Análise da complexidade de algoritmos

Definição: A análise da complexidade de algoritmos é uma forma de estudar os possíveis métodos de se analisar o desempenho de um determinado algoritmo.

Esse estudo é importante pois auxilia o programador durante o processo de codificação a desenvolver algoritmos mais eficientes e otimizados, escolhendo o melhor caminho possível.

Medição de tempo em segundos (empírica): A forma mais óbvia de se medir o desempenho de um algoritmo seria pela medição do tempo em segundos que o mesmo leva para executar determinada entrada.

Entretanto, essa é uma forma ineficaz, pois diversos fatores podem alterar a resposta desejada, de execução para execução.

São eles:

- Velocidade de processamento do computador;

- Linguagens de programação utilizadas;

- Compiladores;

- Sistema operacional;

Método analítico: O método analítico é a forma mais viável de se medir o desempenho de um determinado algoritmo, o qual não está baseado em calcular o tempo de execução em segundos, mas sim na representação matemática, por meio de uma função, que traduz o desempenho baseado na quantidade de instruções de um código.

- Considerar o pior caso **SEMPRE**!!!

Vejamos o número de instruções que comandos em linguagens de programação possuem:

- Custo computacional = 1 (constante)
  - ✓ Declaração e atribuição de variáveis;
  - ✓ Incremento ou decremento de variáveis;
  - ✓ Operações matemáticas complexas (pow, sqrt...);
  - ✓ Acesso a elementos de arrays

- ✓ Operações lógicas (if, else...)
- ✓ Operações de entrada e saída

Ex de código (apenas 1 instruções):

int somaSimples(int x, int y){

int s; 1 instrução (declaração de variável)

s = x+y; 1 instrução (atribuição de valor à variável)

if(s%2==0){ 1 instrução (verificação da condição)

return 1; 1 instrução (caso a condição acima seja verdadeira)

}

return 0; 1 instrução (caso a condição for falsa)

}

Análise analítica no pior caso:

T(n) = 1+1+1+1 = 4

- Custo computacional = log n
  - ✓ Busca binária

Ex de código (log n)

int buscaBinaria(int arr[], int tamanho, int alvo) {
int esquerda = 0, direita = tamanho - 1; 1 instrução (declaração de variáveis)

while (esquerda <= direita) { log n instruções (divisão do vetor ao meio a cada iteração)
int meio = (esquerda + direita) / 2; 1 instrução (declaração de variável)
if (arr[meio] == alvo) { 1 instrução (verificação da condição)
return meio; 1 instrução (caso a condição seja verdadeira)
}
else if (arr[meio] < alvo) {2 instruções (else e if)
esquerda = meio + 1; 1 instrução (atribuição de valor à variável)
}
else { 1 instrução (else)
direita = meio - 1; 1 instrução (atribuição de valor à variável)

```
        }
    }

    return -1;  1 instrução (retorno de valor)
}
```

Análise analítica no pior caso:

T(n) = 1 + log n(1+1+2+1+1) + 1= 1 + 6*log n + 1 = 2 + 6*log n

- Custo computacional = n
  - ✓ Laços de repetição, no geral (que vão até n) (for e while)
  - ✓ Chamada recursiva única

Ex de código (n)

```
int soma(int a, int b){

    if(b==0){ 1 instrução (verificação de condição)

        return 0;

    }

    if(b==1){ 1 instrução (verificação de condição)

        return a;

    }

    return a + soma(a, b-1); 1 instrução (operação aritmética a+(soma(a,
b-1))  e n instruções (chamada recursiva soma(a, b-1))

}
```

Análise analítica no pior caso:

T(n) = 3n

- Custo computacional = n log n
  - ✓ Algoritmos de ordenação eficientes (merge sort, quick sort e heap sort)

- Custo computacional = n^2
  - ✓ Algoritmos de ordenação simples (bubble sort, insertion sort e slection sort)

Ex de código (n^2):

```
void ordenarVetor(int *A, int tam_a){
  int pos_menor, aux; 1 instrução (declaração de variáveis)
  for(int i=0; i<tam_a-1; i++){ 1 instrução (i=0), n instruções (i<tam_a-1), (n-1) instruções (i++)
    pos_menor = i; 1 instrução (atribuição)
    for(int j=i+1; j<tam_a; j++){ 1 instrução (j=i+1), (n+1) instruções (j<tam_a), n instruções (j++)
      if(A[j]<A[pos_menor]){ 1 instrução (verificação de condição)
        pos_menor = j; 1 instrução (atribuição)
      }
    }
    aux = A[i]; 1 instrução (atribuição)
    A[i] = A[pos_menor]; 1 instrução (atribuição)
    A[pos_menor] = aux; 1 instrução (atribuição)
  }
}
```

