Programación 2

Tecnicatura Universitaria en Inteligencia Artificial

2022

Unidad 3

1. ¿Qué es un algoritmo?

La palabra algoritmo proviene del matemático persa *Al-Khwarizm* quien vivió cerca del año 825 después de Cristo. Este matemático escribió un tratado sobre aritmética y álgebra. A partir de entonces se empezó a utilizar la palabra *algorism* para describir los procesos de realización de cálculos utilizando los números arábigos, que fue el tema central del libro de *Al-Khwarizm* .

De allí evolucionó la palabra algoritmo, que usamos actualmente para referirnos a una forma de describir el proceso de resolución de un problema, explicando paso a paso como se debe proceder para llegar a una respuesta encuadrada dentro de los datos disponibles, en un tiempo finito.

Definir exactamente que es un algoritmo no es fácil, y de hecho es un tema que preocupó a los grandes matemáticos del siglo XX, como Alan Turing o Alonzo Church. Sin embargo, sí existe consenso sobre una serie de características que debe cumplir un procedimiento para ser calificado de algoritmo:

- Debe ser **correcto**, es decir, debe resolver efectivamente todas las instancias del problema para el que fue propuesto como solución.
- Debe ser **preciso**, es decir, debe indicar claramente, paso a paso y sin ambigüedades como resolver el problema.
- Debe ser **exacto**, es decir, si utilizamos el mismo algoritmo dos veces con los mismos datos de entrada, las dos veces nos debería dar el mismo resultado.
- Debe ser **finito**, es decir, no aceptamos como algoritmos secuencias de acciones que no tengan un final claro, dado que nunca podríamos cumplirlas en un tiempo finito.

En líneas generales, existen muchos algoritmos, posiblemente infinita cantidad de algoritmos, que resuelven una misma tarea. El objetivo de esta unidad es presentar el problema del ordenamiento y utilizarlo para demostrar como distintos algoritmos pueden resolverlo, así como también adentrarnos en las consecuencias teóricas y prácticas que puede traer el uso de uno u otro algoritmo. Veremos que si bien todos los algoritmos resuelven la misma tarea, cada uno tendrá ventajas y desventajas particulares que tendremos que tener en cuenta para decidirnos por uno o por otro al momento de escribir nuestros programas.

2. El problema de ordenamiento

Presentar una lista o conjunto de elementos en un orden determinado, por ejemplo, alfabético o numérico, ascendente o descendente, es un problema que aparece con gran frecuencia en las aplicaciones de la Informática. Es un problema ampliamente estudiado dado que a pesar de que es relativamente fácil de formular sirve de introducción a los conceptos generales de algoritmia. En la vida cotidiana, muchas veces nos disponemos a colocar objetos de una manera especial, siguiendo algún criterio, es decir, los ordenamos. Apilamos un conjunto de platos por tamaño, acomodamos varias carpetas por colores, clasificamos un conjunto de palabras por orden alfabético, acomodamos un mazo de cartas por palos o por números. En síntesis, clasificamos un conjunto finito de acuerdo con un criterio prefijado. Generalmente no nos detenemos a pensar cómo hacemos esto, pero para poder darle estas mismas órdenes a una computadora es fundamental que lo conceptualicemos.

Notemos además que ordenar un conjunto de datos es en principio más difícil que chequear si ya esta ordenado. Pensemos en ordenar un mazo de cartas, primero según su número y luego según su palo. Chequear si están ordenadas es fácil, simplemente revisamos, carta a carta, que estén en el lugar correcto. Ahora bien, si tenemos que ordenarlas, puede no ser claro como arrancar. Vamos a introducir algunas restricciones al problema sin perder generalidad:

- En las secciones que siguen, asumiremos siempre que estamos ordenando una lista de números enteros en Python. Extender las ideas para otros tipos de datos es trivial en Python, siempre que los elementos provean los métodos mágicos necesarios para ser ordenables. Además, las ideas que subyacen a estos algoritmos son transversales a todos los lenguajes de programación.

- Asumiremos que estamos ordenando siempre de menor a mayor. Ordenar de mayor a menor es trivial cambiando el sentido de todas las desigualdades, o lo que es lo mismo, ordenar de menor a mayor y luego listar los elementos en orden inverso.

