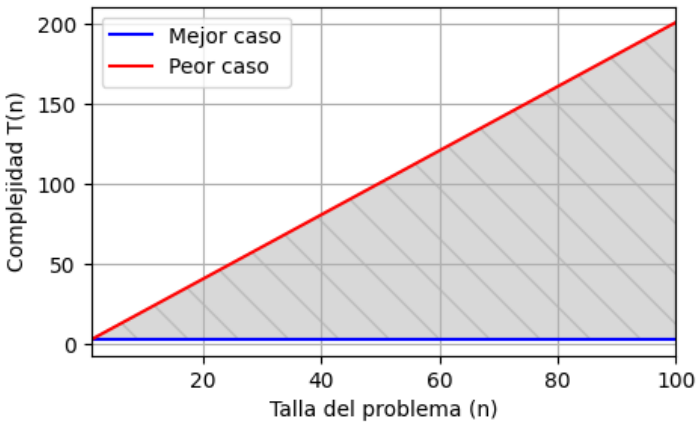

Figura 2.2: Cotas de complejidad de `buscar`.

$$T(n) \in \mathcal{O}(g(n)) \iff \exists n_0, c > 0 : T(n) \leq c \cdot g(n) \, \forall n \geq n_0$$

En otras palabras, una función $T(n)$ pertenece al conjunto de funciones $\mathcal{O}(g(n))$ si existen constantes positivas c y n_0 tales que, a partir de $n \geq n_0$, la función $T(n)$ está acotada superiormente por $c \cdot g(n)$. En esta definición:

- $T(n)$ representa la función de complejidad de un algoritmo para un tamaño de entrada n.
- $g(n)$ es una función que proporciona una cota superior simple para $T(n)$ cuando $n \to \infty$.
- c es una constante positiva que escala la función $g(n)$ para que acote a $T(n)$.
- n_0 es el valor a partir del cual la cota se hace válida, indicando que la comparación solo es significativa para valores grandes de n.

La clave para entender la notación "Big O" es que abstrae el comportamiento de un algoritmo para entradas grandes, permitiendo centrarse en lo que realmente importa: cómo escala el algoritmo. Los factores constantes y los términos de menor orden, que también influyen en la complejidad práctica, se ignoran en esta notación, proporcionando una manera simplificada pero eficaz de analizar y comparar algoritmos en escenarios extremos.

Para aclarar cómo las funciones concretas se clasifican bajo la notación asintótica, consideremos las categorías de complejidad más comunes:[1]

- Complejidad **constante** $\mathcal{O}(1)$: Independientemente del tamaño de la entrada n, la complejidad del algoritmo no cambia. Ejemplo: $T(n) = 3 \in \mathcal{O}(1)$.
- Complejidad **logarítmica** $\mathcal{O}(\log n)$: La complejidad aumenta logarítmicamente con el tamaño de entrada. Ejemplo: $T(n) = 4 \log_{10} n + 6 \in \mathcal{O}(\log n)$.
- Complejidad **lineal** $\mathcal{O}(n)$: La complejidad aumenta directamente en proporción al tamaño de la entrada. Ejemplo: $T(n) = 2n + 3 \in \mathcal{O}(n)$.
- Complejidad **lineal-logarítmica** $\mathcal{O}(n \log n)$: Combina el crecimiento lineal y logarítmico de la complejidad. Ejemplo: $T(n) = 2n \log n + 4n + 3 \in \mathcal{O}(n \log n)$.
- Complejidad **cuadrática** $\mathcal{O}(n^2)$: La complejidad aumenta cuadráticamente en función del tamaño de la entrada. Ejemplo: $T(n) = n^2 + n \log n + n + 1 \in \mathcal{O}(n \log n)$.
- Complejidad **exponencial** $\mathcal{O}(2^n)$: La complejidad crece exponencialmente, lo cual hace que el algoritmo sea ineficiente para tallas grandes. Ejemplo: $T(n) = 2^n + n^2 \in \mathcal{O}(2^n)$.

Como se puede deducir de algunos ejemplos anteriores, los órdenes de complejidad se pueden organizar jerárquicamente:

$$\mathcal{O}(1) \subset \mathcal{O}(\log n) \subset \mathcal{O}(n) \subset \mathcal{O}(n \log n) \subset \mathcal{O}(n^2) \subset \mathcal{O}(2^n)$$

Entender esta jerarquía ayuda a comparar la eficiencia de distintas alternativas algorítmicas rápidamente. En lo sucesivo, cuando hagamos el cálculo de la complejidad de un algoritmo, estaremos interesados en saber a qué orden de complejidad pertenece.