Análise analítica no <u>pior caso</u>:

$T(n) = 1+1+n*(1+1+(n+1)*(1+1) + n)+(n-1)+1+1+1$

$T(n) = 1+1+n*(1+1+n+n+1+1+n)+(n-1)+1+1+1$

$T(n) = 1+1+n+n+n^2+n^2+n+n+n^2+n-1+1+1+1$

$T(n) = 3n^2 + 5n + 4$

Análise dos loops do exemplo de cima: tam_a = 5, logo n = 5

1° for:

condição do loop: i<tam_a-1, ou seja, i vai até 4

i=0 -> 1 instrução -> i++

i=1 -> 1 instrução -> i++

i=2 -> 1 instrução -> i++

i=3 -> 1 instrução -> i++

i=4 -> 1 instrução -> verificação para saída do loop (4<4? (F)), saiu

total = n instruções

2° for:

condição do loop: j<tam_a, ou seja j vai até 5

i=0 -> 1 instrução -> i++

i=1 -> 1 instrução -> i++

i=2 -> 1 instrução -> i++

i=3 -> 1 instrução -> i++

i=4 -> 1 instrução -> i++

i=5 -> 1 instrução -> verificação para saída do loop (5<5? (F)), saiu

total = n+1 instruções

A ordem de prioridade fica:

const < log n < n < n log n < n^2

Exemplos práticos:

01)

```
void func(int n){
    for(int i=0; i<n; i++){
        printf("%d", i);
    }
}
```

1 – Analisando a estrutura do loop:

int i=0 -> 1 instrução

i<n -> n+1 instruções (1 comparação é realizada a mais para sair do loop)

i++ -> n instruções

2 – Analisando as instruções internos do loop:

print(i) -> n instruções (realizadas junto com o loop)

3 – Montando a função correspondente à complexidade do algoritmo

T(n) = 1 + (n+1) + n + n

T(n) = 1 + n + 1 + n + n

T(n) = 3n + 2

4 – Complexidade Big O: O(n)

BuscaLinear(A, n, k)

1 $i = 1$

2 **enquanto** $i \leq n$ e $A[i]$.chave $\neq k$ **faça**

3     $i = i + 1$

4 **se** $i \leq n$ e $A[i]$.chave $== k$ **então**

5   **devolve** i

6 **devolve** -1

Reescrevendo as linhas 1 e 2 -> for(i=1; i<=n && A[i].chave!=k; i++);

Logo, T(n) = 1+(n+1)+n+2+1 = 1 + n + 1 + n + 2 + 1 = 5 + 2n

Exemplos práticos em C:

```c
/*
    EX 01:
    Codifique uma função que multiplique um dado inteiro A por B, usando 
somas sucessivas.
*/

int soma(int a, int b)
{
    if(b==0) // 1
    {
        return 0;
    }
    if(b==1) //1
    {
        return a;
    }
    return a + soma(a, b-1); //1 e n
}

//f(n) = 3n -> Logo, O(n)

//Funcionamento
soma(4, 3) = 12
    soma(4, 3) = 4 + soma(4, 2) = 4 + 8 = 12
            soma(4, 2) = 4 + soma(4, 1) = 4 + 4 = 8
                soma(4, 1) = 4

/*
    EX 02:
    Faça uma função recursiva que descubra o tamanho de uma string
*/


int tamanho(char *str)
{
    if(*str=='\0') // 1
    {
        return 0;
    }
    return 1 + tamanho(str+1); // 1 (operação aritmética) e n (chamada 
recursiva)
}

T(n) = 2n -> Logo, O(n)

/*
    EX 02:
    Crie uma função recursiva que calcule a potência de um número usando 
multiplicações sucessivas
```

```
*/
int pot(int a, int b)
{
    if(b==0) //1
    {
        return 1;
    }
    if(b==1) //1
    {
        return a;
    }
    return a*pot(a, b-1); // 1 e n
}

t(n) = 3n -> Logo, O(n)

pot(4, 3) = 64
    4*pot(4, 2) = 4*16 = 64 //pot(4, 3)
        4*pot(4, 1) = 4*4 = 16 //pot(4, 2)
            4 //pot(4, 1)

int fatorial(int x)
{
    if(x==0 || x==1) //1 -> independente se é o melhor ou pior caso,
sempre será verificado
    {
        return 1;
    }
    return x*fatorial(x-1); // 1 (operação x*fatorial(x-1)) e n (x-1)
}
Logo, f(n) = 2n -> Logo, O(n)
```