# Atividade 4 - Complexidade Assintótica

25 de setembro de 2025

## Problema A

**Procedimento**:

```
#include <stdio.h>
#include <stdlib.h>

int encontrarMaior(int vetor[], int tamanho) {
    int maior = vetor[0];
    for (int i = 1; i < tamanho; i++) {
        if (vetor[i] > maior) {
            maior = vetor[i];
        }
    }
    return maior;
}

int main() {
    int n;
    scanf("%d", &n);
    int *vetor = (int*)malloc(n * sizeof(int));
    for (int i = 0; i < n; i++) {
        scanf("%d", &vetor[i]);
    }
    int maiorElemento = encontrarMaior(vetor, n);
    printf("O maior elemento do vetor é: %d\n", maiorElemento);
    free(vetor);
    return 0;
}
```

**Resposta**: $O(n)$

**Explicação**:
Para computar qual o maior elemento do vetor, o algoritmo percorre todos os seus elementos. Como o vetor tem tamanho $n$, a sua complexidade assintótica é $O(n)$, que é o custo de percorrer todo o *array*.

# Problema B

**Procedimento**:

```
#include <stdio.h>
#include <stdlib.h>

void ordenaVetor(int vetor[], int tamanho) {
    for (int i = 0; i < tamanho - 1; i++) {
        int indiceMinimo = i;
        for (int j = i + 1; j < tamanho; j++) {
            if (vetor[j] < vetor[indiceMinimo]) {
                indiceMinimo = j;
            }
        }
        if (indiceMinimo != i) {
            int temp = vetor[i];
            vetor[i] = vetor[indiceMinimo];
            vetor[indiceMinimo] = temp;
        }
    }
}

void imprimirVetor(int vetor[], int tamanho) {
    for (int i = 0; i < tamanho; i++) {
        printf("%d", vetor[i]);
        if (i < tamanho - 1) {
            printf(" ");
        }
    }
    printf("\n");
}

int main() {
    int n;
    scanf("%d", &n);
    int *vetor = (int*)malloc(n * sizeof(int));
    for (int i = 0; i < n; i++) {
        scanf("%d", &vetor[i]);
    }
    printf("Vetor original: ");
    imprimirVetor(vetor, n);
    ordenaVetor(vetor, n);
    printf("Vetor ordenado: ");
    imprimirVetor(vetor, n);
    free(vetor);
    return 0;
}
```

**Resposta**: $O(n^2)$

**Explicação**:
O laço principal da função `ordenaVetor()` executa um total de $n-1$ vezes, onde $n$ é o tamanho do vetor. A cada iteração, ele seleciona um elemento para ser colocado em ordem.

Já o laço interno depende do valor de `i`, que altera ao longo das iterações do laço principal. Como o valor de `i` vai de 0 até $n-1$, avançando de um em um, o número de comparações que ele realiza segue a forma:

$$(n-1) + (n-2) + (n-3) + \cdots + (n-(n-1))$$

Como o valor de `i` cresce, o número de iterações do laço interno diminui, formando uma série aritmética. Essa série é o mesmo que a soma dos primeiros $n-1$ números inteiros positivos, que pode ser expressa pela fórmula:

$$S = \frac{(n-1)\cdot(1+(n-1))}{2} = \frac{(n-1)\cdot n}{2} = \frac{n^2-n}{2}$$

Na análise de complexidade assintótica ignoramos termos de menor ordem (como o $-n$) e constantes (como $\frac{1}{2}$), pois o termo que cresce mais rapidamente, $n^2$, domina o tempo de execução à medida que o tamanho do vetor aumenta.

Portanto, a complexidade é determinada pelo termo $n^2$, o que resulta em uma complexidade assintótica de $O(n^2)$.

# Problema C

**Procedimento**:

```
#include <stdio.h>

int main(){
    for(int i = 0; i < 1000000; i++){
        printf("Eu amo Estrutura de Dados <3.\n");
    }
    return 0;
}
```

**Resposta**: $O(1)$

**Explicação**:
Embora o programa realize 1000000 iterações, que é uma grande quantidade, a complexidade do programa não altera de acordo com a entrada. Sendo assim, o programa possui complexidade constante, ou seja, se enquadra na classe $O(1)$.

