Guia de Estudo Estratégico: Algoritmos e Estruturas de Dados em Java para a Prova

---

Parte I: Fundamentos Essenciais - As Ferramentas da Sua Caixa

Objetivo

Antes de mergulhar nos algoritmos, é fundamental entender as ferramentas que eles utilizam. Esta seção aborda o "porquê" por trás da escolha de cada estrutura de dados em Java. Compreender suas forças e fraquezas não apenas justifica seu uso em cada algoritmo, mas também solidifica o conhecimento para resolver problemas de forma mais intuitiva e eficiente.

1.1. Desmistificando as Coleções Java: A Ferramenta Certa para o Trabalho Certo

ArrayList vs. LinkedList

A escolha entre ArrayList e LinkedList é uma das decisões mais fundamentais ao se trabalhar com listas em Java, e a diferença reside inteiramente em sua estrutura interna.

- **ArrayList**: Pense em um ArrayList como um "super array" ou um array dinâmico. Internamente, ele usa um array simples para armazenar os elementos.
    - **Ponto Forte**: Seu principal benefício é o acesso extremamente rápido a qualquer elemento através do seu índice. A operação get(i) é executada em tempo constante, ou O(1), porque a localização do elemento pode ser calculada matematicamente a partir do endereço de memória inicial do array. Isso o torna ideal para situações onde a leitura de dados em posições aleatórias é frequente.
    - **Ponto Fraco**: Adicionar ou remover elementos do início ou do meio da lista é uma

operação lenta, com complexidade de tempo O(n). Isso ocorre porque, para manter a contiguidade da memória, todos os elementos subsequentes à posição modificada precisam ser deslocados, um por um. Adicionar ao final é geralmente rápido (amortizado O(1)), mas pode desencadear uma operação de redimensionamento do array interno se a capacidade for excedida.

- **LinkedList**: Um LinkedList é uma cadeia de nós. Cada nó contém o dado em si e dois ponteiros: um para o nó anterior e outro para o próximo (formando uma lista duplamente encadeada).
    - **Ponto Forte**: Sua força reside na manipulação da estrutura. Adicionar ou remover elementos no início ou no final da lista (addFirst, addLast, removeFirst, removeLast) são operações muito rápidas, com complexidade O(1), pois exigem apenas a atualização de alguns ponteiros, sem a necessidade de mover outros elementos.
    - **Ponto Fraco**: O acesso a um elemento em uma posição específica no meio da lista é lento, com complexidade O(n). Para encontrar o elemento no índice i, o algoritmo precisa percorrer a lista a partir do início (ou do fim, o que for mais perto) até chegar à posição desejada.

A Fila (Queue): Por que ArrayDeque é o Campeão de Performance?

A interface Queue representa uma coleção do tipo FIFO (First-In, First-Out), ou "o primeiro a entrar é o primeiro a sair". É a estrutura de dados que define o comportamento da Busca em Largura (BFS).

- **LinkedList como Fila**: Por implementar a interface Queue, LinkedList é uma escolha tradicional e funcional. As operações de enfileirar (add()) e desenfileirar (poll() ou remove()) são eficientes, executadas em tempo O(1).
- **ArrayDeque como Fila**: Esta é a implementação moderna e preferida para uma fila em Java. ArrayDeque significa "Array Double Ended Queue" (fila de duas pontas baseada em array). Embora use um array internamente, ele é implementado como um **array circular**. Isso significa que ele não precisa deslocar elementos ao adicionar ou remover das pontas. As operações de enfileirar e desenfileirar são executadas em tempo **amortizado O(1)**.
- **Causa e Efeito**: A principal vantagem do ArrayDeque sobre o LinkedList na prática é a **localidade de referência**. Como os elementos do ArrayDeque estão em um bloco de memória contíguo, o processador pode carregá-los no cache de forma muito mais eficiente. Em contraste, os nós de um LinkedList podem estar espalhados pela memória, resultando em mais "cache misses" e, consequentemente, menor performance. Para algoritmos como o BFS, onde o volume de operações de enfileirar/desenfileirar é alto e a performance geral é o que importa, o ArrayDeque é quase sempre a escolha mais rápida.

