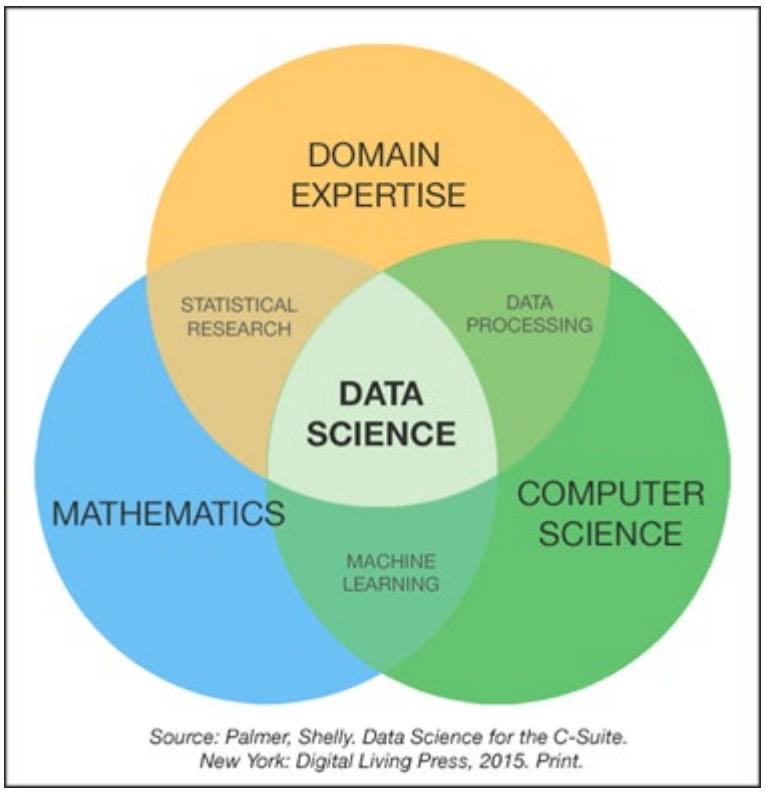
UNIVERSIDAD NACIONAL DE INGENIERÍA

OTI UNI

CIENCIA DE DATOS I



SESION 03

ALGORITMICA

INTRODUCCION

La **algoritmia** es una disciplina que se ocupa del **estudio y diseño de algoritmos**, que son conjuntos de instrucciones o reglas que describen cómo resolver un problema en particular. Un algoritmo es una secuencia de pasos bien definidos y precisos que se utilizan para resolver un problema o llevar a cabo una tarea específica.

La notación **Big O**, también conocida como notación asintótica, se utiliza en ciencias de la computación para analizar el rendimiento de los algoritmos y describir su complejidad. En términos simples, la notación Big O permite estimar cuánto **tiempo** o **espacio** requerirá un algoritmo a medida que crece el tamaño de los datos de entrada.

La notación Big O se representa como **O(f(n))**, donde "**f(n)**" es una función que describe la cantidad de operaciones o recursos necesarios en función del tamaño de entrada **"n"**. La notación Big O se utiliza para expresar el **peor escenario** de complejidad temporal o espacial de un algoritmo.

La notación Big O se utiliza para **comparar** la eficiencia relativa de diferentes algoritmos y determinar cuál es **más óptimo** en términos de tiempo o espacio requerido. Por ejemplo, si

INTRODUCCION

un algoritmo tiene una complejidad **O(n)**, significa que su tiempo de ejecución **aumentará linealmente** a medida que el tamaño de entrada **"n" aumente**. En contraste, un algoritmo con una complejidad $O(n^2)$ tendrá un tiempo de ejecución cuadrático, lo que significa que aumentará más rápidamente a medida que el tamaño de entrada aumente.

En resumen, la notación Big O es una herramienta importante en algoritmia para analizar y comparar la eficiencia de los algoritmos. Nos permite predecir cómo crecerá el tiempo o el espacio requerido por un algoritmo a medida que aumente el tamaño de entrada, lo cual es esencial para tomar decisiones informadas sobre qué algoritmo utilizar en diferentes situaciones.

IMPORTANCIA EN LA CIENCIA DE DATOS

La complejidad algorítmica es un **aspecto crucial en la ciencia de datos** por varias razones.

A continuación, se detallan las principales razones de su importancia:

1. **Eficiencia de Procesamiento**

Escalabilidad: Los conjuntos de datos en ciencia de datos pueden ser extremadamente grandes. La eficiencia de los algoritmos es esencial para procesar estos datos de manera oportuna. Un algoritmo con una complejidad de tiempo alta puede volverse impracticable con conjuntos de datos grandes.

Uso de Recursos: Los algoritmos eficientes no solo ahorran tiempo, sino que también utilizan menos recursos computacionales (memoria, CPU). Esto es especialmente importante en **entornos con recursos limitados** o cuando se trabaja con **grandes volúmenes de datos**.

2. Predicción del Rendimiento

Estimación del Tiempo de Ejecución: Comprender la complejidad algorítmica permite a los científicos de datos predecir cuánto tiempo tomará ejecutar un algoritmo en función del tamaño del conjunto de datos. Esto es crucial para la planificación y gestión de proyectos.

Identificación de Cuellos de Botella: Analizar la complejidad algorítmica ayuda a identificar partes del código que pueden ser optimizadas para mejorar el rendimiento general.

3. Comparación de Algoritmos

Selección del Mejor Algoritmo: La complejidad algorítmica permite comparar diferentes algoritmos para una tarea específica y seleccionar el más eficiente. Por ejemplo, al elegir entre diferentes algoritmos de clasificación o clustering, la complejidad algorítmica es un criterio clave.

Optimización: Incluso después de seleccionar un algoritmo, es posible que se necesiten optimizaciones adicionales. Conocer la complejidad ayuda a enfocar los esfuerzos de optimización en las partes del algoritmo que más lo necesitan.

