Algoritmos de Busca para Maratonas de Programação em Python

1. Busca Linear (Linear Search)

python

```python
def busca_linear(lista, alvo):
    """
    Implementação de busca linear.
    Retorna o índice do elemento encontrado ou -1 se não encontrar.

    Complexidade: O(n)
    """
    for i in range(len(lista)):
        if lista[i] == alvo:
            return i
    return -1

# Exemplo de uso
numeros = [5, 2, 8, 4, 9, 3, 1]
resultado = busca_linear(numeros, 9)
print(f"Busca Linear: O elemento foi encontrado no índice {resultado}")
```

2. Busca Binária (Binary Search)

```
python
```

```python
def busca_binaria(lista, alvo):
    """
    Implementação de busca binária (requer lista ordenada).
    Retorna o índice do elemento encontrado ou -1 se não encontrar.

    Complexidade: O(log n)
    """
    esquerda, direita = 0, len(lista) - 1

    while esquerda <= direita:
        meio = (esquerda + direita) // 2

        if lista[meio] == alvo:
            return meio
        elif lista[meio] < alvo:
            esquerda = meio + 1
        else:
            direita = meio - 1

    return -1

# Exemplo de uso
numeros_ordenados = [1, 2, 3, 4, 5, 7, 9, 10]
resultado = busca_binaria(numeros_ordenados, 7)
print(f"Busca Binária: O elemento foi encontrado no índice {resultado}")

# Implementação recursiva
def busca_binaria_recursiva(lista, alvo, esquerda=None, direita=None):
    """
    Implementação recursiva de busca binária.

    Complexidade: O(log n)
    """
    if esquerda is None and direita is None:
        esquerda, direita = 0, len(lista) - 1

    if esquerda > direita:
        return -1

    meio = (esquerda + direita) // 2

    if lista[meio] == alvo:
        return meio
```

```
    elif lista[meio] < alvo:
        return busca_binaria_recursiva(lista, alvo, meio + 1, direita)
    else:
        return busca_binaria_recursiva(lista, alvo, esquerda, meio - 1)
```

3. Busca em Largura (BFS - Breadth-First Search)

```
python
```

```python
from collections import deque

def bfs(grafo, inicio, alvo=None):
    """
    Implementação de Breadth-First Search (Busca em Largura).

    Parâmetros:
    grafo: dicionário onde as chaves são os nós e os valores são as listas de vizinhos
    inicio: nó de início
    alvo: nó alvo (opcional)

    Retorna:
    - Caminho do nó inicial até o alvo se o alvo for especificado e encontrado
    - Dicionário de distâncias e dicionário de predecessores se o alvo não for especifi

    Complexidade: O(V + E) onde V são vértices e E são arestas
    """
    fila = deque([inicio])
    visitados = {inicio}
    distancia = {inicio: 0}
    predecessores = {inicio: None}

    while fila:
        no_atual = fila.popleft()

        # Se encontramos o alvo, reconstruímos o caminho e retornamos
        if alvo and no_atual == alvo:
            caminho = []
            while no_atual is not None:
                caminho.append(no_atual)
                no_atual = predecessores[no_atual]
            return caminho[::-1]  # Inverter o caminho para ter início -> alvo

        # Explorar vizinhos
        for vizinho in grafo.get(no_atual, []):
            if vizinho not in visitados:
                fila.append(vizinho)
                visitados.add(vizinho)
                distancia[vizinho] = distancia[no_atual] + 1
                predecessores[vizinho] = no_atual

    # Se o alvo não foi encontrado ou não foi especificado
    if alvo:
```

```
        return None  # Alvo não encontrado
    else:
        return distancia, predecessores  # Retorna informações de todos os nós acessíve

# Exemplo de uso
grafo = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}

# Buscar caminho de A para F
caminho = bfs(grafo, 'A', 'F')
print(f"BFS: Caminho de A para F: {caminho}")

# Obter distâncias de todos os nós a partir de A
distancias, predecessores = bfs(grafo, 'A')
print(f"BFS: Distâncias a partir de A: {distancias}")
```