A Pilha (Stack): Por que ArrayDeque Também é o Novo Padrão?

A Pilha (Stack) é uma estrutura LIFO (Last-In, First-Out), ou "o último a entrar é o primeiro a sair". É a estrutura essencial para a versão iterativa da Busca em Profundidade (DFS).

- A classe java.util.Stack é uma implementação antiga e legada. Seu principal problema é que seus métodos são sincronizados, o que adiciona uma sobrecarga de performance desnecessária para aplicações single-threaded.
- **ArrayDeque como Pilha**: A documentação oficial do Java recomenda o uso de ArrayDeque como uma implementação de pilha. Os métodos push() (empilhar) e pop() (desempilhar) correspondem perfeitamente à funcionalidade de uma pilha, com as mesmas vantagens de performance (tempo amortizado O(1) e localidade de referência) vistas em seu uso como fila.

A Fila de Prioridade (PriorityQueue): A Fila "VIP"

Uma PriorityQueue é fundamentalmente diferente de uma Queue normal. Ela não segue a regra FIFO. Em vez disso, ela organiza os elementos com base em sua "prioridade", garantindo que o elemento de maior prioridade seja sempre o primeiro a ser removido.

- **Estrutura Interna**: Em Java, a PriorityQueue é implementada usando uma estrutura de dados chamada **Heap**. Por padrão, é um **Min-Heap**, o que significa que o elemento com o **menor valor** é considerado o de maior prioridade.
- **Operações e Complexidade**: As operações mais importantes são add() (ou offer()) para inserir um elemento e poll() para remover o elemento de maior prioridade. Ambas as operações têm uma complexidade de tempo eficiente de O(logn), onde n é o número de elementos na fila.
- **Aplicação Principal**: É a peça central que torna o algoritmo de Dijkstra eficiente. Dijkstra precisa, a cada passo, encontrar o vértice ainda não visitado que tem a menor distância acumulada da origem. Uma PriorityQueue permite encontrar este vértice em tempo O(logV) em vez de uma busca linear de O(V), otimizando drasticamente o algoritmo.

1.2. Representação de Grafos em Java: A Lista de Adjacência

Para a maioria dos problemas de algoritmos, especialmente em grafos que não são extremamente densos (grafos esparsos), a **lista de adjacência** é a forma de representação mais comum e eficiente.

- **Implementação**: Para grafos onde os vértices são numerados de 0 a N−1, uma lista de adjacência pode ser implementada de forma muito eficaz em Java usando um ArrayList de ArrayLists, como ArrayList<ArrayList<Integer>> para grafos não ponderados ou ArrayList<ArrayList<Aresta>> para grafos ponderados (onde Aresta é uma classe que

armazena o destino e o peso).
- **Funcionamento**: A estrutura adj é um ArrayList onde o índice corresponde ao vértice de origem. Assim, adj.get(u) retorna uma outra lista contendo todos os vizinhos (ou arestas de saída) do vértice u. Esta representação é eficiente em termos de espaço, pois armazena apenas as conexões que realmente existem, e permite iterar sobre os vizinhos de um nó de forma rápida.

Parte II: Algoritmos de Travessia em Grafos

Objetivo

Dominar as duas formas fundamentais de explorar um grafo é o primeiro passo para resolver problemas mais complexos. Esta seção detalha o DFS e o BFS, focando na diferença crucial entre suas estratégias de exploração e como isso se reflete nas estruturas de dados que eles utilizam.