4. Escalabilidad y Mantenimiento

Crecimiento de Datos: A medida que los datos crecen, los algoritmos con baja complejidad (como O(n log n)) se vuelven cada vez más valiosos en comparación con aquellos con alta complejidad (como O(n^2) o más complejos). Esto asegura que las soluciones implementadas hoy seguirán siendo viables mañana.

Evolución del Proyecto: Los proyectos de ciencia de datos a menudo evolucionan con el tiempo. Algoritmos eficientes aseguran que las soluciones puedan adaptarse al crecimiento y cambios en los datos sin necesidad de ser reemplazadas completamente.

IMPORTANCIA EN LA CIENCIA DE DATOS

5. Aplicaciones en Tiempo Real

Procesamiento en Tiempo Real: En aplicaciones como la detección de fraudes, recomendaciones en línea o análisis de redes sociales, la capacidad de procesar datos en tiempo real es crucial. La complejidad algorítmica determina si un algoritmo puede cumplir con los requisitos de tiempo real.

Experiencia del Usuario: Algoritmos eficientes mejoran la experiencia del usuario final al proporcionar resultados rápidos y precisos, lo cual es crítico en aplicaciones interactivas.

La complejidad algorítmica es de suma importancia para las **bibliotecas de ciencia de datos** porque determina la eficiencia, escalabilidad y capacidad de las bibliotecas para manejar grandes volúmenes de datos y proporcionar resultados rápidos y precisos. Algoritmos bien diseñados y optimizados permiten que estas bibliotecas sean herramientas robustas y confiables para científicos de datos, desarrolladores e ingenieros.

Aunque es posible usar bibliotecas de ciencia de datos sin un conocimiento profundo de la complejidad algorítmica, tener una comprensión básica de la complejidad algorítmica puede ofrecerte varios beneficios importantes:

¿Porque debemos tener nociones de complejidad algorítmica?

1. Mejor Elección de Algoritmos

Selección Informada: Saber cómo funciona la complejidad algorítmica te ayuda a seleccionar los algoritmos más adecuados para tus tareas. Por ejemplo, entender que un algoritmo tiene complejidad O(n^2) te permitirá evitarlo para grandes conjuntos de datos y optar por uno con complejidad O (n log n).

2. Optimización del Rendimiento

Tuning de Hiperparámetros: Al comprender la complejidad, puedes ajustar mejor los hiperparámetros de tus modelos para equilibrar la precisión y el tiempo de ejecución.
Identificación de Cuellos de Botella: Puedes identificar qué partes de tu pipeline de datos son ineficientes y centrarte en optimizarlas.

3. Manejo de Grandes Volúmenes de Datos

Escalabilidad: Conocer la complejidad te permite anticipar cómo escalarán tus soluciones con el aumento del tamaño de los datos. Esto es crucial para la planificación y gestión de proyectos.

Elección de Herramientas: Puedes elegir las herramientas y bibliotecas más adecuadas que implementan algoritmos eficientes para tus necesidades específicas.

IMPORTANCIA EN LA CIENCIA DE DATOS

4. Resolución de Problemas y Depuración

Diagnóstico de Problemas: Si tu código está funcionando más lentamente de lo esperado, entender la complejidad algorítmica te puede ayudar a diagnosticar y solucionar el problema.

Evitar Pitfalls Comunes: Puedes evitar errores comunes que conducen a un rendimiento subóptimo, como el uso de algoritmos inapropiados para grandes conjuntos de datos.

5. Eficiencia de Recursos

Optimización de Recursos Computacionales: Saber cómo funcionan los algoritmos te permite optimizar el uso de memoria y CPU, lo que es especialmente importante en entornos con recursos limitados.

Reducción de Costos: En entornos de computación en la nube, algoritmos más eficientes pueden reducir significativamente los costos operativos al disminuir el tiempo de cómputo.

6. Mejora de la Comprensión General

Comprensión de la Documentación: Muchas veces, la documentación de las bibliotecas menciona la complejidad de los algoritmos implementados. Entender estos conceptos te permitirá interpretar mejor esta documentación.

Comunicación Efectiva: Ser capaz de discutir sobre la complejidad algorítmica te permitirá comunicarte de manera más efectiva con otros profesionales de la ciencia de datos y desarrolladores.

ANALISIS DE ALGORTIMOS

Antes de comenzar con este tema revisemos una comparativa del hardware usado en el proyecto APOLO con el hardware actual:

Especificaciones del Apollo Guidance Computer (AGC)

Memoria

Memoria de Núcleo Magnético (RAM):

El AGC tenía aproximadamente 2 kB (2048 palabras) de memoria RAM, donde cada palabra era de 16 bits.

Memoria de Núcleo Magnético (ROM):

El AGC incluía aproximadamente 36 kB de memoria ROM (Read-Only Memory) para almacenar el software del sistema. Esta memoria era no volátil y retenía los datos incluso sin energía.

Espacio en Disco

No tenía disco duro:

El AGC no tenía un disco duro en el sentido moderno. Utilizaba memoria de núcleo magnético para almacenar tanto el software como los datos necesarios para la navegación y el control en tiempo real.

Velocidad del Procesador

Velocidad de Reloj:

El AGC operaba a una velocidad de reloj de aproximadamente 2.048 MHz (2.048 millones de ciclos por segundo).

Ciclos por Instrucción:

La mayoría de las instrucciones se ejecutaban en 12 ciclos de reloj, lo que daba una velocidad de procesamiento de alrededor de 85,000 instrucciones por segundo.

Comparación con Tecnología Moderna

Para poner estas especificaciones en perspectiva, aquí hay algunas comparaciones con la tecnología moderna:

Memoria

Un teléfono inteligente moderno típico tiene varios GB de RAM (millones de veces más que el AGC).

ANALISIS DE ALGORTIMOS

Espacio de Almacenamiento

Los dispositivos modernos tienen almacenamiento en disco (SSD o HDD) de cientos de GB a varios TB, una capacidad increíblemente superior a la del AGC.

