Resumo de Análise de Algoritmos

Este resumo abrange os principais tópicos de Análise de Algoritmos, com base nas apostilas e questões de provas anteriores fornecidas. O objetivo é consolidar o conhecimento para a preparação da prova.

1. Notação Assintótica

A Notação Assintótica é fundamental para analisar a eficiência de algoritmos, descrevendo como o tempo de execução ou o espaço de memória crescem em função do tamanho da entrada (n). Ela permite focar no comportamento do algoritmo para grandes entradas, ignorando constantes e termos de menor ordem.

1.1. Notação Big O (O) - Limite Superior

Define um limite superior para o tempo de execução de um algoritmo. Se $f(n) \in O(g(n))$, significa que a taxa de crescimento de $f(n)$ é menor ou igual à de $g(n)$ para valores grandes de n. Formalmente, $f(n) \in O(g(n))$ se existem constantes positivas c e n_0 tais que $0 \le f(n) \le c \cdot g(n)$ para todo $n \ge n_0$.

Exemplos de complexidade O(n): * Percorrer um array. * Soma dos elementos de um vetor. * Encontrar o maior elemento em um vetor.

Exemplos de complexidade O(n^2): * Loops aninhados onde cada loop executa `n` vezes. * Criação de matriz identidade.

Exemplos de complexidade O(log n): * Divisão sucessiva de um número por 2 (como na contagem de divisões por 2).

1.2. Notação Omega (Ω) - Limite Inferior

Define um limite inferior para o tempo de execução de um algoritmo. Se $f(n) \in \Omega(g(n))$, significa que a taxa de crescimento de $f(n)$ é maior ou igual à de $g(n)$ para valores grandes de n. Formalmente, $f(n) \in \Omega(g(n))$ se existem constantes positivas c e n_0 tais que $0 \le c \cdot g(n) \le f(n)$ para todo $n \ge n_0$.

1.3. Notação Theta (Θ) - Limite Superior e Inferior (Assintoticamente Justo)

Define um limite superior e inferior para o tempo de execução de um algoritmo. Se $f(n) \in \Theta(g(n))$, significa que a taxa de crescimento de $f(n)$ é assintoticamente igual à de $g(n)$. Formalmente, $f(n) \in \Theta(g(n))$ se existem constantes positivas c_1, c_2 e n_0 tais que $0 \le c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n)$ para todo $n \ge n_0$.

Observações importantes: * Um algoritmo com complexidade $O(n^2)$ pode, em certas instâncias, executar em tempo linear. A notação Big O representa o pior caso ou um limite superior. * A presença de loops aninhados não garante automaticamente complexidade quadrática; depende do número de iterações de cada loop.

2. Algoritmos Gulosos

Algoritmos gulosos são uma abordagem para resolver problemas de otimização. Eles fazem a escolha localmente ótima em cada etapa, na esperança de que essa escolha leve a uma solução globalmente ótima. Nem todos os problemas podem ser resolvidos de forma ótima por algoritmos gulosos.

2.1. Problema da Mochila Fracionária

Neste problema, o objetivo é maximizar o valor total de itens que podem ser colocados em uma mochila com capacidade limitada, onde os itens podem ser fracionados. A estratégia gulosa para este problema é escolher os itens com o maior valor por unidade de peso primeiro. Este problema é um bom exemplo onde a abordagem gulosa funciona e leva à solução ótima.

3. Teoria dos Grafos

Grafos são estruturas matemáticas usadas para modelar relações entre objetos. Um grafo G é um par (V, E), onde V é um conjunto de vértices (nós) e E é um conjunto de arestas (conexões).

3.1. Definições Básicas

- **Vértice (Vertex):** Um ponto ou nó no grafo.
- **Aresta (Edge):** Uma conexão entre dois vértices.
- **Ordem do Grafo:** Número de vértices ($|V|$ ou n).
- **Tamanho do Grafo:** Número de arestas ($|E|$ ou m).

- **Extremidades:** Os dois vértices que uma aresta conecta.
- **Adjacência:** Dois vértices são adjacentes se estão conectados por uma aresta. Duas arestas são adjacentes se compartilham um vértice.
- **Vizinhança:** O conjunto de vértices adjacentes a um determinado vértice.
- **Loop:** Uma aresta que conecta um vértice a si mesmo.
- **Arestas Paralelas:** Múltiplas arestas conectando o mesmo par de vértices.
- **Grafo Simples:** Um grafo sem loops e sem arestas paralelas.