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

- **Conceito**: A estratégia do DFS é "ir o mais fundo possível por um único caminho antes de voltar atrás (backtracking)". A analogia clássica é a de um explorador em um labirinto que sempre segue um caminho até o fim (um beco sem saída ou um ponto já visitado). Somente então ele retorna ao último cruzamento para tentar uma direção diferente.
- **Estrutura de Dados Chave**: A **Pilha (Stack)**. Essa estrutura LIFO (Last-In, First-Out) é o que permite ao algoritmo "lembrar" dos pontos de decisão para onde ele deve retornar. Isso pode ser a pilha de chamadas implícita de uma função recursiva ou uma pilha explícita (como ArrayDeque) em uma implementação iterativa.
- **Implementação Iterativa (Passo a Passo)**:
    1. Crie uma pilha (ex: ArrayDeque<Integer> stack) e um conjunto para os nós visitados (ex: boolean visited).
    2. Empilhe o nó inicial: stack.push(startNode).
    3. Inicie um laço que continua enquanto a pilha não estiver vazia (!stack.isEmpty()).
    4. Dentro do laço, desempilhe um nó: int u = stack.pop().
    5. Verifique se o nó u já foi visitado. Se !visited[u]:
    6. Marque u como visitado: visited[u] = true.
    7. Processe o nó u (por exemplo, imprima seu valor).
    8. Para cada vizinho v do nó u, verifique se ele **não foi visitado**. Se !visited[v], empilhe-o: stack.push(v).

- **O Papel do visited**: O array (ou Set) visited é absolutamente crucial. Em grafos que contêm ciclos, sem um mecanismo para rastrear os nós já visitados, o DFS entraria em um loop infinito, navegando para frente e para trás entre os mesmos nós repetidamente. Ele garante que cada nó seja processado apenas uma vez.
- Teste de Mesa para a Questão da Prova (DFS):
  Vamos aplicar o DFS iterativo ao grafo da Questão 1, partindo do vértice 0.
  - **Grafo (Lista de Adjacência):**
    - 0:
    - 1:
    - 2:
    - 3:
    - 4:
    - 5:
  - **Execução Passo a Passo:**

Passo	Nó Atual ('u')	Estado da Pilha (Topo à direita)	Nós Visitados {0, 1, 2, 3, 4, 5}	Ordem de Visita (Saída)
1	-	``	{F, F, F, F, F, F}	
2	0	->	{**T**, F, F, F, F, F}	0
3	1	->	{T, **T**, F, F, F, F}	0 1
4	3	->	{T, T, F, **T**, F, F}	0 1 3
5	5	->	{T, T, F, T, F, **T**}	0 1 3 5
6	4	->	{T, T, F, T, **T**, T}	0 1 3 5 4
7	2	->	{T, T, **T**, T, T, T}	0 1 3 5 4 2
8	-	``	{T, T, T, T, T, T}	(Fim)

Observação: Na pilha, os vizinhos de um nó são empilhados. A ordem exata (ex: empilhar 2 e depois 1, ou 1 e depois 2) pode variar, alterando a ordem final da travessia. A ordem acima assume que os vizinhos são empilhados na ordem inversa da lista de adjacência para simular a ordem da recursão.
Saída Final: 0 1 3 5 4 2

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

- **Conceito**: A estratégia do BFS é explorar o grafo em "camadas". Primeiro, ele visita o nó inicial. Em seguida, visita todos os vizinhos diretos do nó inicial (camada 1). Depois, visita todos os vizinhos desses vizinhos (camada 2), e assim por diante. A analogia é a de uma onda se expandindo a partir do ponto onde uma pedra foi jogada na água.
- **Estrutura de Dados Chave**: A **Fila (Queue)**. A natureza FIFO (First-In, First-Out) da fila garante que os nós sejam processados na ordem em que foram descobertos, mantendo a exploração camada por camada.
- **Aplicação Principal**: Por sua natureza de exploração em camadas, o BFS é o algoritmo

ideal para encontrar o **caminho mais curto em termos de número de arestas** entre dois nós em um grafo não ponderado.

