UNPAZ

Universidad Nacional de José C. Paz

Algoritmos y Programación

Profesores:
Gustavo Funes
Rómulo Arceri

Licenciatura en Gestión de Tecnología de la Información

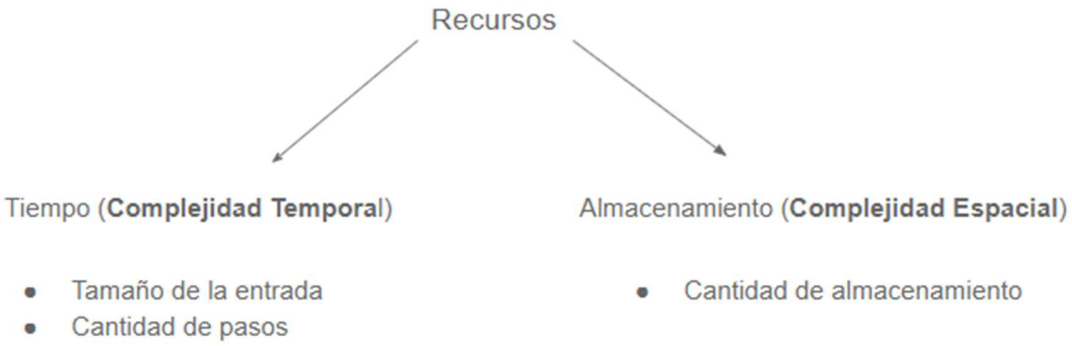
Clase 8

Contenido

Análisis de Algoritmos.

Definición

El análisis de algoritmos se refiere al proceso de encontrar la complejidad computacional de un algoritmo que resuelva un problema computacional dado, con el objetivo de proveer estimaciones teóricas de los recursos que necesita.



Los recursos a los cuales se hace referencia son el tiempo (complejidad temporal) y el almacenamiento (complejidad espacial).

La **complejidad temporal** involucra determinar una función que relaciona la longitud o el tamaño de la entrada del algoritmo con el número de pasos que realiza.

La **complejidad espacial** busca la cantidad de ubicaciones de almacenamiento que utiliza.

Como estudiar la **complejidad espacial** depende de muchos factores específicos para cada caso nos centraremos en estudiar la **complejidad temporal**.

Distintos algoritmos pueden utilizarse para resolver un mismo problema y a su vez los algoritmos pueden estudiarse de forma independiente del lenguaje de programación a utilizar y de la máquina donde se ejecutará. Esto significa que se necesitan técnicas que permitan comparar la eficiencia de los algoritmos antes de su implementación.

Pruebas de referencia

Una forma de medir el tiempo de ejecución de una función es hacer un análisis de pruebas de referencia (**benchmark**), medir el tiempo real requerido.

En Python, podemos hacer una prueba de referencia de una función observando el tiempo de inicio y el tiempo de finalización con respecto al sistema que estamos utilizando.

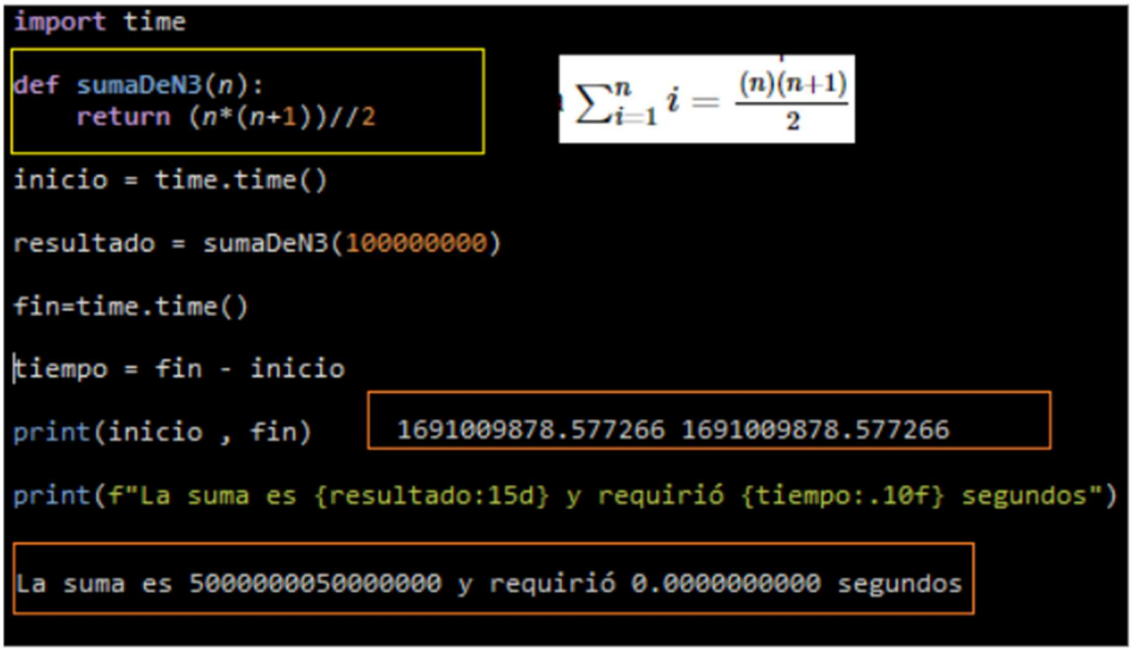
En el módulo **time** hay una función llamada **time** que devolverá el tiempo actual del reloj del sistema medido en segundos desde algún punto de inicio arbitrario. Al llamar a esta función dos veces, al inicio y al final, y luego calcular la diferencia, podemos obtener un número exacto de segundos (fracciones en la mayoría de los casos) de la ejecución.

La siguiente función ***sumaDeN2()*** realiza la acumulación de la suma desde 0 hasta n, calcula la suma de los primeros n enteros. El algoritmo utiliza la idea de una variable acumuladora ***LaSuma*** que se inicializa en 0. La solución itera entonces a través de los n enteros, agregando cada uno a la variable acumuladora.

```
import time

def sumaDeN2(n):
   inicio = time.time()

   laSuma = 0
   for i in range(1,n+1):
      laSuma = laSuma + i

   final = time.time()

   return laSuma,final-inicio

resultado = sumaDeN2(10000)
print(f"La suma es {resultado[0]:15d} y requirió {resultado[1]:.10f} segundos")
resultado = sumaDeN2(100000)
print(f"La suma es {resultado[0]:15d} y requirió {resultado[1]:.10f} segundos")
resultado = sumaDeN2(1000000)
print(f"La suma es {resultado[0]:15d} y requirió {resultado[1]:.10f} segundos")
resultado = sumaDeN2(10000000)
print(f"La suma es {resultado[0]:15d} y requirió {resultado[1]:.10f} segundos")
```

```
La suma es        50005000 y requirió 0.0013728142 segundos
La suma es      5000050000 y requirió 0.0108773708 segundos
La suma es    500000500000 y requirió 0.1193704605 segundos
La suma es  50000005000000 y requirió 1.1267480850 segundos
```

Ahora observemos la siguiente función ***sumaDeN3(n)***, que a primera vista puede parecer extraña, pero esta función está haciendo esencialmente lo mismo que la anterior.



Los dos algoritmos resuelven el problema, pero vean la diferencia de tiempo que tarda el primero y el segundo. El segundo resolvió el cálculo de cien millones en 0 segundos. mientras que el primer algoritmo tardó más de 1 segundo en resolver el cálculo de 10 millones.

¿Cuál es la diferencia más notable?