- Asumiremos, por simplicidad, que todos los elementos que estamos ordenando son distintos, es decir, que no hay elementos duplicados o repetidos.

Basados en estas presunciones, presentaremos algunos métodos de ordenamiento.

Un primer procedimiento podría ser mezclar el mazo como lo haríamos para jugar algún juego y luego chequear si están ordenadas. Si no están ordenadas, repetimos el procedimiento hasta que lo estén. De más esta decir que realizar este procedimiento y esperar tener éxito es de ingenuos. De hecho este procedimiento es tan malo, que ni siquiera podemos calificarlo como algoritmo, ya que no cumple la condición de finitud: podríamos repetir el proceso indefinidamente con la mala suerte de caer siempre en un orden incorrecto.

Otra posibilidad para ordenar el mazo de cartas es probar todos los posibles órdenes que haya. Pensemos que la distribuimos sobre una mesa, chequeamos si están en orden y si no lo están, intercambiamos las dos últimas cartas. Chequeamos si están en orden y si no lo están, intercambiamos la última carta con la penúltima. Así seguimos, intercambiando en cada paso las últimas dos cartas que no hayamos intercambiado entre sí. Ahora si estamos propiamente frente a un algoritmo, lo podemos describir de forma concisa y se puede demostrar que eventualmente nos llevará al orden correcto. Sin embargo, no es de sorprenderse que este algoritmo será considerado pésimo, si intentáramos ordenar las cartas con este método, probablemente no terminaríamos en unos cuantos billones de años. [1]

Esto nos da una primera cuestión a la que vamos a prestar atención cuando estemos analizando algoritmos, y es el tiempo que nos llevará ejecutarlos. Como esto generalmente depende del procesador en que lo ejecutemos [2], lo que se hace es buscar una medida que represente ese tiempo, pero que sea independiente de

[1] Para quienes estén interesados en seguir las cuentas, primero notemos que pensar este algoritmo es equivalente a pensar cuantas formas distintas existen de que esté mezclado un mazo. Luego, se puede leer la nota del matemático y divulgador Adrián Paenza al respecto. https://www.pagina12.com.ar/diario/contratapa/13-274866-2015-06-14.html

[2] No es lo mismo ejecutar el mismo algoritmo codificado en lenguaje C que en lenguaje Python, como tampoco es lo mismo hacerlo en un procesador Intel Core i7, lanzado en 2022, que en un procesador Intel 4004, lanzado en 1971.

la máquina y del lenguaje específico que estemos utilizando. En el caso particular de algoritmos de ordenamiento, utilizaremos la cantidad de comparaciones necesarias para ordenar el conjunto como una medida del tiempo necesario para ejecutar el algoritmo. Este número es en general una función del número de elementos que hay que ordenar. A este concepto lo llamaremos **complejidad temporal del algoritmo**.

3. Ordenamiento por selección

Dada una lista de n números enteros, el método de selección para ordenarlo en forma ascendente, es el siguiente:

1. encontrar el menor valor de la lista (seleccionar el valor mínimo de la lista).
2. intercambiar el elemento encontrado con el primero de la lista.
3. repetir estas operaciones con los $n - 1$ elementos restantes, seleccionando, el segundo elemento, así sucesivamente.

Veamos como funciona este algoritmo aplicándolo a una lista pequeña.

```
A = [5, 8, 7, 1, 3, 6]
```

Lo primero que hacemos es identificar el menor elemento de la lista. En este caso, el valor 1. Luego, lo intercambiamos con el primer elemento de la lista. Nos queda

```
A = [1, 8, 7, 5, 3, 6]
```

Una vez que completamos este paso, el primer elemento de la lista ya está ordenando, en el sentido de que es menor que todos los que le siguen. Entonces el problema se reduce a ordenar los elementos restantes.

El menor elemento entre los restantes es el 3. Lo intercambiamos entonces con el segundo elemento de la lista y nos queda:

```
A = [1, 3, 7, 5, 8, 6]
```

Una vez que completamos este paso, la sublista formada por los dos primeros elementos está ordenada, seguimos trabajando con los elementos restantes.