- **Implementação (Passo a Passo)**:
    1. Crie uma fila (ex: ArrayDeque<Integer> queue) e um conjunto para os nós visitados (ex: boolean visited).
    2. Marque o nó inicial como visitado e enfileire-o: visited[startNode] = true; queue.add(startNode);.
    3. Inicie um laço que continua enquanto a fila não estiver vazia (!queue.isEmpty()).
    4. Dentro do laço, desenfileire um nó: int u = queue.poll().
    5. Processe o nó u (por exemplo, imprima seu valor).
    6. Para cada vizinho v do nó u:
    7. Verifique se v **não foi visitado**. Se !visited[v]:
    8. Marque v como visitado (visited[v] = true;) e enfileire-o (queue.add(v);).
- Teste de Mesa para a Questão da Prova (BFS):
  Vamos aplicar o BFS ao mesmo grafo da Questão 1, partindo do vértice 0.
    - **Grafo (Lista de Adjacência):**
        - 0:
        - 1:
        - 2:
        - 3:
        - 4:
        - 5:
    - **Execução Passo a Passo:**

Passo	Nó Desenfileirado ('u')	Estado da Fila (Frente à esquerda)	Nós Visitados {0, 1, 2, 3, 4, 5}	Ordem de Visita (Saída)
1	-	``	{**T**, F, F, F, F, F}	
2	0	->	{T, **T**, **T**, F, F, F}	0
3	1	->	{T, T, T, **T**, **T**, F}	0 1
4	2	-> (4 já visitado)	{T, T, T, T, T, F}	0 1 2
5	3	->	{T, T, T, T, T, **T**}	0 1 2 3
6	4	-> (5 já visitado)	{T, T, T, T, T, T}	0 1 2 3 4
7	5	``	{T, T, T, T, T, T}	0 1 2 3 4 5
8	-	``	{T, T, T, T, T, T}	(Fim)

Saída Final: 0 1 2 3 4 5

Parte III: Árvores Geradoras Mínimas (MST)

Objetivo

Esta seção é dedicada a desvendar o conceito de Árvore Geradora Mínima e a dissecar o algoritmo de Kruskal, com um foco cirúrgico na implementação KruskalSimples fornecida pelo professor. O entendimento da estrutura de dados UnionFind é o ponto mais crítico aqui.

3.1. Conceitos Fundamentais da MST

- **O que é uma Árvore Geradora (Spanning Tree)?** Dado um grafo conectado e não direcionado, uma árvore geradora é um subconjunto de arestas que conecta todos os vértices do grafo original, usando o número mínimo possível de arestas (V−1, onde V é o número de vértices) e sem formar nenhum ciclo. Um grafo pode ter muitas árvores geradoras diferentes.
- **O que é uma Árvore Geradora Mínima (MST)?** Em um grafo ponderado, cada árvore geradora tem um "custo" total, que é a soma dos pesos de suas arestas. A MST é a árvore geradora que possui o menor custo total possível.
- **Propriedades Importantes (para Questões Teóricas)**:
    - **Unicidade**: Uma MST é **única** se todos os pesos das arestas do grafo forem distintos. Se houver arestas com pesos iguais, podem existir múltiplas MSTs, todas com o mesmo custo total mínimo.
    - **Propriedade do Ciclo**: Para qualquer ciclo no grafo, a aresta com o **maior peso** nesse ciclo **nunca** fará parte de uma MST. Remover essa aresta e manter as outras do ciclo ainda mantém a conectividade, mas com um custo menor. Essa propriedade é a base lógica do algoritmo de Kruskal.
    - **Propriedade do Corte**: Para qualquer "corte" que divida os vértices do grafo em dois conjuntos disjuntos, a aresta de **menor peso** que cruza esse corte (conecta um vértice de um conjunto a um vértice do outro) **deve** fazer parte de toda e qualquer MST. Essa propriedade é a base do algoritmo de Prim.

