Nome: Thaíssa Fernandes Silva **Curso:** Engenharia da Computação

Período: 2º **Matéria:** AEDS II - Noturno

Análise de complexidade

A Análise de complexidade nos permite medir o quão rápido um programa executa suas computações. Exemplos de computações são: Operações de adição e multiplicação; comparações; pesquisa de elementos em um conjunto de dados; determinar o caminho mais curto entre diferentes pontos; ou até verificar a presença de uma expressão regular em uma string. Claramente, a computação está sempre presente em programas de computadores.

Sendo o algoritmo um agrupamento para se executar uma tarefa, o tempo que leva para um algoritmo ser executado é baseado no número de passos.

Notação O

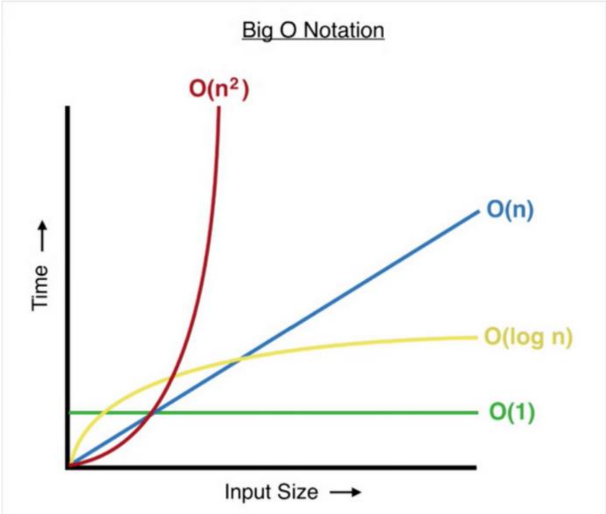
A notação O (Big O Notation) é uma notação que indica a complexidade de um algoritmo. Na realidade, existem várias notações: big e small O, ômega de theta.

A notação O é sobre um “limite por cima”, a ômega é um “limite por baixo” e a theta é uma combinação de ambos.

O Big O é mais utilizada porque o interesse está em verificar, sem tanta rigidez, o máximo de recursos que o algoritmo vai utilizar, e tentar reduzir isso quando possível.

Um exemplo é a função do tempo T(n), ppois ela representa a complexidade do algoritmo, tal que T(n) = O(n), afirmando que um algoritmo tem uma complexidade linear de tempo, pois o tempo está relacionado a N, isto é, o tamanho de input do algoritmo.

Em Big O Notation, as complexidades de tempo linear, logarítmica, cúbica e quadrática são representações de complexidades diferentes de relação entre T e N em um algoritmo. Também é usada para determinar quanto espaço é consumido pelo algoritmo.



Constantes O (1)

A complexidade constante é aquela em que o número de operações não muda mesmo que o input varia. As constantes de um algoritmo são normalmente ignoradas. Isto porque a notação Big O se importa com o comportamento do algoritmo à medida que a entrada cresce, e não com os detalhes exatos para cada tamanho.

Quanto maior a entrada fica, menos importantes se tornam as constantes. Por isso, todo algoritmo com número de operações constantes tem tempo de execução O(1).

Complexidade Linear O(n)

Toda complexidade diferente da constante se dá em relação ao número que a sua função recebe. Logo, quanto maior o imput, maior o tempo de execução do algoritmo. No caso da complexidade linear, a diferença é bem proporcional ao tamanho da entrada.

Neste caso, como temos operações relacionadas a N, não há porque se preocupar com as que são constante. Isso porque a parte linear já dominará a notação de complexidade, então se deixarmos de lado as constantes sobram apenas as O(n).

Complexidade Logarítma O(log n)

Através do logaritmo de N podemos encontrar o número de operações realizadas durante o runtime.

Um exemplo de algoritmo com complexidade linear O(log n) é uma busca binária em uma lista já ordenada.

Você parte o input ao meio e compara para verificar se o item a ser buscado é menor ou maior que o item no meio do array. Quando isso acontece, metade da lista é descartada ficando apenas a parte menor. Esse processo é repetido até que se encontre o item da busca, diminuindo cada vez mais o processamento.

