CAPÍTULOS

- Pesquisa binária: um algoritmo para encontrar um valor específico em uma lista ordenada.
- Busca em largura (BFS): um algoritmo para visitar todos os nós de um grafo, começando de um nó específico.
- Busca em profundidade (DFS): um algoritmo para visitar todos os nós de um grafo, começando de um nó específico e explorando todos os seus filhos antes de explorar seus irmãos.
- Ordenação de mesclagem: um algoritmo para ordenar uma lista, dividindo-a em duas metades e mesclando-as de volta em uma ordem ordenada.
- Ordenação rápida: um algoritmo para ordenar uma lista, escolhendo um elemento pivote e dividindo a lista em duas metades, uma menor que o pivote e uma maior que o pivote.

Introdução

Olá, desenvolvedor iniciante!

Se você está aqui, é porque está interessado em aprender mais sobre algoritmos. Parabéns! Você está tomando a decisão certa.

Os algoritmos são a base da programação. Eles são o que permite que os computadores executem tarefas e resolvem problemas. Sem algoritmos, os computadores seriam apenas máquinas sem utilidade.

Existem muitos algoritmos diferentes, mas alguns são mais importantes do que outros. Neste livro, você aprenderá sobre os cinco principais algoritmos que todo desenvolvedor deve saber.

Esses algoritmos são usados em uma variedade de aplicações, incluindo processamento de dados, análise de dados, inteligência artificial e aprendizado de máquina.

Este livro é direcionado a desenvolvedores iniciantes que desejam aprender os fundamentos de algoritmos. Não é necessário conhecimento prévio de algoritmos ou programação.

Aqui está um breve resumo dos cinco algoritmos que você aprenderá neste livro:

- Pesquisa binária: um algoritmo para encontrar um valor específico em uma lista ordenada.
- Busca em largura (BFS): um algoritmo para visitar todos os nós de um grafo, começando de um nó específico.
- Busca em profundidade (DFS): um algoritmo para visitar todos os nós de um grafo, começando de um nó específico e explorando todos os seus filhos antes de explorar seus irmãos.
- Ordenação de mesclagem: um algoritmo para ordenar uma lista, dividindo-a em duas metades e mesclando-as de volta em uma ordem ordenada.
- Ordenação rápida: um algoritmo para ordenar uma lista, escolhendo um elemento pivote e dividindo a lista em duas metades, uma menor que o pivote e uma maior que o pivote.

Por que aprender algoritmos?

Aprender algoritmos é importante por vários motivos. Em primeiro lugar, os algoritmos são uma parte essencial da programação. Sem um entendimento básico de algoritmos, você não será capaz de escrever programas eficazes.

Em segundo lugar, os algoritmos são usados em uma variedade de aplicações, incluindo processamento de dados, análise de dados, inteligência artificial e aprendizado de máquina. Se você deseja trabalhar em qualquer uma dessas áreas, é importante ter um conhecimento sólido de algoritmos.

Em terceiro lugar, aprender algoritmos pode ajudá-lo a melhorar suas habilidades de resolução de problemas. Os algoritmos são uma forma de pensar logicamente sobre problemas e encontrar soluções eficientes.

O que você aprenderá neste livro?

Neste livro, você aprenderá os seguintes tópicos sobre cada algoritmo:

- Definição: uma explicação do que o algoritmo faz.
- Funcionamento: uma explicação de como o algoritmo funciona.
- Complexidade: uma avaliação de quanto tempo o algoritmo leva para ser executado.

- Exemplos de uso: exemplos de como o algoritmo pode ser usado em aplicações práticas.

Além disso, você aprenderá sobre os seguintes tópicos gerais:

- Abordagem passo a passo: uma abordagem passo a passo para aprender algoritmos.
- Conceito de base: uma introdução a conceitos de base importantes, como variáveis, tipos de dados, operadores e loops.
- Resolução de problemas: dicas para resolver problemas usando algoritmos.

Como usar este livro

Este livro é projetado para ser usado por desenvolvedores iniciantes que desejam aprender os fundamentos de algoritmos. Não é necessário conhecimento prévio de algoritmos ou programação.

A melhor maneira de usar este livro é começar no início e seguir em frente. Cada capítulo se baseia no conhecimento do capítulo anterior, portanto, é importante entender os conceitos apresentados antes de passar para o próximo capítulo.

