ALGORITMOS E ESTRUTURAS DE DADOS

RICARDO ROUCO
INÊS FERREIRA

Ricardo Rouco & Inês Ferreira

Algoritmos e Estruturas de dados

https://ricardodsr.github.io

"A recursão é a magia da programação: permite que se crie soluções simples para problemas complexos. “

- Edsger Dijkstra

Este e-book é dedicado a todas as pessoas curiosas, teimosas e perseverantes que terminam por alcançar os seus objetivos!

CONTEÚDO

INTRODUÇÃO

No reino da computação, os algoritmos reinam supremos. Eles são os arquitetos invisíveis por trás de cada software, cada aplicativo e cada website que utilizamos diariamente. Desde antes da era dos computadores, quando existiam apenas em mentes humanas brilhantes, até os dias de hoje, onde permeiam cada aspeto da nossa vida digital, os algoritmos moldam a forma como interagimos com o mundo da informação.

Este e-book é um convite para desvendar os mistérios por trás desses mestres da computação. Uma jornada no universo dos algoritmos, onde a simplicidade e a clareza se entrelaçam com a profundidade e o rigor matemático. Através de uma linguagem acessível e exemplos práticos, o autor nos guia por um leque de algoritmos, desde os mais básicos até os mais complexos, desvendando seus segredos e revelando sua beleza intrínseca.

Cada capítulo é uma porta de entrada para um novo universo algorítmico. Seja um algoritmo clássico, uma técnica de projeto inovadora, uma área de aplicação intrigante ou um tópico

relacionado, cada página nos oferece um vislumbre da vastidão e da riqueza do mundo algorítmico.

A busca pela eficiência é a estrela guia deste livro. Análises cuidadosas de cada algoritmo nos guiam na compreensão de sua performance, revelando seus pontos fortes e fracos, e nos preparando para tomar decisões informadas sobre sua aplicabilidade em diferentes cenários.

Se você busca desvendar os segredos dos algoritmos, seja como estudante em busca de conhecimento, profissional em busca de aprimoramento ou simplesmente um curioso fascinado pela tecnologia, este e-book é o seu guia. Através de uma jornada empolgante e enriquecedora, você se tornará um conhecedor dos algoritmos, apto a compreender, projetar e implementar soluções computacionais eficientes e inovadoras.

Prepare-se para desvendar os mistérios dos algoritmos e mergulhar na fascinante aventura da computação!

CAPÍTULO 1 - ALGORITMOS

"Informalmente, um algoritmo é qualquer procedimento computacional bem definido que toma algum valor ou conjunto de valores como entrada e produz algum valor ou conjunto de valores como saída. Portanto, um algoritmo é uma sequência de etapas computacionais que transformam a entrada na saída."

O que é um Algoritmo?

Em resumo, um algoritmo é como uma receita para solucionar problemas em um computador. Ele define os passos precisos a serem seguidos para transformar dados de entrada em dados de saída.

Características principais:

- Sequência de instruções: Um conjunto de instruções passo a passo para resolver um problema específico.
- Entrada e saída: Recebe dados e gera resultados.
- Solução de problemas: Uma ferramenta para resolver problemas computacionais específicos.

Características importantes de um algoritmo:

- Precisão: Deve funcionar corretamente para qualquer entrada válida.
- Clareza: Deve ser fácil de entender e seguir.
- Eficiência: Deve usar recursos computacionais de forma otimizada.
- Generalidade: Deve ser aplicável a diferentes tipos de problemas.

Exemplo: Algoritmo de ordenação:

- Problema: Ordenar uma sequência de números.
- Entrada: Sequência de números (ex: 31, 41, 59, 26, 41, 58).
- Saída: Sequência ordenada (ex: 26, 31, 41, 41, 58, 59).
- Importância: Operação fundamental em ciência da computação.

- Vários algoritmos disponíveis: Cada um com vantagens e desvantagens.
- Fatores a considerar na escolha do algoritmo: Número de itens, ordenação inicial, restrições de valores, arquitetura do computador, tipo de armazenamento.

Outras definições importantes:

- Instância: Conjunto de dados que serve como entrada para o algoritmo.
- Solução: Resultado correto do algoritmo para uma instância específica.
- Algoritmo incorreto: Pode não terminar ou fornecer resultado errado para algumas instâncias.
- Taxa de erro controlável: Algoritmo incorreto com probabilidade de erro conhecida.
- Especificação do algoritmo: Pode ser em linguagem natural, código de programação ou projeto de hardware.

Importância:

Os algoritmos são fundamentais na ciência da computação, pois permitem automatizar tarefas complexas de forma eficiente e precisa. Existem diversos tipos de algoritmos, cada um com suas vantagens e desvantagens, e a escolha do algoritmo ideal depende do problema a ser resolvido e dos recursos disponíveis.

Conclusão:

Algoritmos são essenciais para resolver problemas computacionais de forma eficiente e precisa. A escolha do algoritmo ideal depende das características do problema e dos recursos disponíveis.

CAPÍTULO 2 - ALGORITMOS COMO TECNOLOGIA

Imagine que os computadores fossem instantâneos e a memória infinita. Você ainda precisaria estudar algoritmos? A resposta é sim! Mesmo nesse cenário, os algoritmos ainda teriam grande utilidade.

Por quê?

- Garantir a solução correta: Mesmo com um computador rápido, é crucial ter certeza de que o método escolhido para resolver um problema realmente funciona e fornece o resultado correto.
- Boa prática de software: Mesmo que a velocidade não seja um problema, escrever um código bem documentado, organizado e fácil de entender é fundamental para qualquer programador.

- Recursos limitados na vida real: Na realidade, computadores são rápidos, mas não infinitos. A memória também tem um custo, mesmo que baixo. Por isso, é importante usar esses recursos com inteligência.

- Eficiência: Algoritmos eficientes otimizam o tempo de processamento e o uso da memória, o que se torna ainda mais importante quando os recursos são limitados.

- Em resumo, estudar algoritmos ajuda-te a escrever programas corretos, fáceis de entender e que utilizam os recursos do computador de forma eficiente, mesmo em um mundo hipotético de computadores superpoderosos.

Eficiência

Imagine dois algoritmos diferentes para resolver o mesmo problema. Qual seria a melhor escolha? A resposta depende da eficiência de cada algoritmo, que se refere à quantidade de recursos computacionais que ele utiliza para alcançar a solução.

Recursos computacionais:

- Tempo de execução: O tempo que o algoritmo leva para processar os dados e fornecer o resultado.
- Memória: A quantidade de memória necessária para armazenar os dados e as variáveis durante a execução do algoritmo.
- Outras medidas: Em alguns casos, outros recursos como consumo de energia ou uso de dispositivos de entrada/saída podem ser relevantes.

Comparando eficiências:

Para comparar a eficiência de dois algoritmos, podemos analisar seu comportamento em diferentes cenários:

- Tamanho da entrada: Como o tempo de execução e o uso de memória variam conforme a quantidade de dados a serem processados? Um algoritmo pode ser mais eficiente para pequenos conjuntos de dados, enquanto outro se torna mais vantajoso para conjuntos maiores.

- Características da entrada: O tipo de dados (números, *strings*, etc.) ou a organização dos dados (ordenados, aleatórios, etc.) podem influenciar o desempenho dos algoritmos.

- Hardware: A arquitetura do computador, a velocidade do processador e a quantidade de memória disponível podem afetar a eficiência de cada algoritmo.

Exemplo:

Imagine dois algoritmos de ordenação:

- Algoritmo A: Tempo de execução O(n^2), ou seja, cresce quadraticamente com o tamanho da entrada. Uso de memória constante.

- Algoritmo B: Tempo de execução O(n log n), ou seja, cresce logaritmicamente com o tamanho da entrada. Uso de memória proporcional ao tamanho da entrada.

Para pequenas entradas, o algoritmo A pode ser mais rápido. No entanto, para conjuntos de dados maiores, o algoritmo B se

torna mais eficiente, pois seu tempo de execução cresce mais lentamente.

Escolhendo o algoritmo ideal:

A escolha do algoritmo ideal depende do problema específico, dos recursos disponíveis e dos requisitos da aplicação. É importante analisar os diferentes algoritmos, considerar os cenários de uso e escolher aquele que oferece o melhor equilíbrio entre eficiência e outros fatores relevantes.

Lembre-se:

- A eficiência é apenas um dos fatores a serem considerados ao escolher um algoritmo. Outros fatores como facilidade de implementação, confiabilidade e legibilidade do código também podem ser importantes.
- Em alguns casos, um algoritmo mais simples e menos eficiente pode ser a melhor escolha, especialmente se o problema for pequeno ou se a performance não for crítica.

- É importante ter conhecimento sobre diferentes tipos de algoritmos e suas características para poder fazer uma escolha consciente.

CAPÍTULO 3 - ESTRUTURAS DE DADOS EFICIENTES

Buffers

Queues

- Queue e um tipo de dados abstrato baseado no principio de *First In First Out* (FIFO), ou seja, o primeiro elemento a sair e o primeiro a ser acrescentado.
- Operação "*enqueue*" → inserção de dados no conjunto.
- Operação "*dequeue*" → remoção de dados do conjunto.
- Operação "*peek*" ou "*front*" → devolve o valor da frente da *queue*, mas sem o remover.

Stacks

• *Stack* e um tipo de dados abstrato baseado no princípio de *Last In First Out* (LIFO), ou seja, o primeiro elemento a sair e o último a ser acrescentado.

- Operação "*push*" → inserção de dados no conjunto.
- Operação "*pop*" → remoção de dados do conjunto.
- Operação "*top*" ou "*peek*" → devolve o valor no topo da *stack*, mas sem o remover.

Arvores Binárias

Arvore Binaria de Pesquisa é uma estrutura de dados de arvore binaria baseada em nós, onde todos os nós da sub-arvore esquerda possuem um valor numérico inferior ao nó raiz e todos os nos da subárvore direita possuem um valor superior ao nó raiz. As sub-arvores esquerda e direita devem também ser arvores binarias de procura e não devem existir nodos duplicados.

Numa arvore completa, todos os nos (excepto possivelmente um) são equilibrados: as alturas das suas duas sub-arvores são iguais.

O tempo de execução:

- e $\Theta(\log n)$ no caso da pesquisa binaria num vetor ordenado;
- numa BST em geral a operação de pesquisa executa em $\Omega(1)$ e em O(n);
- numa BST cuja altura seja logarítmica no número de nodos da arvore, a operação de
- pesquisa executa em $\Theta(\log n)$;

Heaps

Max/Min Heap

- Arvore binaria em que a raiz e maior/menor que os filhos.
- Inserção e na primeira posição livre.

- 2*p+1 (1º filho)
- 2*p+2 (2º filho)
- (p-1)/2 (pai)
- p → posição onde esta
- Remoção só na raiz, pega-se no último elemento do vetor, poe-se no topo e reordena-se.

AVL

- AVL → arvore binaria de procura balanceada; elementos a esquerda menores que a raiz; elementos a direita maiores que a raiz.
- Nas arvores AVL cada nodo pode estar equilibrado, mas também desequilibrado para a esquerda ou para a direita, desde que a diferença entre as alturas não exceda 1,
- Este relaxamento permite que a altura da arvore permaneça logarítmica, pelo que o mesmo acontece com o tempo de pesquisa.

- O tempo de execução dos algoritmos de inserção e remoção, modificados por forma a efetuarem o necessário ajuste das arvores preservando sempre o invariante.
- As operações adicionais necessárias para a "autorregulação" do equilíbrio dos nos (gestão de informação auxiliar e rotações) executam em tempo $\Theta(1)$.
- Tendo as arvores sempre altura logarítmica, as consultas são feitas em tempo $\Theta(\log n)$.
- As operações de inserção e remoção, que numa BST de altura logarítmica executam em tempo também logarítmico, não são perturbadas pelas operações adicionais, mantendo-se em $\Theta(\log n)$.

Tabelas de *Hash*

- Generalização dos arrays para grande universo de chaves.