Podemos ver que las soluciones iterativas parecen estar haciendo más trabajo ya que algunos pasos del programa se están repitiendo. El tiempo requerido para la solución iterativa parece aumentar a medida que aumentamos el valor de n.

Sin embargo, hay un problema. Si ejecutamos la misma función en una computadora diferente o usamos un lenguaje de programación diferente, es probable que obtengamos resultados diferentes. Podría tomar aún más tiempo ejecutar el segundo algoritmo si la computadora fuera más antigua.

Necesitamos una mejor manera de describir o representar estos algoritmos con respecto al tiempo de ejecución. La técnica de pruebas de referencia calcula el tiempo de ejecución real. Esa técnica en verdad no nos proporciona una medida útil, ya que depende de una máquina, programa, hora del día, compilador y lenguaje de programación en particular.

Como verán esta técnica depende de muchos factores externos, para independizarnos de estos factores existen otras técnicas de medición. como la **Notación O-grande.**

Entonces para saber cuánto va a tardar en procesar todos los documentos, cuántas transacciones por segundo soporta nuestra aplicación, cuántos usuarios podemos manejar, qué tanta memoria ocupa, cuántos servidores vamos a necesitar para soportar este tráfico y para que nosotros nos demos cuenta cómo se comporta un algoritmo no tenemos por qué siempre ejecutarlos para ver el resultado, muchas veces podemos analizarlos y determinar su complejidad utilizando la **anotación de Big O**.

Esta medición nos resultará muy útil para comparar la eficiencia de dos algoritmos, por ejemplo, de ordenamiento. Es válido para cualquier lenguaje de programación.

El tiempo de ejecución de un algoritmo depende de la entrada. Vemos que el comportamiento podría crecer de manera diferente y para poder clasificar todas estas diferentes formas en que crecen los algoritmos dependiendo de la entrada, usamos la **notación asintótica** *o* ***notación «O grande»***

Notación O Grande (Big-O)

En ciencias de la computación se conoce como **notación asintótica** o **notación** «**O grande**» a la forma de describir el crecimiento de los requerimientos de recursos (ciclos de procesamiento, tiempo, memoria, almacenamiento) de un algoritmo respecto del tamaño de los datos de entrada, o más explícitamente respecto del tamaño del trabajo a procesar.

La Notación Big-O es una forma de medir el tiempo, como escala un programa o un algoritmo y el tiempo que tardará en ejecutar.

O(1)

¿Cómo podemos analizar nuestro código?

Para determinar cuál es su complejidad, tenemos que saber que cualquier función o línea de código se considera **Big O de 1** o **tiempo constante** siempre y cuando no sea un ciclo, no tenga recursión o no sea una llamada a una función que a su vez no sea de tiempo constante, si es el tiempo constante pues sigue siendo tiempo constante.

```
x = 10                                  # O(1)
entrada = input("Ingrese su nombre: ") # O(1)

if entrada == "juan":                   # O(1)
    print("Hola " * x)                  # O(1)
```

```
x = 10                # O(1)
secuencia = "-" * x   # O(1)

for i in secuencia: # O(1) secuencia siempre contendra 10 -
    print(i)        # O(1)
```

O(n)

Los ciclos se consideran **Big O de n** o **tiempo lineal** cuando la variable del ciclo va incrementando o decrementando por un número constante o sea que vaya subiendo de uno en uno o de dos en dos o de tres en tres, etcétera y siempre y cuando este ciclo vaya iterando en base a la entrada, que vaya recorriendo los números que nosotros le pasamos o vaya recorriendo los caracteres del string o que vaya haciendo algo con la entrada.

```
secuencia = input("Ingrese una palabra: ") # O(1)
for i in secuencia:    # O(n) n puede variar en la cantidad de elementos
    print(i)           # O(1)

vueltas = int(input("Ingrese cantidad de vueltas: ")) # O(1)
for i in range(vueltas):    # O(n) n puede variar en la cantidad de ciclos a dar no es contante
    print(i)                # O(1)
```

Pero si ese ciclo itera tres veces y siempre itera tres veces independientemente de la entrada que le pusimos se sigue considerando de tiempo constante **Big O de 1**

```
vueltas = int(input("Ingrese cantidad de vueltas: ")) # O(1)
while vueltas > 0:
    print(vueltas)
    vueltas -= 1
```

O(n^2)

Cuando tenemos ciclos anidados la complejidad depende de cuántas veces se ejecuta el ciclo que está más adentro llegando a **Big O n^2** si son dos ciclos o si son tres ciclos n^3 , n^4 , n^5… depende de cuántos ciclos anidados tengamos dentro y si todos esos ciclos también iteran basándose en la entrada o en variables que a su vez están basándose en la entrada

```
print("Mostrar todas las combinaciones posibles de un candado de 2 digitos, desde el 0 a n")  # O(1)
n = int(input("Ingresar el valor de n: "))+1 # O(1)

for i in range(n): # O(n)
    print()
    for j in range(n):  # O(n)
        print(f" {i} {j} " )

O(1) + O(1) + O(n * n) + O(1) + O(1) = O(n^2)
```

```
n = int(input("Ingresar el valor de n: ")) # O(1)
i = 0                                     # O(1)
while i < n:                              # O(n)
    j = 0                                 # O(1)
    while j < n:                          # O(n)
        j += 1                            # O(1)
        print(i , j)                      # O(1)
    i += 1                                # O(1)

# O(6) + O(n * n) = O(n^2)
```

O(log [n])

Cuando la variable del ciclo en lugar de estar incrementando por un número constante va multiplicándose o dividiéndose esta complejidad se convierte en **logaritmo de n**.

En **Big O** el enemigo del logaritmo de n es **n^2** etc, con esto ya podemos analizar algunos algoritmos

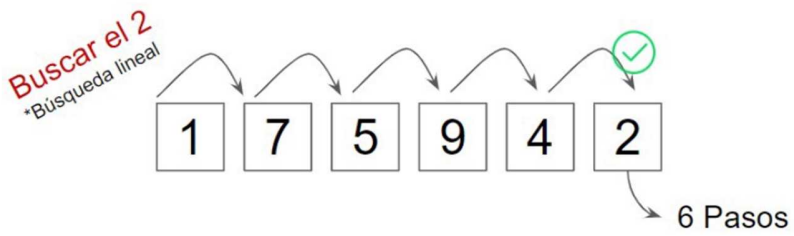
```
// Donde "n" es la entrada
for (int i = 1; i <=n; i *= c) { // O(log[n])
  // Cualquier sentencia O(1)
}
for (int i = n; i > 0; i /= c) { // O(log[n])
  // Cualquier sentencia O(1)
}
```

Por ejemplo, buscar gente en una guía telefónica es **O(log n)**. No necesitas comprobar cada persona en la guía telefónica para encontrar la correcta; en cambio, puedes simplemente dividir y conquistar, y sólo necesitas explorar una pequeña fracción de todo el espacio antes de que finalmente encuentres el número de teléfono de alguien.

Ejemplos de Notación O grande

Entonces la notación **Big O** es un sistema que nos permite comparar dos algoritmos.

Entendiendo la notación seremos capaces de analizar el mejor de los escenarios, cuando tengamos múltiples opciones para poder elegir el algoritmo más efectivo y el que va a funcionar mejor cuando tengamos una carga elevada.



