* Big O - grande operação.

Introdução:

Big O mede quanto "tempo" um algoritmo leva para ser executado.

Medimos a velocidade de um algoritmo com base no número aproximado de Etapas que ele leva para ser executado. Esse cálculo é feito em relação ao tamanho da entrada, que denotamos por n. Portanto, se recebermos um vetor de comprimento n estamos dizendo que queremos saber aproximadamente quantas operações o algoritmo exigiria para esse comprimento.

. Melhor solução ?

Algoritmo 1 -

Resolve o problema em $O(n)$

☑

no pior caso n vezes

Algoritmo 2 -

Resolve o problema em $O(n^2)$

☐

no pior caso $n \times n$ vezes

O que Big O significa ?

Big O avalia a rapidez com que nosso algoritmo lida com um grande número de itens.

1. Julgamos a velocidade com base no número de Etapas e não no tempo calculado (ou seja, segundos ou minutos).

2. Nós nos importamos com as etapas que aumentam com base no tamanho da entrada.

Exemplo prático: considere um n de entrada = 1.000.000

1º)
```
int vinte = 20;
int i;
for(i=0; i< vinte; i++){
    printf("Hello World\n");
}
```

2º)
```
int n, i;
scanf("%d",&n);
for(i=0; i<n; i++){
    printf("%d", i);
}
```

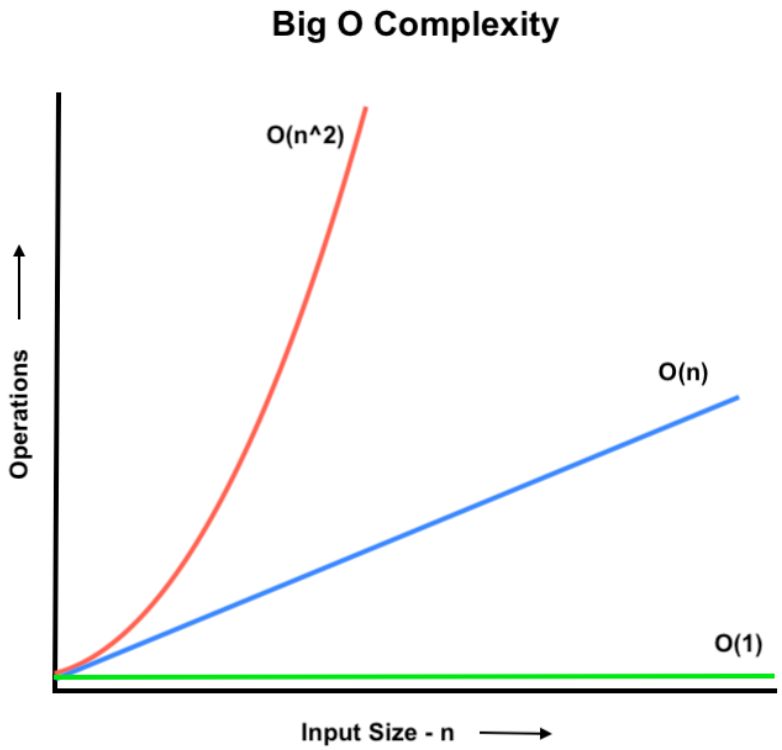
3º)
```
int n, i, j;
scanf("%d",&n);
for(i=0; i<n; i++){
    for(j=0; j<n; j++){
        printf("%d", i*j);
    }
}
```

O 1º não depende da entrada. Não importa o tamanho dos dados ele sempre realizará o mesmo número de operações. Portanto é um algoritmo de complexidade O(1) - constante. Ele sempre executa o mesmo número de operações, porque o loop sempre vai de 0 - 20.

Já o 2º usa um loop for, ele intera a entrada para os 1.000.000 itens, isso é o que chamamos de O(n).

Com o 3º) que possui for aninhados, ele irá interagir 1.000.000 vezes para todos os 1.000.000 de elementos para um total de 1.000.000.000.000. Isso o torna um algoritmo $O(n^2)$. Isso é 1 milhão contra 1 trilhão de operações para o 2º) vs 3º).

Graficamente:



O pior Caso:

Quando calculamos a complexidade do tempo Big O de um algoritmo, devemos nos basear na possibilidade do pior caso.

Exemplo Simples:

```c
#include <stdio.h>

int main (void) {
    int vetor [] = {4, 6, 7, 5, 2, 0, 3, 1};
    int i;

    // Melhor caso: encontrar o 4 na posição [0] - intera 1 item

    for (i = 0; i < 8; i++) {
        if (vetor[i] == 4) {
            printf("%d\n",i);
            break;
        }
    }

    // No caso médio: encontrar o 5 na posição [3] - intera 4 itens

    for (i = 0; i < 8; i++) {
        if (vetor[i] == 5) {
            printf("%d\n",i);
            break;
        }
    }

    // No pior caso: encontrar o 1 na última posição [7] - intera todos os 'n' itens

    for (i = 0; i < 8; i++) {
        if (vetor[i] == 1) {
            printf("%d\n",i);
            break;
        }
    }

    return 0;
}
```

1,1 Tudo

melhor caso

No melhor dos hipótes, iteramos apenas 1 item, então realmente encontramos o item no tempo $O(1)$ - nesse caso, o tamanho do vetor não irá importar, porém esse é um caso extremo e muito raro, além de não fornecer muito significado.

No caso médio, iremos iterar metade do vetor, que é $O(0,5n)$

No pior caso, teríamos que iterar todo o vetor para encontrar o o último item, que é $O(n)$.

Assim dizemos que nossa complexidade Big O é o pior cenário, o que nos permite contabilizar com segurança qualquer resultado.

Por Paulo Henrique Diniz de Lima Alencar.

Referências:

BIG O NATATION TIME AND SPACE COMPLEXITY. Skilled.dev, 2020. Disponível em: <https://skilled.dev/course/big-o-time-and-space-complexity>. Acesso em: 25 dez. de 202