En esta clase veremos uno de los temas más importantes sobre algoritmos y estructuras de datos.

Ese tema es el beaucoup.

Qué es el pico u el pico?

Es simplemente una métrica que se utiliza en ingeniería del software y que sirve para analizar la eficiencia

en nuestros algoritmos.

Es un tema básico para determinar cómo se comportarán nuestros algoritmos cuando los sometamos a cargas

de trabajo altas.

Dominar este tema es esencial para hacerlo bien en las entrevistas técnicas, pero también es importante

para ser un gran desarrollador.

Si desconoces este término, es muy probable que implementen algoritmos poco eficientes o menos eficientes

de lo que podrían ser, lo que te afectará cuando tu sistema tenga que soportar muchos usuarios o carga

de trabajo.

Por lo tanto, antes de pasar a los siguientes módulos, asegúrate de que dominas el concepto de BBCode.

Vamos a explicar el término haciendo uso de una analogía.

Imagínate que quieres obtener toda la colección de juegos de tu consola favorita.

Para ello hay dos opciones comprar y descargar la versión digital de los juegos o comprar la versión

física y esperar a que el paquete llegue a tu casa.

Si quieres jugar a un juego lo más rápido posible, la opción que elegiría sería la primera, ya que

podrás tener el juego en unos minutos o unas horas dependiendo de la velocidad de descarga.

En cambio, el envío tardaría unos días.

Pero qué pasaría si queremos tener una cantidad enorme de juegos?

Llegará un momento en que las descargas tardarán semanas en completarse, mientras que el envío con

todos los juegos seguiría tardando unos días en llegar.

Esto es lo que se conoce como tiempo de ejecución asintótica o Big ou.

Por una parte, tenemos el tiempo que nos llevará a obtener una cantidad determinada de juegos en formato

digital.

De esta forma, el tiempo que nos llevará a tener los juegos viene determinado por la cantidad de juegos

que queramos.

A más juegos, más datos que necesitamos descargar, por lo que nos llevará más tiempo.

Esto se representa como DJ o DJ, siendo J el número de juegos.

Por otra parte, tenemos la opción de la compra de juegos en formato físico.

Como hemos mencionado, esto no depende del número de juegos que compres.

Da igual si compras un juego o mil juegos, todos se enviarán y llegarán a tu casa en tiempo constante.

Más o menos, evidentemente.

Esto se representa como o D1 con una muestra muy pequeña o de DJ supera en tiempo de ejecución a o de

uno.

Pero eso no es lo que nos interesa para determinar la eficiencia de un algoritmo.

La eficiencia de un algoritmo vendrá dada por su comportamiento en los extremos, por el tiempo de ejecución

asintótica o backbone.

Como veremos a continuación, estos no son los únicos tiempos de ejecución posibles.

Los más comunes son o de log de n o de n loc n o de n o de en el cuadrado o de dos elevado a n.

Los trataremos en detalle en un rato.

El tiempo no es lo único importante.

Cuando queremos analizar un algoritmo, también puede ser muy relevante la cantidad de memoria utilizada.

Normalmente se hacen compromisos con respecto a la complejidad temporal y espacial, ya que es habitual

que la mejora de uno implique empeorar la otra.

Por lo tanto, el objetivo debe ser encontrar un buen balance en ambas complejidades y elegir la que

más nos convenga dadas nuestras condiciones.

Pongamos un ejemplo.

Imaginemos que tenemos una lista de personas sin ningún orden concreto.

A continuación, nos irán pidiendo que dado un país, devolvamos todas las personas pertenecientes a

ese país.

Lo primero que puedes estar pensando es utilizar la fuerza bruta.

La opción más simple.

Creamos una lista temporal vacía.

Recorremos toda nuestra lista de personas.

Y si la persona actual pertenece al país que estamos buscando, la añadimos a la lista.

Al acabar, devolvemos la lista con el conjunto de personas.

Por ejemplo, tenemos aquí una lista de personas y queremos encontrar a las personas que sean de España.

Recorremos 1 a 1 los elementos de la lista.

El primero, como vemos, es de España.

Entonces lo añadimos a nuestra lista.

El segundo es de Italia.

Nos lo saltamos.

El tercero de Reino Unido nos lo saltamos.

Y la última es de España, por lo que la añadimos a la lista y devolvemos nuestro resultado.