Você também pode usar este livro como um recurso de referência. Se você estiver trabalhando em um projeto e precisar usar um algoritmo específico, poderá consultar este livro para obter instruções.

O que você aprenderá ao final deste livro

Ao final deste livro, você será capaz de:

- Definir e explicar os cinco principais algoritmos que todo desenvolvedor deve saber.
- Descrever como cada algoritmo funciona.
- Avaliar a complexidade de cada algoritmo.
- Usar cada algoritmo em aplicações práticas.
- Resolver problemas usando algoritmos.

Espero que você aproveite este livro e aprenda muito sobre algoritmos!

Capítulo 1: Pesquisa binária

Definição

Pesquisa binária é um algoritmo de busca que encontra um valor específico em uma lista ordenada. O algoritmo funciona dividindo a lista ao meio em cada iteração, comparando o valor a ser encontrado com o valor no meio da lista. Se o valor a ser encontrado for maior que o valor no meio, a metade esquerda da lista é descartada. Se o valor a ser encontrado for menor que o valor no meio, a metade direita da lista é descartada. O processo continua até que o valor a ser encontrado seja encontrado ou até que a lista esteja vazia.

Funcionamento

Aqui está um exemplo de como a pesquisa binária funciona:

Java

```
public static int binarySearch(int[] array, int value) {
  int low = 0;
  int high = array.length - 1;

  while (low <= high) {
    int mid = (low + high) / 2;

    if (array[mid] == value) {
      return mid;
    } else if (array[mid] < value) {
      low = mid + 1;
    } else {
      high = mid - 1;
    }
  }

  return -1;
}
```

Use o código com cuidado. Saiba mais
content_copy

Este código funciona dividindo a lista em duas metades, comparando o valor a ser encontrado com o valor no meio da lista. Se o valor a ser encontrado for igual ao valor no meio, o algoritmo retorna o índice do valor. Se o valor a ser encontrado for maior que o valor no meio, a metade esquerda da lista é descartada. Se o valor a ser encontrado for menor que o valor no meio, a metade direita da lista é descartada. O processo continua até que o valor a ser encontrado seja encontrado ou até que a lista esteja vazia.

Complexidade

A complexidade de tempo da pesquisa binária é O(log n), onde n é o tamanho da lista. Isso significa que o tempo de execução do algoritmo aumenta linearmente com o tamanho da lista.

Exemplos de uso

Pesquisa binária pode ser usada em uma variedade de aplicações, incluindo:

- Encontrar um valor específico em uma lista ordenada
- Localizar um item em uma árvore binária
- Comparar dois conjuntos de dados

Implementação em Java

Aqui está uma implementação de pesquisa binária em Java:

Java
```
public static int binarySearch(int[] array, int value) {
  int low = 0;
  int high = array.length - 1;

  while (low <= high) {
    int mid = (low + high) / 2;

    if (array[mid] == value) {
      return mid;
    } else if (array[mid] < value) {
      low = mid + 1;
    } else {
      high = mid - 1;
    }
  }

  return -1;
}
```
Use o código com cuidado. Saiba mais
content_copy

Este código funciona da mesma maneira que a implementação em Python, dividindo a lista ao meio em cada iteração e comparando o valor a ser encontrado com o valor no meio da lista. Se o valor a ser encontrado for igual ao valor no meio, o algoritmo retorna o índice do valor. Se o valor a ser encontrado for maior que o valor no meio, a metade esquerda da lista é descartada. Se o valor a ser encontrado for menor que o valor no meio, a metade direita da lista é descartada. O processo continua até que o valor a ser encontrado seja encontrado ou até que a lista esteja vazia.