- Tendencialmente as operações executam em tempo constante $\Theta(1)$.
- Factor de carga ("*load*"); aconselhável limite < 0.8
- α = no de chaves inseridas / capacidade
- Se α > k, fazer: duplicar a capacidade ("*rehash*"). Θ (1) em termos amortizados.
- Condicionamento + "*hashing*".
- h(x) = x % N, com N primo (exemplo de função de *hash* aceitável)
- Uma função de *hash* mapeia chaves para o domínio de um array concreto. Deve ser não injetiva. Se as chaves geradas não forem números naturais, devem ser previamente condicionadas, i.e. transformadas em números naturais, novamente de forma determinista é tao uniforme quanto possível.
- Altura da arvore: log2 N

Colisões

• Acontecem quando a função de *hash* calcula a mesma posição do array para duas chaves diferentes, principalmente quando a capacidade da tabela e muito menor que a cardinalidade do conjunto de chaves.

Soluções:
Open Addressing

- Inserir a chave noutra posição do array, usando um método que possa ser reproduzido ao efetuar consultas.
- Método mais natural linear *probing*: resolve-se a colisão inserindo na próxima posição livre do array.
- E essencial armazenar as chaves juntamente com os valores, uma vez que na operação de consulta não basta procurar a posição respetiva a chave, e necessário repetir o *linear probing* até esta ser encontrada.

- As posições do array devem ser inicializadas como "*empty*" e marcadas como "*deleted*" quando e removido um elemento da tabela.
- Problema: formação de clusters aumento da probabilidade de inserção em posições subsequentes as já preenchidas. Pode ser evitado recorrendo a *quadratic probing*.
- A tabela deve ser redimensionada quando necessário, assegurando um facto de carga pequeno.
- Vantagens: se devidamente optimizador (ex. *quadratic probing*, redimensionamento dinâmico) e boa escolha porque não é penalizador em termos de espaço e é totalmente estático (vantagens em termos de *caching*).
- Desvantagens: dificuldade em lidar com remoções.

Closed Addressing (Chaining)

- Alterar a estrutura de dados de forma a permitir mais que um par chave → valor na mesma posição do array.

Através da criação de uma lista ligada cujo endereço inicial e guardado no array.

- A consulta envolvera uma pesquisa neste conjunto de pares.
- As listas ligadas em teoria podem ser α > 1, mas não devem crescer indefinidamente, pois o tempo de execução deixa de ser tendencialmente constante.
- A tabela deve ser redimensionada quando necessário, assegurando um factor de carga pequeno.
- Vantagens: de programação simples.
- Desvantagens: custo adicional de espaço relevante.

CAPÍTULO 4 - BFS OU BUSCA EM LARGURA

Breadth-First Search, ou Busca em Largura, é um algoritmo utilizado para percorrer uma estrutura de dados, como uma árvore ou um grafo, de forma sistematizada. Ele explora todos os vértices vizinhos (recorrentemente, designados por adjacentes) de um vértice localizado na posição atual antes de avançar para os vértices mais distantes.

Aspectos-chave:

- Utilização de uma fila: Uma fila é a estrutura de dados principal do BFS. Nela, os nós a serem explorados são armazenados de forma ordenada, seguindo a regra *First* In, *First* Out (FIFO).

- Exploração por nível: O algoritmo explora os nós nível a nível, começando pelo nó raiz (ou nó inicial) e visitando todos os seus vizinhos adjacentes antes de passar para o próximo nível.

- Marcação de nós: Ao visitar um nó, ele é marcado como visitado para evitar ciclos redundantes e garantir que cada nó seja visitado apenas uma vez.
- Resultado retornado: O BFS pode retornar diferentes tipos de informações, como o caminho percorrido, a distância entre os nós ou a estrutura completa do grafo explorado.

Exemplo de código (Python):

```python
from collections import deque
Codeium: Refactor | Explain | Generate Docstring | ×
def bfs_iterative(graph, start):
        visited = set()
        queue = deque([start])

        while queue:
            vertex = queue.popleft()
            print(vertex)

            if vertex not in visited:
                visited.add(vertex)
                neighbors = graph[vertex]
                queue.extend(neighbors)
```

Começamos por definir um conjunto *visited* para armazenar os vértices visitados e uma fila *queue* para armazenar os vértices a serem explorados. Inicialmente, colocamos o vértice de partida na fila. Em cada iteração do *loop*, removemos o primeiro vértice da fila utilizando o método *popleft()* e realizamos as ações desejadas com esse vértice. Em seguida, verificamos se o vértice já foi visitado. Se não tiver sido visitado, adicionamos o vértice ao conjunto *vis*ited e adicionamos os vizinhos desse vértice na fila utilizando o método *extend().* Esse processo repete-se até que a fila esteja vazia, o que indica que todos os vértices foram explorados.

Vantagens do BFS:

- Fácil de implementar: O BFS possui uma estrutura simples e intuitiva, tornando-o fácil de implementar em diferentes linguagens de programação.

- Eficiente para grafos densos: Em grafos densos, onde cada nó possui um grande número de adjacentes, o BFS geralmente apresenta bom desempenho.

- Encontra o caminho mais curto: Para grafos não direcionados e sem pesos nas arestas, o BFS encontra o caminho mais curto entre dois nós do grafo.

Desvantagens do BFS:

- Consumo de memória: O BFS pode consumir mais memória do que outros algoritmos de busca, como o *Depth-First Search* (DFS), devido à utilização da fila.
- Ineficiente para grafos esparsos: Em grafos esparsos, onde cada nó possui um número pequeno de adjacentes, o BFS pode ser menos eficiente do que o DFS.

Aplicações do BFS:

- Roteamento em redes: O BFS é utilizado em algoritmos de roteamento para encontrar o caminho mais curto entre dois dispositivos numa rede.
- Busca em jogos: Em jogos de tabuleiro e videojogos, o BFS pode ser usado para encontrar o caminho mais curto entre dois pontos no mapa do jogo.

- Análise de redes sociais: O BFS pode ser aplicado na análise de redes sociais para identificar comunidades e conexões entre indivíduos.
- Inteligência artificial: O BFS é utilizado em diversas aplicações de inteligência artificial, como robótica e processamento de linguagem natural.

Conclusão:

O *Breadth-First Search* é um algoritmo versátil e eficiente para explorar grafos e árvores. A simplicidade de implementação e a capacidade de encontrar o caminho mais curto tornam-no uma ferramenta valiosa para diversas aplicações em diferentes áreas do conhecimento.

CAPÍTULO 5 –BUSCA BINÁRIA

A Busca Binária é um algoritmo fundamental na ciência da computação, conhecido por sua eficiência na busca por um elemento específico em um conjunto ordenado de dados. Diferentemente da busca sequencial, que verifica cada elemento do conjunto na ordem em que estão armazenados, a Busca Binária utiliza uma estratégia inteligente para reduzir drasticamente o tempo de busca.

Funcionamento:

Imagine um conjunto de livros em uma estante ordenados alfabeticamente. A Busca Binária funciona como se você estivesse procurando por um livro específico:

- Comece no meio: Comece no meio da estante, dividindo o conjunto de livros em duas metades.
- Compare o elemento: Compare o livro do meio com o que o livro que procura. Se for o livro que procura, a busca termina!

- Elimine metade: Se o livro do meio não for o que procura, elimine metade da estante, dependendo se o livro que procura é alfabeticamente anterior ou posterior ao livro do meio.

- Repita em uma metade: Repita o processo de dividir e comparar na metade restante da estante até encontrar o livro que procura ou concluir que ele não está presente.

Vantagens da Busca Binária:

- Eficiência: A Busca Binária reduz drasticamente o tempo de busca em comparação com a busca sequencial, especialmente em conjuntos grandes. Em média, o algoritmo realiza apenas log2(n) comparações, onde n é o número de elementos no conjunto.

- Simplicidade: O algoritmo é relativamente simples de implementar e entender.

- Versatilidade: A Busca Binária pode ser aplicada em diversos tipos de dados ordenados, como arrays, listas e árvores.

Exemplo de código (Python):

```python
def binary_search_iterative(arr, target):
    low = 0
    high = len(arr) - 1

    while low <= high:
        mid = (low + high) // 2

        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1

    return -1
```

Neste método, começamos por definir um intervalo de busca, representado pelas variáveis *low* (limite inferior) e *high* (limite superior). Em cada iteração, calculamos o índice médio (*mid*) do intervalo atual. Comparamos o valor no índice médio com o valor alvo. Se eles são iguais, retornamos o índice médio. Caso contrário, ajustamos os limites *low e high* dependendo se o valor alvo é maior ou menor que o valor no índice médio. Repetimos esse processo até encontrar o elemento desejado ou determinar que ele não está presente.

Aplicações:

- Bancos de dados: A Busca Binária é utilizada em bancos de dados para encontrar registos específicos de forma eficiente.
- Compiladores: Compiladores utilizam a Busca Binária para localizar identificadores, palavras-chave e outras informações no código-fonte.
- Sistemas embarcados: A Busca Binária é frequentemente utilizada em sistemas embarcados com recursos limitados, devido à sua eficiência e baixo consumo de memória.

CAPÍTULO 6 - DEPTH-FIRST SEARCH, OU BUSCA EM PROFUNDIDADE.

A Busca em Profundidade (DFS), também conhecida como *Depth-First Search*, é um algoritmo de busca utilizado para explorar e percorrer grafos e árvores. Diferentemente da Busca em Largura (BFS), que explora os nós nível por nível, a DFS segue um caminho mais profundo, explorando o máximo possível em cada ramificação antes de voltar para trás.

Funcionamento:

Imagine um labirinto e você está procurando a saída. A DFS funcionaria da seguinte maneira:

- Comece em um ponto: Comece em um ponto inicial no labirinto (geralmente o nó raiz no grafo).
- Explore o caminho atual: Siga o caminho atual o mais profundamente possível, marcando cada nó que visita para evitar ciclos.

- Chegou a um beco sem saída? Se chegar a um beco sem saída (um nó sem arestas não exploradas), volte para o último nó não explorado e siga por um caminho diferente.

- Repita até encontrar a saída: Repita o processo de explorar caminhos e voltar até encontrar a saída do labirinto (ou determinar que não existe saída).

Vantagens da DFS:

- Exploração completa: A DFS garante que todos os nós do grafo serão explorados, desde que não haja ciclos infinitos.

- Memória eficiente: A DFS geralmente consome menos memória do que a BFS, pois a pilha utilizada para armazenar os caminhos é menor que a fila utilizada na BFS.

- Deteção de ciclos: A DFS é eficaz na deteção de ciclos em grafos, pois ao marcar os nós visitados, evita revisitá-los e cair em *loops* infinitos.

Exemplo de código(Python):

```python
def dfs_iterative(graph, start):
    visited = set()
    stack = [start]

    while stack:
        vertex = stack.pop()
        print(vertex)

        if vertex not in visited:
            visited.add(vertex)
            neighbors = graph[vertex]
            stack.extend(neighbors)
```

Começamos por definir um conjunto *visited* para armazenar os vértices visitados e uma pilha *stack* para armazenar os vértices a serem explorados. Inicialmente, colocamos o vértice de partida na pilha. Em cada iteração do *loop,* removemos o último vértice da pilha utilizando o *método pop ()* e realizamos as ações desejadas com esse vértice. Em seguida, verificamos se o vértice já foi visitado. Se não tiver sido visitado, adicionamos o vértice ao conjunto *visited* e adicionamos os vizinhos desse vértice na pilha utilizando o método *extend().*

Esse processo é repetido até que a pilha esteja vazia, o que indica que todos os vértices foram explorados.

Aplicações:

- Análise de grafos: A DFS é utilizada para analisar a estrutura de grafos, identificar componentes conectados, encontrar ciclos e determinar a existência de caminhos entre nós.
- Inteligência artificial: A DFS é utilizada em diversas áreas da inteligência artificial, como jogos, robótica e processamento de linguagem natural.
- Redes de computadores: A DFS pode ser utilizada em protocolos de roteamento para encontrar o caminho mais curto entre dois dispositivos em uma rede.

Considerações:

- Ordem de exploração: A ordem em que os nós são explorados pela DFS depende da ordem em que as arestas são armazenadas no grafo.

- Ciclos infinitos: Se o grafo contiver ciclos infinitos, a DFS pode ficar presa em um *loop* infinito, explorando os mesmos nós repetidamente.
- Eficiência em grafos grandes: Em grafos muito grandes e densos, a DFS pode ser menos eficiente do que a Busca em Largura (BFS).

CAPÍTULO 7 – ALGORITMO DE DIJKSTRA

O Algoritmo de *Dijkstra*, concebido pelo cientista da computação holandês *Edsger Dijkstra*, é um algoritmo poderoso e amplamente utilizado para encontrar o caminho mínimo entre um vértice inicial (origem) e todos os outros vértices em um grafo dirigido ou não dirigido com pesos não negativos nas arestas.