4. Busca em Profundidade (DFS - Depth-First Search)

```
python
```

```python
def dfs(grafo, inicio, alvo=None):
    """
    Implementação iterativa de Depth-First Search (Busca em Profundidade).

    Parâmetros:
    grafo: dicionário onde as chaves são os nós e os valores são as listas de vizinhos
    inicio: nó de início
    alvo: nó alvo (opcional)

    Retorna:
    - True se o alvo for encontrado, False caso contrário
    - Lista de nós visitados se o alvo não for especificado

    Complexidade: O(V + E) onde V são vértices e E são arestas
    """
    pilha = [inicio]
    visitados = set()

    while pilha:
        no_atual = pilha.pop()

        if no_atual not in visitados:
            visitados.add(no_atual)

            if alvo and no_atual == alvo:
                return True

            # Adicionar vizinhos não visitados à pilha
            for vizinho in grafo.get(no_atual, []):
                if vizinho not in visitados:
                    pilha.append(vizinho)

    if alvo:
        return False  # Alvo não encontrado
    else:
        return visitados  # Retorna todos os nós visitados

# Implementação recursiva
def dfs_recursivo(grafo, no_atual, visitados=None, alvo=None):
    """
    Implementação recursiva de Depth-First Search.

    Complexidade: O(V + E)
```

```
    """
    if visitados is None:
        visitados = set()

    visitados.add(no_atual)

    if alvo and no_atual == alvo:
        return True

    for vizinho in grafo.get(no_atual, []):
        if vizinho not in visitados:
            if dfs_recursivo(grafo, vizinho, visitados, alvo):
                return True

    if alvo:
        return False  # Alvo não encontrado
    else:
        return visitados  # Retorna todos os nós visitados

# Exemplo de uso
grafo = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}

# Versão iterativa
resultado = dfs(grafo, 'A', 'F')
print(f"DFS Iterativo: Caminho de A para F existe? {resultado}")

# Versão recursiva
visitados = dfs_recursivo(grafo, 'A')
print(f"DFS Recursivo: Nós visitados a partir de A: {visitados}")
```

5. Busca Gulosa (Greedy Search)

```
python
```

```python
def busca_gulosa(grafo, heuristica, inicio, alvo):
    """
    Implementação de busca gulosa (greedy search).

    Parâmetros:
    grafo: dicionário onde as chaves são os nós e os valores são as listas de vizinhos
    heuristica: dicionário com valores heurísticos para cada nó
    inicio: nó de início
    alvo: nó alvo

    Retorna:
    Lista representando o caminho do início até o alvo, ou None se não encontrar
    """
    # Conjunto para armazenar nós já avaliados
    visitados = set()

    # Dicionário para armazenar o caminho
    predecessores = {inicio: None}

    # Nó atual
    atual = inicio

    while atual != alvo:
        visitados.add(atual)

        # Obtém vizinhos não visitados
        vizinhos = [v for v in grafo.get(atual, []) if v not in visitados]

        if not vizinhos:
            return None  # Sem saída

        # Escolhe o vizinho com a menor heurística
        proximo = min(vizinhos, key=lambda n: heuristica[n])

        # Atualiza o predecessores
        predecessores[proximo] = atual

        # Move para o próximo nó
        atual = proximo

    # Reconstrói o caminho
    caminho = []
    while atual is not None:
```

```
            caminho.append(atual)
            atual = predecessores[atual]

    return caminho[::-1]  # Inverte o caminho para ter início -> alvo

# Exemplo de uso
grafo = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}

# Heurística (distância aproximada até o alvo 'F')
heuristica = {
    'A': 3,
    'B': 2,
    'C': 1,
    'D': 3,
    'E': 1,
    'F': 0
}

# Busca caminho de A para F
caminho = busca_gulosa(grafo, heuristica, 'A', 'F')
print(f"Busca Gulosa: Caminho de A para F: {caminho}")
```

Dicas para Maratonas de Programação

1. **Identifique o algoritmo adequado**:
   - Para listas pequenas: busca linear é simples e eficaz
   - Para listas ordenadas: busca binária é muito mais eficiente
   - Para problemas de grafos com caminhos mais curtos: BFS
   - Para explorar todos os caminhos possíveis: DFS
   - Para otimização com heurística: algoritmos gulosos
2. **Otimize seu código**:
   - Use funções da biblioteca padrão quando possível (`bisect`, `collections.deque`)
   - Evite recalcular valores usando memoização ou programação dinâmica

3. **Teste com casos extremos**:
    - Lista vazia
    - Lista com um único elemento
    - Elemento no início, meio e fim da lista
    - Elemento não presente na lista
4. **Atenção aos detalhes**:
    - Condições de parada
    - Índices (evite erros off-by-one)
    - Controle de visitados para evitar ciclos infinitos em grafos
    - Tratamento de casos especiais
5. **Biblioteca úteis do Python**:
    - `collections.deque`: Implementação eficiente de filas (para BFS)
    - `bisect`: Módulo para busca binária e inserção em listas ordenadas
    - `heapq`: Implementação de filas de prioridade (útil em algoritmos gulosos)