Velocidad del Procesador

Los procesadores modernos funcionan a velocidades de reloj de varios GHz (miles de millones de ciclos por segundo) y pueden ejecutar miles de millones de instrucciones por segundo, con múltiples núcleos y capacidades de procesamiento paralelo.

Mas información:

https://www.ibiblio.org/apollo/#gsc.tab=0
https://www.youtube.com/watch?v=2KSahAoOLdU
https://en.wikipedia.org/wiki/Hidden_Figures

Ahora si introduzcámonos en el análisis de algoritmos:

Por lo general, hay múltiples formas de resolver el problema usando un programa de computadora. Por ejemplo, hay varias formas de ordenar elementos en una lista: puede utilizar la ordenación por combinación, la ordenación por burbuja, la ordenación por inserción, etc.

Todos estos algoritmos tienen sus propios pros y contras y el trabajo del desarrollador es sopesarlos para poder elegir el mejor algoritmo para usar en cualquier caso de uso. En otras palabras, la pregunta principal es qué algoritmo usar para resolver un problema específico cuando existen múltiples soluciones al problema.

El análisis de algoritmos se refiere al análisis de la complejidad de diferentes algoritmos y la búsqueda del algoritmo más eficiente para resolver el problema en cuestión. La notación Big-O es una medida estadística utilizada para describir la complejidad del algoritmo.

Nota: La notación Big-O es una de las medidas utilizadas para la complejidad algorítmica. Algunos otros incluyen Big-Theta y Big-Omega. Big-Omega, Big-Theta y Big-O son intuitivamente iguales a la mejor, media y peor complejidad de tiempo que puede lograr un algoritmo. Por lo general, usamos Big-O como medida, en lugar de los otros dos, porque podemos garantizar que un algoritmo se ejecuta con una complejidad aceptable en el peor de los casos, funcionará en el promedio y en el mejor de los casos también, pero no viceversa.

¿Por qué es importante el análisis de algoritmos?

Para entender por qué es importante el análisis de algoritmos, tomaremos la ayuda de un ejemplo simple. Supongamos que un gerente asigna una tarea a dos de sus empleados para diseñar un algoritmo en Python que calcule la factorial de un número ingresado por el usuario.

El algoritmo desarrollado por el primer empleado se ve así:

ANALISIS DE ALGORTIMOS

```python
def fact(n):
    product = 1
    for i in range(n):
        product = product * (i+1)
    return product

print(fact(5))
```

Observemos que el algoritmo simplemente toma un número entero como argumento. Dentro de la función fact(), una variable llamada producto se inicializa en 1. Un bucle se ejecuta de 1 a n y durante cada iteración, el valor del producto se multiplica por el número que itera el bucle y el resultado se almacena en el producto. variable de nuevo. Después de que se ejecute el ciclo, la variable del producto contendrá la factorial.

De manera similar, el segundo empleado también desarrolló un algoritmo que calcula la factorial de un número. El segundo empleado usó una función recursiva para calcular la factorial del número n:

```python
def fact2(n):
    if n == 0:
        return 1
    else:
        return n * fact2(n-1)

print(fact2(5))
```

El gerente tiene que decidir **qué algoritmo usar**. Para hacerlo, han decidido elegir qué algoritmo se ejecuta **más rápido**. Una forma de hacerlo es encontrar el tiempo necesario para ejecutar el código en la misma entrada.

Podemos usar el literal %timeit seguido de la llamada a la función para encontrar el tiempo que tarda la función en ejecutarse:

```
%timeit fact(50)
```

Esto nos dará:

```
9 µs ± 405 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

El resultado dice que el algoritmo tarda 9 microsegundos (más/menos 405 nanosegundos) por ciclo.

Del mismo modo, podemos calcular cuánto tiempo tarda en ejecutarse el segundo enfoque:

ANALISIS DE ALGORTIMOS CON NOTACION BIG-O

```
%timeit fact2(50)
```

Esta vez no da:

```
15.7 µs ± 427 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

El segundo algoritmo que implica la recursividad tarda 15.7 microsegundos (más/menos 427 nanosegundos).

El tiempo de ejecución muestra que el primer algoritmo es más rápido en comparación con el segundo algoritmo que involucra recursividad. Cuando se trata de grandes entradas, la diferencia de rendimiento puede volverse más significativa.

Sin embargo, el **tiempo de ejecución no es una buena métrica** para medir la complejidad de un algoritmo, ya que depende del **hardware**. Se necesita una métrica de análisis de complejidad más objetiva para un algoritmo. Aquí es donde entra en juego la notación Big O.

La notación Big-O significa la relación entre la entrada al algoritmo y los pasos necesarios para ejecutar el algoritmo. Se denota con una gran "O" seguida de un paréntesis de apertura y cierre. Dentro del paréntesis, la relación entre la entrada y los pasos tomados por el algoritmo se presenta usando "n".

La conclusión clave es: Big-O no está interesado en una instancia particular en la que ejecuta un algoritmo, como fact(50), sino qué tan bien se escala con una entrada cada vez mayor.
Esta es una métrica mucho mejor para evaluar que el tiempo concreto para una instancia concreta

Por ejemplo, si existe una relación lineal entre la entrada y los pasos que da el algoritmo para completar su ejecución, la notación Big-O utilizada será O(n). De manera similar, la notación Big-O para funciones cuadráticas es $O(n^2)$.

Para desarrollar la intuición:

O(n): en n=1, se da 1 paso. En n=10, se toman 10 pasos.

$O(n^2)$: en n=1, se da 1 paso. En n=10, se toman 100 pasos.