Conceitos-chave do algoritmo de *Dijkstra:*

- Grafo com pesos: O algoritmo funciona em grafos onde cada aresta possui um peso associado, representando o custo de se percorrer aquela aresta. Os pesos devem ser não negativos (maiores ou iguais a zero).

- Busca por custo mínimo: O objetivo é encontrar o caminho com o menor custo total entre o vértice inicial e todos os outros vértices. O custo total é a soma dos pesos das arestas percorridas no caminho.

- Distâncias estimadas: Inicialmente, todas as distâncias estimadas são definidas como infinito, exceto a do vértice inicial, que é zero. O algoritmo atualiza essas estimativas durante a iteração.

- Conjunto de vértices finalizados: O algoritmo mantém um conjunto de vértices já processados (finalizados), onde a distância mínima a partir do vértice inicial é garantida.

Funcionamento:

Imagine uma rede de estradas com distâncias marcadas entre as cidades. O algoritmo de *Dijkstra* funciona como um planejador de rotas eficiente:

Processo:

- Inicialização: O algoritmo atribui um valor de distância inicial para todos os vértices do grafo. Geralmente, o valor inicial é infinito para todos os vértices, exceto o vértice de partida, para o qual a distância inicial é definida como zero.

- Seleção do vértice: O algoritmo seleciona o vértice de menor distância entre os vértices não visitados. Inicialmente, esse vértice será o vértice de partida.

- “Relaxamento” das arestas: O algoritmo atualiza as distâncias dos vértices vizinhos ao vértice selecionado, se for possível alcançá-los por meio de uma aresta com um peso menor do que a distância atualmente registada. Isso é conhecido como "relaxamento" das arestas.

- Marcação do vértice: Após o relaxamento das arestas, o vértice selecionado é marcado como visitado e não será considerado novamente.

- Repetição: Os passos 2 a 4 são repetidos até que todos os vértices sejam visitados ou até que o vértice de destino seja alcançado.

- Resultado: Após a conclusão do algoritmo, a menor distância para cada vértice é determinada. Além disso, o caminho mais curto de um vértice de partida a qualquer outro vértice pode ser reconstruído usando as informações armazenadas durante o processo.

O algoritmo de *Dijkstra* é frequentemente implementado utilizando uma estrutura de dados chamada "fila de prioridade" (*priority queue*), que permite recuperar eficientemente o vértice não visitado de menor distância a cada iteração.

Código Fonte Python:

```python
import heapq
Codeium: Refactor | Explain | Generate Docstring | X
def dijkstra(graph, start):
        distances = {vertex: float('inf') for vertex in graph}
        distances[start] = 0

        pq = [(0, start)]

        while pq:
            current_distance, current_vertex = heapq.heappop(pq)

            if current_distance > distances[current_vertex]:
                continue

            for neighbor, weight in graph[current_vertex].items():
                distance = current_distance + weight

                if distance < distances[neighbor]:
                    distances[neighbor] = distance
                    heapq.heappush(pq, (distance, neighbor))

        return distances
```

A função *dijkstra* recebe o grafo representado por um dicionário, onde as chaves são os vértices e os valores são outros dicionários contendo os vizinhos e os pesos das arestas. A função inicializa um dicionário *distances* com valores infinitos

para todos os vértices, exceto o vértice de partida, que é definido como zero. A fila de prioridade é usada para armazenar as distâncias e os vértices não visitados. O algoritmo itera pela fila de prioridade, atualizando as distâncias e visitando os vértices conforme necessário.

Vantagens:

- Caminho mínimo garantido: O algoritmo de *Dijkstra* encontra o caminho com o menor custo total entre a origem e todos os outros vértices, desde que os pesos das arestas sejam não negativos.
- Versatilidade: Funciona tanto em grafos dirigidos quanto não dirigidos.
- Eficiente para grafos esparsos: Em grafos esparsos, onde cada vértice possui um número pequeno de vizinhos, o algoritmo de *Dijkstra* apresenta bom desempenho.

Desvantagens:

- Pesos não negativos: O algoritmo não funciona corretamente com pesos negativos nas arestas. Para lidar com pesos negativos, é necessário utilizar algoritmos como o algoritmo de *Bellman-Ford*.

- Menos eficiente para todos os pares: Se o objetivo for encontrar o caminho mínimo entre todos os pares de vértices, o algoritmo de *Floyd-Warshall* pode ser mais eficiente para grafos com pesos não negativos.

Aplicações do Algoritmo de *Dijkstra:*

- Sistemas de GPS: O algoritmo de *Dijkstra* é utilizado em sistemas de GPS para encontrar a rota mais rápida entre dois pontos em um mapa.

- Roteamento de redes: Redes de computadores utilizam o algoritmo *de Dijkstra* para roteamento de pacotes, encontrando o caminho de menor custo para enviar dados entre dispositivos.

- Planejamento logístico: O algoritmo pode ser aplicado em planejamento logístico para encontrar rotas de entrega mais eficientes.

O algoritmo de *Dijkstra* é amplamente utilizado em problemas de caminho mais curto, como encontrar a rota mais rápida entre dois pontos em um mapa ou otimizar a rede de transporte. Sua complexidade de tempo é O((V + E) log V), onde V é o número de vértices e E é o número de arestas no grafo.

CAPÍTULO 8 - HEAP SORT

O *Heap Sort* é um algoritmo de classificação baseado em estruturas de dados chamadas "*heaps*", que são árvores binárias especiais. Ele utiliza a propriedade de um *heap*, em que o elemento pai de cada nó é sempre maior ou igual aos elementos filhos, para ordenar os elementos em uma lista.

Funcionamento:

Construção do *Heap:*

- O algoritmo inicia convertendo o array de entrada em um *heap* máximo. Um *heap* máximo é uma árvore binária completa onde cada nó pai é maior ou igual aos seus filhos.
- A construção do *heap* pode ser realizada de duas maneiras:

- *Heapify*: Percorre o array de baixo para cima, ajustando os elementos para garantir a propriedade de heap máximo.
- *Build-Heap*: Cria o *heap* iterativamente, inserindo os elementos um a um e ajustando-os para manter a propriedade de *heap* máximo.

Ordenação passo a passo:

- Após a construção do *heap*, o algoritmo realiza a ordenação passo a passo:
  - Extração do Maior Elemento:
    - O maior elemento do *heap,* que é a raiz (*heap* máximo), é extraído e colocado na sua posição final no array ordenado.
    - A raiz do *heap* é substituída pelo último elemento do *heap,* que ainda não está ordenado.

        - O *heap* é ajustado para manter a propriedade de *heap* máximo após a remoção da raiz.

    - Repetição:

        - O processo de extração do maior elemento e ajuste do *heap* é repetido até que todos os elementos do array estejam ordenados.

Código Fonte Python:

```
Codeium: Refactor | Explain | Generate Docstring | X
def heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and arr[left] > arr[largest]:
        largest = left

    if right < n and arr[right] > arr[largest]:
        largest = right

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

Codeium: Refactor | Explain | Generate Docstring | X
def heap_sort(arr):
    n = len(arr)

    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
```

A função *heapify* é responsável por garantir a propriedade do *heap* para um nó específico. Ela recebe a lista *arr*, o tamanho do *heap* n e o índice do nó i. A função compara o nó i com seus filhos esquerdo (*left*) e direito (*right*), e se necessário, troca o nó

com o filho de maior valor. Esse processo é realizado recursivamente para garantir que todo o *heap* esteja em ordem.

A função *heap_sort* realiza o processo de construção do *heap* inicial chamando a função *heapify* para todos os nós não folha. Em seguida, o algoritmo realiza a extração do máximo, trocando o primeiro elemento (máximo) com o último elemento da lista e ajustando o *heap* com a função *heapify*. Isso é repetido até que todos os elementos estejam ordenados.

O *Heap Sort* é um algoritmo de classificação eficiente com complexidade de tempo O(n log n) em todos os casos. Ele é adequado para classificar grandes conjuntos de dados e também possui a vantagem de ser i*n-place*, ou seja, não requer espaço adicional para armazenar os elementos durante o processo de classificação.

Vantagens do *Heap Sort*:

- Eficiência: O *Heap Sort* possui uma complexidade temporal média e pior caso de O (n log n), o que o torna um algoritmo de ordenação eficiente para conjuntos de dados grandes.

- *In-place:* O algoritmo realiza a ordenação no próprio array de entrada, sem a necessidade de memória auxiliar, o que o torna mais eficiente em termos de uso de memória.

- Estabilidade: O *Heap Sort* preserva a ordem original dos elementos iguais, o que o torna um algoritmo estável.

Desvantagens do *Heap Sort*:

- Estrutura de Dados: O algoritmo depende da estrutura de dados *heap*, que pode ser menos intuitiva para alguns programadores.

- Espaço: O *Heap Sort* requer espaço adicional para armazenar a estrutura de dados *heap*, o que pode ser um problema para conjuntos de dados muito grandes em sistemas com memória limitada.

CAPÍTULO 9 - HUFFMAN COMPRESSION

A compressão de *Huffman* é um algoritmo utilizado para compactar dados, reduzindo o tamanho de arquivos sem perda de informação. Ele é baseado na construção de códigos de comprimento variável, chamados de códigos de *Huffman*, que atribuem sequências de bits mais curtas para os caracteres ou símbolos mais frequentes e sequências de bits mais longas para os menos frequentes.

Conceitos-chave:

- Frequência de ocorrência: O algoritmo analisa os dados de entrada (por exemplo, um texto) e calcula a frequência de ocorrência de cada caractere ou símbolo presente. Isso pode ser feito por meio de uma contagem dos caracteres ou usando uma tabela de frequências pré-calculada.
- Construção da Árvore de *Huffman:* Com base nas frequências de ocorrência, o algoritmo constrói uma

árvore binária especial chamada Árvore de *Huffman.* Essa árvore é construída de baixo para cima, começando pelos caracteres menos frequentes e terminando nos caracteres mais frequentes. Os nós internos da árvore representam a soma das frequências dos caracteres filhos.

- Atribuição de códigos: A partir da Árvore de *Huffman,* atribui-se um código binário único para cada caracter ou símbolo. O código é obtido seguindo o caminho da raiz até o nó correspondente ao caracter desejado. Atribui-se 0 para o caminho à esquerda e 1 para o caminho à direita.

- Compressão: Os dados de entrada são substituídos pelos códigos de *Huffman* correspondentes a cada caractere. Essa substituição reduz o tamanho do arquivo original, já que os códigos de *Huffman* são otimizados para que os caracteres mais frequentes sejam representados por sequências de bits mais curtas.

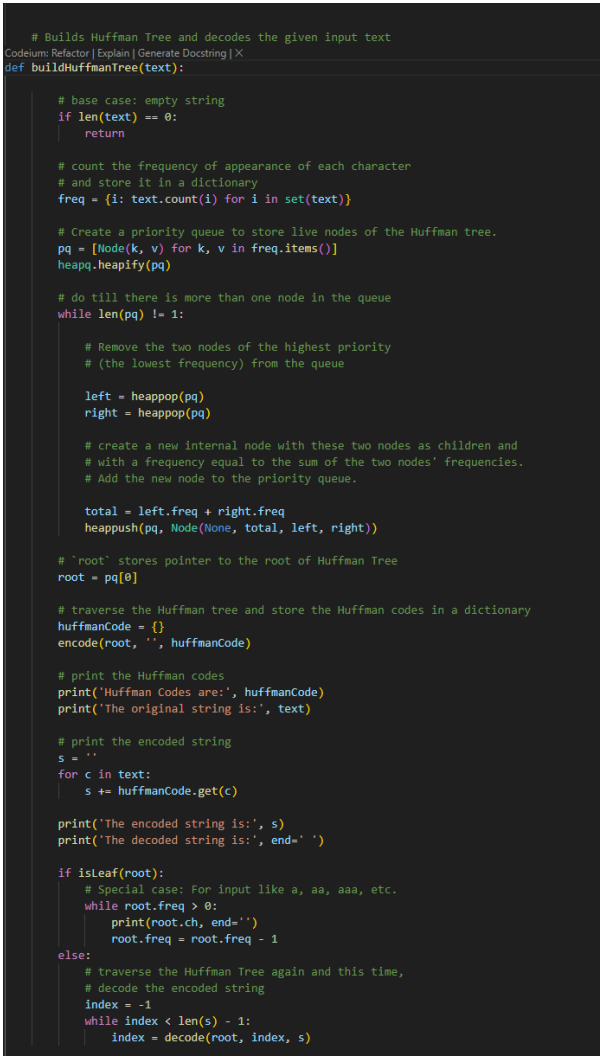
- Descompressão: Para descompactar os dados, é necessário ter acesso à Árvore *de Huffman* original. Os códigos de *Huffman* são lidos sequencialmente do arquivo comprimido e a cada código lido, a árvore é percorrida a partir da raiz até a folha correspondente. O caracter ou símbolo correspondente ao código é recuperado e adicionado ao arquivo descomprimido.