Este algoritmo tiene una complejidad lineal, es decir, ADN, ya que necesitamos visitar todas las

personas del array para encontrar las que tienen el país que nos solicitan.

Este problema se podría plantear de otra forma, utilizando una estructura de datos más apropiada para

este caso, como son las tablas hash.

Para el que no sepa lo que es una tabla hash es simplemente una estructura que almacena pares de clave

a valor.

Dada una clave te proporciona un acceso muy rápido en tiempo constante al valor asociado a dicha clave.

Piensa en qué pasaría si hacemos un pre procesamiento de la lista de personas creando un hash map que

tenga como clave el nombre del país de la persona y como valor la lista de personas de ese país, Tendríamos

que recorrer la lista completa una única vez al inicio, añadiendo al hash map cada persona según su

país.

A continuación, no importa cuántas solicitudes de países tengamos, podríamos acceder a la lista de

personas de un país concreto de forma constante o de uno, ya que las tablas nos ofrecen este direccionamiento

directo.

No importa cuál sea el tamaño de nuestro conjunto de personas, nuestra función de búsqueda siempre

tardará lo mismo en encontrar la lista de personas que necesitamos.

Esto contrasta con el caso anterior, el cual tardaría más según la lista de

personas sea mayor, vaya

creciendo.

Por otra parte, el tener que almacenar la lista de personas en una tabla nos obliga a hacer un procesamiento

y a necesitar o DN memoria adicional.

Como vemos, haciendo un compromiso con la complejidad espacial, conseguimos una mejora en la complejidad

temporal.

Mencionar también que para analizar la complejidad espacial de nuestros algoritmos tan solo contamos

con la memoria adicional necesaria para ejecutarlos.

No se deben contar los parámetros de entrada a nuestro algoritmo.

Por eso, a pesar de que en ambos tengamos una lista desordenada de personas, en el primer caso la

complejidad espacial es constante o de uno, ya que la lista es uno de nuestros parámetros de entrada.

Mientras que en este caso.

En el segundo caso es lineal o DN.

Ya que necesitamos almacenar un mapa mayores y este no es un parámetro de entrada.

Por lo tanto, de forma resumida, el bigote describe el ritmo con el que crece la complejidad o el

tiempo de ejecución de un algoritmo.

Como hemos visto previamente, para input seleccionado es de forma específica o DN es más rápido que

de uno.

La complejidad lineal es más rápida que la constante para ciertos inputs.

En el caso de los juegos de la consola.

Pero eso no es relevante.

Lo que nos interesa saber es cómo se comportará nuestro algoritmo en el extremo.

Es por eso que las constantes no son relevantes y se deben eliminar en el cálculo del pico.

Veamos el siguiente ejemplo Podemos ver en la imagen de la izquierda un bucle que recorre un array de

enteros, calculando una suma y un producto de forma acumulativa.

Podemos deducir de forma muy rápida que este algoritmo tiene una complejidad temporal de ADN lineal,

siendo n el tamaño del array, ya que tenemos que pasar una vez por cada uno de los elementos del array.

Si el tamaño de este array crece, la complejidad de nuestro algoritmo, el tiempo de ejecución del

mismo crecerá de forma lineal.

Si crece el tamaño del array, la complejidad de nuestro algoritmo, su tiempo de ejecución crecerá

de forma lineal.

Veamos ahora la imagen de la derecha.

Como podemos ver aquí, lo que hacemos es separar el procesamiento en dos bucles for uno para la suma

y otro para el producto.

Es muy fácil salir rápidamente con que este algoritmo tiene una complejidad o de dos por n, ya que

nuestro algoritmo recorre el array dos veces.

Pero esto no es así, ya que el número de instrucciones no influye en la complejidad algorítmica.

La complejidad algorítmica no mide eso.

Mide la forma en la que crece el tiempo de ejecución de un algoritmo según los valores de entrada.

Veamos en una gráfica el porqué las constantes se deben eliminar del pico.

Podemos ver representadas o DN o de cinco n y o de dos elevado a n.

En primer lugar, comentar que la constante que podemos ver es la del cinco n, el

dos en o de dos elevado

a n no es una constante.

Se trata de una complejidad exponencial.

Como podemos ver, el ritmo al que crecen el n y el cinco n va a ser siempre superado por una complejidad

mayor como puede ser la exponencial.

No importa si estamos ante un o de n o de cinco n o de 100 millones n siempre va a ser superado, ya

que estas complejidades crecen linealmente según crece?