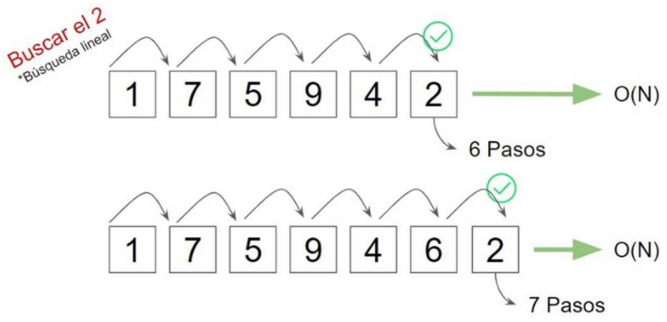
Por ejemplo, tenemos un Algoritmo de búsqueda lineal (o sea comprobar cada elemento de una lista uno por uno) ese mismo algoritmo tarda 200 pasos para un array de 200 elementos y 120 para un array de 120 elementos en el peor de los escenarios.

Cuando definimos la eficiencia de un algoritmo no lo hacemos indicando el número de pasos como tal como podría ser "un algoritmo de 200 pasos y otro de 120 pasos". Esto es debido a que el número de pasos no puede ser reducido a un número para todos los casos.

Calcular la eficiencia de un algoritmo con Big O

¿Qué significa O(N) ?:

Quiere decir que la "**complejidad = n**", donde "**n**" es el número de elementos. quiere decir que ese algoritmo va a tener un tiempo de ejecución lineal, si introduces 10 elementos, tardará 10 pasos, mientras que si introduces 100 tardará 100 pasos.



Por lo tanto, como podemos ver en la imagen, ambos procesos están en la categoría O(N) pese a uno tener un paso extra. Ya que cuando definimos **Big O** lo hacemos de forma relativa a sus datos de entrada.

Entonces **O(N)** tiene un número de pasos proporcional al número de elementos de entrada.

Algoritmos de Búsqueda

La búsqueda es el proceso algorítmico de encontrar un ítem particular en una colección de ítems.

Una búsqueda normalmente devuelve **True** o **False** según el ítem esté o no presente, respectivamente. En ocasiones, el algoritmo se puede modificar para devolver la posición donde se encuentre el ítem.

En ocasiones, el algoritmo se puede modificar para devolver la posición donde se encuentre el ítem.

Para nuestros propósitos, simplemente nos ocuparemos de la pregunta de saber si existe o no el ítem devolveremos o sea **True** o **False**

En Python, hay una manera muy fácil de preguntar si un ítem está en una lista de ítems. Utilizando el operador **in**.

Operador	Significado
in	Verdadero si el valor/variable/ítem se encuentra en la secuencia
not in	Verdadero si el valor/variable/ítem NO se encuentra en la secuencia

```
secuencia = (1,4,7,8,9,3)

print( 3 in secuencia)
print( 100 in secuencia)

if 8 in secuencia:
    print("Numero 8 encontrado")
else:
    print("Numero 8 no encontrado")
```

```
True
False

Numero 8 encontrado
```

Pero como estamos aprendiendo algoritmos y programación debemos saber y conocer que hay muchas maneras diferentes de buscar el ítem.

Algoritmo de búsqueda secuencial

Cuando los ítems de datos se almacenan en una colección, por ejemplo, en una lista, decimos que tienen una relación lineal o secuencial. Cada ítem de datos se almacena en una posición relativa a los demás. En las listas de Python, hay posiciones relativas que son los valores de los índices de los ítems individuales. Dado que estos valores de los índices están ordenados, es posible para nosotros visitarlos en secuencia. Este proceso da lugar a la búsqueda secuencial.

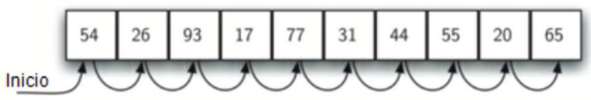


Fig 1

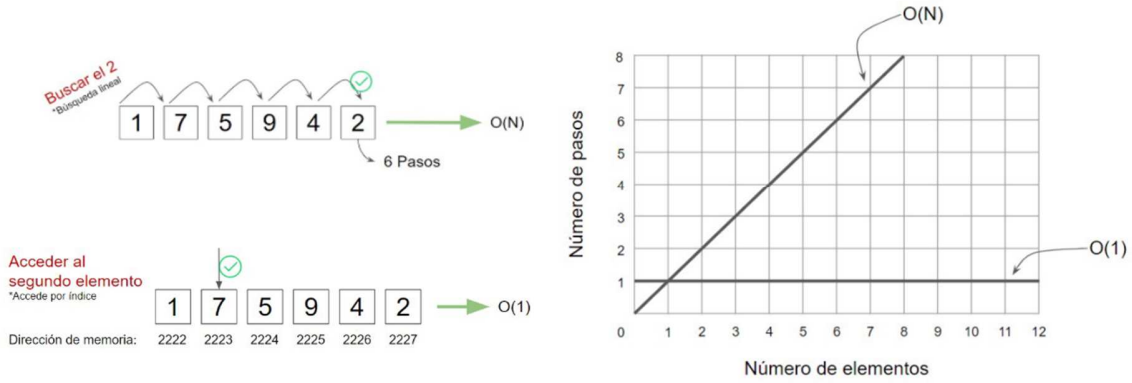
La Figura 1 muestra cómo funciona esta búsqueda. Comenzando en el primer ítem de la lista, simplemente nos trasladamos de un ítem a otro, siguiendo el orden secuencial subyacente hasta que encontremos lo que buscamos o nos quedemos sin ítems. Si nos quedamos sin ítems, hemos descubierto que el ítem que estábamos buscando no estaba presente.

```
def busquedaSecuencial(Lista, item):
    pos = 0
    while pos < len(Lista):
        if Lista[pos] == item:
            return True
        else:
            pos = pos+1

    return False

secuencia = [1, 2, 32, 8, 17, 19, 42, 13, 0]

print(busquedaSecuencial(secuencia, 3))
print(busquedaSecuencial(secuencia, 13))
```

```
False
True
```

En la implementación en Python para este algoritmo se muestra que la función necesita la lista y el ítem que estamos buscando y devuelve un valor booleano que indica si el ítem está o no presente.

Análisis de la búsqueda secuencial



Vemos que **O(1)** se utiliza cuando es un solo paso, y **O(N)** cuando es el mismo número de pasos que elementos de entrada como en el peor de los casos de la búsqueda secuencial.

Esta lista de ítems no está ordenada de ninguna manera. La probabilidad de que el ítem que estamos buscando esté en una posición determinada es exactamente la misma para cada posición de la lista.

En realidad, hay tres escenarios diferentes que pueden ocurrir. En el mejor de los casos encontraremos el ítem en el primer lugar que miramos, al principio de la lista. Sólo necesitaremos una comparación. En el peor de los casos, no descubriremos el ítem hasta la última comparación, la n-ésima comparación.

¿Cómo sería el caso promedio?