Código fonte em python:

Parte I

```python
import heapq
from heapq import heappop, heappush


Codeium: Refactor | Explain | Generate Docstring | ×
def isLeaf(root):
        return root.left is None and root.right is None


    # A Tree node
Codeium: Refactor | Explain
class Node:
        Codeium: Refactor | Explain | Generate Docstring | ×
        def __init__(self, ch, freq, left=None, right=None):
            self.ch = ch
            self.freq = freq
            self.left = left
            self.right = right

        # Override the `__lt__()` function to make `Node` class work with priority queue
        # such that the highest priority item has the lowest frequency
        Codeium: Refactor | Explain | Generate Docstring | ×
        def __lt__(self, other):
            return self.freq < other.freq


    # Traverse the Huffman Tree and store Huffman Codes in a dictionary
Codeium: Refactor | Explain | Generate Docstring | ×
def encode(root, s, huffman_code):

        if root is None:
            return

        # found a leaf node
        if isLeaf(root):
            huffman_code[root.ch] = s if len(s) > 0 else '1'

        encode(root.left, s + '0', huffman_code)
        encode(root.right, s + '1', huffman_code)


    # Traverse the Huffman Tree and decode the encoded string
Codeium: Refactor | Explain | Generate Docstring | ×
def decode(root, index, s):

        if root is None:
            return index

        # found a leaf node
        if isLeaf(root):
            print(root.ch, end='')
            return index

        index = index + 1
        root = root.left if s[index] == '0' else root.right
        return decode(root, index, s)
```

Parte II



```
    # Builds Huffman Tree and decodes the given input text
Codeium: Refactor | Explain | Generate Docstring | X
def buildHuffmanTree(text):

        # base case: empty string
        if len(text) == 0:
            return

        # count the frequency of appearance of each character
        # and store it in a dictionary
        freq = {i: text.count(i) for i in set(text)}

        # Create a priority queue to store live nodes of the Huffman tree.
        pq = [Node(k, v) for k, v in freq.items()]
        heapq.heapify(pq)

        # do till there is more than one node in the queue
        while len(pq) != 1:

            # Remove the two nodes of the highest priority
            # (the lowest frequency) from the queue

            left = heappop(pq)
            right = heappop(pq)

            # create a new internal node with these two nodes as children and
            # with a frequency equal to the sum of the two nodes' frequencies.
            # Add the new node to the priority queue.

            total = left.freq + right.freq
            heappush(pq, Node(None, total, left, right))

        # `root` stores pointer to the root of Huffman Tree
        root = pq[0]

        # traverse the Huffman tree and store the Huffman codes in a dictionary
        huffmanCode = {}
        encode(root, '', huffmanCode)

        # print the Huffman codes
        print('Huffman Codes are:', huffmanCode)
        print('The original string is:', text)

        # print the encoded string
        s = ''
        for c in text:
            s += huffmanCode.get(c)

        print('The encoded string is:', s)
        print('The decoded string is:', end=' ')

        if isLeaf(root):
            # Special case: For input like a, aa, aaa, etc.
            while root.freq > 0:
                print(root.ch, end='')
                root.freq = root.freq - 1
        else:
            # traverse the Huffman Tree again and this time,
            # decode the encoded string
            index = -1
            while index < len(s) - 1:
                index = decode(root, index, s)
```

Parte III

```
# Huffman coding algorithm implementation in Python
if __name__ == '__main__':

    text = 'Huffman coding is a data compression algorithm.'
    buildHuffmanTree(text)
```

Vantagens da Compressão de *Huffman*:

- Eficiente para dados com distribuição de frequência desigual: A Compressão de *Huffman* é particularmente eficiente para dados onde alguns caracteres ocorrem com mais frequência do que outros.

- Sem perda de dados: A compressão de *Huffman* é um algoritmo sem perdas, o que significa que o texto descomprimido será exatamente igual ao texto original.

Desvantagens da Compressão de *Huffman*:

- Tabela de códigos: A compressão requer a transmissão ou armazenamento da tabela de códigos junto com os dados comprimidos, o que pode adicionar *overhead* em alguns casos.

- Menos eficiente para dados aleatórios: A Compressão de *Huffman* não é tão eficiente para dados aleatórios onde todos os caracteres têm frequências semelhantes.

Aplicações da Compressão de *Huffman*:

- Arquivos de texto: A Compressão de *Huffman* é usada para compactar arquivos de texto, como documentos e código-fonte.
- Imagens: Formatos de imagem como JPEG usam técnicas semelhantes à Compressão de *Huffman* para reduzir o tamanho do arquivo.
- HTTP: A Compressão *de Huffman* pode ser utilizada para compactar dados transmitidos pela web, reduzindo o tempo de carregamento das páginas.

Conclusão:

A Compressão de *Huffman* é uma técnica valiosa para reduzir o tamanho de dados digitais sem perda de informação. Sua eficiência em dados com distribuição de frequência desigual a torna uma ferramenta importante em diversas

aplicações, como compactação de arquivos e transmissão de dados.

CAPÍTULO 10 - INSERTION SORT

Insertion Sort, ou Ordenação por Inserção, é um algoritmo de ordenação simples e eficiente para pequenos e médios conjuntos de dados. Ele funciona de forma similar a como organizamos cartas em nossas mãos em um jogo: comparando elementos adjacentes e inserindo-os na posição correta

Processo

- Divisão em sublistas: O algoritmo divide a lista em duas partes: uma sublista já ordenada e uma sublista não ordenada. Inicialmente, a sublista ordenada contém apenas o primeiro elemento da lista, enquanto a sublista não ordenada contém o restante dos elementos.

- Seleção do elemento: O algoritmo seleciona o próximo elemento da sublista não ordenada.

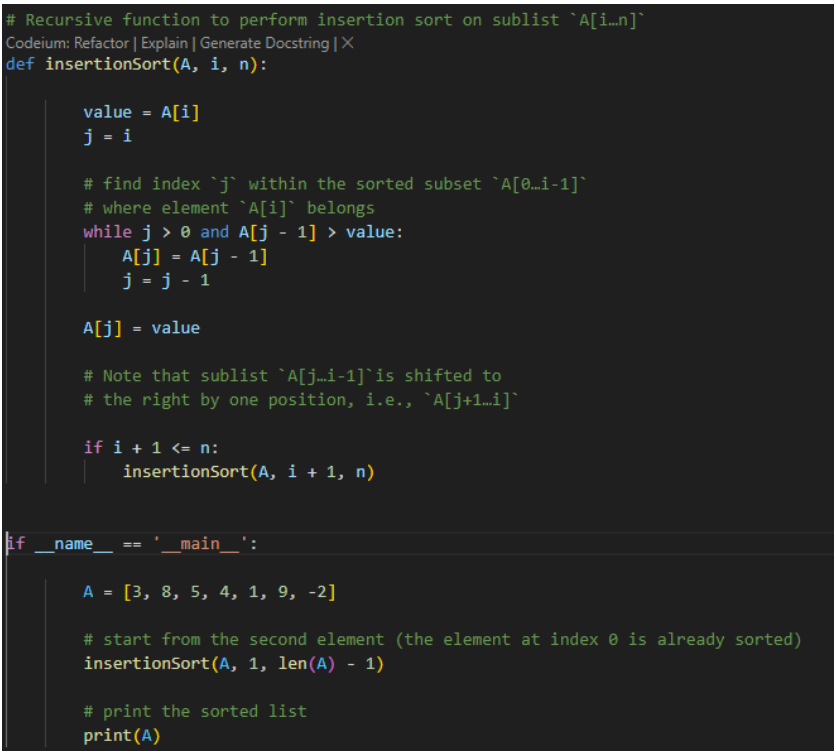
- Inserção na posição correta: O elemento selecionado é inserido na posição correta na sublista ordenada, deslocando os elementos maiores para a direita. Para encontrar a posição correta, o algoritmo compara o elemento selecionado com os elementos da sublista ordenada, começando do último elemento e movendo-se para a esquerda até encontrar uma posição em que o elemento selecionado seja maior.

- Repetição: Os passos 2 e 3 são repetidos até que todos os elementos da sublista não ordenada sejam inseridos na sublista ordenada. Ao final do processo, a lista estará completamente ordenada.

Código fonte Python

```python
# Function to perform insertion sort on a list
Codeium: Refactor | Explain | Generate Docstring | X
def insertionSort(A):

    # Start from the second element
    # (the element at index 0 is already sorted)
    for i in range(1, len(A)):

        value = A[i]
        j = i

        # find index `j` within the sorted subset `A[0…i-1]`
        # where element `A[i]` belongs
        while j > 0 and A[j - 1] > value:
            A[j] = A[j - 1]
            j = j - 1

        # Note that sublist `A[j…i-1]`is shifted to
        # the right by one position, i.e., `A[j+1…i]`

        A[j] = value


if __name__ == '__main__':

    A = [3, 8, 5, 4, 1, 9, -2]

    insertionSort(A)

    # print the sorted list
    print(A)
```

Código fonte Python (Recursivo)



```python
# Recursive function to perform insertion sort on sublist `A[i…n]`
Codeium: Refactor | Explain | Generate Docstring | ×
def insertionSort(A, i, n):

    value = A[i]
    j = i

    # find index `j` within the sorted subset `A[0…i-1]`
    # where element `A[i]` belongs
    while j > 0 and A[j - 1] > value:
        A[j] = A[j - 1]
        j = j - 1

    A[j] = value

    # Note that sublist `A[j…i-1]`is shifted to
    # the right by one position, i.e., `A[j+1…i]`

    if i + 1 <= n:
        insertionSort(A, i + 1, n)


if __name__ == '__main__':

    A = [3, 8, 5, 4, 1, 9, -2]

    # start from the second element (the element at index 0 is already sorted)
    insertionSort(A, 1, len(A) - 1)

    # print the sorted list
    print(A)
```

O *Insertion Sort* é eficiente para classificar pequenas listas ou quando a lista já está quase ordenada. No entanto, sua complexidade de tempo média é O(n^2), onde n é o número de elementos a serem classificados. Portanto, para listas grandes, outros algoritmos de classificação, como o *Merge Sort* ou o *Quick Sort*, geralmente são preferidos.

Vantagens do *Insertion Sort*:

- Simples de implementar: O *Insertion Sort* é um algoritmo fácil de entender e implementar, tornando-o ideal para iniciantes em programação.
- Eficiente para dados quase ordenados: *O Insertion Sort* apresenta bom desempenho em dados já parcialmente ordenados, pois o número de trocas de elementos é menor.
- Memória eficiente: *O Insertion Sort* requer pouca memória extra para realizar a ordenação, pois manipula o array original.

Desvantagens do *Insertion Sort:*

- Ineficiente para grandes conjuntos: O *Insertion Sort* torna-se ineficiente para grandes conjuntos de dados, pois o número de comparações e trocas de elementos cresce quadraticamente (n^2).

- Instável (opcional): Dependendo da implementação, o *Insertion Sort* pode alterar a ordem original de elementos iguais no array.

Conclusão:

Insertion Sort é um algoritmo de ordenação versátil, sendo uma boa escolha para iniciantes e para cenários onde a simplicidade e a eficiência em dados pequenos ou quase ordenados são prioridades. No entanto, para grandes conjuntos de dados, algoritmos como *Merge Sort* ou *Quick Sort* costumam ser mais performáticos.

.

CAPÍTULO 11 - MERGE SORT

O *Merge Sort* é um algoritmo de classificação eficiente e popular que segue a abordagem de dividir e conquistar. Ele divide a lista não ordenada em pequenas *sub-listas*, ordena essas *sub-listas* recursivamente e depois combina-as para obter a lista final ordenada.