Exemplo de uso

Aqui está um exemplo de como usar pesquisa binária para encontrar um valor específico em uma lista ordenada:

Java

```
int[] array = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19};

int value = 13;

int index = binarySearch(array, value);

if (index == -1) {
  System.out.println("O valor não foi encontrado.");
} else {
  System.out.println("O valor foi encontrado no índice " + index);
}
```

Use o código com cuidado. Saiba mais
content_copy

Este código irá imprimir o seguinte:

```
O valor foi encontrado no índice 6.
```

Vantagens e desvantagens

Pesquisa binária é uma técnica de busca muito eficiente, com complexidade de tempo O(log n). Isso significa que o tempo de execução do algoritmo

CAPÍTULO 2

Busca em largura (BFS)

Introdução

Busca em largura (BFS) é um algoritmo de busca que visita todos os nós de um grafo, começando de um nó específico. O algoritmo funciona visitando cada vizinho do nó inicial, depois visitando cada vizinho dos vizinhos do nó inicial, e assim por diante. O processo continua até que todos os nós do grafo tenham sido visitados.

BFS é um algoritmo de busca recursivo. Isso significa que ele chama a si mesmo para visitar cada nó no grafo.

Funcionamento

Aqui está um exemplo de como a busca em largura (BFS) funciona:

Java

```
public class BFS {

    public static void bfs(Graph graph, Node startNode) {
        Queue<Node> queue = new LinkedList<>();
        queue.add(startNode);

        while (!queue.isEmpty()) {
            Node node = queue.poll();

            System.out.println(node);

            for (Node neighbor : node.neighbors) {
                if (!neighbor.visited) {
                    neighbor.visited = true;
                    queue.add(neighbor);
                }
            }
        }
    }
}
```

Use o código com cuidado. Saiba mais
content_copy

Este código funciona criando uma fila para armazenar os nós que ainda não foram visitados. O nó inicial é adicionado à fila. Enquanto a fila não estiver vazia, o nó no topo da fila é removido e visitado. Todos os vizinhos do nó visitado são adicionados à fila. O processo continua até que todos os nós do grafo tenham sido visitados.

Complexidade

A complexidade de tempo da busca em largura (BFS) é O(V + E), onde V é o número de nós do grafo e E é o número de arestas do grafo. Isso significa que o tempo de execução do algoritmo aumenta linearmente com o número de nós e arestas do grafo.

Exemplos de uso

Busca em largura (BFS) pode ser usada em uma variedade de aplicações, incluindo:

- Encontrar o caminho mais curto entre dois nós em um grafo
- Encontrar todos os componentes conectados em um grafo
- Encontrar todos os caminhos possíveis de um nó para outro

Implementação em Java

Aqui está uma implementação completa da busca em largura (BFS) em Java:

Java

```
public class BFS {

    public static void bfs(Graph graph, Node startNode) {
        Queue<Node> queue = new LinkedList<>();
        queue.add(startNode);

        while (!queue.isEmpty()) {
            Node node = queue.poll();

            System.out.println(node);

            for (Node neighbor : node.neighbors) {
                if (!neighbor.visited) {
                    neighbor.visited = true;
                    queue.add(neighbor);
                }
            }
        }
    }
}

public class Node {

    private String name;
    private List<Node> neighbors;
    private boolean visited;

    public Node(String name) {
        this.name = name;
        this.neighbors = new ArrayList<>();
        this.visited = false;
    }

    public void addNeighbor(Node neighbor) {
        this.neighbors.add(neighbor);
    }

    public String getName() {
        return this.name;
    }

    public List<Node> getNeighbors() {
        return this.neighbors;
    }

    public boolean isVisited() {
        return this.visited;
    }

    public void setVisited(boolean visited) {
        this.visited = visited;
    }
}

public class Graph {

    private List<Node> nodes;
```

```
    public Graph() {
        this.nodes = new ArrayList<>();
    }

    public void addNode(Node node) {
        this.nodes.add(node);
    }

    public List<Node> getNodes() {
        return this.nodes;
    }
}
```

Capítulo 3: Busca em largura (BFS)

Introdução

Busca em largura (BFS) é um algoritmo de busca que visita todos os nós de um grafo, começando de um nó específico. O algoritmo funciona visitando cada vizinho do nó inicial, depois visitando cada vizinho dos vizinhos do nó inicial, e assim por diante. O processo continua até que todos os nós do grafo tenham sido visitados.

BFS é um algoritmo de busca recursivo. Isso significa que ele chama a si mesmo para visitar cada nó no grafo.