# Problema D

**Procedimento**:

```
int** multiplicaMatrizes(int **A, int **B, int n){
    int **resultado = (int**) malloc (sizeof(int*) * n);
    for(int i = 0; i < n; ++i){
        resultado[i] = (int*) calloc (n, sizeof(int));
    }
    for(int i = 0; i < n; i++){
        for(int j = 0; j < n; j++){
            for(int k = 0; k < n; k++){
                resultado[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    return resultado;
}
```

**Resposta**: $O(n^3)$

**Explicação**:
Na função, fica claro que a complexidade será determinada pelo número de iterações que os três laços aninhados realizam. No caso, todos os casos iteram de 0 até $n$. Dessa forma, o número de operações realizadas pela função é basicamente $n \cdot n \cdot n = n^3$, ou seja, $O(n^3)$.

# Problema E

**Procedimento**:

```
#include <stdio.h>

void funct(int n){
    int x = 0;
    for(int i = 0; i < n; ++i){
        for(int j = 1; j <= n; j *= 2){
            x++;
        }
    }
}

int main(){
    int n;
    scanf("%d", &n);
    funct(n);
    return 0;
}
```

**Resposta**: $O(n \cdot log(n))$

**Explicação**:
A função possui 2 laços aninhados: o primeiro indo de 0 até $n$ de 1 em 1, já a outra vai de 1 até $n$ de acordo com as potências de 2.

Para determinar a quantidade de iterações realizadas pelo loop mais interno, precisamos encontrar o valor de $k$ tal que $2^k >= n$. Dessa forma, temos:

$$2^k >= n$$
$$log(2^k) >= log(n)$$
$$k >= log(n)$$

Assim, o laço mais interno executa um total de $log(n)$ vezes.

Como o laço principal executa um total de $n$ vezes, temos a complexidade assintótica de $O(n \cdot log(n))$.

# Problema F

**Procedimento**:

```
#include <stdio.h>
#include <stdlib.h>

int buscaElemento(int *vetor, int n, int elemento){
    int inicio = 0, fim = n - 1;
    while(inicio <= fim){
        int meio = (inicio + fim) / 2;
        if(vetor[meio] == elemento){
            // Elemento foi encontrado. Retorna sua posição
            return meio;
        } else if (vetor[meio] > elemento){
            fim = meio - 1;
        } else {
            inicio = meio + 1;
        }
    }
    // Não encontrou o elemento no vetor
    return -1;
}
int main(){
    int n;
    scanf("%d", &n);
    int *vetor = (int*) malloc (sizeof(int) * n);
    for(int i = 0; i < n; ++i){
        vetor[i] = i;
    }
    int elemento;
    scanf("%d", &elemento);
    int resposta = buscaElemento(vetor, n, elemento);
    if(resposta == -1){
        printf("Elemento nao foi encontrado.\n");
    } else {
        printf("Elemento encontrado na posicao %d!\n", resposta);
    }
```

```
    return 0;
}
```

**Resposta**: $O(log(n))$

**Explicação**:

Desconsiderando a operação de leitura, o procedimento mais custoso que o código realiza é a função `buscaElemento()`. Nessa função, sempre o algoritmo particiona o vetor em 2 partes, a partir da escolha do meio. Caso o elemento buscado esteja dentro do intervalo $[inicio, meio]$, a busca descarta a parcela $[meio+1, fim]$, e o contrário quando o elemento buscado está dentro do intervalo $[meio+1, fim]$. A busca só termina quando o elemento é encontrado.

Como estamos realizando uma análise do comportamento assintótico da função, podemos considerar que ela só encerrará quando $inicio == fim$, que determinará se o elemento pertence ao vetor ou não.

Podemos observar que a função sempre divide o vetor ao meio, essa propriedade só é possível porque o vetor está ordenado. Caso isso não fosse verdade, poderíamos descartar uma parcela que poderia conter o elemento.

Levando isso em conta, o algoritmo divide $n$ sucessivamente por 2, até que o valor restante em $n$ seja igual a 1. Dessa forma, como vimos anteriormente, esse comportamento é característico de uma complexidade $log(n)$.

Sendo assim, a complexidade assintótica do procedimento é $O(log(n))$.