Antes de acabar la sección, conviene mencionar —aunque no vayamos a tratarlo en la asignatura— que la notación $\mathcal{O}$ es parte de un conjunto más amplio de notaciones que incluye Ω, que representa los límites inferiores, y Θ, que indica un límite estricto tanto superior como inferior.

[1] En el Apéndice A se muestran algunas demostraciones de pertenencia a órdenes de complejidad.

2.4. Cálculo de complejidades

Calcular la complejidad de un algoritmo es esencial para entender su eficiencia y prever su comportamiento en diferentes escenarios de aplicación. Para realizar este cálculo de manera sistemática, se pueden seguir los siguientes pasos:

1. **Determinar la talla del problema:** Esto implica definir cuál es el tamaño de la entrada del algoritmo, que generalmente se denota como n. Este tamaño puede representar distintos aspectos, como el número de elementos en una lista, el número de nodos en un grafo, o cualquier otra medida cuantitativa que influya en la ejecución del algoritmo.

2. **Determinar los casos mejor o peor:** Si existen, es crucial identificar el mejor y el peor caso de ejecución. El mejor caso representa la situación en la que el algoritmo realiza el menor gasto posible de recursos, mientras que el peor caso se refiere a la situación opuesta. Los algoritmos también podrían ser analizados bajo el *caso promedio*, que considera un rendimiento típico bajo una distribución estadística de todas las entradas posibles, pero no lo consideraremos en esta asignatura.

3. **Calcular la complejidad asintótica de cada caso:** Una vez identificados los casos relevantes, se analiza la complejidad con respecto a la talla del problema (n). Este análisis se realiza utilizando la notación asintótica para describir estos cambios de forma que se pueda entender el comportamiento del algoritmo en el límite de la talla.

Esta metodología sirve para abordar el cálculo de complejidad de diferentes tipos de algoritmos. En las siguientes secciones, nos centraremos en los algoritmos iterativos y los recursivos.

2.4.1. Complejidad de algoritmos iterativos

Los algoritmos iterativos se caracterizan por su uso de bucles, que repiten secuencias de operaciones hasta que se cumple una condición específica. La complejidad de estos algoritmos viene determinada por el número de veces que se ejecutan los bucles, que en la mayoría de los casos depende directa o indirectamente de la talla del problema.

Para analizar la complejidad de los algoritmos iterativos es importante identificar los bucles principales, esencialmente aquellos que dependen de la

talla del problema. Esto incluye bucles anidados, que son particularmente críticos ya que el número total de iteraciones del bucle interno se multiplica por el número de iteraciones del bucle externo. A continuación, se debe prestar atención a cómo el número de iteraciones de cada bucle depende de la talla. Esto puede ser directo, como un bucle que recorre todos los elementos de un vector, o más complejo (como un recorrido no lineal). Por último, hay que identificar el coste de las operaciones realizadas en cada iteración del bucle. Esto puede variar desde una simple operación hasta llamadas a funciones más complejas.

Para ejemplificar cómo se calcula la complejidad en algoritmos iterativos, vamos a utilizar dos ejemplos: la "multiplicación de matrices" y la "ordenación por inserción".

Complejidad de "multiplicación de matrices"

La multiplicación de matrices es una operación fundamental en álgebra lineal que implica combinar dos matrices para producir una tercera matriz. Para multiplicar dos matrices, el número de columnas de la primera matriz debe ser igual al número de filas de la segunda matriz. El elemento en la fila i y columna j de la matriz resultante se calcula como la suma de los productos de los elementos correspondientes de la fila i de la primera matriz y la columna j de la segunda matriz.

Por simplicidad, vamos a asumir que el algoritmo solo multiplica matrices cuadradas. El siguiente código realiza el producto de dos matrices cuadradas A y B, y almacena el resultado en una matriz C:

```
función producto_matrices_cuadradas(A, B)
    n := dimensión(A)
    C := ceros(A)
        para i := 1 hasta n
            para j := 1 hasta n
                suma := 0
                para k := 1 hasta n
                    suma := suma + A[i][k] * B[k][j]
                C[i][j] := suma
    devuelve C
```

Para analizar la complejidad del algoritmo comenzamos identificando la talla del problema. Aquí hay dos alternativas igualmente justificables: utilizar el tamaño de las matrices ($d = n \times n$) o simplemente el tamaño de cada dimensión (n). Dado que los bucles dependen de una única dimensión,

parece más conveniente calcular la complejidad en función de n. Por otro lado, se puede observar que no hay caso mejor o peor, sino que el algoritmo ejecuta exactamente los mismos pasos independientemente del contenido de las matrices.

