

Trilha de Estrutura de Dados

Módulo 03: Recursividade

Instruções para a melhor prática de Estudo

1. **Leia atentamente todo o conteúdo:** Antes de iniciar qualquer atividade, faça uma leitura detalhada do material fornecido na trilha, compreendendo os conceitos e os exemplos apresentados.
2. **Não se limite ao material da trilha:** Utilize o material da trilha como base, mas busque outros materiais de apoio, como livros, artigos acadêmicos, vídeos, e blogs especializados. Isso enriquecerá o entendimento sobre o tema.
3. **Explore a literatura:** Consulte livros e publicações reconhecidas na área, buscando expandir seu conhecimento além do que foi apresentado. A literatura acadêmica oferece uma base sólida para a compreensão de temas complexos.
4. **Realize todas as atividades propostas:** Conclua cada uma das atividades práticas e teóricas, garantindo que você esteja aplicando o conhecimento adquirido de maneira ativa.
5. **Evite o uso de Inteligência Artificial para resolução de atividades:** Utilize suas próprias habilidades e conhecimentos para resolver os exercícios. O aprendizado vem do esforço e da prática.
6. **Participe de debates:** Discuta os conteúdos estudados com professores, colegas e profissionais da área. O debate enriquece o entendimento e permite a troca de diferentes pontos de vista.
7. **Pratique regularmente:** Não deixe as atividades para a última hora. Pratique diariamente e revise o conteúdo com frequência para consolidar o aprendizado.
8. **Peça feedback:** Solicite o retorno dos professores sobre suas atividades e participe de discussões sobre os erros e acertos, utilizando o feedback para aprimorar suas habilidades.

Essas instruções são fundamentais para garantir um aprendizado profundo e eficaz ao longo das trilhas.

Recursividade

1. Definição e Exemplos

Recursividade é uma técnica de programação onde uma função chama a si mesma para resolver subproblemas de uma questão maior. Um problema é decomposto em instâncias menores dele próprio até que se atinja uma condição base, a qual termina as chamadas recursivas.

Uma função recursiva possui três elementos fundamentais:

1. **Caso base** - condição que encerra a recursão
2. **Chamada recursiva** - chamada da função a si mesma
3. **Progresso** - cada chamada deve se aproximar do caso base

Exemplo Simples: Cálculo do Fatorial

O fatorial de um número n (representado como $n!$) é o produto de todos os números inteiros de 1 até n . Recursivamente, o fatorial pode ser definido como:

- $n! = n \times (n-1)!$ se $n > 1$
- $1! = 1$

Exemplo em Pseudocódigo:

```
function fatorial(n) {
    if(n === 1) {
        return 1; // Caso base
    } else {
        return n * fatorial(n - 1); // Chamada recursiva
    }
}
```

Neste exemplo, a função `fatorial` continua chamando a si mesma com o valor $n-1$ até que n seja 1, quando a recursão para.

Outro Exemplo: Sequência de Fibonacci

A sequência de Fibonacci é outra aplicação clássica da recursividade. A sequência é definida como:

- $F(0) = 0$
- $F(1) = 1$
- $F(n) = F(n-1) + F(n-2)$ se $n > 1$

Exemplo em Pseudocódigo:

```
function fibonacci(n) {
  if(n === 0) {
    return 0; // Caso base
  } else if(n === 1) {
    return 1; // Caso base
  } else {
    return fibonacci(n - 1) + fibonacci(n - 2); // Chamadas recursivas
  }
}
```

Neste exemplo, a função `fibonacci` chama a si mesma duas vezes até alcançar os casos base $F(0)F(0)F(0)$ ou $F(1)F(1)F(1)$.

2. Relacionamento com Estruturas de Dados

Recursividade é amplamente utilizada em algoritmos que manipulam estruturas de dados, principalmente em árvores e grafos. Algumas das operações sobre essas estruturas podem ser implementadas de forma mais intuitiva e eficiente usando recursão.

- **Árvores:** A travessia de uma árvore (como as ordens *in-order*, *pre-order* e *post-order*) é naturalmente recursiva, já que as árvores possuem uma natureza hierárquica, onde cada subárvore pode ser tratada como uma árvore completa.