En promedio, encontramos el ítem alrededor de la mitad de la lista; es decir, compararemos contra **n/2** ítems. Recordemos, sin embargo, que a medida que **n** se hace grande, los coeficientes, sean cuales sean, se vuelven insignificantes en nuestra aproximación, por lo que la complejidad de la búsqueda secuencial es **O(n)**

Tabla 1: Comparaciones utilizadas en una búsqueda secuencial en una lista no ordenada

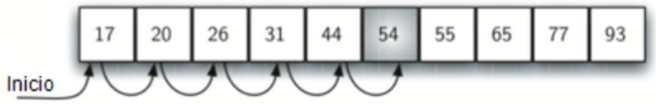
Caso	Mejor caso	Peor caso	Caso promedio
El ítem está presente	1	n	$\frac{n}{2}$
El ítem no está presente	n	n	n

¿Qué pasaría con la búsqueda secuencial si los ítems estuvieran ordenados de alguna manera? ¿Seríamos capaces de mejorar en algo la eficiencia en nuestra técnica de búsqueda?

```python
def busquedaSecuencial(Lista, item):
    pos = 0
    while pos < len(Lista):
        if Lista[pos] == item:
            return True
        elif Lista[pos] > item:
            return False
        else:
            pos = pos+1
    return False

secuencia = [0, 1, 2, 8, 13, 17, 19, 32, 42]
print(busquedaSecuencial(secuencia, 3))
print(busquedaSecuencial(secuencia, 13))
```

```
False
True
```

Crearemos una lista de ítems en orden ascendente, de menor a mayor. Si el ítem que estamos buscando está presente en la lista, la posibilidad de que esté en alguna de las **n** posiciones sigue siendo la misma que antes. Aún tendremos que hacer el mismo número de comparaciones para encontrar el ítem. Sin embargo, si el ítem no está presente hay una ligera ventaja.



La Figura muestra este proceso a medida que el algoritmo busca el ítem 50. Observe que los ítems aún se comparan en secuencia hasta el 54. No obstante, en este punto, sabemos algo más. No sólo el 54 no es el ítem que estamos buscando, sino que ningún otro ítem más allá de 54 servirá ya que la lista está ordenada. En este caso, el algoritmo no tiene que seguir mirando a lo largo de todos los ítems para reportar que no se encontró el elemento. Puede detenerse inmediatamente.

Tabla 2: Comparaciones usadas en la búsqueda secuencial en una lista ordenada

Caso	Mejor caso	Peor caso	Caso promedio
El ítem está presente	1	n	$\frac{n}{2}$
El ítem no está presente	1	n	$\frac{n}{2}$

La Tabla 2 resume estos resultados. Note que en el mejor de los casos podríamos descubrir que el ítem no está en la lista mirando únicamente un ítem. En promedio, lo sabremos solamente después de mirar **n/2** ítems. Sin embargo, esta técnica sigue siendo **O(n)**.

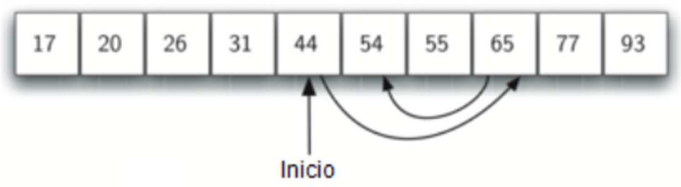
En resumen, una búsqueda secuencial se mejora ordenando la lista sólo en caso que no encontremos el ítem.

Algoritmo de búsqueda binaria

La búsqueda binaria es un algoritmo eficiente para encontrar un elemento en una lista ordenada de elementos. Funciona al dividir repetidamente a la mitad la porción de la lista que podría contener al elemento, hasta reducir las ubicaciones posibles a solo una.

Las listas ordenadas son aprovechadas en la búsqueda binaria. La búsqueda binaria comenzará examinando el ítem central. Si ese ítem es el que estamos buscando, hemos terminado. Si no es el ítem correcto, podemos utilizar la naturaleza ordenada de la lista para eliminar la mitad de los ítems restantes. Si el ítem que buscamos es mayor que el ítem central, sabemos que toda la mitad inferior de la lista, así como el ítem central, se pueden ignorar de la consideración posterior. El ítem, si es que está en la lista, debe estar en la mitad superior.

Podemos entonces repetir el proceso con la mitad superior. Comenzar en el ítem central y compararlo con el valor que estamos buscando. Una vez más, o lo encontramos o dividimos la lista por la mitad, eliminando por tanto otra gran parte de nuestro espacio de búsqueda posible.



Antes de pasar al análisis, debemos observar que este algoritmo es un gran ejemplo de una estrategia de dividir y conquistar. Dividir y conquistar significa que dividimos el problema en partes más pequeñas, resolvemos dichas partes más pequeñas de alguna manera y luego ensamblamos todo el problema para obtener el resultado.

Cuando realizamos una búsqueda binaria en una lista, primero verificamos el ítem central. Si el ítem que estamos buscando es menor que el ítem central, podemos simplemente realizar una búsqueda binaria en la mitad izquierda de la lista original. Del mismo modo, si el ítem es mayor, podemos realizar una búsqueda binaria en la mitad derecha.

```
def busquedaBinaria(Lista, item):
    primero = 0
    ultimo = len(Lista)-1

    while primero<=ultimo:
        puntoMedio = (primero + ultimo)//2
        if Lista[puntoMedio] == item:
            return True
        else:
            if item < Lista[puntoMedio]:
                ultimo = puntoMedio-1
            else:
                primero = puntoMedio+1

    return False

listaPrueba = [0, 1, 2, 8, 13, 17, 19, 32, 42,]
print(busquedaBinaria(listaPrueba, 3))
print(busquedaBinaria(listaPrueba, 13))
```

```
False
True
```

Algoritmo de búsqueda binaria recursiva

Veamos una llamada recursiva a la función de búsqueda binaria pasándole una lista más pequeña.

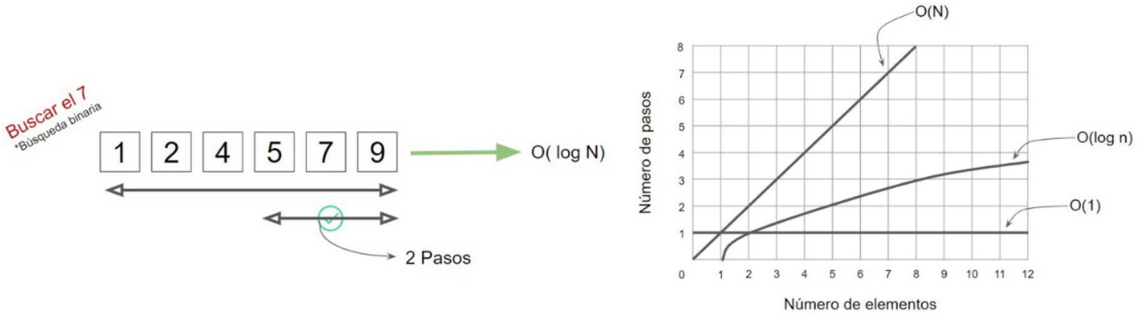
```
def busquedaBinaria(unaLista, item):
    if len(unaLista) == 0:
        return False
    else:
        puntoMedio = len(unaLista)//2
        if unaLista[puntoMedio]==item:
          return True
        else:
          if item<unaLista[puntoMedio]:
            return busquedaBinaria(unaLista[:puntoMedio],item)
          else:
            return busquedaBinaria(unaLista[puntoMedio+1:],item)

listaPrueba = [0, 1, 2, 8, 13, 17, 19, 32, 42]
print(busquedaBinaria(listaPrueba, 3))
print(busquedaBinaria(listaPrueba, 42))
```

```
False
True
```

En la solución recursiva la llamada recursiva, **busquedaBinaria(unaLista[:puntoMedio],item)** usa el operador de partición para crear la mitad izquierda de la lista que se pasa a la siguiente invocación (similarmente para la mitad derecha también).

La búsqueda binaria utilizando la partición no funcionará estrictamente en tiempo logarítmico. El operador de partición requiere un tiempo constante. Sabemos que el operador de partición en Python es realmente O(k), por lo que se puede decir que no es más rápida que la implementación iterativa.