Dentro de los elementos restantes, el menor elemento es el 5. Lo intercambiamos entonces con el tercer elemento de la lista.

```
A = [1, 3, 5, 7, 8, 6]
```

En la siguiente iteración, el mínimo valor que no ha sido ordenado aún es el 6.

```
A = [1, 3, 5, 6, 8, 7]
```

Finalmente, el mínimo valor que no ha sido ordenado aùn es el 7.

```
A = [1, 3, 5, 6, 7, 8]
```

Una vez que completamos este paso, la lista quedó ordenada. No es necesario hacer el último paso, ya que si todos los elementos anteriores están en el orden correcto, el último número también habrá quedado forzosamente ordenado.

El proceso puede verse ilustrado en la Figura 1.

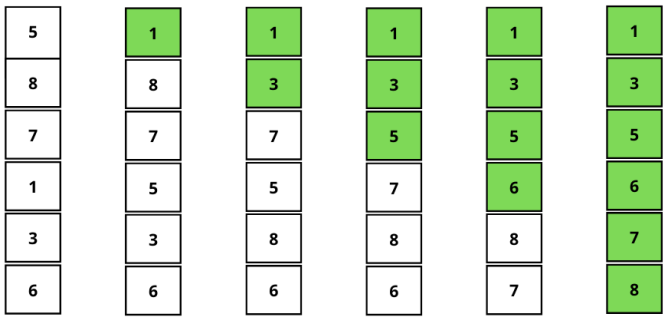

Figura 1: Ejemplo de ordenamiento por selección

3.1. Implementación en Python

```
def ordenar_por_seleccion(A):
    # Guardamos en n la cantidad de elementos a ordenar
    n = len(A)
    # En esta variable guardaremos el mínimo encontrado en cada iteración.
    min = None
    # En esta variable guardaremos la posición donde se encuentra el mínimo.
    pos = None
    for i in range(n - 1):
        # Ordenamos el elemento en la posición i
        min = A[i]
        pos = i
        for j in range(i, n):
            if A[j] < min:
                # Actualizamos los valores de min y pos
                min = A[j]
                pos = j
        # Intercambiamos los valores en pos y en i
        A[pos], A[i] = A[i], A[pos]
```

4. Ordenamiento por inserción

Dada una lista de n números enteros, el método de inserción para ordenarlo en forma ascendente, es el siguiente:

1. Comenzamos considerando la sub-lista formado solo por el primer elemento. Esta lista esta trivialmente ordenada.

2. Consideramos el siguiente elemento de la lista. En este momento, solo hay dos posibilidades, dado que por simplicidad suponemos que no hay elementos repetidos:

   - El elemento que vemos es menor que el primer elemento, y por lo tanto lo ubicamos a la izquierda del mismo.

   - El elemento que vemos es mayor que el primer elemento, y por lo tanto lo ubicamos a la derecha del mismo.

3. Una vez realizado este, tenemos una sub-lista de tamaño dos ordenado. Repetimos para toda la secuencia, en cada paso consideramos el siguiente elemento de la lista, y decidimos en que posición tenemos que *insertarlo*

Veamos como funciona este algoritmo aplicándolo a una lista pequeña.

```
A = [8, 6, -1, 2]
```

Empezamos considerando la sublista A[:1], es decir, [8]. Esta lista esta trivialmente ordenada.
Ahora consideramos el elemento 6. Para conservar la propiedad de orden, ¿en que posición debemos insertarlo? Para conservar la propiedad de orden entre los elementos que ya vimos, la ubicamos antes que el 8. Ahora tenemos una lista ordenada de tamaño 2, A[:2], es decir, [6, 8]
El siguiente elemento de la lista es -1. Para conservar la propiedad de orden, ¿en que posición debemos insertarlo? Para conservar la propiedad de orden entre los elementos que ya vimos, lo ubicamos en la posición 0. Ahora tenemos una lista ordenada de tamaño 3, A[:3], es decir, [-1, 6, 8]
El último elemento de la lista es -. Para conservar la propiedad de orden, ¿en que posición debemos insertarlo? Para conservar la propiedad de orden entre los elementos que ya vimos, lo ubicamos en la posición 1, es decir, entre los números -1 y 6. Ahora tenemos una lista ordenada de tamaño 4, A[:3], es decir, [-1, 2, 6, 8]. Como ya hemos terminado de recorrer la lista, hemos terminado.