Funcionamento

Aqui está um exemplo de como a busca em largura (BFS) funciona:

Java

```
public class BFS {

    public static void bfs(Graph graph, Node startNode) {
        Queue<Node> queue = new LinkedList<>();
        queue.add(startNode);

        while (!queue.isEmpty()) {
            Node node = queue.poll();

            System.out.println(node);

            for (Node neighbor : node.neighbors) {
                if (!neighbor.visited) {
                    neighbor.visited = true;
                    queue.add(neighbor);
                }
```

```
            }
        }
    }
}
```
Use o código com cuidado. Saiba mais
content_copy

Este código funciona criando uma fila para armazenar os nós que ainda não foram visitados. O nó inicial é adicionado à fila. Enquanto a fila não estiver vazia, o nó no topo da fila é removido e visitado. Todos os vizinhos do nó visitado são adicionados à fila. O processo continua até que todos os nós do grafo tenham sido visitados.

Complexidade

A complexidade de tempo da busca em largura (BFS) é O(V + E), onde V é o número de nós do grafo e E é o número de arestas do grafo. Isso significa que o tempo de execução do algoritmo aumenta linearmente com o número de nós e arestas do grafo.

Exemplos de uso

Busca em largura (BFS) pode ser usada em uma variedade de aplicações, incluindo:

- Encontrar o caminho mais curto entre dois nós em um grafo
- Encontrar todos os componentes conectados em um grafo
- Encontrar todos os caminhos possíveis de um nó para outro

Implementação em Java

Aqui está uma implementação completa da busca em largura (BFS) em Java:

Java
```
public class BFS {

    public static void bfs(Graph graph, Node startNode) {
        Queue<Node> queue = new LinkedList<>();
        queue.add(startNode);

        while (!queue.isEmpty()) {
            Node node = queue.poll();

            System.out.println(node);

            for (Node neighbor : node.neighbors) {
```

```
                if (!neighbor.visited) {
                    neighbor.visited = true;
                    queue.add(neighbor);
                }
            }
        }
    }
}

public class Node {

    private String name;
    private List<Node> neighbors;
    private boolean visited;

    public Node(String name) {
        this.name = name;
        this.neighbors = new ArrayList<>();
        this.visited = false;
    }

    public void addNeighbor(Node neighbor) {
        this.neighbors.add(neighbor);
    }

    public String getName() {
        return this.name;
    }

    public List<Node> getNeighbors() {
        return this.neighbors;
    }

    public boolean isVisited() {
        return this.visited;
    }

    public void setVisited(boolean visited) {
        this.visited = visited;
    }
}

public class Graph {

    private List<Node> nodes;

    public Graph() {
        this.nodes = new ArrayList<>();
    }

    public void addNode(Node node) {
        this.nodes.add(node);
    }

    public List<Node> getNodes() {
        return this.nodes;
    }
}
```

Este código implementa um gráfico básico usando uma lista de nós. Cada nó no gráfico tem uma lista de vizinhos. O método `bfs()` implementa o algoritmo de busca em largura (BFS).

Capítulo 3

Busca em profundidade (DFS)

Definição

Busca em profundidade (DFS) é um algoritmo de busca que visita todos os nós de um grafo, começando de um nó específico. O algoritmo funciona visitando cada vizinho do nó inicial, depois visitando cada vizinho dos vizinhos do nó inicial, e assim por diante. O processo continua até que um nó sem vizinhos seja encontrado. Em seguida, o algoritmo volta para o último nó visitado e continua a explorar seus vizinhos.

Funcionamento

Aqui está um exemplo de como a busca em profundidade (DFS) funciona:

Python
```
def dfs(graph, start_node):
    visited = set()

    def dfs_recursive(node):
        visited.add(node)
        print(node)

        for neighbor in graph[node]:
            if neighbor not in visited:
                dfs_recursive(neighbor)

    dfs_recursive(start_node)
```
Use o código com cuidado. Saiba mais
content_copy

Este código funciona criando um conjunto para armazenar os nós que já foram visitados. O nó inicial é adicionado ao conjunto. Em seguida, o algoritmo chama a função `dfs_recursive()`, passando o nó inicial como parâmetro. A função

`dfs_recursive()` visita o nó passado como parâmetro e, em seguida, visita cada vizinho do nó que ainda não foi visitado. O processo continua até que todos os nós do grafo tenham sido visitados.