Processo

- Divisão: O algoritmo divide a lista não ordenada ao meio até que haja apenas um elemento em cada *sub-lista*. Isso é feito recursivamente até que não seja mais possível dividir.

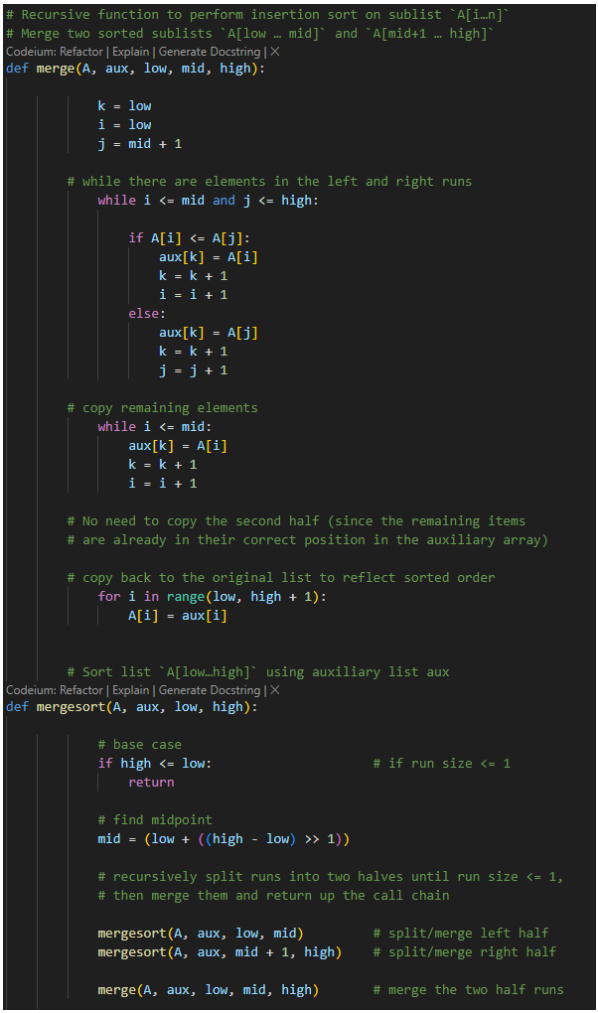
- Ordenação: Após a divisão, o algoritmo começa a ordenar as *sub-listas* combinando-as em uma lista ordenada. Para fazer isso, compara-se o primeiro elemento de cada *sub-lista* e os coloca em ordem crescente. Esse processo continua até que todas as

sub-listas sejam combinadas em uma única lista ordenada.

- Combinação: Nesta etapa, as *sub-listas* ordenadas são combinadas repetidamente para formar *sub-listas* maiores, até que toda a lista original seja recriada na ordem correta.

Código fonte Python:

Parte I



```python
# Recursive function to perform insertion sort on sublist `A[i…n]`
# Merge two sorted sublists `A[low … mid]` and `A[mid+1 … high]`
Codeium: Refactor | Explain | Generate Docstring | X
def merge(A, aux, low, mid, high):

        k = low
        i = low
        j = mid + 1

    # while there are elements in the left and right runs
        while i <= mid and j <= high:

            if A[i] <= A[j]:
                aux[k] = A[i]
                k = k + 1
                i = i + 1
            else:
                aux[k] = A[j]
                k = k + 1
                j = j + 1

    # copy remaining elements
        while i <= mid:
            aux[k] = A[i]
            k = k + 1
            i = i + 1

    # No need to copy the second half (since the remaining items
    # are already in their correct position in the auxiliary array)

    # copy back to the original list to reflect sorted order
        for i in range(low, high + 1):
            A[i] = aux[i]


    # Sort list `A[low…high]` using auxiliary list aux
Codeium: Refactor | Explain | Generate Docstring | X
def mergesort(A, aux, low, high):

        # base case
        if high <= low:                     # if run size <= 1
            return

        # find midpoint
        mid = (low + ((high - low) >> 1))

        # recursively split runs into two halves until run size <= 1,
        # then merge them and return up the call chain

        mergesort(A, aux, low, mid)         # split/merge left half
        mergesort(A, aux, mid + 1, high)    # split/merge right half

        merge(A, aux, low, mid, high)       # merge the two half runs
```

Parte II

```
def isSorted(A):

            prev = A[0]
            for i in range(1, len(A)):
                if prev > A[i]:
                    print("MergeSort Fails!!")
                    return False

                prev = A[i]

            return True


        # Implementation of merge sort algorithm in Python
if __name__ == '__main__':

            A = [12, 3, 18, 24, 0, 5, -2]
            aux = A.copy()

            # sort list `A` using auxiliary list `aux`
            mergesort(A, aux, 0, len(A) - 1)

            if isSorted(A):
                print(A)
```

Vantagens do *Merge Sort*

- Eficiência: O *Merge Sort* possui uma complexidade temporal média e pior caso de O(n log n), o que o torna

um algoritmo de ordenação eficiente para conjuntos de dados grandes.

- Estabilidade: O *Merge Sort* preserva a ordem original dos elementos iguais, o que o torna um algoritmo estável.
- Versatilidade: Funciona tanto em listas quanto em arrays, e pode ser implementado de forma iterativa ou recursiva.

Desvantagens do *Merge Sort:*

- Espaço: O *Merge Sort* requer memória auxiliar para armazenar as *sub-listas* durante a divisão e combinação.
- Custo de Combinação: A etapa de combinação pode ser um pouco mais complexa do que a simples comparação e troca de elementos em outros algoritmos.

Conclusão:

O *Merge Sort* destaca-se por sua eficiência, estabilidade e versatilidade, tornando-o uma escolha popular para ordenação de grandes conjuntos de dados. A sua implementação recursiva, dividindo e conquistando o problema da ordenação, demonstra a elegância desse algoritmo clássico.

CAPÍTULO 12 – QUICKSELECT

O *Quickselect* é um algoritmo relacionado ao *Quicksort* e é utilizado para encontrar o *k-ésimo* menor elemento em uma lista não ordenada. Ele segue a abordagem de dividir e conquistar, semelhante ao *Quicksort*, mas, em vez de ordenar toda a lista, foca apenas na parte relevante para encontrar *o k-ésimo* menor elemento.

Processo

- Escolha do pivô: O algoritmo seleciona um pivô a partir da lista não ordenada. Geralmente, o pivô é escolhido aleatoriamente, mas também pode ser escolhido de outras formas, como o primeiro ou o último elemento da lista.

- Particionamento: O algoritmo rearranja os elementos da lista de forma que todos os elementos menores que o pivô fiquem à esquerda dele e todos os elementos maiores fiquem à direita. Isso é feito movendo os

elementos menores que o pivô para a esquerda e os maiores para a direita.

- Verificação da posição do pivô: Após o particionamento, o pivô está em sua posição final na lista. Se a posição do pivô for igual a k - 1 (onde k é o índice do elemento desejado), então o elemento pivô é exatamente o k-*ésimo* menor elemento. Caso contrário, o algoritmo decide em qual parte (esquerda ou direita) continuar a busca, com base na posição do pivô em relação a k - 1.

- Repetição: Os passos 1 a 3 são repetidos recursivamente para a parte relevante da lista até que o *k-ésimo* menor elemento seja encontrado.

Código Fonte em Python

```python
def quickselect(arr, k):
    if k < 1 or k > len(arr):
        return None

    pivot = arr[random.randint(0, len(arr) - 1)]
    smaller = [x for x in arr if x < pivot]
    larger = [x for x in arr if x > pivot]

    if k - 1 < len(smaller):
        return quickselect(smaller, k)
    elif k - 1 >= len(arr) - len(larger):
        return quickselect(larger, k - (len(arr) - len(larger)))
    else:
        return pivot
```

Vantagens do *Quickselect:*

- Eficiência: O *Quickselect* apresenta uma complexidade temporal média e pior caso de O(n log n), o que o torna um algoritmo eficiente para encontrar o *k-ésimo* menor elemento em grandes conjuntos de dados.

- Menor espaço de memória: Comparado ao *Quicksort*, o *Quickselect* requer menos memória auxiliar, pois não precisa ordenar toda a lista.

- Versatilidade: Pode ser aplicado para encontrar o *k-ésimo* maior elemento, alterando a direção das comparações.

Desvantagens do *Quickselect:*

- Aleatoriedade: O desempenho do *Quickselect* pode ser afetado pela escolha do pivô. Pivôs inadequados podem levar a um pior desempenho na recursão.
- Instabilidade: O *Quickselect* não ordena a lista completa, apenas retorna o *k-ésimo* menor elemento.

Conclusão:

O *Quickselect* destaca-se como uma ferramenta eficiente para encontrar *o k-ésimo* menor elemento em grandes conjuntos de dados, especialmente quando a ordenação completa da lista não é necessária. A sua combinação de eficiência, baixo consumo de memória e versatilidade torna-o uma opção vantajosa para diversas aplicações, como seleção de estatísticas, análise de dados e algoritmos de mineração de dados.

CAPÍTULO 13 - QUICKSORT

O *Quicksort* é um algoritmo eficiente de classificação baseado na abordagem de dividir e conquistar. Ele divide a lista em torno de um elemento chamado "pivô", rearranja os elementos de forma que os menores que o pivô fiquem à esquerda e os maiores fiquem à direita, e, em seguida, repete o processo recursivamente nas duas metades resultantes até que a lista esteja completamente ordenada.

Processo

- Escolha do pivô: O algoritmo seleciona um elemento da lista para ser o pivô. Geralmente, o pivô é escolhido como o último elemento da lista, mas também pode ser escolhido de outras formas, como o primeiro elemento ou um elemento aleatório.

- Particionamento: O algoritmo rearranja os elementos da lista de forma que todos os elementos menores que o pivô fiquem à esquerda dele e todos os elementos

maiores fiquem à direita. Isso é feito movendo os elementos menores que o pivô para a esquerda e os maiores para a direita.

- Recursão: O algoritmo repete o processo descrito acima para as duas metades resultantes da lista, ou seja, para a sublista à esquerda do pivô e para a sublista à direita do pivô. Esse processo é repetido recursivamente até que a lista esteja completamente ordenada.

- Combinação: Como o *Quicksort* é um algoritmo de classificação *in-place*, ou seja, não requer espaço adicional para armazenar as *sub-listas*, não há uma etapa de combinação explícita. A lista é ordenada "*in-place*" durante o processo de particionamento e recursão.

Código Fonte em Python

Parte I

```python
def swap(A, i, j):

    temp = A[i]
    A[i] = A[j]
    A[j] = temp


# Partition using the Lomuto partition scheme
Codeium: Refactor | Explain | Generate Docstring | X
def partition(a, start, end):

    # Pick the rightmost element as a pivot from the list
    pivot = a[end]

    # elements less than the pivot will be pushed to the left of `pIndex`
    # elements more than the pivot will be pushed to the right of `pIndex`
    # equal elements can go either way
    pIndex = start

    # each time we find an element less than or equal to the pivot,
    # `pIndex` is incremented, and that element would be placed
    # before the pivot.
    for i in range(start, end):
        if a[i] <= pivot:
            swap(a, i, pIndex)
            pIndex = pIndex + 1

    # swap `pIndex` with pivot
    swap(a, end, pIndex)

    # return `pIndex` (index of the pivot element)
    return pIndex
```

Parte II

```python
# Quicksort routine
Codeium: Refactor | Explain | Generate Docstring | X
def quicksort(a, start, end):

    # base condition
    if start >= end:
        return

    # rearrange elements across pivot
    pivot = partition(a, start, end)

    # recur on sublist containing elements less than the pivot
    quicksort(a, start, pivot - 1)

    # recur on sublist containing elements more than the pivot
    quicksort(a, pivot + 1, end)


# Python implementation of the Quicksort algorithm
if __name__ == '__main__':

    a = [9, -3, 5, 2, 6, 8, -6, 1, 3]

    quicksort(a, 0, len(a) - 1)

    # print the sorted list
    print(a)
```

Vantagens do *Quicksort:*

- Eficiência: *O Quicksort* possui uma complexidade temporal média de O(n log n), o que o torna um

algoritmo de ordenação muito eficiente para grandes conjuntos de dados.

- *In-place*: O *Quicksort* realiza a ordenação na própria lista original, sem a necessidade de memória auxiliar, o que o torna mais eficiente em termos de uso de memória.

