

UFRN
Disciplina EDB1 – IMD 0029 - Turma 05
Nome: Cristovao Lacerda Cronje
Matricula: 20230002694

Trabalho 1 – Análise de Algoritmos

Questão 1:

Complexidade de Tempo - Busca Sequencial Recursiva

Link do código completo: https://github.com/Croncl/EDB1/blob/main/Atividade%2016%2004%202024/questao1_busca_sequencial_recursiva.cpp

```
int busca_sequencial_recursiva(int *vetor, int tamanho, int valor) {  
    if (tamanho == 0) {  
        return -1; // Se o vetor estiver vazio, retorna -1  
    }  
    if (vetor[tamanho - 1] == valor) {  
        return tamanho - 1; // Retorna o índice se encontrar o valor  
    } else {  
        // Chama recursivamente com um subvetor  
        return busca_sequencial_recursiva(vetor, tamanho - 1, valor);  
    }  
}
```

- Melhor caso: $O(1)$ - o elemento está na última posição do vetor.
- Pior caso: $O(n)$ - o elemento não está no vetor ou está na primeira posição.

Número de Instruções Executadas

- Atribuição de variáveis: 0 (não há atribuições diretas de variáveis, mas operações de comparação e retorno são contadas como operações básicas).
- Verificação do tamanho: 1 operação.
- Verificação do valor no vetor (na primeira chamada recursiva): 1 operação.
- Chamada recursiva (para o pior caso): n chamadas recursivas (tamanho - 1, tamanho - 2, ..., 1).
- Total de operações (para o pior caso): $1 + 1 + n = n + 2$ operações.

Demonstração Matemática

Considerando um vetor com 5 elementos {1, 2, 3, 4, 5} e buscando o elemento 5:

- Verificação do tamanho: 1 operação.
- Verificação do valor no vetor (na primeira chamada recursiva): 1 operação.
- Chamada recursiva (para o pior caso): 5 chamadas recursivas (tamanho - 1, tamanho - 2, ..., 1).
- Total de operações (para o pior caso): $1 + 1 + 5 = 7$ operações.

Portanto, a complexidade de tempo é $O(n)$ para o pior caso, pois o número de chamadas recursivas aumenta linearmente com o tamanho do vetor. No entanto, no melhor caso, a complexidade de tempo é $O(1)$, pois o elemento está na última posição do vetor, e a função retorna imediatamente sem chamar recursivamente.

Questao 2:

Complexidade de Tempo - Busca Sequencial Iterativa

Link do código completo: https://github.com/Croncl/EDB1/blob/main/Atividade%2016%2004%202024/questao2_busca_sequencial_iterativa.cpp

```
int busca_sequencial_iterativa(int *vetor, int tamanho, int valor) {
    for (int indice = 0; indice < tamanho; ++indice) {
        if (valor == vetor[indice]) {
            return indice; // Encontrou o valor, retorna o índice
        }
    }
    return -1; // Valor não encontrado
}
```

- Melhor Caso: $O(1)$ - o elemento está na primeira posição do vetor.
- Pior Caso: $O(n)$ - o elemento não está no vetor ou está na última posição.

Número de Instruções Executadas

- Atribuição de variáveis: 2 operações.
- Comparação 'índice < tamanho': n operações.
- Comparação 'valor == vetor[indice]': n operações.
- Retorno do índice: 1 operação.

$$= 2n + 3$$

Demonstração Matemática

Para exemplificar, considerando um vetor com 5 elementos {1, 2, 3, 4, 5} e buscando o elemento 1:

- Atribuição de variáveis: 2 operações.
- Comparação 'índice < tamanho': 1 operação.
- Verificação do valor no vetor (na primeira iteração): 1 operação.
- Total de operações (para o melhor caso): $2 + 1 + 1 = 4$ operações.

Vamos considerar o exemplo de busca pelo elemento 5 em um vetor com 5 elementos {1, 2, 3, 4, 5}:

- Atribuição de variáveis: 2 operações.
- Verificação do tamanho: 1 operação.
- Iterações no loop:
 - Na primeira iteração: 1 operação de comparação.
 - Na segunda iteração: 1 operação de comparação.

- ...
- Na última iteração (para o pior caso): 1 operação de comparação.

Neste caso, o número total de operações para o pior caso, onde o elemento não está presente no vetor ou está na última posição, é: 2 (atribuição de variáveis) + 1 (verificação do tamanho) + 1 (verificação na primeira iteração) + 1 (verificação na segunda iteração) + ... + 1 (verificação na última iteração) = $n + 3$ operações.

A complexidade de tempo é $O(1)$ para o melhor caso e $O(n)$ para o pior caso, pois a função percorre todo o vetor em busca do elemento.