Seguiremos el proceso examinando los pasos del algoritmo, los bucles y las operaciones que se ejecutan dentro de estos. Si seguimos el algoritmo por bloques:

- Inicialización: la primera línea obtiene la dimensión de la matrices, con un coste constante $\mathcal{O}(1)$. A continuación, se inicializa una matriz C del mismo tamaño que A y rellenada con ceros. Inicializar una matriz de tamaño $n \times n$ podría implicar un coste de orden $\mathcal{O}(n^2)$, ya que se establece cada elemento a cero.

- Bucles anidados: El código tiene tres bucles anidados, cada uno iterando n veces. Cada bucle corresponde a una dimensión: el bucle exterior recorre las filas de A, el bucle medio recorre las columnas de B, y el bucle interior realiza la multiplicación de elementos y suma para calcular un elemento $C[i][j]$. Dentro del bucle más interno, la operación de suma se realiza n veces por cada combinación de i y j. Cada operación de este tipo implica una multiplicación y una suma, con un coste constante.

Teniendo en cuenta el desglose anterior, podríamos establecer una función de complejidad $T(n) = \mathcal{O}(1) + \mathcal{O}(n^2) + \mathcal{O}(n^3)$. Por lo tanto, la complejidad del algoritmo es de coste cúbico: $T(n) \in \mathcal{O}(n^3)$.

Complejidad de "ordenación por inserción"

Vamos a analizar ahora el algoritmo de "ordenación por inserción", uno de los métodos de ordenación fundamentales en las ciencias de la computación. Analizar algoritmos de ordenación es importante porque la ordenación es un problema ubicuo en la informática, aplicable en áreas que van desde la gestión de bases de datos hasta la inteligencia artificial, así como la optimización de otros algoritmos que requieren datos previamente ordenados para funcionar eficientemente. Además, desde el punto de vista de la algoritmia son interesantes porque existen multitud de alternativas, cada una con sus ventajas y desventajas.[2]

El algoritmo de "ordenación por inserción" simula el proceso de ordenar cartas en una mano. Comienza con un elemento, y en cada paso, toma el

[2]Los algoritmos de ordenación tendrán un papel fundamental en el Capítulo 3.

siguiente elemento y lo inserta en su posición correcta relativa dentro de la porción ya ordenada de la lista. Este proceso se repite hasta que todos los elementos están ordenados. El algoritmo se ejecuta en linea, lo que significa que no necesita un vector adicional para realizar la ordenación, y es *estable*, es decir, mantiene el orden relativo de los elementos que son iguales.

Consideremos el siguiente pseudocódigo:

```
función ordenación_por_inserción(v):
    para i := 2 hasta |v|:
        valor := v[i]
        j := i - 1
        mientras j >= 0 y v[j] > valor:
            v[j + 1] := v[j]
            j := j - 1
        v[j + 1] := valor
```

Para calcular su complejidad, comenzamos identificando la talla del problema. En este caso, la talla viene representada por el tamaño del vector a ordenar (n). Después observamos que el algoritmo de ordenación por inserción sí presenta distintos casos:

- Mejor caso: ocurre cuando el vector ya está ordenado. En este caso, dentro del bucle interior, `v[j]` siempre es menor o igual que `valor`. Por lo tanto, la complejidad en el mejor caso se concentra en el bucle exterior, siendo por tanto del orden $\mathcal{O}(n)$.

- Peor caso: sucede cuando el vector está ordenado en orden inverso. Cada elemento `valor` tiene que compararse e intercambiarse con cada uno de los elementos anteriores ya ordenados, lo que lleva a $\frac{n(n-1)}{2}$ comparaciones e intercambios, resultando en una complejidad de $\mathcal{O}(n^2)$.

2.4.2. Complejidad de algoritmos recursivos

A diferencia de los algoritmos iterativos, que generalmente tienen un análisis más directo basado en la cantidad de veces que se ejecutan los distintos bloques del algoritmo, el análisis de algoritmos recursivos presenta un desafío particular.