- Simplicidade: O algoritmo *Quicksort* tem uma estrutura relativamente simples e intuitiva, facilitando sua compreensão e implementação.

Desvantagens do *Quicksort:*

- Complexidade temporal pior caso: O *Quicksort* apresenta uma complexidade temporal no pior caso de O(n^2), que pode ocorrer em casos específicos, como quando a lista já está ordenada ou quando o pivô escolhido é sempre o menor ou maior elemento.

- Instabilidade: O *Quicksort,* em sua implementação básica, não preserva a ordem original de elementos iguais na lista.

- Aleatoriedade: O desempenho do *Quicksort* pode ser afetado pela escolha do pivô. Pivôs inadequados podem levar a um pior desempenho na recursão.

Conclusão:

O *Quicksort* destaca-se como um algoritmo de ordenação eficiente e versátil, sendo uma escolha popular para grandes conjuntos de dados. A sua simplicidade, baixo uso de memória e boa performance média tornam-no uma ferramenta valiosa para diversos cenários. No entanto, é importante considerar seu pior caso de complexidade e a instabilidade em algumas implementações.

CAPÍTULO 14 - SELECTION SORT

O *Selection Sort* é um algoritmo simples de classificação que encontra repetidamente o menor elemento restante em uma lista não ordenada e o coloca em sua posição correta. Ele percorre a lista várias vezes, dividindo-a em uma parte ordenada e outra não ordenada.

Processo

- Repetição Principal: Repita o processo de ordenação até que todos os elementos da lista estejam em seus lugares corretos.
- Menor Elemento: Em cada repetição, encontre o menor elemento da lista não ordenada. Este menor elemento pode ser encontrado através de comparações simples entre os elementos restantes.
-

- Troca: Troque o menor elemento encontrado com o elemento na primeira posição da lista não ordenada. Essa troca coloca o menor elemento em sua posição correta (final da lista ordenada até o momento).

- Atualização: Após a troca, a parte ordenada da lista aumenta em um elemento, e a parte não ordenada diminui em um elemento. Isso significa que, na próxima repetição, o algoritmo considerará apenas os elementos restantes na parte não ordenada.

Código Fonte em Python

```python
def selection_sort(array):
  for i in range(len(array)):
    menor_indice = i
    for j in range(i + 1, len(array)):
      if array[j] < array[menor_indice]:
        menor_indice = j
    array[i], array[menor_indice] = array[menor_indice], array[i]

  return array
```

Vantagens do *Selection Sort*:

- Simplicidade: O *Selection Sort* é um algoritmo muito simples de entender e implementar, sendo ideal para iniciantes em programação.
- Eficiência em casos específicos: O *Selection Sort* apresenta bom desempenho em casos onde a lista já está parcialmente ordenada, pois o número de trocas de elementos é menor.
- Baixo uso de memória: O algoritmo requer pouca memória extra, pois manipula a lista original e não utiliza estruturas auxiliares complexas.

Desvantagens do *Selection Sort*:

- Ineficiência para grandes conjuntos: O *Selection Sort* torna-se ineficiente para grandes conjuntos de dados, pois o número de comparações e trocas de elementos cresce quadraticamente (n^2).

- Instabilidade (opcional): Dependendo da implementação, o *Selection Sort* pode alterar a ordem original de elementos iguais no array.

- Sem otimizações: O *Selection Sort* não possui otimizações inerentes como outros algoritmos de ordenação mais eficientes.

Conclusão:

O *Selection Sort* é um algoritmo de ordenação simples e com baixo custo de implementação, sendo uma boa escolha para iniciantes ou para cenários com conjuntos de dados pequenos ou parcialmente ordenados. No entanto, para grandes conjuntos de dados, algoritmos como *Merge Sort* ou *Quick Sort* costumam ser mais performáticos e eficientes.

CAPÍTULO 15 - ALGORITMOS FUNDAMENTAIS SOBRE GRAFOS

Conceitos, Notas e Terminologia

- Grafo orientado: par (V,E) com V conjunto finito de vértices (ou nos) e E uma relação binaria em V - o conjunto de arestas (ou arcos) do grafo.
- Grafo não orientado: o conjunto E é constituído por pares não-ordenados (conjuntos com 2 elementos) neste tipo de grafo. Assim, (i,j) e (j,i) correspondem ao mesmo arco.
- Um arco (i,i) designa-se por anel. Os anéis são interditos nos grafos não-orientados.
- i e j são, respetivamente, vértices de origem e destino do arco (i,j). j diz-se adjacente a i.

- A relação de adjacência pode não ser simétrica num grafo orientado.
- Grau de entrada (de saída) de um vértice num grafo orientado e o número de arcos com destino (origem) no vértice. O grau do vértice e a soma de ambos.
- A distância d(s,v) do vértice s ao vértice v define-se como caminho mais curto entre esses vértices.
- Um sub-caminho de um caminho e uma qualquer sua subsequência contigua.
- Um caminho diz-se simples se todos os seus vértices são distintos.
- Um ciclo e um caminho de comprimento ≥ 1 com início e fim no mesmo vértice. Note-se que existe sempre um caminho de comprimento 0 de um vértice para si próprio, que não se considera ser um ciclo.
- Um grafo diz-se acíclico se não contem ciclos. Um grafo orientado acíclico e usualmente
- Designado por DAG (Directed Acyclic Graph).

- Um grafo não-orientado diz-se ligado se para todo o par de vértices existe um caminho que os liga. Os componentes ligados de um grafo são os seus maiores sub-grafos ligados.

- Um grafo e fortemente ligado se, para todo o par de vértices m,n existem caminhos de m para n e de n para m. Os componentes fortemente ligados de um grafo são os seus maiores sub-grafos fortemente ligados.

Arvores

- Uma floresta e um grafo não-orientado acíclico.

- Uma arvore e um grafo não-orientado, acíclico e ligado.

- Arvores com raiz: a escolha de um vértice arbitrário para raiz de uma arvore define noções de descendentes de um vértice e de sub-árvore com raiz num vértice. Existe um caminho único da raiz para qualquer vértice. Uma arvore com raiz pode também ser vista como um caso particular de grafo orientado.

- Arvores Ordenadas: uma arvore ordenada e uma arvore com raiz em que a ordem dos descendentes de cada vértice e relevante.

Representação de Grafos em Computadores

Listas de adjacências

Consiste num vector Adj de |V| listas ligadas. A lista Adj[i] contem todos os vértices j tais que (i,j)∈E, ie, todos os vértices adjacentes a i.

- A soma dos comprimentos das listas ligadas e |E|.
- Se o grafo for pesado (ie, se contiver informação associada aos arcos), o peso de cada arco pode ser incluído no vértice respetivo numa das listas ligadas.
- No caso de um grafo não-orientado, esta representação pode ser utilizada desde que antes se converta o grafo num grafo orientado, substituindo cada arco (não-orientado) por um par de arcos (orientados). A

representação contém informação redundante e o comprimento total das listas e 2|E|.

- O espaço necessário de memoria em qualquer dos casos e Θ(V+E).

Matriz de adjacências

- É uma representação estática, apropriada para grafos densos, em que |E| se aproxima de $|V|_2$.
- Tem dimensão |V| x |V|, A = (a_{ij}), com a_{ij} = 1 se (i,j)∈E; a_{ij} = 0 em caso contrário.
- Se o grafo for pesado, o peso de cada arco pode ser incluído na respetiva posição da matriz (em vez de 1).
- Se o grafo for não-orientado a matriz de adjacências e simétrica. E possível armazenar apenas o triangulo acima da diagonal principal.
- Vantagem em relação as listas de adjacências: e imediato verificar a adjacência de dois vértices (Θ(1) sem ter de percorrer uma lista ligada).

- Espaço de memoria necessário e $\Theta(V_2)$ – independente do número de arcos.

Travessias em Grafos

Pesquisa em Largura ("*Breadth-first Search*")

Dado um grafo G = (V,E) e um seu vértice s, um algoritmo de pesquisa explora o grafo passando por todos os vértices alcançáveis a partir de s.

- O algoritmo de pesquisa em largura calcula a distância (menor número de arcos) de s a cada vértice;
- Produz uma arvore (sub-grafo de G) com raiz s e contendo todos os vértices alcançáveis a partir de s;
- Nessa arvore o caminho da raiz s a cada vértice corresponde ao caminho mais curto entre os dois vértices;
- Algoritmo para grafos orientados e não-orientados.

ALGORITMO

- Utiliza a estratégia: todos os vértices a distância k de são visitados antes de qualquer vértice a distância k+1 de s.

- Pinta os vértices (inicialmente brancos) de cinzento ou preto. “Branco” ainda não foi descoberto, “cinzento” já foi visitado, mas pode ter adjacentes ainda não descobertos, “preto” só tem adjacentes já descobertos. Os “cinzentos” correspondem a fronteira entre descobertos e não-descobertos.

- A arvore de travessia em largura e expandida atravessando-se a lista de adjacências de cada vértice cinzento u; para cada vértice adjacente v acrescenta-se a arvore o arco (u,v).

- Utiliza-se uma fila de espera FIFO.

- O sub-grafo dos antecessores de G e uma arvore de pesquisa em largura (APL).

- Adequado para utilização com representação por listas de adjacências.

- Tempo de execução Θ(V+E).

Pesquisa em Profundidade "*Depth-first Search*"

ALGORITMO

- Utiliza a estratégia: os próximos arcos a explorar tem origem no mais recente vértice descoberto que ainda tenha vértices adjacentes não explorados.
- Quando todos os adjacentes a um vértice v tiverem sido explorados, o algoritmo recua para explorar vértices com origem no vértice a partir do qual v foi descoberto.
- O grafo dos antecessores de G e uma floresta composta de várias arvores de pesquisa em profundidade (APP).
- O algoritmo faz ainda uma etiquetagem dos vértices com marcas temporais; para o instante em que o vértice é descoberto e quando todos os seus adjacentes são descobertos. O algoritmo não tem, no entanto, necessariamente que produzir essa etiquetagem.
- Tempo de execução Θ(V+E).

- Utiliza-se uma fila de espera LIFO.

Arvores Geradoras Mínimas (MST - “*Minimum Spanning Trees*”)

- Seja G = (V,E) um grafo não-orientado, ligado. Uma arvore geradora de G e um sub-grafo (V,T) acíclico e ligado de G;
- (V,T) e necessariamente uma arvore e liga todos os vértices de G entre si;
- Associe-se a cada arco um peso numérico;
- Arvores geradoras mínimas de G são aquelas para as quais o peso total e mínimo.

Algoritmo de Prim para Determinação de MSTs

- O algoritmo considera em cada instante da execução o conjunto de vértices dividido em três conjuntos distintos: (1) vértices da arvore construída até ao momento (2) os vértices na orla, alcançáveis a partir dos anteriores (3) restantes vértices. Em cada passo

seleciona-se um arco (com origem em 1 e destino em 2) para acrescentar a arvore. O vértice destino desse arco também e acrescentado.

- O algoritmo de Prim seleciona sempre o arco com menor peso nestas condições.

ESTRUTURA GERAL:

void MST((V,E)) {

selecionar vértice arbitrário x para início da árvore;

while(existem vértices na orla) {

selecionar um arco de peso mínimo entre um vértice da árvore e um vértice na orla;

acrescentar esse arco e o respetivo vértice destino à árvore;

}

}

- Grafo implementado por listas de adjacências.

- São mantidos vetores adicionais para o estado de cada vértice (indicando se esta na arvore, na orla, ou nos restantes), para a construção da arvore, e para o peso dos arcos candidatos.
- E mantida uma lista ligada correspondente aos vértices na orla.
- A pesquisa e remoção do arco candidato de menor peso e feita por uma travessia da orla e consulta direta dos pesos dos arcos candidatos.
- Se se colocar na arvore os nos da orla e os respetivos arcos candidatos, a substituição de um arco candidato e feita alterando-se o vetor *parent* e o vetor de pesos dos arcos candidatos.
- Tempo de execução $\Theta(V_2+E)$.