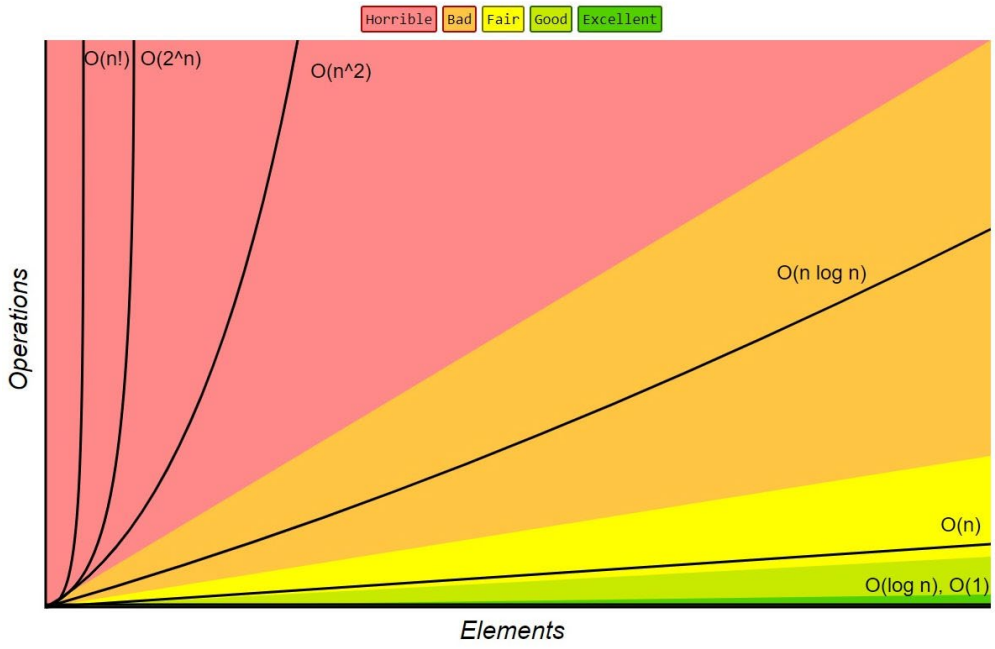
En n=1, ¡estos dos harían lo mismo! Esta es otra razón por la que observar la relación entre la entrada y el número de pasos para procesar esa entrada es mejor que solo evaluar funciones con alguna entrada concreta.

Las siguientes son algunas de las funciones Big-O más comunes:

ANALISIS DE ALGORTIMOS CON NOTACION BIG-O

Name	Big O
Constant	O(c)
Linear	O(n)
Quadratic	$O(n^2)$
Cubic	$O(n^3)$
Exponential	$O(2^n)$
Logarithmic	O(log(n))
Log Linear	O(nlog(n))

Visualicémoslas:



En términos generales, cualquier cosa peor que lineal se considera una mala complejidad (es decir, ineficiente) y debe evitarse si es posible. La complejidad lineal está bien y generalmente es un mal necesario. Logarítmico es bueno. ¡Constante es increíble!

Nota: dado que Big-O modela las relaciones de entrada a pasos, generalmente eliminamos las constantes de las expresiones. O(2n) es el mismo tipo de relación que O(n) - ambos son lineales-, por lo que podemos denotar ambos como O(n). Las constantes no cambian la relación.

Para tener una idea de cómo se calcula un Big-O, revisemos algunos ejemplos de complejidad constante, lineal y cuadrática.

COMPLEJIDAD CONSTANTE - O(c)

Se dice que la complejidad de un algoritmo es constante si los pasos necesarios para completar la ejecución de un algoritmo permanecen constantes, independientemente del número de entradas. La complejidad constante se denota por O(c) donde c puede ser cualquier número constante.

Escribamos un algoritmo simple en Python que encuentre el cuadrado del primer elemento de la lista y luego lo imprima en la pantalla:

```
def constant_algo(items):
    result = items[0] * items[0]
    print(result)

constant_algo([4, 5, 6, 8])
```

En el script anterior, independientemente del tamaño de entrada o la cantidad de elementos en los elementos de la lista de entrada, el algoritmo realiza solo 2 pasos:

Hallar el cuadrado del primer elemento

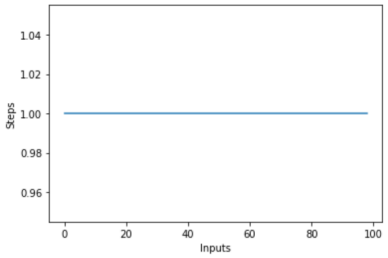
Imprimiendo el resultado en pantalla.

Por lo tanto, la complejidad permanece constante.

Si dibuja un diagrama de líneas con el tamaño variable de los elementos ingresados en el eje X y el número de pasos en el eje Y, obtendrá una línea recta. Vamos a crear un script corto para ayudarnos a visualizar esto. No importa el número de entradas, el número de pasos ejecutados sigue siendo el mismo:

```
steps = []
def constant(n):
    return 1

for i in range(1, 100):
    steps.append(constant(i))
plt.plot(steps)
```



COMPLEJIDAD LINEAL - O(n)

Se dice que la complejidad de un algoritmo es lineal si los pasos necesarios para completar la ejecución de un algoritmo aumentan o disminuyen linealmente con el número de entradas. La complejidad lineal se denota por O(n).

En este ejemplo, escribamos un programa simple que muestre todos los elementos de la lista en la consola:

```
def linear_algo(items):
    for item in items:
        print(item)

linear_algo([4, 5, 6, 8])
```

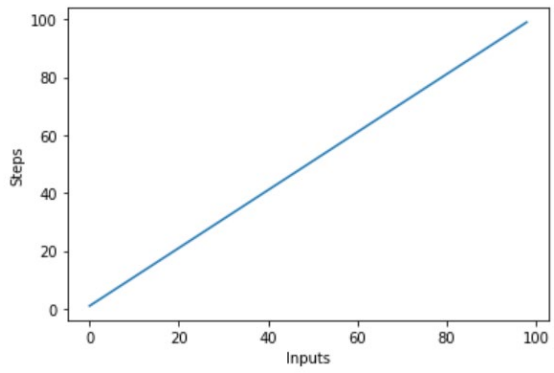
La complejidad de la función linear_algo() es lineal en el ejemplo anterior, ya que el número de iteraciones del bucle for será igual al tamaño de la matriz de elementos de entrada. Por ejemplo, si hay 4 elementos en la lista de elementos, el bucle for se ejecutará 4 veces.