Análisis del algoritmo de búsqueda binaria



El tiempo de ejecución aumenta lentamente a medida que aumenta el tamaño de la entrada.

Si tenemos un array de enteros ordenados y queremos buscar un número, tenemos dos opciones, o buscamos uno por uno utilizando una búsqueda lineal, o por el contrario podemos hacer una búsqueda binaria donde, comparamos el elemento del medio si es mayor o menor y sobre ese subgrupo volvemos a comparar.

Debemos repetir el proceso hasta que encontremos el resultado. En este ejemplo, cómo podemos observar utilizando la búsqueda binaria únicamente tardamos dos pasos, mientras que en la búsqueda lineal tardaríamos 5. Esto se debe a que cada vez vamos dividiendo por 2 el número de elementos que vamos a comprobar.

¿Qué son los Logaritmos?

Los logaritmos son la parte inversa a los componentes, como recordamos un componente es "elevar" por ejemplo 2**3 se traduce en 2*2*2 con el resultado de 8.

Por lo que lo opuesto sería un logaritmo que es cuantas veces tienes que dividir el número por 2 hasta que obtienes 1. 8/2/2/2 = 1 lo que se traduce en log2 8 = 3.

Para analizar el algoritmo de búsqueda binaria, necesitamos recordar que cada comparación elimina aproximadamente la mitad de los ítems restantes de la consideración. ¿Cuál es el número máximo de comparaciones que este algoritmo requerirá para examinar la lista completa? Si empezamos con n ítems, alrededor de n/2

Comparaciones	Número aproximado de ítems restantes
1	$\frac{n}{2}$
2	$\frac{n}{4}$
3	$\frac{n}{8}$
...	
i	$\frac{n}{2^i}$

Cuando dividimos la lista suficientes veces, terminamos con una lista que tiene un único ítem. Ya sea aquél ítem único el valor que estamos buscando o no lo sea. En todo caso, habremos terminado.

El número de comparaciones necesarias para llegar a este punto es **i** donde **n/2^i=1**. La solución para **i** nos da **i= log n**. El número máximo de comparaciones es logarítmico con respecto al número de ítems de la lista. **Por lo tanto, la búsqueda binaria es O(log n)**

A pesar de que una búsqueda binaria es generalmente mejor que una búsqueda secuencial, es importante tener en cuenta que, para valores pequeños de n, el costo adicional del ordenamiento probablemente no vale la pena. De hecho, siempre debemos considerar si es rentable asumir el trabajo extra del ordenamiento para obtener beneficios en la búsqueda.

Si podemos ordenar una sola vez y luego buscar muchas veces, el costo del ordenamiento no es tan significativo. Sin embargo, para listas grandes, incluso ordenar una vez puede resultar tan costoso que simplemente realizar una búsqueda secuencial desde el principio podría ser la mejor opción.

Algoritmos de Ordenamiento

Definición

Ordenar es el proceso de ubicar elementos de una colección en algún orden.

Por ejemplo, una lista de palabras podría ordenarse alfabéticamente o por longitud. Una lista de ciudades podría ordenarse por población, por área o por código postal. Ya hemos visto una serie de algoritmos que fueron capaces de beneficiarse de tener una lista ordenada (recuerde la búsqueda binaria).

Al igual que la búsqueda, la eficiencia de un algoritmo de ordenamiento está relacionada con el número de ítems que se están procesando. Para las pequeñas colecciones, un método de ordenamiento complejo puede resultar más problemático que beneficioso. La sobrecarga puede ser demasiado alta. Por otra parte, para colecciones más grandes, queremos aprovechar tantas mejoras como sean posibles.

Discutiremos varias técnicas de ordenamiento y las compararemos respecto a sus tiempos de ejecución.

¿Qué operaciones se pueden utilizar para analizar un proceso de ordenamiento?

Primero, será necesario comparar dos valores para ver cuál es más pequeño (o más grande). Para ordenar una colección, será necesario contar con una manera sistemática de comparar los valores para ver si no están en orden. El número total de comparaciones será la forma más común de medir un procedimiento de ordenamiento.

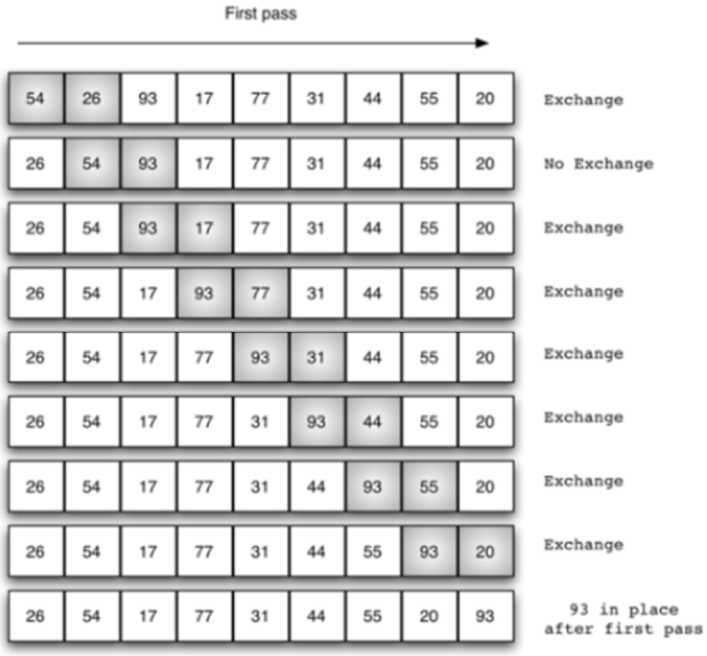
En segundo lugar, cuando los valores no están en la posición correcta con respecto a los otros, puede ser necesario intercambiarlos. Este intercambio es una operación costosa y el número total de intercambios también será importante para evaluar la eficiencia global del algoritmo.

Algoritmos de Ordenamiento. Burbuja

El ordenamiento burbuja hace múltiples pasadas a lo largo de una lista. Compara los ítems adyacentes (próximos) e intercambia los que no están en orden. Cada pasada a lo largo de la lista ubica el siguiente valor más grande en su lugar apropiado.

En esencia, cada ítem "**burbujea**" hasta el lugar al que pertenece.

La siguiente Figura muestra la primera pasada de un ordenamiento burbuja. Los ítems sombreados se comparan para ver si no están en orden. Si hay **n** ítems en la lista, entonces hay **n–1**, parejas de ítems que deben compararse en la primera pasada. Es importante tener en cuenta que, una vez que el valor más grande de la lista es parte de una pareja, éste avanzará continuamente hasta que la pasada se complete.

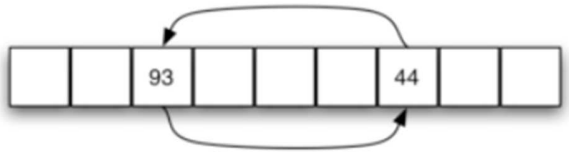


Al comienzo de la segunda pasada, el valor más grande ya está en su lugar. Quedan **n–1** ítems por ordenar, lo que significa que habrá **n–2** parejas. Puesto que cada pasada ubica al siguiente valor mayor en su lugar, el número total de pasadas necesarias será **n –1**. Después de completar la pasada **n –1** (Todas), el ítem más pequeño debe estar en la posición correcta (posición 0) sin requerir procesamiento adicional. La función recibe la lista como un parámetro, y lo modifica intercambiando ítems según sea necesario.