Caminhos Mais Curto ("*Shortest Paths*")

- Algoritmo de *Dijkstra* para determinação de caminhos mais curtos → muito semelhante ao algoritmo de Prim para MST.
- Em cada passo seleciona um vértice da orla para acrescentar a arvore que vai construindo.
- O algoritmo vai construindo caminhos cada vez mais longos (ie, com peso cada vez maior) a partir de v, dispostos numa arvore e para quando alcançar w.
- A diferença para o algoritmo de Prim e o critério de seleção do arco candidato. Arcos candidatos:
    1. para cada vértice z na orla, existe um caminho mais curto v, v_1,...v_k na arvore construída, tal que $(v_k,z)\in E$.
    2. se existem vários caminhos v, v_1,...v_k e arco (v_k,z) nas condições anteriores, o arco candidato (único) de z será aquele para o qual $d(v_k,v) + w(v_k,z)$ for mínimo.

3. Em cada passo o algoritmo seleciona um vértice da orla para acrescentar a arvore. Este será o vértice z com valor $d(v_k,v)+w(v_k,z)$ mais baixo.

- Tempo de execução $\Theta(V\log(V) +E)$.

Variantes do Problema dos Caminhos Mais Curtos

Seja G um grafo orientado,

Single-source Shortest Paths: determinar todos os caminhos mais curtos com origem num vértice v dado e destino em cada vértice de G. Pode ser resolvido por uma versão ligeiramente modificada do algoritmo de *Dijkstra*.

Single-destination Shortest Paths: determinar todos os caminhos mais curtos com destino num vértice w dado e e origem em cada vértice de G. Pode ser resolvido por um algoritmo de resolução do problema anterior, operando sobre um grafo obtido do original invertendo-se o sentido de todos os arcos.

- *All-pairs Shortest Paths*: determinar caminhos mais curtos entre todos os pares de vértices de G.

Estratégias Algorítmicas - Algoritmos “*Greedy*”

- Podem ser usados na resolução de problemas de otimização.
- Um algoritmo *greedy* efetua uma sequência de escolhas; em cada ponto de decisão do algoritmo esta estratégia elege a solução que “parece” melhor.
- Nem sempre esta estratégia resulta em soluções globalmente ótimas, pelo que e necessário provar que a estratégia e adequada.
- Para que um problema seja resolúvel por uma estratégia “*greedy*” e condição necessária que possua sub-estrutura ótima, ou seja:
    1. é efetuada uma escolha, da qual resulta um (unico) sub-problema que deve ser resolvido.
    2. Essa escolha e efetuada localmente sem considerar soluções de sub-problemas – o novo sub-problema resulta da escolha efetuada; a escolha greedy não pode depender da solução do sub-problema criado.

3. Trata-se de um método "*top-down*": cada problema e reduzido a um mais pequeno por uma escolha *greedy* e assim sucessivamente.

Fecho Transitivo de um Grafo

O objetivo é, dado um grafo orientado G = (V,E) com o conjunto de vértices V = {1,2,...,n}, descobrir se existe um caminho em G desde i ate j para todos os pares de vértices i,j∈V. O fecho transitivo de G e definido como o grafo G* = (V, E*), onde E* = {(i,j) : existe um caminho desde o vértice i ate o vértice j em G}.

Algoritmo:

```
void TC(int A[][], int R[][], int n) {
/* G representado por A, fecho transitivo em R */
R = A; /* copia matriz... */
for (i=1 ; i<=n; i++)
for (j=1 ; j<=n ; j++)
for (k=1 ; k<=n ; k++)
```

```
if (R[i][k] && R[k][j]) R[i][j] = 1;
}
```

- Algoritmo de Warshall, mais eficiente, executa em tempo θ (|V|3):

```
void TC(int A[][], int R[][], int n) {
/* G representado por A, fecho transitivo em R */
R = A; /* copia matriz... */
for (k=1 ; k<=n ; k++)
for (j=1 ; j<=n ; j++)
for (i=1 ; i<=n; i++)
if (R[i][k] && R[k][j]) R[i][j] = 1;
}
```

Programacao Dinamica

Consiste na otimização de uma definição recursiva quando há margem para armazenamento de resultados intermédios, que se calculam “bottom-up”.

Exemplo:

- sequencia de Fibonacci. Basta calcular os valores sequencialmente e armazena-los num vetor (tempo Θ(n)), as custas de espaço adicional também em Θ(n).
- algoritmo de Floyd-Warshall.

CAPÍTULO 16 - ANÁLISE E CORRECÇÃO DE PROGRAMAS

CORREÇÃO PARCIAL VS CORREÇÃO TOTAL:

A correção parcial garante que, se o programa terminar (ou seja, se ele não entrar em um *loop* infinito ou travar), ele produzirá o resultado correto. No entanto, não garante que o programa terminará.

A correção total, por outro lado, garante não apenas que o programa produzirá o resultado correto se terminar, mas também que ele terminará sempre para as entradas especificadas (ou seja, que ele é parcialmente correto e termina para todas as entradas válidas). Isso implica tanto a correção parcial quanto a garantia de terminação.

DEFINIÇÕES RELEVANTES:

Para garantir a correção de programas foi desenvolvido por Charles Hoare em 1969 um sistema formal com um conjunto de regras lógicas que garante um raciocínio rigoroso sobre a *corretude* de computação. Assim a expressão fundamental que introduz este tema é denominada por Triplos de Hoare e representam-se do seguinte modo:

{P} S {Q}, sendo {P} um conjunto de pré-condições, S um qualquer programa computacional e {Q} um conjunto de pós-condições.

Quer {P} quer {Q} são assunções (fórmulas da lógica de predicados de 1º ordem) que definem, respetivamente, condições que se querem ter verificadas antes e após S, sendo S um conjunto de operações computacionais.

As regras de Hoare têm a seguinte estrutura:

$$\frac{H1 \ldots Hn}{C} regra$$

Acima do eixo representam-se todas as assunções que se assumem válidas e abaixo do eixo o que se concluiu por um

processo dedução do que é válido nas assunções. O nome de cada regra é escrito à direita do eixo.

Outro detalhe relevante para o que se aborda de seguida são os invariantes de ciclo (*loop*) que ao longo deste capítulo serão, frequentemente, tratados pela letra I.

Um invariante de ciclo é uma condição que é verdadeira antes e depois de cada iteração do ciclo. Invariantes de ciclo são usados para provar a correção de ciclos, garantindo que, mesmo após várias iterações, uma certa propriedade continue verdadeira.

Analisemos o exemplo abaixo:

```
soma = 0
for elemento in lista:
    soma += elemento
```

Neste exemplo um possível invariante de ciclo é que, após cada iteração, a variável soma é igual à soma dos elementos processados até aquele ponto na lista.

Fortalecimento de condições

Quanto à teoria de Hoare o fortalecimento de um conjunto de condições corresponde a aumentar o leque de assunções desse conjunto. Ao crescer o conjunto estamos a fortalecer a especificação dada ao programa.

$$\frac{\mathrm{R} \Rightarrow \mathrm{P} \quad \{\mathrm{P}\}\ \mathrm{S}\ \{\mathrm{Q}\}}{\{\mathrm{R}\}\ \mathrm{S}\ \{\mathrm{Q}\}}\ (Fort)$$

Enfraquecimento de condições

Enfraquecer um conjunto de condições é o processo oposto ao fortalecimento, ou seja, está-se a diminuir o espaço de assunções e, portanto, a dar maior flexibilidade de especificação a um programa.

$$\frac{\{\mathrm{P}\}\ \mathrm{S}\ \{\mathrm{Q}\} \quad \mathrm{Q} \Rightarrow \mathrm{R}}{\{\ \mathrm{P}\ \}\ \mathrm{S}\ \{\mathrm{R}\}}\ (Enfraq)$$

Atribuição – 1

Para atribuição 1 não é declarada nenhuma assunção assumida verdadeira porque se pretende que seja trivialmente verdade que se valide que uma pré-condição seja válida antes e

depois de qualquer atribuição feita às variáveis livres da expressão que representa P.

$$\frac{}{\{P[x \setminus E]\}\ x :=\ E\ \{P\}}\ (Atrib1)$$

Atribuição – 2

A regra de atribuição 2 prova que assumindo que é verdade se P acontece então é válido substituir as variáveis livres x por E em Q, e que com isso se pode concluir que com a pré-condição P e atribuindo E a uma expressão x a pós-condição se mantém válida. Esta regra garanta que após atribuições uma pré-condição se mantenha válida.

$$\frac{P\ \Rightarrow (Q[x \setminus E])}{\{\ P\ \}\ x =\ E\ \{Q\}}\ (Atrib2)$$

Sequência

A regra da sequência prova que nos casos em que a pós condição de um conjunto de operações computacionais corresponde à pré-condição de outro conjunto podemos então concluir que, se começarmos com a condição P e executarmos a sequência de comandos S1;S2 isso leva-nos à condição Q. Por

outras palavras, a condição R atua como uma "ponte" entre as duas partes do programa, permitindo que as garantias de correção sejam combinadas.

$$\frac{\{P\}\ S1\ \{R\} \qquad \{R\}\ S2\ \{Q\}}{\{P\}\ S1; S2\ \{Q\}}\ (;)$$

Condicional

A regra condicional prova que no caso de termos um conjunto computacional (S1) em que a pré-condição é formada por uma condição P e uma condição c. E um outro conjunto (S2) em que a pré-condição é formada pela condição P e a negação da condição c. Se ambos os conjuntos tiverem a mesma pós-condição podemos concluir que, começando com a condição P, a execução da estrutura condicional (*if* c *then* S1 *else* S2) levará à condição Q, independentemente de c ser verdadeira ou falsa.

$$\frac{\{P\ \wedge c\}\ S1\ \{Q\} \qquad \{P \wedge \neg c\}\ S2\ \{Q\}}{\{P\}\ if\ c\ then\ S1\ else\ S2\ \{Q\}}\ (If)$$

Ciclo - 1

A regra While1 abaixo descrita assume válido que tendo como pré-condição um invariante de ciclo I e uma condição c se se correr um programa S o invariante se mantém verdadeiro e que se pode daqui concluir que partindo de um variante I válido e executar um ciclo que corre S enquanto a condição c é verificada então mesmo quando deixamos de estar no corpo do ciclo o invariante I ainda é válido. Esta regra assegura a validade de um invariante dentro e após um ciclo.

$$\frac{\{I \wedge c\}\, S\, \{I\}}{\{I\}\, while\ c\ S\, \{I\ \wedge \neg c\}}\ (While1)$$

Ciclo – 2

Na regra While2 assumem-se válidas três assunções que (i) se uma proposição P acontece então o invariante de ciclo I é verdade, é verificado que (ii) juntamente com I se verificar uma condição c e se se correr um programa S no final I se mantém válido e que (iii) se temos uma invariante I válido e uma condição c não satisfeita então Q é uma proposição válida. Esta regra conclui que tendo como verdade i, ii e iii se conclui que se correrrmos um programa S a partir da pré-condição {P} em ciclo

enquanto a condição c é verdadeira então {Q} é uma pré-condição válida. Esta é uma regra que generaliza que a pré e pós condição têm que ser válidas antes e após um ciclo, respetivamente.

$$\frac{P \Rightarrow I \quad \{I \wedge c\}\, S\, \{I\} \quad (I \wedge \neg c) \Rightarrow Q}{\{P\}\, while\ c\ S\, \{Q\}}\ (While2)$$

A lógica de Hoare padrão proporciona apenas correção parcial. Para obter correção total aplicam-se as regras da correção parcial mencionadas acima, substituindo-se apenas os parênteses curvos por retos. As exceções são os casos "Ciclo-1" e "Ciclo-2".

