Análise de Complexidade em Algoritmos.

Introdução

Um algoritmo é capaz de resolver um problema através de uma entrada de dados e produzir seus resultados na saída. Podemos definir um problema como um conjunto de dados em que desejamos processar a fim de obter um resultado. Todo esse processo demanda tempo e muitas das vezes é impossível de ser resolvido na prática devido o tempo necessário. Adotandos boas práticas e uma boa análise matemática podemos diminuir esses problemas ou até mesmo contorná-los.

A importância da análise matemática

A Análise Matemática é um dos meios de se avaliar a complexidade de um algoritmo. Comparar o tempo que um algoritmo leva parar executar determinado código não são parâmetros para dizer qual código é melhor. A análise matemática busca utilizar fórmulas menos complexas e que chegam no mesmo resultado de forma mais eficiente e utilizando menos processamento.

Contando instruções de um algoritmo

Por mais que dois algoritmos resolvam o mesmo problema, nem sempre eles terão a mesma eficiência. A complexidade de um algoritmo pode ser medida em cima das funções que o algoritmo é capaz de fazer (acesso, inserção, remoção, são exemplos mais comuns em computação), e o volume de dados (quantidade de elementos a processar). Quanto maior o número de instruções um algoritmo possui, mais processamento será necessário para resolver o problema.

```
maior := lista[0]

for indice := 0; indice < n; indice++ {
  if lista[indice] > maior {
    maior = lista[indice]
  }
}
```

No algoritmo acima podemos definir que sua taxa de crescimento é de **f(n) = 4 + 2n** caso o corpo do looping for não seja considerado.

Comportamento assintótico

O comportamento assintótico é a curva de crescimento de uma função gerada pelo processo de análise de algoritmos. O comportamento assintótico de f(**n**) representa o limite do comportamento do custo quando **n** cresce. Na análise de complexidade, apenas contamos o termo que mais cresce de acordo com a entrada. Para chegarmos nesse termo, podemos remover todas as constantes e manter o termo que mais cresce.

Tipos de análise assintótica

A notação O nos fornece uma simbologia simplificada para representar um **limite superior** de desempenho para um algoritmo. Um limite máximo de tempo que um algoritmo leva para ser executado.

A notação Ω nos fornece uma simbologia simplificada para representar um limite inferior de desempenho para um algoritmo. Um **limite mínimo** de tempo que um algoritmo leva para ser executado. Ou seja, a notação Ω é o inverso da notação Big O.

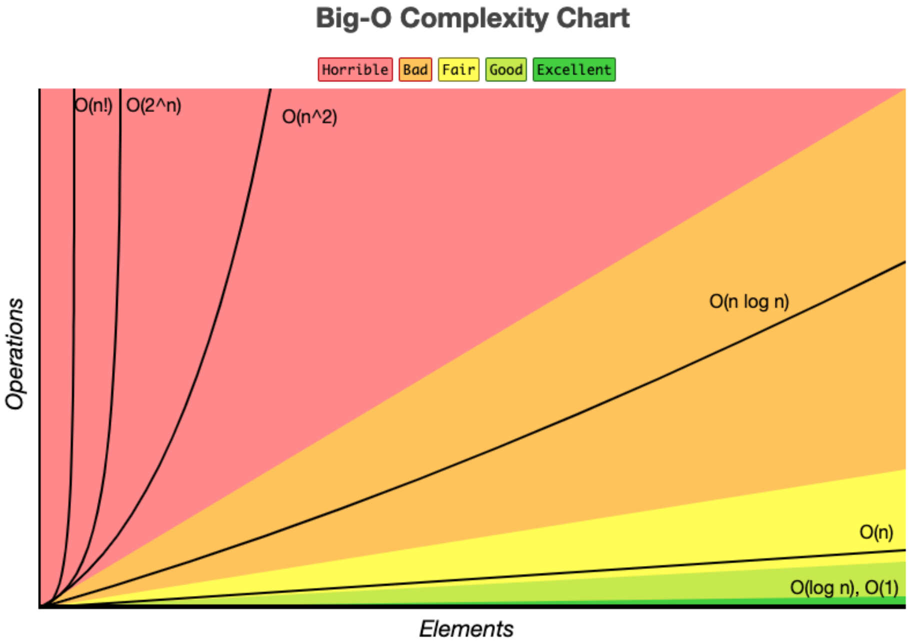
A notação Θ nos fornece uma simbologia simplificada para representar um limite justo de desempenho para um algoritmo. Um **limite exato** de tempo que um algoritmo leva para ser executado. Ou seja, a notação Θ representa o ponto de encontro entre as notações Ω (limite inferior) e Big O (limite superior).

Classes de problemas

As classes dos problemas se dividem da seguinte forma:

1. **f(n) = O(1)** Algoritmos dessa classe possuem uma complexidade constante que não se altera independentemente do valor de n.
2. **f(n) = O(log n)** Algoritmos dessa classe crescem de forma logarítmica, são conhecidos por dividir um problema em outros menores como por exemplo a busca binaria.
3. **f(n) = O(n)** Algoritmos dessa classe crescem de forma linear em função do n. Ao dobrar o valor de n, o tempo de processamento também dobra.
4. **f(n) = O(n log n)** Algoritmos dessa classe dividem um problema, em problemas menores e depois realiza a combinação das soluções obtidas.
5. **f(n) = O(n^2)** Algoritmos dessa classe possuem complexidade quadrática e seu tempo de processamento se multiplica por 4 ao dobrarmos o valor de n.
6. **f(n) = O(n^3)** Algoritmos dessa classe possuem complexidade cúbica e seu tempo de processamento se multiplica por 8 ao dobrarmos o valor de n.
7. **f(n) = O(2^n)** Algoritmos dessa classe possuem complexidade exponencial, não são uteis na prática por possuir um alto consumo de processamento à medida que n aumenta.
8. **f(n) = O(n!)** Algoritmos dessa classe possuem complexidade exponencial, também não são uteis na prática e possuem um tempo de processamento

ainda maior. Pequenos problemas levariam séculos para ser resolvidos nesta classe.



É notório que o grau de complexidade dos problemas cresce cada vez mais chegando a um ponto de ser inviável na prática. No gráfico acima podemos observar que problemas da classe **f(n) = O(n²)** são considerados horríveis devido o crescente número de operações que precisam ser realizadas para solucionar o problema.

Conclusão

Concluímos então que de nada importa se a pessoa que estiver escrevendo o código não tiver uma noção fundamental de algoritmos. construir algoritmos de forma eficiente é de extrema importância para desenvolver aplicações de forma eficiente. Mesmo que dois algoritmos tenham a capacidade de resolver o mesmo problema, um deles com uma boa aplicação pode gerar uma grande economia de tempo e processamento.

Discente: Rafael Rodrigues Monteiro

2º período, Engenharia da Computação, Matutino.