En Python es posible realizar la asignación simultánea. La instrucción **a,b=b,a** dará lugar a que se realicen dos instrucciones de asignación al mismo tiempo. Usando la asignación simultánea, la operación de intercambio se puede hacer en una sola instrucción.

```
def ordenamientoBurbuja(unaLista):
    for numPasada in range(len(unaLista)-1,0,-1):
        for i in range(numPasada):
            if unaLista[i]>unaLista[i+1]:
                unaLista[i], unaLista[i+1] =  unaLista[i+1], unaLista[i]

unaLista = [54,26,93,17,77,31,44,55,20]
print(unaLista)
ordenamientoBurbuja(unaLista)
print(unaLista)
```

```
[54, 26, 93, 17, 77, 31, 44, 55, 20]

[17, 20, 26, 31, 44, 54, 55, 77, 93]
```



En Python, el intercambio se puede hacer como dos asignaciones simultáneas

Análisis del algoritmo de la burbuja.

Un ordenamiento burbuja se considera como el método de ordenamiento más **ineficiente** ya que debe intercambiar ítems antes de que se conozca su ubicación final.

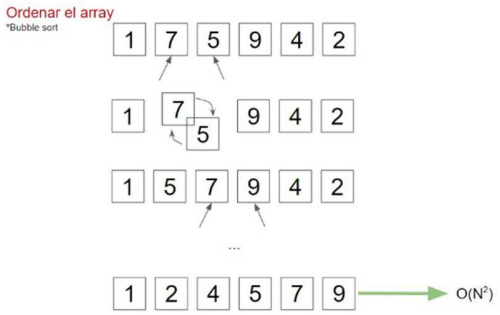
Para analizar el ordenamiento burbuja, debemos tener en cuenta que independientemente de cómo están dispuestos los ítems en la lista inicial, se harán **n −1** pasadas para ordenar una lista de tamaño **n**.

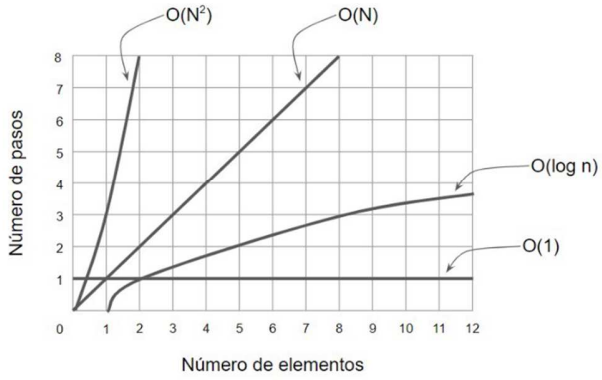
La siguiente imagen muestra el número de comparaciones para cada pasada. El número total de comparaciones es la suma de los primeros **n−1** enteros.

```
def ordenamientoBurbuja(unaLista):
    for numPasada in range(len(unaLista)-1,0,-1):
        print("\n",*unaLista, end= " : | ")
        for i in range(numPasada):
            if unaLista[i]>unaLista[i+1]:
                unaLista[i], unaLista[i+1] =  unaLista[i+1], unaLista[i]
                print(f"{unaLista[i+1]} -> {unaLista[i]}", end=" | ")

unaLista = [54,26,93,17,77,31,44,55,20]
ordenamientoBurbuja(unaLista)
```

```
54 26 93 17 77 31 44 55 20 : | 54 -> 26 | 93 -> 17 | 93 -> 77 | 93 -> 31 | 93 -> 44 | 93 -> 55 | 93 -> 20 |
26 54 17 77 31 44 55 20 93 : | 54 -> 17 | 77 -> 31 | 77 -> 44 | 77 -> 55 | 77 -> 20 |
26 17 54 31 44 55 20 77 93 : | 26 -> 17 | 54 -> 31 | 54 -> 44 | 55 -> 20 |
17 26 31 44 54 20 55 77 93 : | 54 -> 20 |
17 26 31 44 20 54 55 77 93 : | 44 -> 20 |
17 26 31 20 44 54 55 77 93 : | 31 -> 20 |
17 26 20 31 44 54 55 77 93 : | 26 -> 20 |
17 20 26 31 44 54 55 77 93 : |
```

El **ordenamiento de la burbuja** está en el orden de **O(n^2)** comparaciones. En el mejor de los casos, si la lista ya está ordenada, no se realizarán intercambios. Sin embargo, en el peor de los casos, cada comparación causará un intercambio. En promedio, intercambiamos la mitad de las veces.





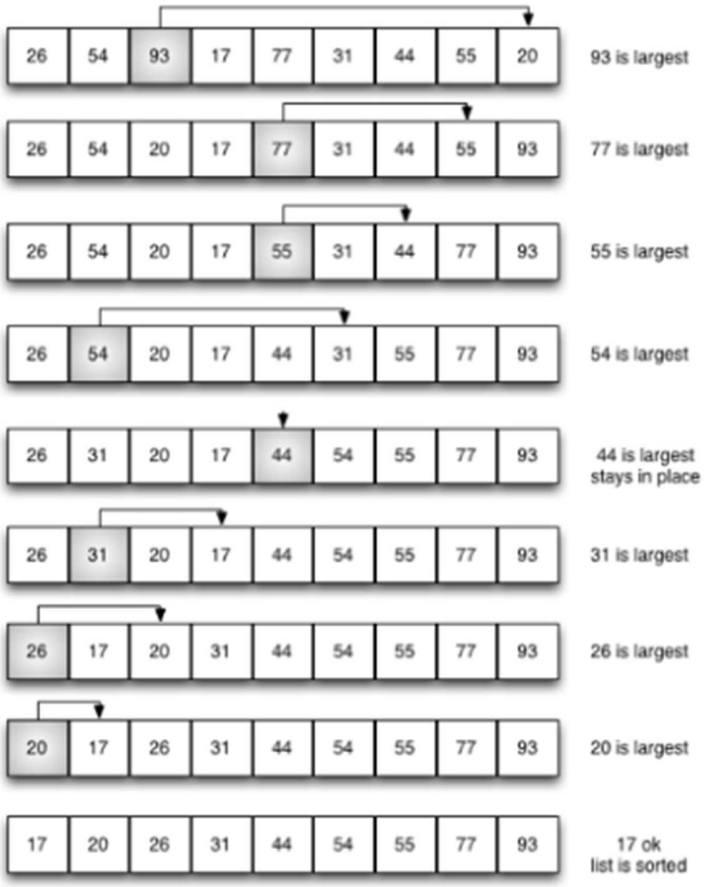
Entonces en el algoritmo bubble sort comparamos el primer elemento con el segundo, y si el primero es mayor los intercambiamos, debemos hacer este proceso tantas veces como elementos tenga el array, y lo hacemos sobre el array entero. A efectos de programación es un bucle for dentro de otro.

Este tipo de algoritmo destaca por ser mucho más lento que **O(N)**, y además tiene un nombre especial, algoritmo cuadrático.

Algoritmos de Ordenamiento. Por Selección

El **ordenamiento por selección** mejora el ordenamiento burbuja haciendo un sólo intercambio por cada pasada a través de la lista. Para hacer esto, un **ordenamiento por selección** busca el valor mayor a medida que hace una pasada y, después de completar la pasada, lo pone en la ubicación correcta. Al igual que con un ordenamiento burbuja, después de la primera pasada, el ítem mayor está en la ubicación correcta. Después de la segunda pasada, el siguiente mayor está en su ubicación. Este proceso continúa y requiere **n−1** pasadas para ordenar los n ítems, ya que el ítem final debe estar en su lugar después de la **(n−1) -ésima** pasada.