Para o Ciclo-1 proporcionar uma correção total tem de ser acrescentada uma prova de decréscimo da variante do ciclo à regra do Ciclo-1 anteriormente apresentada. A variante é representada por V e é uma função que mapeia o estado do programa para um número natural e que diminui a cada iteração do loop, garantindo a terminação. O valor inicial da variante é representado por v0. Se I ∧ c ∧ (V = v0) é verdadeiro antes da execução de S, então I ∧ (V < v0) será verdadeiro depois da execução de S. Isso garante que V está a diminuir, o que é

necessário para garantir a terminação do *loop*. Assim o Ciclo-1 passa a ser representado pela equação abaixo:

$$(I \wedge c) \Rightarrow V \geqslant 0 \qquad (\boldsymbol{p1})$$

$$[I \wedge c \wedge (V = v0)] \; S \; [I \wedge (V < v0)] \qquad (\boldsymbol{p2})$$

$$\frac{(p1) \qquad (p2)}{[I] \; while \; c \; S \; [I \wedge \neg c]} \; (While - 1)$$

Para o Ciclo-2 proporcionar uma correção total tem de ser acrescentada uma prova de decréscimo da variante do ciclo à regra do Ciclo-2 anteriormente apresentada, assim como foi feito para o Ciclo-1. Desta forma, o resultado do ciclo-2 passa a ser representado pela equação abaixo:

$$P \Rightarrow I \qquad (p1)$$

$$(I \wedge c) \Rightarrow V \geqslant 0 \qquad (p2)$$

$$[I \wedge c \wedge V = v0] \; S \; [I \wedge V < v0] \qquad (p3)$$

$$(I \wedge \neg c) \Rightarrow Q \qquad (p4)$$

$$\frac{(p1) \quad (p2) \quad (p3) \quad (p4)}{[P] \; while \; c \; S \; [Q]} \; (While - 2)$$

CAPÍTULO 17 - PROBLEMA NP-COMPLETO

Problemas de otimização vs problemas de decisão

Um problema de otimização consiste em encontrar a melhor solução possível dentro de um conjunto de soluções possíveis, de acordo com algum critério específico. A resposta para um problema de otimização geralmente envolve a seleção de uma solução ótima que minimiza ou maximiza uma determinada função objetivo.

Um exemplo de um problema de otimização seria dado um grafo G com pesos nas arestas, encontrar uma árvore geradora com o menor peso total, sendo que o "peso" de uma árvore geradora é a soma dos pesos das arestas que a compõem. A solução para este problema seria a própria árvore geradora mínima, ou seja, o subgrafo que tem o menor peso possível.

Um problema de decisão é um tipo de problema onde a solução é simplesmente uma resposta "sim" ou "não". Em vez de

buscar a melhor solução, um problema de decisão pergunta se existe alguma solução que satisfaz uma determinada condição.

Um exemplo de um problema de decisão seria descobrir se dado um grafo G e um valor k, existe uma árvore geradora de G cujo peso total é menor ou igual a k. A solução para esse problema seria "sim" ou "não". "Sim" se existe uma árvore geradora com peso ≤ k, e "não" se não existe tal árvore.

Classe P

A Classe P é composta por problemas de decisão que podem ser resolvidos em tempo polinomial, ou seja, em tempo O(P(n)), onde P(n) é um polinômio em n, que representa o tamanho da entrada. Isso significa que o tempo de execução cresce a uma taxa que pode ser expressa como n^k *para algum inteiro k*, em vez de, por exemplo, exponencialmente 2^n, o que seria muito mais ineficiente.

Por exemplo um algoritmo que resolve um problema em tempo O(n^2) é limitado polinomialmente porque o tempo de execução é um polinômio de grau 2 em relação ao tamanho da entrada n.

Essa classe inclui tanto problemas considerados “razoáveis” quanto problemas cuja resolução pode ser desafiante, especialmente quando o polinômio envolvido cresce rapidamente.

Se um problema não pertence à Classe P, a sua resolução é, em geral, considerada impraticável, pois não se conhece um algoritmo que o resolva eficientemente. Um algoritmo é considerado limitado polinomialmente se, no pior caso, o tempo de execução pode ser descrito por uma função polinomial em relação ao tamanho da entrada. Consequentemente, um problema é limitado polinomialmente se existe um algoritmo desse tipo capaz de resolvê-lo.

Além disso, é importante notar que a Classe P é fechada sob várias operações, como adição e multiplicação. Isso significa que, ao combinar dois problemas que pertencem a P através dessas operações, o problema resultante também estará na Classe P, mantendo-se dentro dos limites de tempo polinomial.

Algoritmos Não-Determinísticos

Algoritmos não-determinísticos são conceitos teóricos usados principalmente para a classificação de problemas na

teoria da complexidade computacional. Esses algoritmos diferem dos tradicionais, pois podem apresentar comportamentos diferentes em execuções distintas, mesmo com a mesma entrada. Eles funcionam em duas fases principais:

Fase Não-Determinística: Nesta fase inicial, o algoritmo "escreve" em algum lugar da memória uma *string* arbitrária s. Essa *string* pode variar a cada execução do algoritmo, representando uma espécie de tentativa de "adivinhar" ou explorar possíveis soluções para o problema. Não há uma regra determinística para a escolha dessa *string*, o que significa que, em diferentes execuções, s pode assumir valores completamente diferentes.

Fase Determinística: Após gerar a *string* s na fase anterior, o algoritmo passa para uma fase determinística, onde s é lida e processada. Nesta fase, o algoritmo segue regras específicas e previsíveis para verificar se a *string* s corresponde a uma solução válida para o problema. Dependendo do processamento, podem ocorrer três resultados:

O algoritmo pára e responde "sim", indicando que s é uma solução válida.

O algoritmo pára e responde "não", indicando que s não é uma solução válida.

O algoritmo não pára, o que significa que ele entrou em um *loop* infinito ou está preso em uma computação sem fim.

A fase não-determinística pode ser vista como uma fase de "palpite", onde o algoritmo tenta adivinhar uma possível solução, enquanto a fase determinística verifica a correção desse palpite.

Algumas características dos Algoritmos Não-Determinísticos são nomeadamente:

Diversidade de resultados: Diferente dos algoritmos tradicionais, onde a mesma entrada sempre produz o mesmo resultado, algoritmos não-determinísticos podem gerar respostas diferentes ("sim" ou "não") em execuções distintas com a mesma entrada, dependendo do valor arbitrário escolhido na fase não-determinística.

Resposta definida pela existência de solução: A resposta de um algoritmo não-determinístico a uma entrada x é definida como "sim" se existe pelo menos uma execução do algoritmo com x que resulta em um output "sim". Ou seja, se alguma

execução consegue encontrar uma solução válida, o problema é considerado resolvido positivamente. Isso implica que a resposta "sim" de um algoritmo não-determinístico para uma entrada x indica que existe uma solução válida para o problema com esse input.

Execução em duas fases: O tempo de execução de um algoritmo não-determinístico é a soma do tempo gasto nas duas fases (não-determinística e determinística). Enquanto a fase determinística segue um comportamento previsível, a fase não-determinística é caracterizada pela escolha arbitrária da string s, que pode influenciar diretamente o resultado final.

Classe de Problemas NP *(Nondeterministic Polynomial-bounded)*

A classe de problemas NP, de forma informal, é composta por problemas de decisão em que a verificação de uma solução pode ser feita em tempo polinomial. Um algoritmo não-determinístico A é considerado limitado polinomialmente se, para qualquer entrada de dimensão n e uma resposta "sim", existe um polinômio P(n) tal que A pode produzir essa resposta em tempo O(P(n)). Em outras palavras, se uma solução existe,

ela pode ser gerada e verificada eficientemente, ou seja, em tempo polinomial.

A Classe NP inclui todos os problemas de decisão para os quais existe um algoritmo não-determinístico que seja limitado polinomialmente. A Classe P, que abrange problemas de decisão resolvidos diretamente em tempo polinomial por um algoritmo determinístico, está contida dentro da Classe NP. Isso acontece porque, se um problema pode ser resolvido em tempo polinomial, qualquer solução encontrada também pode ser verificada em tempo polinomial.

Uma abordagem de força bruta, que testa todas as possíveis soluções até encontrar a correta, pode ser aplicada para resolver problemas em NP. No entanto, essa estratégia tende a exigir tempo exponencial, uma vez que o número de soluções possíveis cresce rapidamente com o tamanho da entrada. Assim, mesmo que a verificação de soluções em NP seja rápida, encontrar a solução correta pode ser extremamente demorado, dependendo do método utilizado.

Problemas NP-completos

A classe de complexidade NP-completo é um subconjunto dos problemas de decisão dentro da classe NP, caracterizado pelo facto de que qualquer problema em NP pode ser reduzido, em tempo polinomial, a um dos problemas NP-completos. Isso significa que, se conseguirmos resolver eficientemente (em tempo polinomial) um único problema NP-completo, então todos os problemas em NP também podem ser resolvidos eficientemente. No entanto, provar que P=NP, o que implica que todos os problemas em NP poderiam ser resolvidos em tempo polinomial, é amplamente considerado improvável. Para demonstrar que P=NP, seria suficiente provar que qualquer problema NP-completo pode ser resolvido por um algoritmo polinomial, mas a crença geral na comunidade científica é de que essa equivalência não ocorre.

Dado que os problemas NP-completos são notoriamente difíceis de resolver de maneira eficiente, algumas estratégias práticas podem ser adotadas:

Escolha do algoritmo exponencial mais eficiente: Embora a resolução de problemas NP-completos geralmente exija

algoritmos com tempo de execução exponencial, é possível escolher o mais eficiente entre eles, otimizando o desempenho dentro das limitações conhecidas.

Foco na análise do caso médio: Em vez de se concentrar apenas no pior caso, que pode ser extremamente raro, é mais prático analisar o desempenho do algoritmo no caso médio, o que pode resultar em soluções mais rápidas e adequadas para a maioria das situações.

Estudo dos inputs frequentes: Analisar os tipos de entrada que ocorrem com mais frequência pode ajudar a escolher um algoritmo que, embora não seja o mais eficiente em todos os cenários, ofereça um desempenho superior para os casos mais comuns.

Dependência de resultados empíricos: Em muitos casos, a escolha do melhor algoritmo para um problema NP-completo pode depender mais de experimentação e resultados empíricos do que de uma análise teórica rigorosa. Testes práticos podem revelar otimizações ou comportamentos que não são evidentes apenas pela teoria.

Algoritmos de Aproximação ou Heurísticos

Algoritmos de Aproximação ou Heurísticos têm como princípio a utilização de algoritmos rápidos (pertencentes à classe P) que, embora não garantam encontrar a solução ótima, produzem soluções próximas do ideal. Muitas heurísticas são simples e eficientes, oferecendo soluções que, apesar de, possivelmente, não serem a melhor entre as existentes, ficam muito próximas da melhor. A definição de "proximidade à solução ótima" varia conforme o problema em questão.

Por exemplo, uma solução aproximada para o problema do caixeiro-viajante não é aquela que passa por "quase todos" os vértices do grafo, mas sim uma que visita todos os vértices e cujo peso total é próximo do mínimo possível. Em outras palavras, embora a solução não seja necessariamente a melhor possível, ela deve ser viável e derivada de uma execução de um algoritmo não-determinístico para o problema, com um custo próximo ao da solução ótima.

CAPÍTULO 18 - OPTIMIZAÇÕES

Na otimização de algoritmos, diversas técnicas podem ser aplicadas para melhorar a eficiência e o desempenho, dependendo do contexto e do tipo de problema que se está tentando resolver. Neste capítulo serão inumeradas apenas algumas delas.

Exploração de Estruturas de Dados Eficientes

Escolher a estrutura de dados correta pode ter um impacto profundo na eficiência do algoritmo. Diferentes estruturas de dados têm diferentes vantagens e desvantagens em termos de espaço, tempo e facilidade de uso. Por exemplo, os arrays são simples e rápidos de aceder, mas têm um tamanho fixo e exigem elementos deslocados ao inserir ou excluir. As tabelas de *hash* são eficientes para pesquisa, mas têm colisões potenciais e exigem funções de *hash*. As árvores são hierárquicas e adequadas para algoritmos recursivos, mas têm diferentes tipos e equilíbrios que afetam seu desempenho.

Paralelização

Dividir o trabalho do algoritmo em partes que podem ser executadas simultaneamente em múltiplos processadores ou núcleos pode reduzir o tempo de execução de um algoritmo. Técnicas como *threads,* processamento em *GPUs* e computação distribuída podem ser usadas.

Redução de Complexidade

Eliminação de cálculos redundantes: Identificar e eliminar cálculos repetidos ou redundantes pode reduzir significativamente o tempo de execução do algoritmo.

Pré-cálculo *(caching*): Armazenar resultados intermediários para uso posterior pode economizar tempo, especialmente em problemas onde a mesma ”sub-questão“ é resolvida múltiplas vezes.

REFÊNCIAS

Barros, J. B. (s.d.). *Introdução à Análise de Complexidade.*

Barros, J. B. (s.d.). *Introdução à Análise de Correcçãoo de Algoritmos.*

Cormen, T. H. (2012). *Algoritmos - Teoria e Prática.* Elsevier.

Jorge Sousa Pinto, M. A. (2017). Algoritmos e Complexidade. Apoio na Web via Blackboard.

Wayne, R. S. (2014). *Algorithms.*

AUTORES

Ricardo Rouco

Inês Ferreira