3.2. Algoritmo de Kruskal: A Estratégia "Gulosa"

O algoritmo de Kruskal é um exemplo clássico de um algoritmo guloso (greedy). Sua lógica é elegantemente simples: construir a MST adicionando, a cada passo, a aresta mais barata disponível que não crie um ciclo com as arestas já selecionadas.

- **Passos do Algoritmo**:
    1. Crie uma lista contendo todas as arestas do grafo.
    2. Ordene essa lista em ordem crescente de peso.
    3. Inicialize uma estrutura de dados **Union-Find** com V conjuntos, um para cada vértice.

4. Itere pela lista de arestas ordenadas. Para cada aresta (u,v):
5. Verifique se os vértices u e v pertencem a conjuntos diferentes (usando a operação find). Se find(u)!= find(v), significa que adicionar a aresta (u,v) não formará um ciclo.
6. Nesse caso, adicione a aresta à MST e una os conjuntos de u e v (usando a operação union).
7. Continue até que a MST tenha V−1 arestas.

3.3. Análise Profunda do Código KruskalSimples

Vamos analisar o código fornecido, peça por peça, pois os detalhes da implementação são cruciais.

Classe Aresta

Java

```
class Aresta implements Comparable<Aresta>{
  int u, v, peso;
  //... construtor...
  @Override
  public int compareTo(Aresta o) {
    if(this.peso < o.peso) return -1;
    if (this.peso == o.peso) return 0;
    return 1;
  }
}
```

- **implements Comparable<Aresta>**: Isso permite que objetos da classe Aresta sejam comparados entre si.
- **compareTo(Aresta o)**: Este método define a "ordem natural" das arestas. Ao retornar -1 se this.peso < o.peso, ele instrui qualquer método de ordenação, como Collections.sort(), a colocar as arestas de menor peso primeiro. É essa implementação que faz a etapa de ordenação de Kruskal funcionar.

Classe UnionFind (A Peça Mais Importante)

Java

```
class UnionFind{
  int representantes;
  public UnionFind(int n){
    representantes = new int[n];
    for (int i = 0; i < n; i++){
      representantes[i] = i;
    }
  }

  public int find(int x){
    while(representantes[x]!= x){
      x = representantes[x];
    }
    return x;
  }

  public void union (int x, int y){
    int rX = find(x);
    int rY = find(y);
    representantes[rY] = rX;
  }
}
```

- **representantes**: Este array é o coração da estrutura. representantes[i] armazena o "pai" do elemento i. No início, representantes[i] = i, o que significa que cada vértice é o representante (a raiz) de seu próprio conjunto.
- **find(int x)**: Esta função determina a qual conjunto o elemento x pertence, encontrando a raiz de sua árvore. O laço while(representantes[x]!= x) simplesmente sobe na hierarquia, seguindo os ponteiros de "pai" até encontrar um elemento que aponta para si mesmo. Esse elemento é a raiz.
- **union(int x, int y)**: Esta função une os conjuntos que contêm x e y. Ela primeiro encontra os representantes de ambos (rX e rY). Em seguida, de forma arbitrária, faz com que a raiz de y aponte para a raiz de x (representantes[rY] = rX), efetivamente fundindo as duas árvores em uma só.
- **A Armadilha Oculta (Análise de Performance)**: A implementação de UnionFind fornecida é a **versão ingênua**. Ela não utiliza duas otimizações cruciais: **path compression** (compressão de caminho) e **union by rank/size** (união por ranking/tamanho).
    - **Sem essas otimizações**, a estrutura de "árvore" formada pelo array representantes pode se degenerar em uma longa cadeia, semelhante a uma lista

encadeada.

- Nesse pior caso, uma única operação find(x) pode precisar percorrer todos os V vértices, resultando em uma complexidade de tempo de O(V). Isso é muito mais lento do que a complexidade quase constante (O(α(V))) das versões otimizadas.
- Para a prova, é vital simular a execução exatamente como o código está escrito, pois o estado do array representantes será diferente (e as árvores serão mais "altas") do que seria com as otimizações.