Ele é o inverso do exponencial, pois N diminui toda vez que um processamento é feito.

Complexidade Quadrática O(n^2)

Enquanto a complexidade linear sobe em linha reta, a complexidade quadrática desenha uma curva (parábola) em relação ao eixo de tempo para cada vez que N aumenta.

Até certo ponto, para algoritmo linear pode ser pior que um quadrático justamente por causa das constantes. Em algum momento o quadrático vai ficar mais lento.

Complexidade Quadrática O(n^3)

Há ainda a complexidade cúbica, onde há três loops aninhados. A lógica é a mesma que a lógica por trás da complexidade quadrática.

Complexidade O(2^n)

Se o algoritmo tiver complexidade n elevada a alguma variável que também cresce com a entrada, então o algoritmo é exponencial, a pior e mais lenta das complexidades.

Upper bound

A cota superior de um problema varia de acordo com o problema que temos.

Seja dado um problema, por exemplo, multiplicação de duas matrizes quadradas de ordem n. Conhecemos um algoritmo para resolver este problema (pelo método trivial) de complexidade. Sabemos assim que a complexidade deste problema não deve superar O(n^3), uma vez que existe um algoritmo desta complexidade que o resolve.

Lower bound

Às vezes, é possível demonstrar que para um dado problema, qualquer que seja o algoritmo a ser usado, o problema requer pelo menos um certo número de operações. Essa complexidade é chamada cota inferior ou ``lower bound'' do problema. Veja que a cota inferior depende do problema, mas não do particular algoritmo.

Small O notation

A diferença entre a definição formal da notação O-grande, e a definição de o-pequeno é: enquanto a primeira deve ser verdade para *pelo menos uma* constante *M* a segunda deve se verificar para *todas* as constantes positivas ε, mesmo as pequenas. Dessa maneira, a notação o-pequeno faz uma afirmação mais forte que a da notação *O*-grande: toda função que é o-pequeno de *g* também é *O*-grande de *g*, mas nem toda função que é *O*-grande de *g* também é o-pequeno de *g* (por exemplo a própria *g* não é, a menos que ela seja identicamente zero perto de ∞).

Notação Omega-grande

Algumas vezes, queremos dizer que um algoritmo leva *ao menos* uma certa quantidade de tempo, sem fornecer um limite superior. Usamos a notação Ω, que é a letra grega "omega" maiúscula.

Se um tempo de execução é Ω (f(n)Ω(*f*(*n*)), então, para um n suficientemente grande, ele será ao menos kf(n) para uma constante k qualquer. Aqui está um exemplo de um tempo de execução igual a Ω (f(n)Ω(*f*(*n*)).

Dizemos que tal tempo de execução é "Ω de f(n)”. Usamos a notação Ω para limites assintóticos inferiores, uma vez que delimita de modo otimista o aumento do tempo de execução para entradas suficientemente extensas.

Notação Grande-Theta

Quando o código possui uma sobrecarga extra ele possui um valor indeterminado de tempo adicionado ao tempo que o código compilará.

Na prática, simplesmente descartamos fatores constantes e termos de ordem inferior. Outra vantagem de usar a notação Θ é que não temos que nos preocupar com quais unidades de tempo estamos usando.

Por exemplo, suponha que você tenha calculado que um tempo de execução é de 6n^2 + 100n + 3006*n*2+100*n*+3006, n, squared, plus, 100, n, plus, 300 microssegundos. Ou talvez sejam milissegundos. Quando você usa a notação Θ, você não especifica isso. Você também descarta o fator 6 e os termos de ordem inferior 100n + 300100*n*+300100, n, plus, 300, e você simplesmente diz que o tempo de execução é Θ (n^2).

Quando usamos a notação big-Θ estamos dizendo que temos um limite assintoticamente restrito no tempo de execução. "Assintoticamente" porque ele importa apenas para valores grandes de n. "Limite restrito" porque definimos o tempo de execução para um fator constante superior e inferior.