Big O: Complexidade espacial x temporal.

O Big O tem como base ver a aplicabilidade e eficiência em qualquer linguagem de programação que o código esteja escrito. Visto que a complexidade é a quantidade de trabalho para a execução de uma sequência de passos, podemos perceber que a quantidade de N dados afeta diretamente a resposta do programa. Dividimos a complexidade de algoritmos em espacial e temporal. A espacial diz a quantidade de memória que é requisitada para a execução do código na pior perspectiva, já a complexidade temporal se diz a respeito sobre o tempo decorrido para correr o código e a quantidade de números de instruções essenciais para a solução do problema.

Escalas de complexidade: melhor, médio e pior caso.

Escalas de complexidade: melhor, médio e pior caso.

A complexidade de algoritmos é definida em:

- Melhor caso Ω

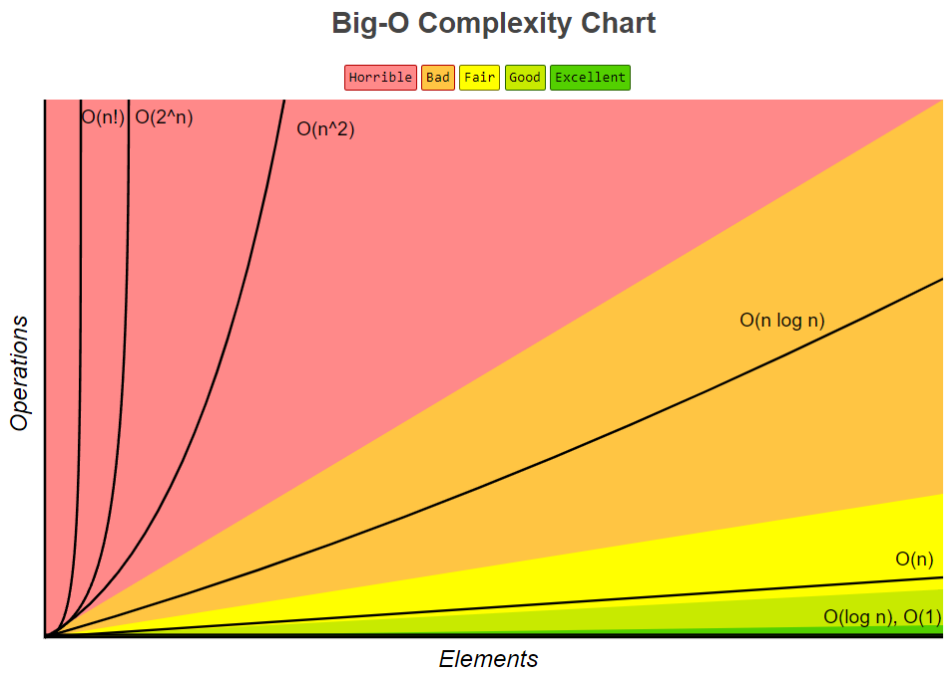
É o menor tempo de execução em uma entrada de tamanho N. Exemplo: Encontrar algum número na lista f(N) = Ω (1), sendo considerado numero na frente da lista.

- Caso médio θ

Sendo as mais difíceis de se estabelecer, para obtê-la devemos ter a média dos tempos de execução de todas as entradas possuindo tamanho N, ou apurando-se em probabilidade de determinada condição acontecer.

- Pior caso O

Tem o maior tempo de execução com as entradas de tamanho N. Considerando a procura de um número no final ou fora da lista f(N) = O (N).



Por que a complexidade de pior caso é a de maior preocupação em desenvolvimento de software? Explique.
Porque nela temos a complexidade temporal e o custo final do algoritmo pode estar ligado a tempo de execução, utilização de memória principal, utilização de disco e consumo de energia e com isso perdendo desempenho de execução.

Defina a notação BigO e dê exemplos de trechos de código para os casos de complexidade: constante, linear, logN, NlogN, N^2, N^3 e 2^N:
A notação Big O é uma notação matemática que descreve o comportamento de uma função e se o valor dela tende a específico ou infinito, assim mostrando como é o desempenho de um algoritmo e como ele escala.