4.1. Implementación en Python

```
def ordenar_por_insercion(A):
    # Guardamos en n la cantidad de elementos a ordenar
    n = len(A)
    for i in range(1, n):
        # Insertamos el elemento en la posición i
        value = A[i]
        j = i - 1
        while j >= 0:
            if value < A[j]:
                # Desplazamos el elemento en la posicion j
                # un lugar a la izquierda
                A[j+1] = A[j]
                j = j - 1
            else:
                break
        #Almacenamos el valor actual en el lugar que nos quedo disponible
        A[j+1] = value
```

5. Ordenamiento Quicksort

Hasta ahora estuvimos viendo algoritmos iterativos para ordenar números enteros. Si bien estos algoritmos también admiten una versión recursiva, se presentan naturalmente como algoritmos iterativos.
Veremos ahora un algoritmo para ordenar que se presenta naturalmente como un algoritmo recursivo.
Este algoritmo comienza eligiendo un elemento más o menos arbitrario de la lista, al que llamaremos *pivote*. Una vez elegido, podemos dividir los elementos restantes de la lista en dos grupos: los elementos menores al pivote y los elementos mayores al pivote.
Luego, podemos ordenar recursivamente cada uno de estos grupos y utilizar esas sublistas ordenadas para tener la lista completa ordenada: primero pondremos el resultado de ordenar la lista compuesta por los números menores que el pivote, luego el pivote mismo, y completamos con los elementos mayores al pivote.
El caso base de este algoritmo recursivo serán las listas de cero o un elementos, las cuales están trivialmente ordenadas.
Veamos como funciona este algoritmo aplicándolo a una lista pequeña. Supondremos que siempre elegimos como pivote el primer elemento de la lista.

A = [5, 8, 7, 1, 3, 6]

Elegimos como elemento pivote el 5. Luego, podemos separar la lista en 3 secciones: los números menores que 5, el 5 mismo, y los números mayores a 5.

A = [1, 3, 5, 8, 7, 6]

Ahora, si suponemos que podemos ordenar recursivamente cada una de las sublistas, obtenemos la lista ordenada

A = [1, 3, 5, 6, 7, 8]

Por supuesto, para ordenar cada una de estas listas se sigue el mismo procedimiento, se elige un pivote, se separan los elementos menores y los mayores al pivote y se ordenan recursivamente. El proceso puede verse ilustrado en la Figura 2.

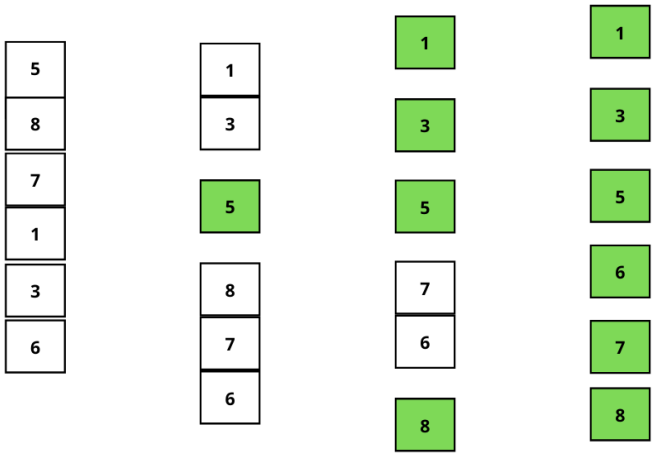

Figura 2: Ejemplo de ordenamiento rápido

El único caso en que no elegiremos pivote es para las listas de cero o un elemento, ya que estas son nuestro caso base: siempre están ordenadas.

Este algoritmo se llama **ordenamiento rápido** o Quicksort y ya veremos mas adelante que hace honor a su nombre. Fue creado por el científico Charles Hoare en la década de los cincuenta.