3.4. Teste de Mesa Detalhado para Kruskal (Questão da Prova)

Vamos aplicar o KruskalSimples ao grafo da **Questão 3**.

- **Vértices**: 6 (0 a 5)
- **Arestas**: (0,1,1), (0,2,5), (1,2,2), (1,3,7), (1,4,6), (2,4,3), (2,5,8), (3,4,4), (4,5,9)
- **MST deve ter**: V−1=5 arestas.

Passo 1: Ordenar as arestas por peso
(0,1,1), (1,2,2), (2,4,3), (3,4,4), (0,2,5), (1,4,6), (1,3,7), (2,5,8), (4,5,9)
Passo 2: Execução do algoritmo

Aresta Ordenada (u,v,peso)	Estado de representantes ANTES	find(u)	find(v)	Ciclo? (find(u)==find(v))	Ação (Adicionar/ Descartar)	Estado de representantes DEPOIS	MST Resultante (Arestas)
Inicialização	``	-	-	-	-	``	{}
(0, 1, 1)	``	0	1	Não	Adicionar, union(0,1)	[0, **0**, 2, 3, 4, 5]	{(0,1,1)}
(1, 2, 2)	``	0	2	Não	Adicionar, union(1,2)	[0, 0, **0**, 3, 4, 5]	{(0,1,1), (1,2,2)}
(2, 4, 3)	``	0	4	Não	Adicionar, union(2,4)	[0, 0, 0, 3, **0**, 5]	{...(2,4,3)}
(3, 4, 4)	``	3	0	Não	Adicionar, union(3,4)	[0, 0, 0, **0**, 0, 5]	{...(3,4,4)}
(0, 2, 5)	``	0	0	**Sim**	Descartar	``	{...(3,4,4)}
(1, 4, 6)	``	0	0	**Sim**	Descartar	``	{...(3,4,4)}
(1, 3, 7)	``	0	0	**Sim**	Descartar	``	{...(3,4,4)}
(2, 5, 8)	``	0	5	Não	Adicionar, union(2,5)	[0, 0, 0, 0, 0, **0**]	{...(2,5,8)}

Fim: A MST tem 5 arestas. Paramos aqui.

- **Arestas da MST**: (0,1,1), (1,2,2), (2,4,3), (3,4,4), (2,5,8)
- **Custo Total da MST**: 1+2+3+4+8=18

Parte IV: Algoritmos de Caminho Mínimo

Objetivo

Esta seção aborda os dois principais algoritmos para encontrar caminhos mais curtos a partir de uma única origem: Dijkstra e Bellman-Ford. O foco é entender suas diferenças fundamentais, restrições (pesos negativos) e os cenários onde cada um deve ser aplicado.

4.1. Algoritmo de Dijkstra

- **Conceito**: O algoritmo de Dijkstra resolve o problema do caminho mais curto de uma única origem (*single-source shortest path*) para todos os outros vértices em um grafo **ponderado cujas arestas têm pesos não-negativos**. É um algoritmo guloso, pois a cada passo ele escolhe o caminho que parece ser o melhor localmente.
- **A Falha com Pesos Negativos**: A abordagem gulosa de Dijkstra é sua maior força e também sua principal fraqueza. O algoritmo mantém um conjunto de nós "finalizados" (settled), para os quais ele assume que a menor distância já foi encontrada. Uma vez que um nó entra nesse conjunto, ele nunca mais é reavaliado. Se o grafo contiver uma aresta de peso negativo, um caminho que passe por ela poderia, mais tarde, revelar uma rota mais curta para um nó já "finalizado", mas Dijkstra não tem como voltar atrás para corrigir essa decisão.
- **O Papel Vital da PriorityQueue**: A eficiência de Dijkstra depende criticamente de sua capacidade de, a cada passo, selecionar o vértice não visitado com a menor distância atualmente conhecida da origem.
    - Uma abordagem ingênua seria percorrer todos os vértices para encontrar esse mínimo, o que levaria tempo O(V).
    - A PriorityQueue (implementada como um Min-Heap) otimiza isso drasticamente. Ela armazena os vértices a serem visitados, priorizados por sua distância. A operação para extrair o vértice com a menor distância (poll()) leva tempo O(logV). Isso reduz a complexidade geral do algoritmo de O(V2) para O(ElogV), que é muito mais rápido para grafos esparsos.
- Teste de Mesa para Dijkstra (Questão da Prova):
  Vamos aplicar Dijkstra ao grafo da Questão 2, partindo do vértice 0.
    - **Estruturas de Dados**:
        - dist: Array para armazenar a menor distância conhecida da origem até cada vértice. Inicializa com dist = 0 e dist[i] = ∞ para os demais.
        - pred: Array para armazenar o predecessor de cada vértice no caminho mais