N, mientras que la otra crece exponencialmente.

O de n i o de cinco n crecen siempre al mismo ritmo, forman la misma figura, una línea recta mientras

que o de dos elevado a n forma una parábola.

Lo mismo pasaría si tomamos el caso de o de dos elevado a n y o de tres elevado a n son equivalentes

en complejidad, ya que ambas son exponenciales y una complejidad mayor como es la factorial va a acabar

superando las siempre.

No importa el tamaño de esa constante.

Es muy importante quedarse con este concepto, ya que es muy común confundir la complejidad de un algoritmo.

Si no lo tenemos claro, lo importante es el ritmo de crecimiento, no el número de instrucciones ejecutadas.

Veamos ahora otro tema los términos no dominantes.

En este código de ejemplo podemos ver que se itera una vez el array en el primer bucle para obtener

la suma total.

Como hemos visto, esto sería una complejidad lineal o DN.

A continuación tenemos dos bucles for anidados para cada elemento del array.

Recorremos otros n elementos.

Por lo tanto, la complejidad aquí sería o de n por n o lo que es lo mismo, o de n al cuadrado.

Si queremos obtener el pico del algoritmo completo, podríamos pensar que sería la suma de ambas complejidades

dándonos un ADN más en el cuadrado.

Pero piénselo de esta forma Hemos comentado previamente que si tenemos una complejidad de o de dos por

n al cuadrado, es decir, o de n al cuadrado más n al cuadrado, debemos eliminar la constante, ya

que no aporta nada en cuanto al crecimiento de la complejidad.

Si no nos interesa ese n al cuadrado extra, por qué nos va a interesar el n de o de n más en el cuadrado?

La respuesta es obvia No nos interesa el ritmo con el que crece la complejidad de un algoritmo viene

dado por el término más dominante.

Si queremos reducir la complejidad de nuestro algoritmo, debemos centrarnos en optimizar la parte dominante,

ya que es la que de verdad nos aportará un grandísimo beneficio.

Optimizar.

Con todo lo que hemos visto anteriormente sobre eliminar las constantes y los términos no dominantes,

es muy común caer en un típico error al analizar la complejidad de un algoritmo.

El caso en el que tenemos múltiples inputs de tamaños diferentes.

Observa el algoritmo que puedes ver en la imagen.

A primera vista podrías ver que son dos bucles for sin nada más anidado, por lo que la complejidad

es o DN.

Pero esto no es así, ya que el array a y el array b pueden tener tamaños muy diferentes y la complejidad

del algoritmo depende de ambos tamaños, los cuales no son constantes.

Uno no es igual al otro.

Es por eso que la complejidad se debe expresar en términos de o de a más b, siendo A el tamaño del

array a y b el tamaño del array b.

Es muy importante tener en cuenta estos casos especiales con múltiples inputs.

Aquí podemos ver los huecos o complejidades más comunes y su crecimiento en una gráfica.

Como podemos observar.

O de uno es el mejor.

La complejidad constante seguida por la complejidad logarítmica o de log n.

Estas dos son complejidades muy buenas y por lo tanto normalmente difíciles de conseguir en algoritmos

complejos.

A continuación viene la complejidad lineal o DN y la complejidad o de n por logaritmo de n, que son

complejidades decentes, ya que el ritmo de crecimiento no es desorbitado.

Por último, tenemos las complejidades bastante malas o muy malas que queremos intentar evitar a toda

costa, ya que los algoritmos con estas complejidades tendrán un mal comportamiento cuando sean sometidos

a un gran número de datos de entrada.

Estas son la complejidad cuadrática o de n al cuadrado, la exponencial que es o de dos elevado a n

y la peor de todas, la factorial o de n factorial.

Para terminar con la parte teórica de este tema, veamos un ejemplo simple de cada una de ellas.

Aquí podemos ver un ejemplo de un algoritmo con complejidad constante.

Es una función que dados dos valores enteros, retorna el máximo de ambos.

Esta función va a tardar siempre lo mismo.

No depende de los datos de entrada y es por eso que tiene una complejidad constante o de uno.

Pasemos ahora a un ejemplo de complejidad logarítmica.

El ejemplo más típico es el del algoritmo de búsqueda binaria, que veremos en profundidad más adelante

en este curso.

Lo que hace es aprovecharse del hecho de que un array de elementos esté ordenado en un caso normal.