- **Constante**: Complexidade O(1), instrução realizada num tamanho fixo de vezes. Ex:

```
Function Vazia(Lista: TipoLista): Boolean;
Begin
        Vazia := Lista.Primeiro = Lista.Ultimo;
End;
```

- **Complexidade Linear**: Complexidade O(N), operação realizada em cada elemento de entrada. Ex:

```
Procedure Busca(Lista: TipoLista; x: TipoElem; Var pos: integer)
Var i: integer;
Begin i
        i:=1;
        while Lista.Elemento[i] <> x do
                i := i+1;
        if i >= Lista.MaxTam then
                pos := -1
        else
                pos := i;
 End;
```

- **Complexidade Logarítmica**: Complexidade O(logN), dividem problemas em menores.

```
int
buscaBinaria (int x, int n, int v[]) {
        int e = -1, d = n;
          while (e < d-1) {
            int m = (e + d)/2;
            if (v[m] < x) e = m;
            else d = m;
          }
        returnvoid fibonacci(numero):
        if numero <= 1
                return number
        else
                return (fibonacci(numero - 1) + fibonacci(numero - 2))
         d;
}
```

- **Complexidade NLogN**: Complexidade O(NlogN), dividem problemas em menores e une depois a resolução dos problemas menores.

```
const mergeSort = (array, p, r) => {
  if (p < r) {
  var q = Math.floor((p + r)/2);
  mergeSort(array, p, q);
  mergeSort(array, q+1, r);
  merge(array, p, q, r);
  }
};
const merge = (array, p, q, r) => {
  const lowHalf=[];
  const highHalf=[];
  let k=p;
  let i,j;
  for(i=0;k<=q;i++,k++){
    lowHalf[i]=array[k];
  }
  for(j=0;k<=r;j++,k++){
    highHalf[j]=array[k];
  }
  k=p;
  for(j=i=0;i<lowHalf.length && j<highHalf.length;){
    if(lowHalf[i]<highHalf[j]){
```

```
      array[k]=lowHalf[i];i++;
    } else {
      array[k]=highHalf[j]; j++;
    }
    k++;
  }
  for(;i<lowHalf.length;){
    array[k]=lowHalf[i];
    i++;
    k++;
    }
  for(;j<highHalf.length;){
    array[k]=highHalf[j];
    j++;
    k++;
  }
};
```

- **Complexidade Quadrática**: complexidade O(N²), loop dentro do outro e itens processados aos pares.

```
Procedure SomaMatriz(Mat1, Mat2, MatRes: Matriz);
        Var i, j: integer;
                Begin for i:=1 to n do
                        for j:=1 to n do
                                MatRes[i,j] := Mat1[i, j] + Mat2[i,j];
```

- **Complexidade Cúbica**: Complexidade O(N³), loop dentro do outros dois e itens processados três a três.

```
Procedure SomaElementos_Vetor_Indices_Matriz (mat: Matriz, vet: Vetor);
Var i, j: integer;
 Begin
         for i:=1 to n do
                for j:=1 to n do
                        for k:=1 to n do
                                mat[i, j] := mat[i, j] + vet[k];
```

- **Complexidade Exponencial**: Complexidade O(2^N), utiliza “Força bruta” para resolver.

Links:

http://ava.femass.edu.br/pluginfile.php/21422/mod_resource/content/2/Prof.%20Braulio%20-%20ED%20II%20-%2001%20Complexidade%20de%20algoritmos.pdf

https://estevestoni.medium.com/iniciando-com-a-nota%C3%A7%C3%A3o-big-o-be996fa3b47b

http://wiki.icmc.usp.br/images/d/de/Analise_complexidade.pdf

https://www.freecodecamp.org/portuguese/news/o-que-e-a-notacao-big-o-complexidade-de-tempo-e-de-espaco/

https://www.felipealencar.net/2022/01/entendendo-complexidade-de-algoritmos-e.html#:~:text=Em%20termos%20simples%2C%20a%20nota%C3%A7%C3%A3o,o%20cen%C3%A1rio%20de%20pior%20caso.

https://www.ime.usp.br/~pf/algoritmos/aulas/bubi.html

https://dev.to/lofiandcode/big-o-part-4-n-log-n-4hgp