Vamos a crear rápidamente una gráfica para el algoritmo de complejidad lineal con el número de entradas en el eje x y el número de pasos en el eje y:

```
steps = []
def linear(n):
    return n

for i in range(1, 100):
    steps.append(linear(i))

plt.plot(steps)
plt.xlabel('Inputs')
plt.ylabel('Steps')
```



COMPLEJIDAD LINEAL - O(n)

Una cosa importante a tener en cuenta es que, **con entradas grandes, las constantes tienden a perder valor**. Esta es la razón por la que generalmente eliminamos las constantes de la notación Big-O, y una expresión como O (2n) generalmente se abrevia a O (n). Tanto O(2n) como O(n) son lineales: lo que importa es la relación lineal, no el valor concreto. Por ejemplo, modifiquemos linear_algo():

```
def linear_algo(items):
    for item in items:
        print(item)

    for item in items:
        print(item)

linear_algo([4, 5, 6, 8])
```

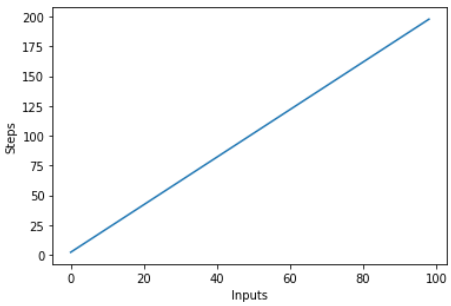
Hay dos bucles for que iteran sobre la lista de elementos de entrada. Por lo tanto, la complejidad del algoritmo se convierte en O(2n); sin embargo, en el caso de infinitos elementos en la lista de entrada, el doble de infinito sigue siendo igual a infinito. Podemos ignorar la constante 2 (ya que en última instancia es insignificante) y la complejidad del algoritmo sigue siendo O(n).

Visualicemos este nuevo algoritmo trazando las entradas en el eje X y el número de pasos en el eje Y:

```
steps = []
def linear(n):
    return 2*n

for i in range(1, 100):
    steps.append(linear(i))

plt.plot(steps)
plt.xlabel('Inputs')
plt.ylabel('Steps')
```



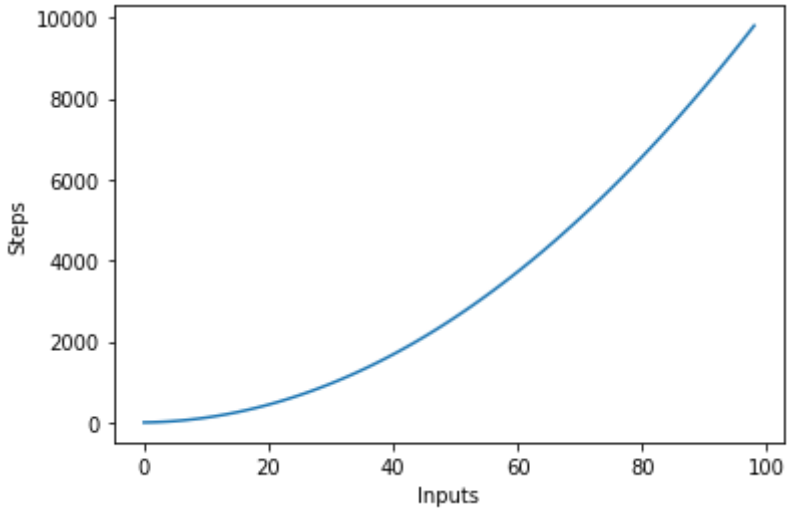
COMPLEJIDAD CUADRATICA - O(n^2)

Se dice que la complejidad de un algoritmo es cuadrática cuando los pasos requeridos para ejecutar un algoritmo son una función cuadrática del número de elementos en la entrada. La complejidad cuadrática se denota como O(n²):

```
def quadratic_algo(items):
    for item in items:
        for item2 in items:
            print(item, ' ' ,item2)

quadratic_algo([4, 5, 6, 8])
```

Tenemos un loop externo que itera a través de todos los elementos de la lista de entrada y luego un loop interno anidado, que nuevamente itera a través de todos los elementos de la lista de entrada. El número total de pasos realizados es n*n, donde n es el número de elementos en la matriz de entrada.



COMPLEJIDAD LOGARITMICA - O(log n)

Algunos algoritmos logran una **complejidad logarítmica**, como la búsqueda binaria. La búsqueda binaria busca un elemento en una lista, comprobando la mitad de una lista y eliminando la mitad en la que no se encuentra el elemento. Lo vuelve a hacer para la mitad restante y continúa con los mismos pasos hasta encontrar el elemento. En cada paso, reduce a la mitad el número de elementos de la lista. Por ejemplo, si la lista tiene 16 elementos, después de una iteración, el algoritmo solo considera 8 elementos, luego 4, luego 2, y finalmente 1. El número de iteraciones necesarias para reducir la lista a un solo elemento es proporcional al logaritmo base 2 del número de elementos en la lista.

Esto requiere que la lista esté ordenada y que hagamos una suposición sobre los datos (en este caso, que están ordenados).

Cuando puede hacer suposiciones sobre los datos entrantes, puede tomar medidas que reduzcan la complejidad de un algoritmo. La complejidad logarítmica es deseable, ya que logra un buen rendimiento incluso con entradas muy escaladas.

```python
def busqueda_binaria(lista, objetivo):
    """
    Realiza una búsqueda binaria en una lista ordenada.

    Parámetros:
    lista (list): Una lista ordenada de elementos.
    objetivo: El elemento a buscar en la lista.

    Retorna:
    int: El índice del elemento si se encuentra en la lista, de lo contrario -1.
    """
    inicio = 0
    fin = len(lista) - 1

    while inicio <= fin:
        medio = (inicio + fin) // 2
        if lista[medio] == objetivo:
            return medio
        elif lista[medio] < objetivo:
            inicio = medio + 1
        else:
            fin = medio - 1

    return -1

# Ejemplo de uso
lista_ordenada = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
objetivo = 7
resultado = busqueda_binaria(lista_ordenada, objetivo)

if resultado != -1:
    print(f"Elemento encontrado en el índice {resultado}.")
else:
    print("Elemento no encontrado en la lista.")
```

COMPLEJIDAD DE FUNCIONES COMPLEJAS

¿Encontrar la complejidad de las funciones complejas?