Si el array no está ordenado y queremos encontrar un elemento, debemos recorrer todo el array elemento

elemento hasta encontrarlo o llegar al final para saber que no se encuentra.

La búsqueda binaria lo que hace es situarse a la mitad del array ordenado.

Si el elemento que queremos encontrar es mayor que el elemento central, quiere decir que se tiene que

encontrar en la parte derecha del array.

En caso contrario, se encuentra en la parte izquierda.

De esta forma, en cada iteración eliminamos la mitad del conjunto de datos, haciendo muchísimas menos

iteraciones para encontrar nuestro elemento.

Veamos un ejemplo.

Queremos saber si nuestro array está en el elemento 15, como es un array ordenado de menor a mayor.

Podemos aplicar la búsqueda binaria.

Elegimos el elemento del medio.

El ocho.

Como 15 es mayor que ocho y el array está ordenado.

Ya sabemos que si el elemento 15 está en el array tiene que estar a la derecha del ocho, por lo que

podemos desechar la mitad del array, la mitad izquierda.

Repetimos el proceso.

Como el array ahora tiene elementos pares, elegimos uno de los elementos centrales, en este caso el

18.

Como 15 es menor a 18.

Sabemos que si el elemento 15 está en el array, tiene que estar a la izquierda del 18, así que podemos

desechar la parte derecha.

De nuevo repetimos el proceso y encontramos al 15.

Como vemos en cada paso, nuestro conjunto de datos se divide entre dos.

Esto es lo que representa el logaritmo.

Si tenemos un n igual a 16, necesitamos, en el peor de los casos, cuatro pasos para encontrar o no

un elemento en el array.

Este número en los cuatro pasos coincide con el logaritmo en base dos de 16.

Esto es, a qué número debo elevar dos para obtener 16?

Este número es el cuatro.

Por eso se dice que este tipo de algoritmos tiene complejidad logarítmica.

Si ves un algoritmo en el que en cada paso tu conjunto de datos se divide entre dos, tu cerebro debe

hacer clic para asociarlo con la complejidad logarítmica.

Veamos ahora otro ejemplo de complejidad lineal para poner en contraste el ejemplo de la búsqueda binaria.

Vamos a ver ahora un ejemplo sencillo de búsqueda lineal.

Dada una función que recibe un array de empleados, queremos buscar al primer empleado que coincida

con un nombre dado.

Para ello se recorre el array de empleados comparando el nombre del empleado actual con el nombre a

buscar.

Si lo encontramos se devuelve, en caso contrario se devuelve null.

Al finalizar con todos los elementos.

Como vemos en este algoritmo, en el peor de los casos se debe recorrer el array entero para buscar

a un empleado, por lo que la complejidad es o dn.

En contraste a la búsqueda binaria que iba reduciendo el tamaño del input de entrada a la mitad en cada

iteración.

Pasemos ahora a la complejidad o de n por logaritmo de n.

Esta es la complejidad que tienen los algoritmos de ordenamiento más eficientes, como son el Merge,

Sort, Absurd y Cuixart.

Estos algoritmos son bastante complejos para explicar en este tema y los veremos en detalle más adelante

en el curso.

Quédate con que si queremos ordenar un array, la mejor o complejidad que podemos obtener es esta o

de n logaritmo de n.

Otro caso de algoritmo con esta complejidad sería, por ejemplo, realizar n veces una búsqueda binaria,

la cual recordemos es o de logaritmo de n.

Veamos ahora la complejidad cuadrática o de n por n o de n al cuadrado.

Aquí podemos ver el código para imprimir todas las parejas de valores que se pueden formar en un array.

Para cada valor del array debemos recorrer el array completo otra vez.

Es decir, para cada valor de n, siendo n la longitud del array, debemos recorrer otros elementos.

Esto nos da n por n, que es igual a Obtén el cuadrado.

Sencilla.

Pasemos a la penúltima complejidad más común la exponencial.

Para ello vamos a poner un ejemplo poco eficiente de cómo calcular la secuencia de Fibonacci.

Para los que no sepan lo que es esta secuencia, es simplemente una fórmula matemática que nos dice

que para n igual a cero el Fibonacci es igual a cero, para n igual a uno el Fibonacci es igual a uno

y para n mayor o igual que dos.

El Fibonacci de N es igual al Fibonacci del número anterior y del número anterior al anterior.

Por lo tanto, el Fibonacci de dos, por ejemplo, es igual al de cero y el de uno, es decir, cero

más uno uno.