A continuación, se muestra todo el proceso de ordenamiento. En cada paso, el ítem mayor restante se selecciona y luego se pone en su ubicación correcta. La primera pasada ubica el **93**, la segunda pasada ubica el **77**, la tercera ubica el **55**, y así sucesivamente



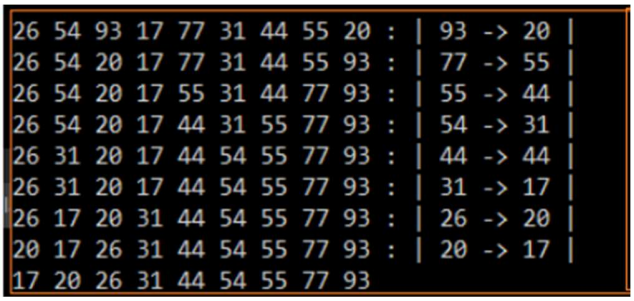
```
def ordenamientoPorSeleccion(unaLista):
    for llenarRanura in range(len(unaLista)-1,0,-1):
        posicionDelMayor=0
        for ubicacion in range(1,llenarRanura+1):
            if unaLista[ubicacion]>unaLista[posicionDelMayor]:
                posicionDelMayor = ubicacion
        unaLista[llenarRanura], unaLista[posicionDelMayor] \
        = unaLista[posicionDelMayor] , unaLista[llenarRanura]

unaLista = [54,26,93,17,77,31,44,55,20]
print(unaLista)
ordenamientoPorSeleccion(unaLista)
print(unaLista)
```

```
[54, 26, 93, 17, 77, 31, 44, 55, 20]
[17, 20, 26, 31, 44, 54, 55, 77, 93]
```

Análisis del algoritmo de Ordenamiento por Selección

El **ordenamiento por selección** hace el mismo número de comparaciones que el ordenamiento burbuja y por lo tanto también es **O(n2)**

```
def ordenamientoPorSeleccion(unaLista):
   for llenarRanura in range(len(unaLista)-1,0,-1):
       posicionDelMayor=0
       print("\n",*unaLista, end= " : | ")
       for ubicacion in range(1,llenarRanura+1):
           if unaLista[ubicacion]>unaLista[posicionDelMayor]:
               posicionDelMayor = ubicacion
       unaLista[llenarRanura], unaLista[posicionDelMayor] = \
                               unaLista[posicionDelMayor] , unaLista[llenarRanura]
       print(f"{unaLista[llenarRanura]} -> {unaLista[posicionDelMayor]}", end=" | ")


unaLista = [26,54,93,17,77,31,44,55,20]
ordenamientoPorSeleccion(unaLista)
print("\n",*unaLista)
```



Sin embargo, debido a la reducción en el número de intercambios, el **ordenamiento por selección** normalmente se ejecuta más rápidamente en pruebas de referencia. De hecho, para nuestra lista, el ordenamiento burbuja hace 20 intercambios, mientras que el ordenamiento por selección hace sólo 8.

Algoritmos de Ordenamiento. Quick Sort (0rdenamiento rápido)

El **ordenamiento rápido** usa dividir y conquistar, Sin embargo, es posible que la lista no se divida por la mitad. Cuando esto sucede, veremos que el desempeño disminuye.

El algoritmo de **ordenamiento rápido** primero selecciona un valor, que se denomina el **valor pivote**. Nosotros usaremos el primer ítem de la lista como pivote. El papel del **valor pivote** es ayudar a dividir la lista.

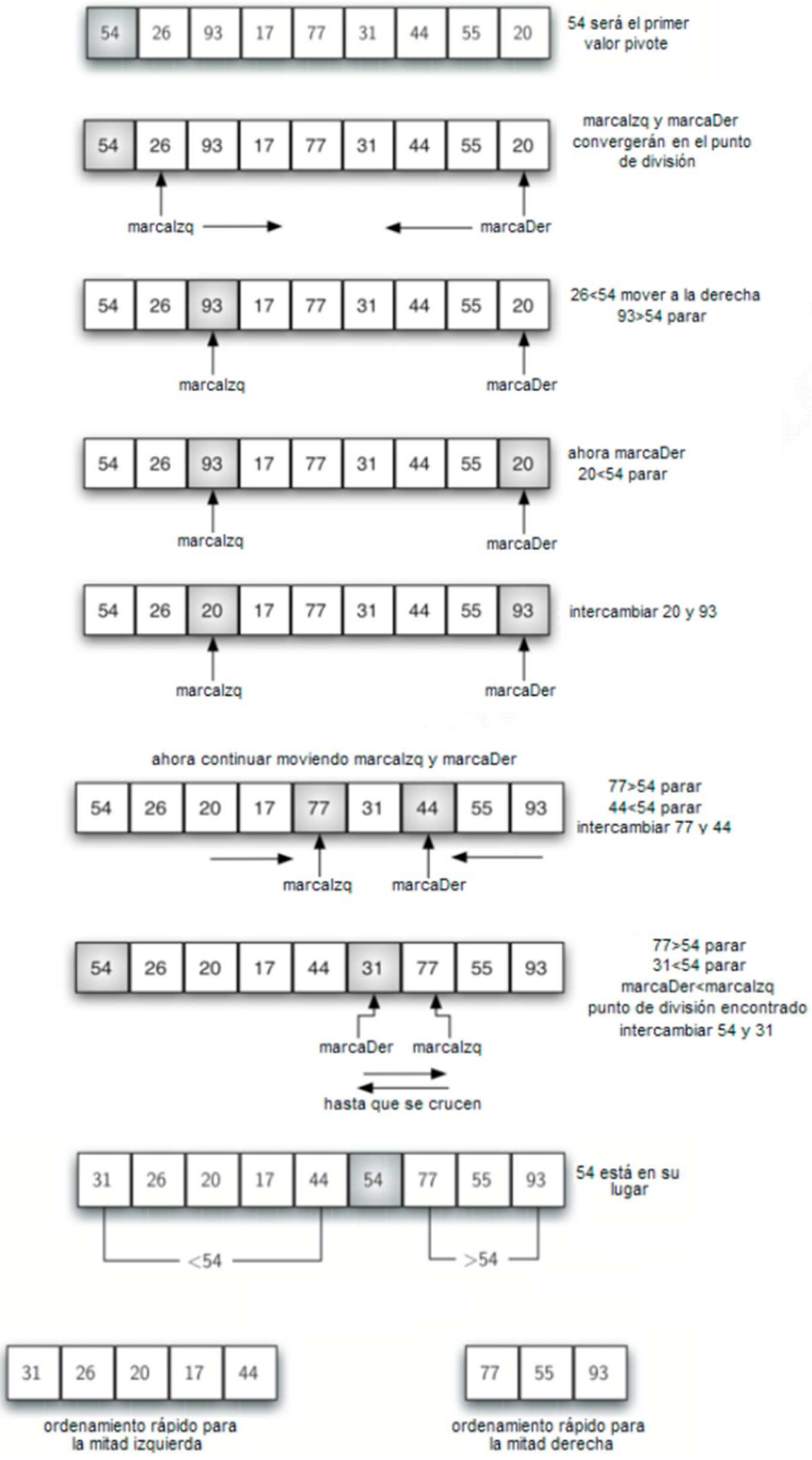
Buscamos el punto de división, **¿qué es el punto de división?** Es la posición real del **pivote** con respecto al resto de los elementos de la lista. Este se utiliza para dividir la lista para las llamadas posteriores a la función de **ordenamiento rápido**.

La siguiente figura muestra que 54 servirá como nuestro primer **valor pivote**.