En ejemplos anteriores, teníamos funciones bastante simples en la entrada. Sin embargo, ¿cómo calculamos el Big-O de las funciones que llaman (múltiples) otras funciones en la entrada?

Vamos a ver:

```
def complex_algo(items):

    for i in range(5):
        print("Python is awesome")

    for item in items:
        print(item)

    for item in items:
        print(item)

    print("Big O")
    print("Big O")
    print("Big O")

complex_algo([4, 5, 6, 8])
```

En el script anterior, se realizan varias tareas, primero, se imprime una cadena 5 veces en la consola usando la declaración de impresión. A continuación, imprimimos la lista de entrada dos veces en la pantalla y, finalmente, se imprime otra cadena tres veces en la consola. Para encontrar la complejidad de dicho algoritmo, necesitamos dividir el código del algoritmo en partes e intentar encontrar la complejidad de los bloques individuales.

COMPLEJIDAD DE FUNCIONES COMPLEJAS

En la primera sección tenemos:

```
for i in range(5):
    print("Python is awesome")
```

La complejidad de esta parte es O (5) ya que se realizan cinco pasos constantes en esta pieza de código independientemente de la entrada.

A continuación, tenemos:

```
for item in items:
    print(item)
```

Sabemos que la complejidad del código anterior es O(n). De manera similar, la complejidad de la siguiente pieza de código también es O(n):

```
for item in items:
    print(item)
```

Finalmente, en el siguiente fragmento de código, una cadena se imprime tres veces, por lo que la complejidad es O (3):

```
print("Big O")
print("Big O")
print("Big O")
```

Para encontrar la complejidad general, simplemente tenemos que agregar estas complejidades individuales:

```
O(5) + O(n) + O(n) + O(3)
```

```
O(8) + O(2n) = O(8+2n)
```

Dijimos anteriormente que cuando la entrada (que tiene una longitud n en este caso) se vuelve extremadamente grande, las constantes se vuelven insignificantes, es decir, el doble o la mitad del infinito sigue siendo infinito. Por lo tanto, podemos ignorar las constantes. La complejidad final del algoritmo será O(n).

COMPLEJIDAD: Worst and Best case

Por lo general, cuando alguien pregunta sobre la complejidad de un algoritmo, está interesado en la complejidad del peor de los casos (Big-O). A veces, también pueden estar interesados en la complejidad del mejor de los casos (Big-Omega).

Para comprender la relación entre estos, revisemos el siguiente bloque de código:

```
def search_algo(num, items):
    for item in items:
        if item == num:
            return True
        else:
            pass
nums = [2, 4, 6, 8, 10]

print(search_algo(2, nums))
```

En el script anterior, tenemos una función que toma un número y una lista de números como entrada. Devuelve verdadero si el número pasado se encuentra en la lista de números, de lo contrario, devuelve Ninguno. Si busca 2 en la lista, se encontrará en la primera comparación. Esta es la complejidad del mejor de los casos del algoritmo en el que el elemento buscado se encuentra en el primer índice buscado. La complejidad del mejor caso, en este caso, es O (1). Por otro lado, si busca 10, se encontrará en el último índice buscado. El algoritmo tendrá que buscar en todos los elementos de la lista, por lo que la complejidad del peor de los casos se convierte en O(n).

Nota: La complejidad del peor de los casos sigue siendo la misma, incluso si intenta encontrar un elemento inexistente en una lista: se necesitan n pasos para verificar que no existe tal elemento en una lista. Por lo tanto, la complejidad del peor de los casos sigue siendo O(n).

Además de la complejidad del mejor y el peor de los casos, también puede calcular la complejidad promedio (Big-Theta) de un algoritmo, que le dice "dada una entrada aleatoria, ¿cuál es la complejidad de tiempo esperada del algoritmo"?

COMPLEJIDAD ESPACIAL

Además de la complejidad del tiempo, donde cuenta la cantidad de pasos necesarios para completar la ejecución de un algoritmo, también puede encontrar la complejidad del espacio, que se refiere a la cantidad de espacio que necesita asignar en la memoria durante la ejecución de un programa.

Revisemos el siguiente ejemplo:

```python
def return_squares(n):
    square_list = []
    for num in n:
        square_list.append(num * num)

    return square_list

nums = [2, 4, 6, 8, 10]
print(return_squares(nums))
```

La función return_squares() acepta una lista de enteros y devuelve una lista con los cuadrados correspondientes. El algoritmo tiene que asignar memoria para el mismo número de elementos que en la lista de entrada. Por lo tanto, la complejidad espacial del algoritmo se convierte en O(n).

P vs NP

El problema P vs NP es uno de los grandes problemas en el mundo de las matemáticas aplicadas y la informática, o más precisamente en la teoría de la complejidad computacional. A pesar de que sus bases teóricas se remontan a Alan Turing, fue planteado paralelamente en los años 70, por los programadores Stephen Cook y Leonid Levin, convirtiéndose en uno de los “problemas del milenio” del famoso Instituto Clay de Matemáticas, que ofrece una recompensa de un millón de dólares a quien pueda solucionarlo. Sin embargo, ¿de qué trata el problema y cuál es su complejidad? ¿por qué es relevante o importante? ¿existe forma de resolverlo?

La complejidad del problema del milenio P vs NP

El problema P vs NP es una de las preguntas más importantes en el campo de las ciencias de la computación, debido a las grandes repercusiones que habría, en caso de encontrarse una solución.