El Fibonacci de tres sería igual al Fibonacci de dos y el Fibonacci de uno, es decir, uno más uno,

dos y así sucesivamente.

Esto se puede reflejar en el código que vemos a continuación.

Con una función recursiva podemos ver que si es menor que dos el Fibonacci scene, es decir cero uno,

retornamos ese valor directamente.

En caso contrario llamamos a la función de manera recursiva con n -1 y n -2.

Veamos ahora las llamadas que se realizan para un ejemplo pequeñito, el Fibonacci de cuatro.

Vemos que el Fibonacci de cuatro es igual al Fibonacci de tres y al de dos.

Por su parte, el Fibonacci de tres es igual al Fibonacci de dos y al de uno, el de uno lo retornamos

directamente, ya que es uno y el de dos.

Tenemos que volver a realizar una llamada recursiva al Fibonacci de uno y al de cero por la otra parte

del procesamiento del Fibonacci de dos.

Volvemos a calcularlo con Fibonacci de uno más uno h de cero.

Si nos fijamos bien, podemos sacar que en cada nodo salen dos llamadas recursivas.

En el primer nivel tenemos una única llamada.

En el segundo nivel, dos.

En el tercero cuatro.

Y podríamos seguir así.

Pero por cuestión de simplicidad, el ejemplo ya finaliza aquí.

Esto es equivalente a decir que el primer nivel tiene dos elevado a cero nodos, el segundo dos elevado

a uno, el tercero dos elevado dos y el último dos elevado a tres.

Bien, no me quiero centrar mucho en las demostraciones matemáticas porque no va a aportar nada útil.

Pero quédate, que si tenemos una estructura en la que en cada paso se duplican las llamadas hasta llegar

al final, tendríamos la siguiente suma de dos elevado a cero, más dos elevado a uno más dos elevado

a dos más, llegando al final dos elevado a N -1, y esto es equivalente a dos elevado a N complejidad

exponencial.

De nuevo, lo importante no es la demostración matemática.

Quédate con que si tienes un algoritmo que se va ramificando creando dos o más ramas en cada paso,

lo que tienes es una complejidad exponencial bajo de dos elevado a n.

Pasemos al último caso.

La complejidad factorial o de n factorial.

En este caso voy a poner un ejemplo directo y sin utilidad real, ya que los

ejemplos reales con esta

complejidad son problemas complejos, como puede ser calcular todas las permutaciones posibles de un

array.

En primer lugar, por si no tienes claro lo que es el factorial de un número, simplemente es el producto

de todos los valores de n hasta dos.

Por ejemplo, el factorial de tres es tres por dos, que es igual a seis.

Por su parte, el factorial de seis es seis por cinco, por cuatro por tres y por dos, que es igual

a 720.

Como ves, esto crece de forma exagerada.

Por eso es una complejidad de evitar a toda costa.

En este ejemplo sencillo, lo que hacemos es recorrer los valores desde cero hasta n n veces.

Como vemos, el número de iteraciones crece de forma factorial.

En cada iteración se hacen n llamadas, las cuales hacen n menos una llamadas, las cuales vuelven a

hacer n dos llamadas.

Así hasta llegar a cero.

Y hasta aquí este tema.

Sé que es un concepto complejo e incluso puede llegar a ser abrumador porque no lo llegas a comprender

del todo.

No te preocupes, todos hemos pasado por esa fase.

A continuación veremos ejemplos prácticos para que mejores el concepto y durante todos los ejercicios

de algoritmos y estructuras de datos del curso se hará un análisis de complejidad, ya que es algo que

te pedirán sí o sí en este tipo de entrevistas técnicas, por lo que irás mejorando este concepto poco

a poco con el paso del curso.

Con lo que te tienes que quedar es que el pico es una forma de analizar como crecerá la complejidad

temporal y espacial de nuestro algoritmo en los extremos.

Recuerda que un algoritmo de mayor complejidad puede ser más rápido que otro con menor complejidad para

rangos de entrada específicamente bajos.

Pero eso no nos importa para un N bajo, todos los algoritmos van a ser rápidos.

Lo que nos importa ver es como se comportan cuando N crece.

Y hasta aquí la teoría del boicot.

En el próximo tema veremos ejemplos prácticos.

Reproducir
Reproducir
+50 ejercicios de entrevistas programación. Big O, algoritmos y