La complejidad de un algoritmo recursivo depende especialmente del número y la naturaleza de sus llamadas recursivas. Estas llamadas suelen seguir una *relación de recurrencia* que describe cómo el problema original se descompone en otras llamadas —típicamente, reduciendo la talla del

problema—, las cuales son resueltas de manera similar. Estas relaciones ayudan a entender cómo la solución a un problema más grande se construye a partir de la solución a problemas más pequeños, proporcionando un marco para analizar la complejidad computacional de un algoritmo recursivo.

Para ilustrar este concepto, utilizaremos dos ejemplos concretos: la búsqueda binaria y el algoritmo de ordenación por selección.

Complejidad de "búsqueda binaria"

La búsqueda binaria es un método eficiente para encontrar un elemento específico en una lista ordenada. Es un ejemplo clásico de cómo la estructura y el orden en los datos pueden ser explotados para mejorar dramáticamente la eficiencia de los algoritmos de búsqueda. Este enfoque es particularmente útil en aplicaciones donde los datos están ordenados y las búsquedas son frecuentes, como en bases de datos y sistemas de indexación.

El algoritmo de búsqueda binaria opera bajo la premisa (no hay que verificarla) de que la lista o vector está ordenado. Comienza con un intervalo que cubre toda la lista. En cada paso, el algoritmo compara el elemento en el centro del intervalo con el valor objetivo:

- Si el valor en el centro es igual al valor objetivo, la búsqueda concluye.
- Si el valor objetivo es menor que el valor en el centro, la búsqueda continúa en la mitad izquierda de la lista (valores menores).
- Si el valor objetivo es mayor que el valor en el centro, la búsqueda continúa en la mitad derecha de la lista (valores mayores).

Este proceso se repite, reduciendo a la mitad el tamaño del intervalo de búsqueda en cada paso, hasta que el valor objetivo es encontrado o el intervalo se reduce a cero (lo cual indica que el valor objetivo no está en la lista).

El pseudocódigo del algoritmo de búsqueda binaria que refleja esta descripción es el siguiente:

```
función búsqueda_binaria(v, z)
    si |v| = 0
        devuelve NO_ENCONTRADO
    m := |v| / 2
    si v[m] = z
        devuelve m
    si v[m] > z
```

```
        devuelve búsqueda_binaria(v[:m], z)
    si no
        devuelve búsqueda_binaria(v[m:], z)
```

Para analizar su complejidad, comenzamos identificando la talla del problema: el tamaño del vector ($n = |v|$). También podemos observar que tenemos distintos casos. En el mejor caso, el valor objetivo coincide con el valor en el centro del vector (`v[m] = z` en la primera comparación). Este caso tiene una complejidad constante $\mathcal{O}(1)$, ya que no depende de la talla sino de algunas operaciones básicas de comparación. En cambio, en el peor caso (el valor no está en la lista), tenemos que completar la recursión desde el caso general hasta el caso base (`|v| = 0`).

Para resolver la complejidad del peor caso, vamos a obtener primero la relación de recurrencia. Podemos comenzar con el caso base: cuando $n = 0$, únicamente tenemos la comparación para la terminación del algoritmo, por lo que $T(n) \in \mathcal{O}(1)$. Cuando no se da el caso base ($n \geq 1$), y asumiendo el peor caso, siempre se ejecutarán las comparaciones y alguna llamada que divide el vector entre dos. Por lo tanto, la relación de recurrencia para el tiempo de ejecución $T(n)$ de la búsqueda binaria se define formalmente como:

$$T(n) = \begin{cases} \mathcal{O}(1) & \text{si } n = 0 \\ T(\frac{n}{2}) + \mathcal{O}(1) & \text{si } n \geq 1 \end{cases} \tag{2.1}$$

Vamos a ver cómo resolver la complejidad de esta relación de recurrencia usando el método de expansión. Iremos expandiendo la relación paso a paso para obtener una visión más clara de cómo el problema se descompone en cada llamada recursiva y cómo se van acumulando los costes temporales:

- Expansión inicial:
$$T(n) = T\left(\frac{n}{2}\right) + \mathcal{O}(1)$$
- Sustitución de la primera expansión:
$$T(n) = \left(T\left(\frac{n}{4}\right) + \mathcal{O}(1)\right) + \mathcal{O}(1) = T\left(\frac{n}{4}\right) + 2 \cdot \mathcal{O}(1)$$
- Sustitución de la segunda expansión:
$$T(n) = \left(T\left(\frac{n}{8}\right) + \mathcal{O}(1)\right) + 2 \cdot \mathcal{O}(1) = T\left(\frac{n}{8}\right) + 3 \cdot \mathcal{O}(1)$$