5.1. Técnicas de elección del pivote

El algoritmo quicksort en no especifica como elegir el elemento pivote, sin embargo dicha elección es crucial para que se preservan ciertas propiedades del algoritmo. En particular, la elección del pivote tendrá consecuencias prácticas a la hora de analizar la eficiencia del algoritmo.

Tomar un elemento cualquiera de la lista, como puede ser el primero o el último, tiene como ventaja que no se requiere realizar ningún cálculo adicional. Sin embargo, esto nos puede causar muchas llamadas recursivas con una lista vacía, si el pivote elegido resulta el elemento mayor o menor de toda la lista. Esto tiene un especial impacto considerando que se ha mostrado empíricamente que, en la práctica, muchas veces las listas vienen casi ordenadas, donde los primeros y los últimos valores de la lista ya están ordenados y solo hay

desorden en el centro. Esta técnica se conoce como **elección a ciegas**.

Un posible paliativo para este problema es usar un módulo generador de números pseudo-aleatorios (como el módulo `random` de python) para elegir un valor al azar de la lista. Sin embargo seguiremos teniendo problemas para ciertas permutaciones de la lista. Esta técnica la llamamos **elección al azar**.

En el mejor caso, el pivote termina en el centro de la lista, dividiéndola en dos sublistas de igual tamaño. Así, puede ser una opción recorrer la lista para saber de antemano cuál será el elemento que ocupará la posición central de la lista, para elegirlo como pivote. Esto puede hacerse y nos asegura que el algoritmo se comportará bastante bien en cualquier permutación de la lista. No obstante, el cálculo adicional es complicado de implementar eficientemente y rebaja bastante la eficiencia para el caso promedio. Esta técnica la llamamos **elección calculada**.

Una opción a medio camino, que combina bastante bien los beneficios de estos enfoques, es la llamada **elección de mejor de tres**. Se toman el primer elemento de la lista, el segundo y el último. Se elige como pivote el número que ocupe la posición central entre estos tres. Esto nos asegura que, por lo menos, no haremos llamadas recursivas innecesarias con listas de tamaño cero y es mucho más eficiente que calcular la mediana de todos los elementos de la lista, sobre todo para listas grandes.

5.2. Implementación en Python

La implementación que daremos usa la técnica de elección de pivote a ciegas.

```
def elegir_pivote(A):
    # Tecnica de elección  a  ciegas , elegimos siempre el
    # primer elemento de la sublista a ordenar
    return A[0]

def quicksort(A):
    if len(A) < 2:
        # Caso base: listas trivialmente ordenadas
        return A

    # Elegimos un pivote
    pivote = elegir_pivote(A)
    # Usamos listas por comprensión para
    # obtener as sublistas que debemos ordenar
    menores = [x for x in A if x < pivote]
    mayores = [x for x in A if x > pivote]
    # Llamamos recursivamente a cada lado
    menores_ordenados = quicksort(menores)
    mayores_ordenados = quicksort(mayores)
    # Construimos la respuesta
    return menores_ordenados + [pivote] + mayores_ordenados
```

6. Cantidad de comparaciones y operaciones efectuadas

Si la cantidad de datos a ordenar es grande, no todos los algoritmos de ordenamiento trabajan con igual eficiencia. Un análisis de las comparaciones y operaciones efectuadas, nos detallará estas características. En las secciones siguientes, analizaremos la eficiencia de los distintos algoritmos considerando que estamos ordenando una lista con n elementos.

6.1. Eficiencia del ordenamiento por selección

En el algoritmo de ordenamiento por selección hay dos estructuras de iteración anidadas, cada una de las cuales recorre una cantidad de elementos relacionada con el tamaño de lista. Esto nos lleva a pensar, en un primer examen, que la cantidad de operaciones que realiza el algoritmo tiene que estará relacionada con n^2. Confirmaremos esto con el siguiente razonamiento informal.
En cada paso del algoritmo tenemos que elegir el valor mínimo entre los que aún no hemos visto. Es decir, en el paso i, tendremos que realizar $n - i$ comparaciones para encontrar el valor mínimo. Realizaremos esta operación para cada valor de i entre 0 y n - 1. La cantidad total de comparaciones que estaremos realizando es:

$$\sum_{i=0}^{n-1} n - i = (n - 0) + (n - 1) + ... + (n - (n - 1) = \frac{n(n-1)}{2} = \frac{n^2 - n}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$$