Exemplo: Percurso In-Order em uma Árvore Binária:

Estrutura de um Nó da Árvore (Simples)

```
function Node(value) {
  this.value = value;
  this.left = null;
  this.right = null;
}
```

Percurso In-Order

```
function inOrder(node) {
  if(node !== null) {
    inOrder(node.left); // Visita a subárvore esquerda
    console.log(node.value); // Visita o nó atual
    inOrder(node.right); // Visita a subárvore direita
  }
}
```

Neste algoritmo, cada sub-árvore é percorrida recursivamente até que os nós folha sejam alcançados.

Neste algoritmo, cada chamada da função trata o nó atual como a raiz de uma subárvore. A recursão continua percorrendo as subárvores esquerda e direita até que sejam alcançados os nós folha. Quando um nó não possui filhos, suas referências são nulas, encerrando naturalmente as chamadas recursivas.

- **Listas:** Embora menos comum em listas, é possível usar a recursividade para percorrer uma lista encadeada, especialmente ao trabalhar com listas recursivas como pilhas e filas.

Exemplo: Contar Nós em uma Lista Encadeada:

Função Recursiva

```
function countNodes(node) {
    if(node === null) {
        return 0; // Caso base: fim da lista
    } else {
        return 1 + countNodes(node.next); // Chamada recursiva
    }
}
```

Aqui, cada nó chama a função para o próximo nó até que o final da lista seja alcançado.

3. Análise de Desempenho de Algoritmos Recursivos

A análise de desempenho de algoritmos recursivos é similar à análise de algoritmos iterativos, mas muitas vezes envolve o uso de **relações de recorrência** para descrever o tempo de execução.

- **Relações de Recorrência:** Uma relação de recorrência é uma equação que define o tempo de execução $T(n)$ de um algoritmo em termos do tempo de execução para instâncias menores do problema.

Exemplo: Análise do Algoritmo Fatorial

Para o algoritmo do fatorial, temos:

- Tempo para o caso base $T(1)=O(1)$ (tempo constante).
- Cada chamada recursiva envolve uma multiplicação e uma chamada recursiva para $n-1$.

A relação de recorrência seria:

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

Isso resulta em uma complexidade de tempo **$O(n)$** , pois haverá n chamadas recursivas.

Exemplo: Análise do Algoritmo Fibonacci

Para o algoritmo Fibonacci recursivo, cada chamada recursiva para n resulta em duas novas chamadas recursivas para $n-1$ e $n-2$. Isso leva a uma relação de recorrência mais complexa:

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

A solução para essa recorrência é exponencial, resultando em uma complexidade de tempo $O(2^n)$. Esta ineficiência ocorre porque muitas chamadas recursivas recalculam os mesmos valores.

Nota

Recursividade é uma ferramenta poderosa, especialmente quando usada em conjunto com estruturas de dados como árvores e listas. No entanto, a análise do desempenho de algoritmos recursivos é crucial para evitar problemas de eficiência, como a recursão excessiva ou o recálculo de subproblemas, que pode ser resolvido com técnicas como memoization.

Lista de Exercícios de Fixação

1. **Fatorial Recursivo:**
 - Implemente uma função recursiva para calcular o fatorial de um número inteiro n .
 - Determine a complexidade de tempo do algoritmo.
2. **Sequência de Fibonacci:**
 - Implemente uma função recursiva que calcule o n -ésimo número da sequência de Fibonacci.
 - Analise o desempenho do algoritmo e sugira uma otimização (por exemplo, usando memoization ou uma abordagem iterativa).
3. **Travessia em Árvores:**
 - Implemente um algoritmo recursivo para realizar o percurso *in-order* de uma árvore binária.
 - Altere o código para implementar os percursos *pre-order* e *post-order*.
4. **Soma dos Elementos de uma Lista Encadeada:**
 - Crie uma função recursiva que percorra uma lista simplesmente encadeada e retorne a soma de todos os elementos da lista.
5. **Busca em uma Árvore Binária de Busca:**
 - Implemente uma função recursiva para buscar um valor em uma árvore binária de busca.
 - Qual é a complexidade de tempo da busca em termos de n , onde n é o número de nós na árvore?
6. **Torre de Hanói:**
 - Implemente o algoritmo recursivo para resolver o problema da Torre de Hanói, movendo n discos de uma haste para outra.

- Determine o número de movimentos necessários para resolver o problema e sua complexidade de tempo.
7. **Contagem de Nós em uma Lista Encadeada:**
- Implemente uma função recursiva que conte o número de nós em uma lista encadeada.
 - Analise o desempenho do algoritmo em termos de complexidade de tempo e espaço.