Su complejidad, está relacionada con la TCC (teoría de la complejidad computacional), puesto que esta teoría -en sentido bastante simplificado-, persigue clasificar los problemas en aquellos que pueden o no ser resueltos con una cantidad determinada de recursos.

Estos recursos para efectos de dicha teoría son, tiempo y memoria. Es decir, el tiempo empleado durante determinado cálculo para resolver un problema dado y la memoria requerida para almacenar y procesar los datos del problema.

En nuestro mundo actual, sería sencillo suponer por intuición o lógica, que todos los problemas que se presenten a un ordenador pueden ser resueltos a pesar de cuán complejos sean. Pero, ¿es realmente así?

Resulta que, en las matemáticas, al igual que en la informática, existen problemas que pueden o no tener solución, así mismo, en caso de existir alguna solución deben ser verificados, y esta requiere una utilización eficiente de recursos (tiempo y memoria) a través de algoritmos óptimos, donde preferiblemente no se disponga de millones de años para solucionarlos.

Una forma práctica de imaginar el universo computacional y su complejidad, así como su relación con el problema P vs NP (que explicaremos más adelante) es intentar ordenar y resolver un puzzle (rompecabezas). El puzzle aumentará su complejidad en la medida que se añadan más y más piezas, y por tanto el tiempo empleado para resolverlo podría llegar a ser abrumador; ¿pero, si existiera un algoritmo capaz de ordenar aquellas piezas en la menor cantidad de tiempo (tiempo polinómico = podría interpretarse como rapidez)? - ¿sería eso una solución al problema del puzzle? -Y ¿cómo lo comprobaríamos? - Su verificación para este

ejemplo quizás sea más sencilla, ver un puzzle completo y resuelto es más simple que ordenar cada una de sus piezas; o como opinaba Stephen Cook… “hay problemas que sí pueden ser resueltos por un ordenador, solo que la máquina tardaría tanto, que el sol moriría antes”.

Así que el problema P vs NP, es algo más que un rompecabezas matemático abstracto. Su objetivo es determinar qué tipos de problemas se pueden resolver con ordenadores y cuáles no.

Si diluimos esta relación, encontraremos que los problemas de clase “P” (de sus siglas en inglés “tiempo polinomial), son “fáciles” de resolver para los ordenadores, es decir, las soluciones a estos problemas pueden ser calculadas en una cantidad de tiempo razonable, en comparación con la complejidad del problema. Por ejemplo, el problema del ordenamiento, es un problema P, todos podemos resolverlo en un tiempo considerablemente rápido (orden polinómico). Así que, aquellos problemas que podemos solucionar y verificar con un algoritmo de orden polinómico, lo llamaremos problema P.

No sucede así, para su entrañable relación con “NP” (de sus siglas en inglés “tiempo no determinista polinomial”), aquí su solución podría ser muy “difícil” de encontrar, quizá requeriría miles de millones de años de computación, pero una vez encontrada, es fácil de comprobar.

Los problemas de la clase NP (Tiempo Polinomial No Determinista) se encuentran en un estado particular dentro de la informática teórica. Estos problemas son aquellos cuyas soluciones son difíciles de encontrar, pero fáciles de verificar una vez encontradas. No se sabe si todos los problemas NP tienen soluciones que puedan ser encontradas en tiempo polinomial (es decir, si NP = P), y esta es una de las grandes preguntas abiertas en la ciencia de la computación.

Naturaleza de los Problemas NP

Difíciles de Resolver: Encontrar una solución puede requerir un tiempo exponencial en el peor de los casos, lo que significa que, para grandes entradas, los tiempos de cálculo pueden ser astronómicamente largos.

Fáciles de Verificar: Si se proporciona una solución, se puede verificar su validez en tiempo polinomial.

P vs NP

Ejemplos y Aplicaciones de Problemas NP

1. Cifrado en Criptografía:

Seguridad: Muchos sistemas criptográficos se basan en problemas NP. Por ejemplo, la factorización de números grandes es un problema NP, y la dificultad de resolverlo garantiza la seguridad de métodos como RSA.

Clave Privada y Pública: La dificultad de resolver estos problemas sin la clave adecuada hace que los sistemas de criptografía de clave pública sean seguros.

2. Modelos de Previsión Financiera:

Optimización de Carteras: Encontrar la combinación óptima de inversiones para maximizar el retorno y minimizar el riesgo puede ser un problema NP, ya que requiere evaluar una gran cantidad de posibles combinaciones de activos.

Análisis de Riesgo: Modelar escenarios financieros y prever riesgos futuros también puede involucrar problemas NP debido a la complejidad y cantidad de variables involucradas.

3. Pliegue de Proteínas:

Biología Computacional: Determinar la estructura tridimensional de una proteína a partir de su secuencia de aminoácidos es un problema NP-completo, debido a la enorme cantidad de posibles configuraciones.

Medicina y Farmacología: Comprender el pliegue de proteínas es crucial para el diseño de medicamentos y tratamientos.

4. Juegos y Rompecabezas (Sudoku):

Sudoku: Resolver un Sudoku implica encontrar una disposición de números que satisfaga todas las restricciones del juego. Mientras que verificar una solución es sencillo, encontrarla puede ser complicado y es un ejemplo clásico de un problema NP-completo.

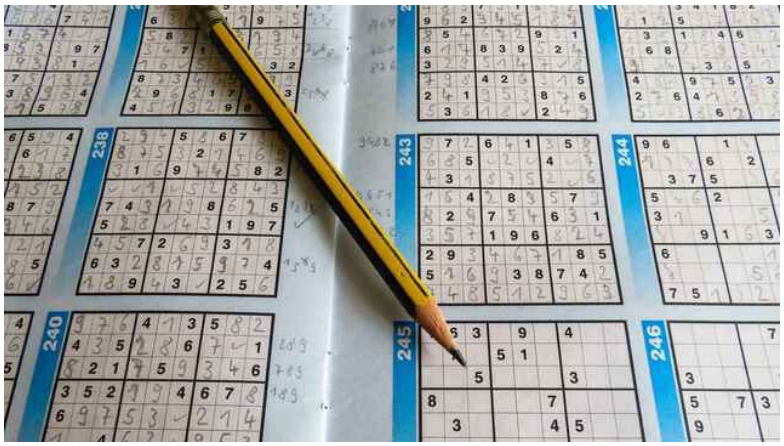
Otros Juegos: Muchos otros juegos de lógica y rompecabezas también son NP-completos, lo que los hace interesantes y desafiantes.