Aquí estamos utilizando la famosa *suma de Gauss* para calcular la suma de los primeros n números naturales. El resultado nos da un polinomio de grado 2, lo que confirma nuestra intuición.
Es interesante remarcar que la cantidad de comparaciones no depende ni de que números estemos ordenando ni de en que orden se encuentran originalmente, siempre se realiza el mismo número de operaciones.
El número de intercambios que se realiza también esta fijo. Se utilizan exactamente $n - 1$ intercambios, puesto que la operación siempre se ejecuta, aún cuando el valor mínimo ya este ubicado correctamente.

6.2. Eficiencia del ordenamiento por inserción

El argumento es similar al de ordenamiento por selección. Se dejan los detalles como ejercicio.

6.3. Eficiencia de quicksort

En principio, al tener una versión recursiva, no resulta inmediato tener una intuición sobre la cantidad de comparaciones que se estará haciendo, por lo que usaremos una técnica conocida como **árbol de recursión**. La eficiencia, como ya adelantamos, esto depende de la elección del pivote. Para simplificar las cuentas, supondremos que $n = 2^k$, es decir, que es una potencia de 2.
En el **mejor de los casos**, el pivote será el punto medio, lo que ocasionará $2n$ operaciones de comparación para armar las sublistas y dos llamadas recursivas con listas de tamaño $\frac{n}{2}$ Las llamadas recursivas pueden acomodarse en un árbol binario. En el primer nivel tenemos entonces las $2n$ comparaciones para armar las listas. En el segundo nivel, tendremos dos llamadas recursivas de tamaño $\frac{n}{2}$. En el siguiente nivel tenemos cuatro llamadas recursivas de tamaño $\frac{n}{4}$. Y así sucesivamente, el número de comparaciones se mantiene constante en cada nivel del árbol: $2n$. El árbol tiene k niveles, por lo tanto, el número total de comparaciones es $k * 2n$, o, expresando todo en términos de n nos queda $2n \log n$. En la figura 3 podemos visualizar este razonamiento.