Complexidade

A complexidade de tempo da busca em profundidade (DFS) é O(V + E), onde V é o número de nós do grafo e E é o número de arestas do grafo. Isso significa que o tempo de execução do algoritmo aumenta linearmente com o número de nós e arestas do grafo.

Exemplos de uso

Busca em profundidade (DFS) pode ser usada em uma variedade de aplicações, incluindo:

- Encontrar o caminho mais curto entre dois nós em um grafo
- Encontrar todos os componentes conectados em um grafo
- Encontrar todos os caminhos possíveis de um nó para outro

Exemplo em Java

Aqui está um exemplo de como implementar a busca em profundidade (DFS) em Java:

Java
```
public class DFS {

    public static void dfs(Graph graph, Node startNode) {
        Set<Node> visited = new HashSet<>();

        dfs(graph, startNode, visited);
    }

    private static void dfs(Graph graph, Node node, Set<Node> visited)
{
        visited.add(node);
        System.out.println(node);

        for (Node neighbor : graph.getNeighbors(node)) {
            if (!visited.contains(neighbor)) {
                dfs(graph, neighbor, visited);
            }
        }
    }
}
```

Este código funciona da mesma forma que o código Python apresentado anteriormente. Ele cria um conjunto para armazenar os nós que já foram visitados e, em seguida, chama a função `dfs()`, passando o nó inicial e o conjunto como parâmetros. A função `dfs()` visita o nó passado como parâmetro e, em seguida, visita cada vizinho do nó que ainda não foi visitado. O processo continua até que todos os nós do grafo tenham sido visitados.

Vantagens e desvantagens

Vantagens:

- A busca em profundidade (DFS) é um algoritmo muito simples de implementar.
- A busca em profundidade (DFS) pode ser usada para encontrar o caminho mais curto entre dois nós em um grafo.
- A busca em profundidade (DFS) pode ser usada para encontrar todos os componentes conectados em um grafo.

Desvantagens:

- A busca em profundidade (DFS) pode ser ineficiente em grafos com muitos loops.
- A busca em profundidade (DFS) pode não encontrar o caminho mais curto entre dois nós em um grafo com ciclos.

CAPÍTULO 4

Ordenação de mesclagem

Definição

Ordenação de mesclagem é um algoritmo de ordenação que funciona dividindo a lista a ser ordenada em duas metades. Em seguida, as metades são ordenadas recursivamente. Finalmente, as duas metades ordenadas são mescladas de volta em uma lista ordenada.

Funcionamento

Aqui está um exemplo de como a ordenação de mesclagem funciona:

Python

```
def merge_sort(array):
    if len(array) < 2:
        return array

    middle = len(array) // 2
    left_array = merge_sort(array[:middle])
    right_array = merge_sort(array[middle:])

    return merge(left_array, right_array)

def merge(left_array, right_array):
    merged_array = []
    i = 0
    j = 0

    while i < len(left_array) and j < len(right_array):
        if left_array[i] < right_array[j]:
            merged_array.append(left_array[i])
            i += 1
        else:
            merged_array.append(right_array[j])
            j += 1

    merged_array += left_array[i:]
    merged_array += right_array[j:]

    return merged_array
```


Este código funciona dividindo a lista em duas metades. As metades são ordenadas recursivamente usando a função `merge_sort()`. Finalmente, as duas metades ordenadas são mescladas de volta em uma lista ordenada usando a função `merge()`.

Complexidade

A complexidade de tempo da ordenação de mesclagem é O(n log n), onde n é o tamanho da lista. Isso significa que o tempo de execução do algoritmo aumenta logaritmicamente com o tamanho da lista.

Exemplos de uso

Ordenação de mesclagem pode ser usada em uma variedade de aplicações, incluindo:

- Ordenação de números
- Ordenação de strings
- Ordenação de objetos

Exemplo em Java

Aqui está um exemplo de como implementar ordenação de mesclagem em Java:

Java

```
public class MergeSort {

    public static void mergeSort(int[] array) {
        if (array.length < 2) {
            return;
        }

        int middle = array.length / 2;
        int[] leftArray = Arrays.copyOfRange(array, 0, middle);
        int[] rightArray = Arrays.copyOfRange(array, middle,
array.length);

        mergeSort(leftArray);
        mergeSort(rightArray);

        merge(array, leftArray, rightArray);
    }

    private static void merge(int[] array, int[] leftArray, int[]
rightArray) {
        int i = 0;
        int j = 0;
        int k = 0;

        while (i < leftArray.length && j < rightArray.length) {
            if (leftArray[i] < rightArray[j]) {
                array[k] = leftArray[i];
                i++;
            } else {
                array[k] = rightArray[j];
                j++;
            }
            k++;
        }

        while (i < leftArray.length) {
            array[k] = leftArray[i];
            i++;
            k++;
        }

        while (j < rightArray.length) {
            array[k] = rightArray[j];
            j++;
            k++;
        }
    }
}
```

Este código é muito semelhante ao código Python, com a diferença de que usa arrays em vez de listas.

CAPÍTULO 5

Ordenação rápida

Definição

Ordenação rápida é um algoritmo de ordenação que funciona escolhendo um elemento pivote da lista e dividindo a lista em duas partes, uma menor que o pivote e uma maior que o pivote. A ordenação rápida é então aplicada recursivamente às duas partes da lista, até que todas as listas sejam ordenadas.

Funcionamento

Aqui está um exemplo de como a ordenação rápida funciona:

Java

```
public class QuickSort {

    public static void quickSort(int[] array) {
        quickSort(array, 0, array.length - 1);
    }

    private static void quickSort(int[] array, int low, int high) {
        if (low < high) {
            int pivot = partition(array, low, high);

            quickSort(array, low, pivot - 1);
            quickSort(array, pivot + 1, high);
        }
    }

    private static int partition(int[] array, int low, int high) {
        int pivot = array[high];

        int i = low;
        for (int j = low; j < high; j++) {
            if (array[j] <= pivot) {
                int temp = array[i];
                array[i] = array[j];
                array[j] = temp;

                i++;
            }
```

```
        }

        int temp = array[i];
        array[i] = array[high];
        array[high] = temp;

        return i;
    }
}
```
Use o código com cuidado. Saiba mais
content_copy

Este código funciona escolhendo um elemento pivote da lista, neste caso o elemento no índice `high`. Em seguida, o algoritmo percorre a lista, começando do índice `low`, e troca cada elemento menor que o pivote com o elemento no índice `i`. O índice `i` é então incrementado.

Quando o algoritmo chega ao final da lista, o elemento no índice `i` é maior ou igual ao pivote. O algoritmo então troca o elemento no índice `i` com o pivote.

Finalmente, o algoritmo aplica a ordenação rápida recursivamente às duas partes da lista, uma menor que o pivote e uma maior que o pivote.

Complexidade

A complexidade de tempo da ordenação rápida é O(n log n), onde n é o tamanho da lista. Isso significa que o tempo de execução do algoritmo aumenta logaritmicamente com o tamanho da lista.

Exemplos de uso

Ordenação rápida pode ser usada em uma variedade de aplicações, incluindo:

- Ordenar listas de dados
- Ordenar arquivos
- Ordenar estruturas de dados

Exemplo de uso

Aqui está um exemplo de como a ordenação rápida pode ser usada para ordenar uma lista de nomes:

Java
```
String[] names = {"John", "Jane", "Mary", "Peter", "David"};

QuickSort.quickSort(names);

System.out.println(Arrays.toString(names));
```

Use o código com cuidado. Saiba mais
content_copy

A saída deste código é a seguinte:

```
[David, John, Mary, Peter, Jane]
```

Exemplo em Java

Aqui está um exemplo de como a ordenação rápida pode ser implementada em Java:

Java
```
public class QuickSort {

    public static void quickSort(int[] array) {
        quickSort(array, 0, array.length - 1);
    }

    private static void quickSort(int[] array, int low, int high) {
        if (low < high) {
            int pivot = partition(array, low, high);

            quickSort(array, low, pivot - 1);
            quickSort(array, pivot + 1, high);
        }
    }

    private static int partition(int[] array, int low, int high) {
        int pivot = array[high];

        int i = low;
        for (int j = low; j < high; j++) {
            if (array[j] <= pivot) {
                int temp = array[i];
                array[i] = array[j];
                array[j] = temp;

                i++;
            }
        }

        int temp = array[i];
        array[i]
```
Use o código com cuidado. Saiba mais