3.2. Famílias de Grafos

- **Grafo Completo:** Todos os pares de vértices distintos são adjacentes.
- **Grafo Vazio:** Não possui arestas.
- **Grafo Bipartido:** O conjunto de vértices pode ser dividido em dois subconjuntos X e Y de forma que todas as arestas conectam um vértice em X a um vértice em Y. Não há arestas dentro de X ou dentro de Y.
    - **Estrela:** Um tipo de grafo bipartido onde um vértice central está conectado a todos os outros vértices, e não há conexões entre os outros vértices.
- **Caminho (Path):** Uma sequência de vértices e arestas onde cada vértice é conectado ao próximo na sequência.
- **Ciclo (Cycle):** Um caminho que começa e termina no mesmo vértice, sem repetir vértices ou arestas intermediárias. Um ciclo tem pelo menos 3 vértices.
- **Grafo Conexo:** Existe um caminho entre qualquer par de vértices no grafo.
- **Grafo Planar:** Pode ser desenhado em um plano sem que suas arestas se cruzem (exceto nos vértices).

3.3. Representações de Grafos

- **Matriz de Incidência:** Uma matriz $n \times m$ (vértices x arestas) onde cada entrada indica se um vértice é incidente a uma aresta.
- **Matriz de Adjacência:** Uma matriz $n \times n$ (vértices x vértices) onde cada entrada indica o número de arestas entre dois vértices. Para grafos simples, é 0 ou 1.
- **Lista de Adjacências:** Para cada vértice, uma lista de seus vértices vizinhos. É uma representação eficiente para grafos esparsos (com poucas arestas).

3.4. Algoritmos de Grafos

3.4.1. Algoritmo de Dijkstra

Encontra o caminho mais curto de um vértice de origem para todos os outros vértices em um grafo com pesos de arestas não negativos. Utiliza uma fila de prioridade para selecionar o próximo vértice a ser processado.

3.4.2. Algoritmo de Bellman-Ford

Encontra o caminho mais curto de um vértice de origem para todos os outros vértices em um grafo que pode conter pesos de arestas negativos. Também é capaz de detectar a presença de ciclos de peso negativo, o que o Dijkstra não consegue fazer.

3.4.3. Algoritmos para Árvore Geradora Mínima (MST)

Uma Árvore Geradora Mínima (MST) de um grafo conexo e ponderado é uma subárvore que conecta todos os vértices com o menor custo total possível das arestas.

- **Algoritmo de Kruskal:** Uma abordagem gulosa que constrói a MST adicionando arestas de menor peso que não formam um ciclo. Utiliza a estrutura UNION-FIND para verificar ciclos e gerenciar conjuntos disjuntos de vértices.
- **Algoritmo de Prim:** Constrói a MST adicionando vértices à árvore um por um, começando de um vértice arbitrário e sempre escolhendo a aresta de menor peso que conecta um vértice na árvore a um vértice fora dela. O grafo induzido pelo conjunto de arestas escolhidas pelo algoritmo de Prim é sempre conexo.

4. Heaps e Heapsort

4.1. Heap

Uma heap é uma estrutura de dados baseada em árvore binária que satisfaz a propriedade de heap: para um heap máximo, o valor de cada nó é maior ou igual ao valor de seus filhos. Para um heap mínimo, o valor de cada nó é menor ou igual ao valor de seus filhos. Heaps são frequentemente implementados usando arrays.

4.2. `maxHeapify`

É um procedimento que mantém a propriedade de heap máximo. Dada uma árvore (ou subárvore) e um nó `i`, `maxHeapify` assume que as subárvores esquerda e direita de `i` já são heaps máximos, mas o nó `i` pode ser menor que seus filhos. Ele corrige essa violação trocando `i` com o maior de seus filhos, e recursivamente chamando `maxHeapify` na subárvore afetada.

4.3. `buildMaxHeap`

É um procedimento que constrói um heap máximo a partir de um array não ordenado. Ele faz isso chamando `maxHeapify` em cada nó não folha, começando do último nó não folha e subindo até a raiz. Este processo garante que a propriedade de heap máximo seja estabelecida em toda a árvore.

4.4. Heapsort

Um algoritmo de ordenação que utiliza a estrutura de heap. Ele funciona em duas fases: 1. **Construção do Heap:** Converte o array de entrada em um heap máximo usando `buildMaxHeap`. 2. **Extração e Ordenação:** Remove repetidamente o maior elemento (a raiz do heap) e o coloca no final do array, reorganizando o heap restante. Isso é feito trocando a raiz com o último elemento do heap, diminuindo o tamanho do heap e chamando `maxHeapify` na nova raiz.

4.5. Fila de Prioridade (Priority Queue)

Uma fila de prioridade é uma estrutura de dados abstrata que armazena elementos com prioridades associadas. Ela suporta operações como inserção de um elemento e extração do elemento com a maior (ou menor) prioridade. Heaps são uma implementação eficiente de filas de prioridade.