Luego empieza la partición sin ordenamiento final de los elementos, se busca encontrar el punto medio para alojar al **pivote** acomodando los valores menores a la izquierda y los mayores a la derecha del pivote.

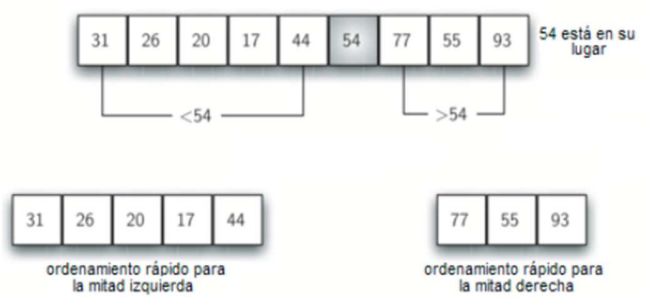
Comienza localizando dos marcadores de posición llamados **marcaIzq** y **marcaDer**, estos marcadores avanzan (**marcaIzq** incrementa posiciones y **marcaDer** decrementa posiciones) mientras que los ítems se posicionarán a la izquierda o derecha según corresponda hasta que los marcadores se encuentren y marcamos este como el punto de división, se intercambian la posición del ítem de **marcaIzq** por el **pivote** en este ejemplo el 31 por el 54.

La lista ahora se puede dividir en el punto de división y el **ordenamiento rápido** se puede invocar recursivamente para las dos mitades. Primero resuelve la izquierda y luego la derecha.


54 será el primer valor pivote
marcalzq y marcaDer convergerán en el punto de división
marcalzq
marcaDer
26<54 mover a la derecha
93>54 parar
ahora marcaDer 20<54 parar
intercambiar 20 y 93
ahora continuar moviendo marcalzq y marcaDer
77>54 parar
44<54 parar
intercambiar 77 y 44
77>54 parar
31<54 parar
marcaDer<marcalzq
punto de división encontrado
intercambiar 54 y 31
hasta que se crucen
54 está en su lugar
<54
>54
ordenamiento rápido para la mitad izquierda
ordenamiento rápido para la mitad derecha

A continuación, vemos la implementación del ordenamiento rápido o **quick sort** y luego como se va partiendo (dividiendo) la lista en subproblemas (aplicando la recursividad) hasta finalizar el ordenamiento.

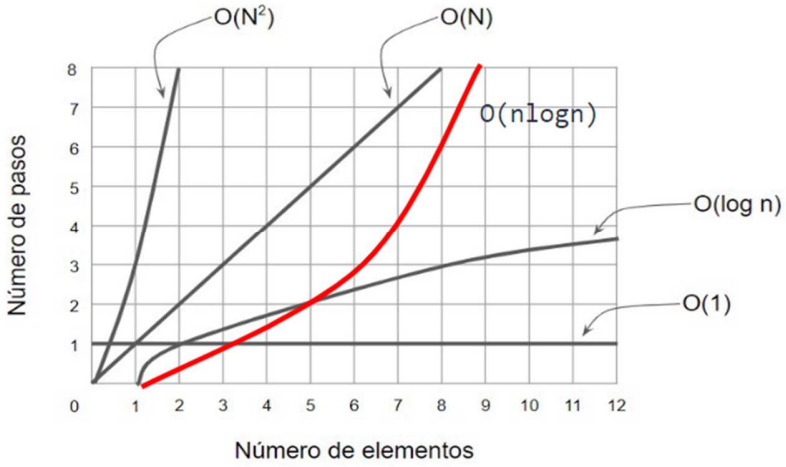
```
def ordenamientoRapido(unaLista):
    ordenamientoRapidoAuxiliar(unaLista,0,len(unaLista)-1)

def ordenamientoRapidoAuxiliar(unaLista,primero,ultimo):
    if primero<ultimo:
        puntoDivision = particion(unaLista,primero,ultimo)
        ordenamientoRapidoAuxiliar(unaLista,primero,puntoDivision-1)
        ordenamientoRapidoAuxiliar(unaLista,puntoDivision+1,ultimo)

def particion(unaLista,primero,ultimo):
    valorPivote = unaLista[primero]
    marcaIzq = primero+1
    marcaDer = ultimo
    hecho = False
    while not hecho:
        print(f"{unaLista} Pivote:{valorPivote} posiciones Izq {marcaIzq}= \
        {unaLista[marcaIzq]} ultimo {marcaDer}={unaLista[marcaDer]}")
        while marcaIzq <= marcaDer and unaLista[marcaIzq] <= valorPivote:
            marcaIzq = marcaIzq + 1

        while unaLista[marcaDer] >= valorPivote and marcaDer >= marcaIzq:
            marcaDer = marcaDer -1

        if marcaDer < marcaIzq:
            hecho = True
        else:
            unaLista[marcaIzq],unaLista[marcaDer] =  \
            unaLista[marcaDer],unaLista[marcaIzq]

    unaLista[primero], unaLista[marcaDer] =  \
    unaLista[marcaDer], unaLista[primero]
    return marcaDer

unaLista = [54,26,93,17,44,77,31,55,20]
ordenamientoRapido(unaLista)
print(unaLista)
```




```
[54, 26, 93, 17, 77, 31, 44, 55, 20] Pivote:54 posiciones Izq 1=26 ultimo 8=20
[54, 26, 20, 17, 77, 31, 44, 55, 93] Pivote:54 posiciones Izq 2=20 ultimo 8=93
[54, 26, 20, 17, 44, 31, 77, 55, 93] Pivote:54 posiciones Izq 4=44 ultimo 6=77
[31, 26, 20, 17, 44, 54, 77, 55, 93] Pivote:31 posiciones Izq 1=26 ultimo 4=44
[17, 26, 20, 31, 44, 54, 77, 55, 93] Pivote:17 posiciones Izq 1=26 ultimo 2=20
[17, 26, 20, 31, 44, 54, 77, 55, 93] Pivote:26 posiciones Izq 2=20 ultimo 2=20
[17, 20, 26, 31, 44, 54, 77, 55, 93] Pivote:77 posiciones Izq 7=55 ultimo 8=93
[17, 20, 26, 31, 44, 54, 55, 77, 93]
```

Análisis del algoritmo de Ordenamiento. Quick Sort

Para analizar la función ordenamientoRapido o quick sort, debemos saber que para una lista de longitud n, si la partición siempre ocurre en el centro de la lista, habrá de nuevo log n divisiones. Con el fin de encontrar el punto de división, cada uno de los n ítems debe ser comparado contra el valor pivote. El resultado es Big O (n log n.)



Cuando la partición no ocurre en el centro, y los puntos de división están muy a la izquierda o a la derecha, dejando una división muy desigual el desempeño disminuye.

Para finalizar veamos que en el cuadro comparativo de algoritmos de ordenamiento, indica que quicksort podría en el peor de los casos ser O(n^2)

ALGORITMOS DE ORDENAMIENTO

NOMBRE	COMPLEJIDAD			MEMORIA	ESTABLE
	MEJOR	MEDIA	PEOR		
SELECCION (SELECTION SORT)	n^2	n^2	n^2	1	no
BURBUJA (BUBBLE SORT)	n	n^2	n^2	1	si
INSERCION (INSERTION SORT)	n	n^2	n^2	1	si
MEZCLA (MERGE SORT)	n log n	n log n	n log n	n	si
RAPIDO (QUICKSORT)	n	n log n	n^2	log n	no

Algoritmos y Programación

Licenciatura en Gestión de Tecnología de la Información