curto.

- pq: Fila de prioridade (PriorityQueue) para armazenar pares {distância, vértice}.

Iter.	Nó 'u' Extraído da PQ	dist {0,1,2,3,4}	pred {0,1,2,3,4}	Vizinhos de 'u'	Ação de Relaxamento (dist[v] > dist[u]+w)	Estado da PQ após relaxar
0	-	{0, ∞, ∞, ∞, ∞}	{-1,-1,-1,-1,-1}	-	-	{{0,0}}
1	**0** (dist=0)	{0, 10, 3, ∞, ∞}	{-, 0, 0, -, -}	1(10), 2(3)	dist=10, dist=3	{{3,2}, {10,1}}
2	**2** (dist=3)	{0, 8, 3, 11, 5}	{-, 2, 0, 2, 2}	1(5), 3(8), 4(2)	dist=8, dist=11, dist=5	{{5,4}, {8,1}, {10,1}, {11,3}}
3	**4** (dist=5)	{0, 8, 3, 10, 5}	{-, 2, 0, 4, 2}	3(5)	dist=10	{{8,1}, {10,1}, {10,3}}
4	**1** (dist=8)	{0, 8, 3, 10, 5}	{-, 2, 0, 4, 2}	3(2)	8+2=10, não menor que dist=10	{{10,1}, {10,3}}
5	**3** (dist=10)	{0, 8, 3, 10, 5}	{-, 2, 0, 4, 2}	-	-	{{10,1}}

Observação: A PriorityQueue pode conter entradas duplicadas para o mesmo vértice com distâncias diferentes. O algoritmo sempre processará a de menor distância primeiro, e as outras serão ignoradas quando extraídas, pois o vértice já terá sido "finalizado".

Resultado Final:

- **Distâncias Mínimas a partir de 0**: dist = {0: 0, 1: 8, 2: 3, 3: 10, 4: 5}
- **Caminhos**:
  - 0 -> 1: 0 -> 2 -> 1 (Custo 3+5=8)
  - 0 -> 2: 0 -> 2 (Custo 3)
  - 0 -> 3: 0 -> 2 -> 4 -> 3 (Custo 3+2+5=10)
  - 0 -> 4: 0 -> 2 -> 4 (Custo 3+2=5)

4.2. Algoritmo de Bellman-Ford

- **Conceito**: Bellman-Ford também resolve o problema do caminho mais curto de uma única origem, mas com uma vantagem crucial: ele funciona corretamente mesmo que o grafo tenha **arestas com pesos negativos**. Sua robustez, no entanto, vem ao custo de uma performance inferior à de Dijkstra.
- **O Mecanismo de Relaxamento**: O núcleo do algoritmo é surpreendentemente simples. Ele repete um processo de "relaxamento" para **todas as arestas** do grafo. Esse

