



→ O melhor algoritmo é o que resolve o problema em menor tempo e ocupando o menor espaço

→ complexidade

Avaliação

- ↳ tempo de processamento e espaço de memória requer para sua execução;
- ↳ atender os requisitos: corretude, simplicidade, eficiência

• corretude → toda entrada (legal) ele produz a saída correta

• simplicidade → código limpo (fácil de entender, implementar e manter)

(↳ contador não é nem `c`, `sum`, `count`, `index` e afins;

• eficiência → tempo e espaço → complexidade de espaço;

↳ tempo
de execução
para saída
correta

→ RAM → espaço para a
execução;

→ Complexidade de espaço

$$S(P) = C + S_p \text{ no espaço variável}$$

espaço em procedimento constante

77

float abc(int a, int b, int c) {

return a+b+b*c + (a+b-c)/(a+b)+4.0

}

→ se declara alguma no procedimento
→ variável

$$S(abc) = 12 \text{ bytes} + 0 = 12 \text{ bytes}$$

↓ int = 4 bytes

→ é um vetor

a - vetor [1..n]

$$n = \{ \mathbb{B}$$

$$a = nB$$

$$\text{Soma} \rightarrow \mathbb{B}$$

$$i \rightarrow \mathbb{B}$$

$$3B + NB = (N+3)B$$

procedure sum(a, n) {

$$\text{Soma} = 0$$

para i : 1 ate n fazer Soma

$$\text{Soma} := \text{Soma} + a[i]$$

Fim para

retorne Soma

}

→ ordem de crescimento de tempo de execução

Complexidade de tempo

O tempo de execução numa determinada entrada, é o número de operações primitivas executadas;

O cada linha requer um tempo constante (operações primitivas)

O linhas diferentes requer tempo diferentes

O somas de cada linha de código

```

int Max (int arr [], int n) {
    tempo vezes
    int i; C1 executada + vez
    int max = arr[0]; C2 , 1 vez;
    for (i=1; i <= n; i++) C3 , n-1 vezes;
        if (arr[i] > max) C4 , n-1 vezes;
        max = arr[i]; C5 , n-1 vezes no pior caso
    return max; C6 , 1 vez
  
```

Tempo Total (soma do tempo de cada linha)

$$T(\max) = C_1 + C_2 + C_3 \cdot (n-1) + C_4 \cdot (n-1) + C_5 \cdot (n-1) + C_6$$

$$\Rightarrow \underbrace{C_1 + C_2 + C_6}_a + (n-1) \underbrace{(C_3 + C_4 + C_5)}_b UT$$

$$T(\max) = a(n-1) + b$$

→ linear (reta) → o tempo de entrada cresce linearmente



→ sempre interessados no pior caso

Pior caso: todos os condições são executadas

Médio caso: metade das "

"

Melhor caso: uma "

" → raramente feita

• Função insertionSort

```
void insertionSort (int arr[], int n) {  
    int i, key, j; → C1 x 1  
    for (i = 1; i < n; i++) { → C2 x (n-1)  
        key = arr[i]; → C3 x (n-1)  
        j = i - 1; → C4 x (n-1)  
        while (j >= 0 && arr[j] > key) { → C5 x (n-1) x (n)  
            arr[j + 1] = arr[j]; → C6 (n-1) x n  
            j = j - 1; → C6 (n-1) x n  
        }  
        arr[j + 1] = key; → C7 x n  
    }  
}
```

• pior caso

$\Theta(n^2)$

• melhor caso

$C_2 + n(C_2 + C_3 + C_6)$

$\cancel{C_2} + C_6 n \rightarrow n$

complexidade de tempo

* constante: sem laço de repetição

& logarítmica: o tempo cresce logarítmicamente → ex: busca binária

* linear: busca sequencial

* quadrática: insertion sort bubble

↳ dois laços de repetição

* polinomial:

* exponencial: \rightarrow força bruta \rightarrow queda de sinal de wifi
e
fatorial

merge sort ($n \log n$)

\rightarrow olhar os laços de repetição

\rightarrow IF - constante

\rightarrow operações na CPU \rightarrow tempo de execução

$\log(n)$ \rightarrow melhor



