EBook Gratis

APRENDIZAJE big-o

Free unaffiliated eBook created from
Stack Overflow contributors.

#big-o

Tabla de contenido

Acerca de

Capítulo 1: Empezando con big-o

Observaciones

Esta sección proporciona una descripción general de qué es big-o y por qué un desarrollador puede querer usarlo.

También debe mencionar los temas grandes dentro de la gran o, y vincular a los temas relacionados. Dado que la Documentación para big-o es nueva, es posible que deba crear versiones iniciales de los temas relacionados.

Examples

¿Qué es la notación Big-O?

La notación Big-O es una notación que se utiliza para hablar sobre las tasas de crecimiento de las funciones a largo plazo. A menudo se usa en el análisis de algoritmos para hablar sobre el tiempo de ejecución de un algoritmo o conceptos relacionados como la complejidad del espacio.

En el uso común, la notación big-O se usa para hablar sobre cómo se escala el tiempo de ejecución de un algoritmo como el tamaño de la entrada. Por ejemplo, diríamos que la ordenación por selección tiene un tiempo de ejecución de $O(n^2)$ porque el tiempo de ejecución crece de forma cuadrática en función del tamaño de la matriz a ordenar. Es decir, si duplica el tamaño de la entrada, el tiempo de ejecución del orden de selección debería duplicarse aproximadamente. Cuando se usa la notación de O grande, la convención es eliminar los coeficientes e ignorar los términos de orden inferior. Por ejemplo, aunque técnicamente no es incorrecto decir que la búsqueda binaria se ejecuta en el tiempo $O(2 \log_2 n + 17)$, se considera un estilo deficiente y sería mejor escribir que la búsqueda binaria se ejecuta en el tiempo $O(\log n)$.

Formalmente, la notación big-O se utiliza para cuantificar el comportamiento a largo plazo de una función. Decimos que $f(n) = O(g)$ (a veces denominado $F(n) \in O(g(n))$ en algunas fuentes) si no están fijadas constantes c y n_0 tal que $f(n) \leq c \cdot g(n)$ para todos $n \geq n_0$. Esta definición formal explica por qué no nos importan los términos de orden bajo (se pueden subsumir haciendo c más grande y aumentando n_0) y factores constantes (el término c los absorbe). Esta definición formal se usa a menudo en el análisis riguroso de algoritmos, pero rara vez se usa coloquialmente.

Calculando Big-O para tu código

Una forma de calcular el valor Big-O de un procedimiento que ha escrito es determinar qué línea de código se ejecuta más veces en su función, dado el tamaño de entrada *n*. Una vez que tenga ese número, elimine todos los términos excepto el de más rápido crecimiento y deshágase de los coeficientes, esa es la notación Big-O de su función.

Por ejemplo, en esta función, cada línea se ejecuta exactamente una vez, y por la misma cantidad

de tiempo, independientemente de lo grande que `a` es:

```
int first(int[] a){
   printf("Returning the first element of a");
   return a[0];
}
```

La función en sí puede tardar 1 milisegundo ((1 ms) * n^0) o 100 milisegundos ((100 ms) * n^0); el valor exacto dependerá de la potencia de la computadora involucrada y de lo que imprima `printf()` . Pero como esos factores no cambian con el tamaño de `a` , no importan para los cálculos de Big-O, son coeficientes constantes, que eliminamos. Por lo tanto, esta función tiene un valor Big-O de *O (1)* .

En esta función, la línea 3 (`sum += a[i];`) se ejecuta una vez para cada elemento en `a` , para un total de `a.length` (o *n*) veces:

```
int sum(int[] a){
   int sum = 0;
   for (int i = 0; i < a.length; i++){
        sum += a[i];
   }
   return sum;
}
```

Las declaraciones `i++` y `i < a.length` cada una también se ejecutan *n* veces, podríamos haber elegido esas líneas, pero no tenemos que hacerlo. Además, `int sum = 0;` , `int i = 0` , y `return sum;` cada ejecución una vez, que es menos de `n` veces, ignoramos esas líneas. No importa cuánto tiempo tome `sum += a[i]` para ejecutarse, es un coeficiente que depende de la potencia de la computadora, así que eliminamos ese coeficiente. Por lo tanto, esta función es *O (n)* .

Si hay varias rutas de código, big-O se suele calcular en el peor de los casos. Por ejemplo, a pesar de que esta función tal vez puede salir de inmediato, no importa lo grande que `a` es (si `a[0]` es `0`), un caso todavía existe que hace que la línea 6 para funcionar `a.length` veces, por lo que sigue siendo O (n):

```
int product(int[] a){
    int product = 0;
    for (int i = 0; i < a.length; i++){
        if (a[i] == 0)
            return 0;
        else
            product *= a[i];
    }
    return product;
}
```

Lea Empezando con big-o en línea: https://riptutorial.com/es/big-o/topic/1340/empezando-con-big-o

Capítulo 2: Calculando Big-O

Examples

O (n) funciones.

Las funciones que son `O(n)` aumentan el número de operaciones de forma lineal, ya que la entrada es muy grande. Un ejemplo simple de una función que es O (n) sería el algoritmo de búsqueda lineal, que se ejecuta una vez para el tamaño de la entrada.

El siguiente pseudocódigo sería `O(n)` , porque siempre estará delimitado por el tamaño de entrada, ya que el algoritmo nunca se ejecutará más veces que el tamaño de entrada.

```
function LinearSearch (SearchArray, SearchFor)
    for each element in SearchArray
        if the element is SearchFor
            return the index of element
```

Lea Calculando Big-O en línea: https://riptutorial.com/es/big-o/topic/4307/calculando-big-o

Creditos

S. No	Capítulos	Contributors
1	Empezando con big-o	Community, intboolstring, templatetypedef, TheHansinator
2	Calculando Big-O	intboolstring