5. Optimización de Horarios (Horarios de Vuelos Aéreos):

P vs NP

Planificación de Vuelos: Crear un horario eficiente que minimice los tiempos de espera y maximice el uso de los recursos (como aviones y tripulaciones) puede ser extremadamente complejo y es un problema NP.

Algoritmos de Optimización: Se utilizan algoritmos avanzados para encontrar soluciones aproximadas en un tiempo razonable.



Luego está una relación aún más compleja, los problemas NP-Completos, que vienen siendo una especie de subclase a los NP (es decir, están contenidos en la clase NP), estos se consideran problemas llave, porque si se encuentra una manera de resolver eficientemente solo uno (01) de ellos, significa que todos -todos los NP– pueden resolverse eficientemente. Así que los ordenadores actuales, tomarían especies de “atajos” para la resolución de dichos problemas, en caso de comprobarse su igualdad.

Ejemplos como el problema del viajero, persiguen explicar con mejor suficiencia este embrollo. Imagina que eres un comerciante que debe entregar paquetes de mercancías en varias ciudades -conociendo la distancia entre ellas- y necesitas la ruta más corta y eficiente para visitarlas una sola vez, sin repetir y volver a tu punto de origen. ¿Cuántas posibles rutas puedes calcular y en cuánto tiempo? ¿cuál es la más eficiente? En dicho ejemplo, las respuestas a las preguntas planteadas se pueden verificar, pero requieren un tiempo ridículamente largo e imposible para resolverlas mediante cualquier procedimiento directo.

Podríamos empezar por calcular cada una de las posibilidades, pero eso llevaría mucho tiempo, como ya hemos detallado suficientemente.

El problema P vs NP cuestiona si los problemas que podemos verificar rápidamente también pueden resolverse rápidamente. Resolver esta cuestión tendría profundas implicaciones en diversas áreas de la ciencia y la tecnología, desde la seguridad de la información hasta la

P vs NP

eficiencia de las soluciones a problemas complejos. La búsqueda de una respuesta a esta pregunta sigue siendo uno de los mayores desafíos en la teoría de la computación.

El acervo científico actual nos dice que P es diferente a NP, pero no existe demostración alguna que corrobore ese planteamiento o lo contradiga. A no ser, que se empleen una serie de pasos, como un algoritmo lo suficientemente capaz de solucionar y demostrar que P es igual a NP. ¿Será esto posible?

ANEXO: Recursos

Libros

"Python for Data Analysis" de Wes McKinney

Este libro es una excelente introducción a la manipulación de datos con Python, utilizando bibliotecas como Pandas, NumPy y IPython.

"Data Science from Scratch" de Joel Grus

Este libro cubre los fundamentos de la ciencia de datos y te enseña cómo implementar algoritmos clave desde cero en Python.

"Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow" de Aurélien Géron

Un libro práctico que cubre desde los conceptos básicos de aprendizaje automático hasta técnicas más avanzadas utilizando Scikit-Learn, Keras y TensorFlow.

"R for Data Science" de Hadley Wickham y Garrett Grolemund

Este libro es ideal si prefieres aprender ciencia de datos usando R. Cubre desde la importación y limpieza de datos hasta la visualización y modelado.

"Introduction to Statistical Learning" de Gareth James, Daniela Witten, Trevor Hastie y Robert Tibshirani

Un libro que ofrece una introducción accesible a los conceptos de aprendizaje estadístico y técnicas de modelado.

Cursos en Línea

Coursera: "Data Science Specialization" de Johns Hopkins University

Una serie de cursos que cubren desde la programación en R hasta la ejecución de proyectos de ciencia de datos.

Coursera: "Machine Learning" de Stanford University (Andrew Ng)

Un curso introductorio a los conceptos y algoritmos de aprendizaje automático.

edX: "Data Science MicroMasters" de MIT

ANEXO: Recursos

Un programa de MicroMasters que cubre una amplia gama de temas de ciencia de datos, desde probabilidades y estadísticas hasta aprendizaje automático.

Kaggle: "Learn Data Science"

Una serie de tutoriales interactivos y prácticos que cubren temas como Python, visualización de datos, machine learning, entre otros.

Udacity: "Data Scientist Nanodegree"

Un programa intensivo que abarca los fundamentos de la ciencia de datos, incluyendo análisis de datos, visualización, machine learning y despliegue de modelos.

Recursos Adicionales

Khan Academy: "Probability and Statistics"

Una excelente fuente para aprender los fundamentos de probabilidad y estadística, esenciales para la ciencia de datos.

HarvardX: "Data Science: R Basics"

Un curso introductorio que enseña los conceptos básicos de programación en R y análisis de datos.

DataCamp: "Data Scientist with Python"

Una serie de cursos enfocados en Python que cubren desde los fundamentos hasta técnicas avanzadas de ciencia de datos.

YouTube: "StatQuest with Josh Starmer"

Un canal educativo que explica conceptos estadísticos y de machine learning de una manera fácil de entender.

Plataformas de Práctica

Kaggle

Kaggle ofrece datasets, competiciones y tutoriales para practicar y mejorar tus habilidades en ciencia de datos.

ANEXO: Recursos

DataCamp

DataCamp proporciona cursos interactivos y proyectos prácticos para aprender y aplicar ciencia de datos.

LeetCode

Aunque es más conocida por la preparación para entrevistas de programación, LeetCode también ofrece problemas relacionados con ciencia de datos y machine learning.