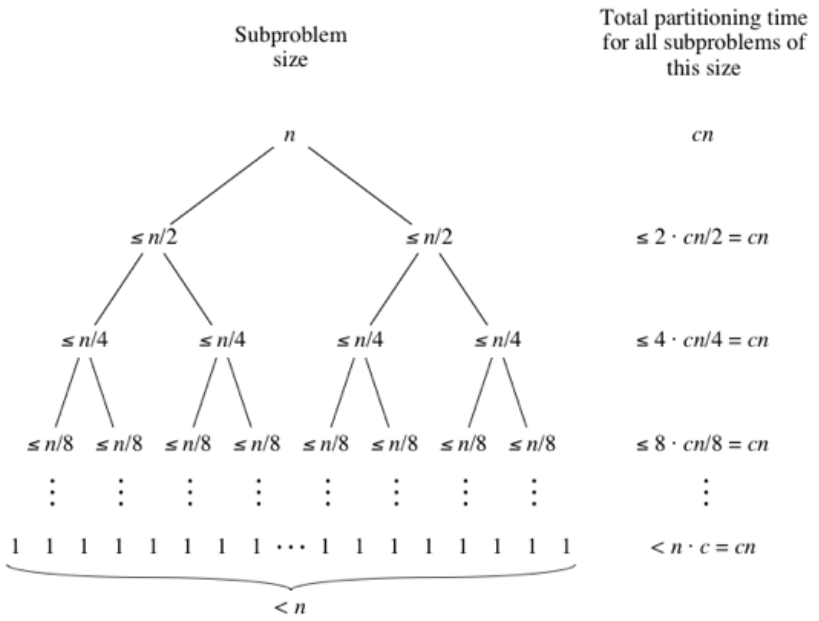

Figura 3: Árbol de recursión para el mejor caso de Quicksort

En el **peor de los casos**, el pivote siempre es el elemento menor o mayor de la lista. Asumiremos aquí que es siempre el menor, el otro caso es análogo. En este caso tenemos $2n$ operaciones de comparación para armar las sublistas y una llamada recursiva con una lista de tamaño $n-1$, y una llamada recursiva con una lista de tamaño 0. El árbol de recursión queda muy desbalanceado. En el primer nivel tenemos entonces las $2n$ comparaciones para armar las listas. En el segundo nivel una hoja y una llamada recursiva de tamaño $n-2$. En el siguiente nivel tendremos nuevamente una hoja y una llamada recursiva de tamaño $n-2$. Y así sucesivamente. Podemos ver esto ilustrado en la figura 4. el número de comparaciones en cada nivel del árbol va disminuyendo en 2, empezamos con $2n$, al siguiente nivel hacemos $2-n2$ y así sucesivamente. Este árbol tendrá n niveles, por lo tanto, el número total de comparaciones es

$$\sum_{i=0}^{n} 2n - 2i = 2\sum_{i=0}^{n} n - i = 2\frac{n(n-1)}{2} = 2\frac{n^2-n}{2} = n^2 - n$$

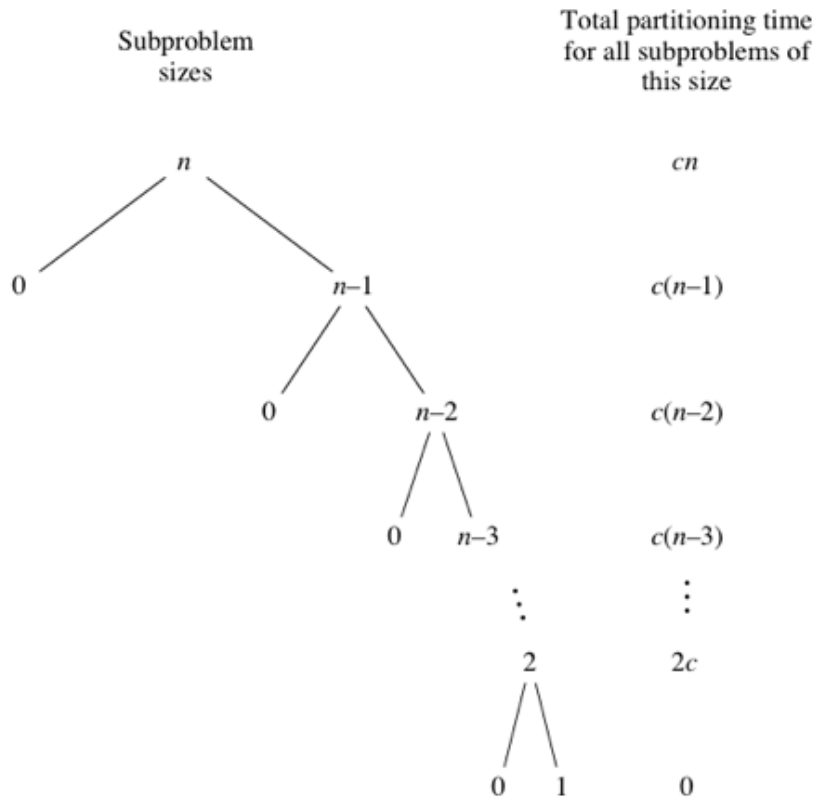

Figura 4: Árbol de recursión para el peor caso de Quicksort

El análisis para la eficiencia de Quicksort en la mayoría de los casos, lo que se conoce como el **caso promedio**, involucra matemáticas avanzadas, por lo que no ahondaremos en él. El número de comparaciones requerido para el caso promedio será $cn \log n$ donde la constante c dependerá de can tan bien los pivotes particionen la lista original.

Referencias

[1] Cormen T. H. et al, 2009. *Introduction to Algorithms*. 3era Edición. The MIT Press. Parte II, en especial el capitulo 7

[2] Wirth N., 1985. *Algorithms and Data Structures*. Capítulo 2

[3] Gagliano M. G. et al, 2014 *Elementos Escenciales para la programación: Algoritmos y estructuras de datos*. Proyecto LatIn. Capítulo 7

Las imagenes 3 y 4 fueron extraídas del sitio khanacademy.org .