Questão 3:

Complexidade de Tempo – Verificar ordenação

Link do código completo: https://github.com/Croncl/EDB1/blob/main/Atividade%2016%2004%202024/questao3_verifica_ordenacao.cpp

```
bool verifica_ordenacao(const vector<int>& vetor) {
    for (size_t i = 0; i < vetor.size() - 1; ++i) {
        if (vetor[i] > vetor[i + 1]) {
            return false;
        }
    }
    return true;
}
```

- Melhor caso: $O(n)$ - Se o vetor estiver ordenado de forma crescente, a função precisará percorrer todo o vetor para verificar a ordenação.
- Pior caso: $O(n)$ - Se o vetor não estiver ordenado a partir dos primeiros elementos, a função precisará percorrer o vetor até encontrar uma violação da ordenação.

Número de Operações Executadas

- Atribuição de variáveis: 1 operação.
- Verificação do tamanho do vetor: 1 operação.
- Verificação da ordenação no loop: $n - 1$ operações no pior caso, onde n é o tamanho do vetor.
- Retorno do resultado: 1 operação.

Considerando um vetor com 5 elementos {1, 2, 3, 4, 5}:

- Verificação do tamanho: 1 operação.
- Verificação da ordenação no loop: 4 operações.
- Total de operações (para o pior caso): $1 + 1 + 4 + 1 = 7$ operações.

Portanto, a complexidade de tempo da função `verifica_ordenacao` é $O(n)$ para o pior caso, pois o número de operações aumenta linearmente com o tamanho do vetor.

Para o vetor {5, 4, 3, 2, 1}:

- Verificação do tamanho: 1 operação.
- Verificação da ordenação no loop: 1 operação.
- Total de operações (para o pior caso): $1 + 1 + 1 = 3$ operações.

Portanto, para o vetor {5, 4, 3, 2, 1}, no melhor caso, a complexidade de tempo é $O(1)$, pois a função retorna imediatamente ao encontrar uma violação da ordenação nos primeiros elementos do vetor.

Questão 4:

Complexidade de Tempo – Fibonacci iterativo

Link do código completo: https://github.com/Croncl/EDB1/blob/main/Atividade%2016%2004%202024/questao4_fibonacci.cpp

```
int fibonacci_iterativo(int n){
    int a = 0, b = 1, c;
    if(n == 0){
        return a;
    }
    for(int i = 2; i <= n; i++){
        c = a + b;
        a = b;
        b = c;
    }
    return b;
}
```

- Melhor caso: $O(1)$ - Quando $n = 0$, a função retorna imediatamente sem executar o loop. Quando $n = 1$, a função retorna imediatamente sem executar o loop, pois o termo de Fibonacci de índice 1 é 1
- Pior Caso: $O(n)$ - Para $n > 1$, a função executa um loop de 2 até n , realizando uma quantidade constante de operações em cada iteração.

Número de Instruções Executadas

1. Atribuições de variáveis: 3 operações.
2. Verificação do valor de n : 1 operação.
3. Execução do loop: $n - 1$ iterações, cada uma com 3 operações.
4. Retorno do valor de b : 1 operação.

Portanto, o número total de operações é $3 + 1 + 3(n - 1) + 1 = 3n + 2$, o que confirma a complexidade $O(n)$ para todos os casos.

Comparando com a versão recursiva do algoritmo de Fibonacci, a versão iterativa é mais eficiente em termos de complexidade temporal, especialmente para valores grandes de n , onde a versão recursiva exponencial se torna impraticável.

Fibonacci Recursivo

```
int fibonacci_recursivo(int n){  
    if (n <= 1) {  
        return n; // Casos base: Fibonacci(0) = 0, Fibonacci(1) = 1  
    } else {  
        return fibonacci_recursivo(n - 1) + fibonacci_recursivo(n - 2); // Recursão  
    }  
}
```

Complexidade de Tempo

- Melhor caso: $O(1)$ - Quando $n = 0$ ou $n = 1$, a função retorna imediatamente sem chamadas recursivas.
- Pior caso: $O(2^n)$ - Cada chamada recursiva gera duas novas chamadas, levando a uma árvore de chamadas exponencial.

Número de Chamadas Recursivas

- Para $n = 0$ ou $n = 1$, a função faz apenas 1 chamada recursiva.
- Para $n > 1$, a função faz duas chamadas recursivas a cada nível da árvore de chamadas, resultando em 2^n chamadas no total.

Comparação

- O algoritmo recursivo tem uma complexidade de tempo exponencial no pior caso, tornando-se muito lento para valores moderados de n .
- O algoritmo iterativo é muito mais eficiente, com uma complexidade de tempo linear, o que o torna preferível para calcular números de Fibonacci grandes.