Podemos ver que si seguimos expandiendo, el patrón que emerge es:

$$T(n) = T\left(\frac{n}{2^k}\right) + k \cdot \mathcal{O}(1)$$

Sabemos que este proceso se expande hasta que el argumento de T se reduzca a 0. Es decir, k es el número de veces que necesitamos dividir n por 2 hasta llegar a 0, es decir, $\frac{n}{2^k} = 0$. Matemáticamente, esto ocurre en la llamada recursiva realizada en $T(1)$ (cuando $\frac{n}{2^k} = 1$), por lo que necesitaremos $\log_2 n$ llamadas para llegar a $T(1)$ y una más para terminar la búsqueda. Por lo tanto:

$$T(n) = T(1) + \log_2 n = 2 \cdot \mathcal{O}(1) + \log_2 n$$

La forma final de la relación de recurrencia nos muestra que el tiempo total de ejecución del algoritmo de búsqueda binaria es proporcional al logaritmo en base 2 del tamaño de la entrada. Por lo tanto, la complejidad de tiempo de la búsqueda binaria pertenece al orden $\mathcal{O}(\log n)$.

Como se puede observar, la búsqueda binaria reduce significativamente el tiempo de búsqueda al dividir repetidamente a la mitad el rango de búsqueda, en lugar de inspeccionar cada elemento uno por uno como se haría en una búsqueda secuencial, que tendría una complejidad $\mathcal{O}(n)$.

Complejidad de "ordenación por selección"

Vamos a analizar ahora otro algoritmo clásico de ordenación: la ordenación por selección. Este algoritmo funciona de la siguiente manera. Dada una lista:

1. Encuentra el elemento mínimo en la lista y lo intercambia con el elemento de la primera posición (su lugar correcto).

2. Llama recursivamente a ordenar el resto de la lista, excluyendo el primer elemento.

Es un algoritmo muy simple, pudiéndose plantear de manera recursiva de la siguiente manera:[3]

```
función ordenación_por_selección(v):
    si |v| = 1:
        devuelve
    índice_mínimo := 1
    para i:=1 hasta |v|:
        si v[i] < v[índice_mínimo]:
```

[3]La implementación típica de este algoritmo no es recursiva sino iterativa (con otro bucle exterior). Sin embargo, usamos aquí la alternativa recursiva para ejemplificar el cálculo de complejidades de este tipo de algoritmos.

```
            índice_mínimo := i
    intercambiar(v[1], v[índice_mínimo])
    ordenación_por_selección(v[2:])
```

Para analizar su complejidad, identificamos primero que el algoritmo de ordenación por selección no tiene un mejor ni peor caso, ya que su complejidad temporal no depende del orden inicial de los elementos en la lista. Siempre ejecuta el mismo número de comparaciones y asignaciones sin importar la entrada, sino que únicamente depende de la talla.

Si representamos la complejidad temporal del algoritmo para una lista de n elementos como $T(n)$, podemos descomponer el proceso de la siguiente manera:

1. Encontrar el mínimo en una lista de n elementos requiere $n-1$ comparaciones. Por tanto, este paso pertenece a $\mathcal{O}(n)$.

2. Intercambiar el mínimo con el primer elemento toma un tiempo constante ($\mathcal{O}(1)$).

3. Ordenar el resto de la lista de $n-1$ elementos tiene una complejidad dada por la complejidad de la llamada recursiva; es decir, $T(n-1)$.

En este caso, podemos expresar la relación de recurrencia como:

$$T(n) = \begin{cases} \mathcal{O}(1) & \text{si } n \leq 1 \\ (n-1) + T(n-1) & \text{si } n > 1 \end{cases} \tag{2.2}$$

Al igual que en el ejemplo anterior, podemos resolver esta relación de recurrencia con el método de expansión:

$$\begin{aligned} T(n) &= (n-1) + T(n-1) \\ T(n) &= (n-1) + (n-2) + T(n-2) \\ T(n) &= (n-1) + (n-2) + (n-3) + \cdots + 1 + T(1) \end{aligned}$$

Dado que $T(1)$ tiene complejidad constante ($\mathcal{O}(1)$), podemos expresar la suma de la siguiente manera:

$$T(n) = (n-1) + (n-2) + (n-3) + \cdots + 1 = \sum_{i=1}^{n-1} i$$

Esta suma aritmética corresponde a:

$$T(n) = \frac{(n-1)n}{2}$$

Por lo tanto, la complejidad temporal del algoritmo de ordenación por selección $T(n) \in \mathcal{O}(n^2)$. O en otras palabras, el tiempo de ejecución del algoritmo de ordenación por selección crece de manera cuadrática en función del número de elementos en la lista.

