

## Trabalho da Disciplina de Estrutura de Dados Básicas – Unidade 1

**Aluno:** Expedito Hebert Firmino da Rocha

**Link do Repositório:** <https://github.com/ShinigameBR/cpp-complexity-algorithms-EDB1>

### Exercício 1:

```
1  int busca_sequencial_recursiva(int *a, int n, int x)
2  {
3      if (n == 0)
4          return -1;
5      else if (x == a[n - 1])
6          return n - 1;
7      else
8          return busca_sequencial_recursiva(a, n - 1, x);
9  }
```

Para a função de busca sequencial recursiva, a complexidade temporal no pior caso é  $O(n)$ , onde  $n$  é o tamanho do array. Vamos demonstrar isso matematicamente:

Se  $T(n)$  é a função que descreve o número de operações necessárias para encontrar um elemento em um array de tamanho  $n$ , então podemos definir:

$T(n) = O(1)$  se  $n = 1$ , ou

$T(n) = T(n - 1) + O(1)$  caso contrário

Então,

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

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

...

$T(n) = T(n - i) + i * O(1)$ , onde  $i$  é tal que  $n - i = 1$ .

$T(n) = T(1) + (n - 1) * O(1)$ , note que,  $T(1) = O(1)$ .

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

$T(n) = n * O(1)$ .

Portanto,  $T(n) \in O(n)$ .

Quanto à comparação com a versão iterativa, sim, ambas as versões têm a mesma complexidade temporal no pior caso, que é  $O(n)$ . Isso ocorre porque ambas precisam percorrer todos os elementos do array no pior caso. A diferença está na forma como esse percurso é realizado: a versão recursiva

usa chamadas de função recursiva, enquanto a versão iterativa usa loops. No entanto, em termos de complexidade temporal, ambas são lineares.

### Exercício 2:

```
1  int busca_binaria_iterativa(int *a, int n, int x)
2  {
3      int begin = 0;
4      int end = n - 1;
5
6      while (begin <= end) {
7
8          int mid = (begin + end) / 2;
9
10         if (a[mid] == x) {
11             return mid;
12         }
13
14         if (a[mid] < x) {
15             begin = mid + 1;
16         } else {
17             end = mid;
18         }
19     }
20
21     return -1;
22 }
```

Para a função de busca binária iterativa, a complexidade temporal no pior caso é  $O(\log n)$ , onde  $n$  é o tamanho do array. Vamos demonstrar isso matematicamente:

Se  $T(n)$  é a função que descreve o número de operações necessárias para encontrar um elemento em um array ordenado de tamanho  $n$ , então podemos definir:

$T(n) = O(1)$  se  $n = 1$ , ou

$T(n) = O(1) + T(n / 2)$ , caso contrário

Então,

$T(n) = O(1) + T(n / 2)$

$$T(n) = O(1) + O(1) + T(n / 4)$$

...

Perceba que a cada iteração, o valor de  $n$  é dividido pela metade. Portanto, após  $k$  iterações, teremos  $n / 2^k = 1$ . Isso implica que  $k = \log_2 n$ . Logo,

$$T(n) = k + 1 = \log_2 n + 1$$

Portanto,  $T(n) \in O(\log n)$ .

Quanto à comparação entre a versão iterativa e a versão recursiva da busca binária, ambas possuem a mesma complexidade temporal no pior caso, que é  $O(\log n)$ . Isso ocorre porque em ambas as versões, o espaço de busca é dividido pela metade a cada iteração, reduzindo o número de elementos a serem considerados.

No melhor caso, o elemento que estamos procurando é o elemento central do array, e podemos encontrá-lo imediatamente sem a necessidade de fazer mais nenhuma comparação adicional. Portanto, tanto a versão iterativa quanto a versão recursiva da busca binária têm complexidade temporal  $O(1)$  no melhor caso, indicando que podem encontrar o elemento desejado em tempo constante quando o elemento está localizado exatamente no meio do array.

Agora com relação a complexidade espacial, na versão recursiva, a pilha de chamadas recursivas consome espaço adicional na memória. Como a busca binária realiza recursão em metades do array, o tamanho máximo da pilha de chamadas recursivas é proporcional ao número de divisões do array, que é  $O(\log n)$ . Na versão iterativa, não há uso significativo de memória adicional, além das variáveis locais necessárias para manter o estado do algoritmo. Não há aumento de consumo de memória com o tamanho do array logo a complexidade espacial é  $O(1)$ .