Apéndice A

Funciones Big O

Vamos a realizar demostraciones sencillas utilizando la definición formal de la notación Big O para dos de los ejemplos del Capítulo 2, especificando las constantes c y n_0 y mostrando cómo se aplican para demostrar que cada función pertenece a su respectiva clase de complejidad.

A.1. Ejemplo 1

- Complejidad lineal $T(n) = 2n + 3$
- **Afirmación:** $T(n) \in \mathcal{O}(n)$

Para demostrar que $T(n) \in \mathcal{O}(n)$, necesitamos encontrar constantes c y n_0 tales que:

$$0 \leq f(n) \leq c \cdot n \text{ para todo } n \geq n_0.$$

Tomemos $c = 5$ y $n_0 = 1$. Entonces para $n \geq n_0$:

$$T(n) = 2n + 3 \leq 2n + 3n = 5n.$$

Por lo tanto, tenemos que:

$$0 \leq 2n + 3 \leq 5n \text{ para todo } n \geq 1.$$

Esto muestra que $T(n) \in \mathcal{O}(n)$ con $c = 5$ y $n_0 = 1$.

A.2. Ejemplo 2

- Complejidad cúbica $T(n) = 2n^3 + n + 3$
- **Afirmación:** $T(n) \in \mathcal{O}(n^3)$

Para demostrar que $T(n) \in \mathcal{O}(n^3)$, necesitamos encontrar constantes c y n_0 tales que:

$$0 \leq f(n) \leq c \cdot n^3 \text{ para todo } n \geq n_0.$$

Tomemos $c = 4$ y $n_0 = 1$. Entonces para $n \geq n_0$:

$$f(n) = 2n^3 + n + 3 \leq 2n^3 + n^3 + n^3 = 4n^3.$$

Por lo tanto, tenemos que:

$$0 \leq 2n^3 + n + 3 \leq 4n^3 \text{ para todo } n \geq 1.$$

Esto muestra que $f(n) \in \mathcal{O}(n^3)$ con $c = 4$ y $n_0 = 1$.

A.2.1. Ejemplo 3

- Complejidad exponencial $T(n) = 2^n + n^2$
- **Afirmación:** $T(n) \in \mathcal{O}(2^n)$

Vamos a realizar la demostración de que una función exponencial con un aditivo cuadrático pertenece a la clase de complejidad $\mathcal{O}(2^n)$.

Para demostrar que $T(n)$ es $\mathcal{O}(2^n)$, necesitamos encontrar constantes c y n_0 tales que:

$$0 \leq f(n) \leq c \cdot 2^n \text{ para todo } n \geq n_0.$$

Observamos que n^2 crece mucho más lentamente que 2^n cuando n es grande. Para hacer esto más formal, podemos considerar cómo se comportan los términos cuando n aumenta:

- 2^n crece exponencialmente,
- n^2 crece polinomialmente.

Elegiremos $c = 3$ y $n_0 = 10$. Esto es algo arbitrario, pero son valores que van a hacer que claramente el término exponencial domine.

Para $n \geq n_0 = 10$, necesitamos mostrar que:

$$2^n + n^2 \leq 3 \cdot 2^n$$

Dividimos todo por 2^n:

$$1 + \frac{n^2}{2^n} \leq 3$$

El término $\frac{n^2}{2^n}$ tiende a cero a medida que n aumenta debido al crecimiento exponencialmente más rápido de 2^n comparado con el crecimiento polinomial de n^2. Específicamente, para $n = 10$, el cálculo muestra:

$$\frac{100}{2^{10}} \approx 0{,}0977$$

Para valores mayores de n, este término será aún menor, asegurando que:

$$1 + \frac{n^2}{2^n} \leq 3$$

Así queda ejemplificado que para $n \geq 10$, $f(n) = 2^n + n^2$ se mantiene por debajo de $3 \cdot 2^n$, confirmando que $f(n)$ es $\mathcal{O}(2^n)$ con $c = 3$ y $n_0 = 10$.