### Exercício 3:

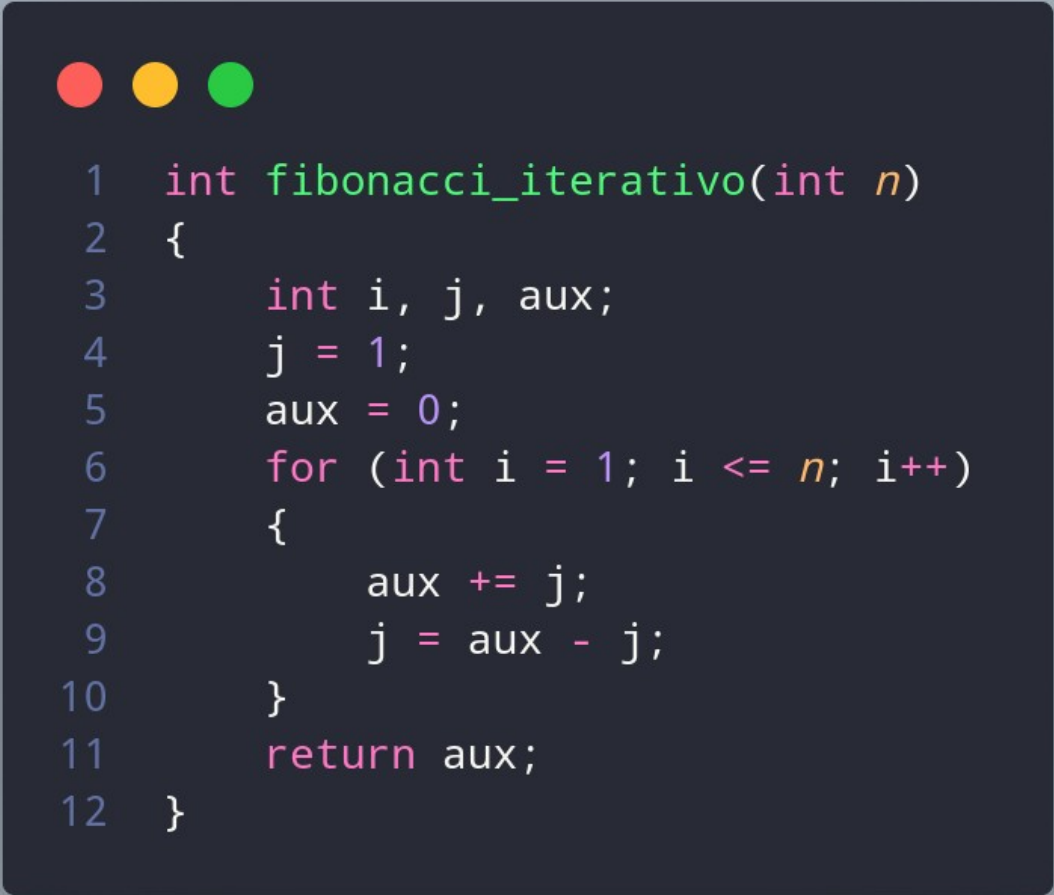
```
1  bool verificar_ordem_crescente(int a[], int n)
2  {
3      for (int i = 0; i < n; i++)
4      {
5          if (a[i] > a[i + 1])
6          {
7              return false;
8          }
9      }
10     return true;
11 }
```

Para a função de verificar se a ordem é crescente, a complexidade temporal no pior caso é  $O(n)$ , onde  $n$  é o tamanho do array. Pois, no pior caso, o algoritmo terá que percorrer todo o array para verificar se os elementos estão em ordem crescente. Isso ocorre quando o array está completamente desordenado, e a condição no interior do loop `if ( a[i] > a [i + 1] )` é sempre verdadeira.

Dentro do loop, há uma única comparação `a[i] > a [i + 1]`. Portanto, a complexidade temporal no pior caso é dada por  $T(n) = O(n)$ .

No melhor caso, o algoritmo não precisará percorrer o array inteiro. Isso ocorre pois no primeiro elemento já se percebe que o vetor não está ordenado de forma crescente fazendo assim a complexidade temporal no melhor caso ser  $O(1)$ . Dentro do loop, então acontecerá uma única comparação `a[i] > a [i + 1]`. Portanto, a complexidade temporal no pior caso é dada por  $T(n) = O(1)$ .

#### Exercício 4:



```
1  int fibonacci_iterativo(int n)
2  {
3      int i, j, aux;
4      j = 1;
5      aux = 0;
6      for (int i = 1; i <= n; i++)
7      {
8          aux += j;
9          j = aux - j;
10     }
11     return aux;
12 }
```

Para a função Fibonacci iterativo, a complexidade temporal no pior caso é  $O(n)$ , onde  $n$  é o tamanho do array. Pois, o algoritmo itera  $n$  vezes, uma vez para cada número na sequência de Fibonacci até o  $n$ -ésimo número. Dentro do loop, há operações de atribuição e adição sendo que ambas são operações de tempo constante. Como o loop é executado  $n$  vezes, a complexidade temporal deste

algoritmo é linear em relação a  $n$ , ou seja,  $O(n)$  no pior caso. Portanto, a complexidade temporal no pior caso é dada por  $T(n) = O(n)$ .

Já a versão recursiva apresentada nas questões tem complexidade temporal exponencial, sendo  $O(2^n)$ , onde  $n$  é o número de chamadas recursivas para calcular o termo de Fibonacci. Isso ocorre porque a função realiza chamadas recursivas para calcular os termos anteriores a cada iteração, resultando em uma árvore de chamadas recursivas exponencial. Isso leva a um alto consumo de recursos computacionais à medida que o valor de  $n$  aumenta, tornando a função ineficiente para valores grandes de  $n$ .

Com relação a uma forma de implementar uma versão recursiva mais eficiente em termos de complexidade temporal, podemos usar uma técnica chamada "memorização" (ou "memoization" em inglês), onde os resultados intermediários são armazenados para evitar recálculos desnecessários. Isso reduz drasticamente o número de chamadas recursivas, melhorando a eficiência do algoritmo.

Para essa implementação em C++, poderíamos usar um `unordered_map` chamado `memo` para armazenar os valores já calculados. Se o valor de  $n$  já estiver presente no `memo`, retornamos o valor correspondente sem precisar recalculá-lo. Isso melhora significativamente a eficiência do algoritmo, reduzindo sua complexidade temporal para  $O(n)$ , onde  $n$  é o número de elementos da sequência de Fibonacci que precisam ser calculados.

```
1  unordered_map<int, int> memo;
2
3  int fibonacci(int n) {
4      if (n <= 2) {
5          return 1;
6      }
7      if (memo.find(n) != memo.end()) {
8          return memo[n];
9      }
10     memo[n] = fibonacci(n - 1) + fibonacci(n - 2);
11     return memo[n];
12 }
```