processo é executado V−1 vezes.
    - Em cada uma das V−1 iterações, o algoritmo percorre cada aresta (u,v) com peso w e verifica se um caminho mais curto para v pode ser obtido passando por u. Ou seja, ele testa se dist[v] > dist[u] + w. Se for verdade, ele atualiza dist[v].
    - A lógica é que o caminho mais curto simples (sem ciclos) em um grafo com V vértices pode ter no máximo V−1 arestas. Após a primeira iteração, o algoritmo encontra todos os caminhos mais curtos de comprimento 1. Após a segunda, todos os de comprimento até 2, e assim por diante. Após V−1 iterações, ele garante ter encontrado todos os caminhos mais curtos simples.
- **Detecção de Ciclos Negativos**: A genialidade do Bellman-Ford está em sua capacidade de detectar ciclos de peso negativo. Após as V−1 iterações, o algoritmo realiza **uma iteração adicional (a V-ésima)**.
    - Se, nesta iteração final, ainda for possível relaxar qualquer aresta (ou seja, se dist[v] for atualizado para algum v), isso prova inequivocamente a existência de um ciclo de peso negativo que é alcançável a partir da origem. Isso ocorre porque as distâncias dos caminhos simples já deveriam ter convergido. Uma melhoria adicional só é possível se o algoritmo puder "dar voltas" em um ciclo que reduz o custo total indefinidamente.
- Teste de Mesa para Bellman-Ford (Questão da Prova):
  Vamos aplicar Bellman-Ford ao grafo da Questão 4, partindo do vértice 0. O grafo tem 5 vértices (V=5), então faremos 4 iterações de relaxamento e uma 5ª para verificação.
    - **Estruturas de Dados**: dist inicializado com {0, ∞, ∞, ∞, ∞}.
    - **Tabela de Execução**: As colunas são os vértices e as linhas são as iterações. Cada célula (i, j) mostra o valor de dist[j] no final da iteração i.

Iteração (i)	dist	dist	dist	dist	dist	Observações da Iteração
0 (Inicial)	0	∞	∞	∞	∞	Distâncias iniciais.
1	0	6	7	∞	∞	Relaxou (0,1) e (0,2).
2	0	2	7	4	5	Relaxou (2,1), (2,3), (2,4) e (1,4).
3	0	2	7	-2	1	Relaxou (4,3) e (1,4). dist ficou -2 via 0->2->1->4->3. dist ficou 1 via 0->2->1->4.
4	0	2	7	-2	1	Nenhuma

						distância foi alterada. O algoritmo poderia parar aqui.
5 (Verif.)	0	2	7	-2	1	Nenhuma distância foi alterada. **Conclusão: Não há ciclos negativos.**

Resultado Final:

- **Distâncias Mínimas a partir de 0**: dist = {0: 0, 1: 2, 2: 7, 3: -2, 4: 1}

Parte V: Ordenação com HeapSort

Objetivo

Esta seção final foca em desmistificar a implementação de HeapSort fornecida pelo professor. A análise detalhada do código é crucial, especialmente para entender a peculiaridade da indexação baseada em 1, que é uma fonte comum de confusão quando se trabalha com arrays baseados em 0 em Java.

5.1. Desvendando o HeapSort do Professor

Conceitos

- **O que é um Max-Heap?** Um Max-Heap é uma árvore binária com duas propriedades principais:
    1. É uma árvore binária **quase completa**, o que significa que todos os níveis estão preenchidos, exceto possivelmente o último, que é preenchido da esquerda para a direita.
    2. Satisfaz a **propriedade do heap**: o valor de cada nó pai é sempre maior ou igual aos valores de seus filhos. Uma consequência direta é que o maior elemento de toda a coleção está sempre na raiz da árvore.

- **Representação em Array**: A beleza de um heap é que, por ser uma árvore quase completa, ele pode ser representado de forma compacta em um array, sem a necessidade de ponteiros explícitos. As relações pai-filho são calculadas matematicamente a partir dos índices.

Análise do Código (A Peça Mais Importante)

Java

```
public class HeapSort {
  public static int parent (int i){ return i / 2; }
  public static int left(int i){ return 2 * i; }
  public static int right(int i){ return 2 * i + 1; }
  //... swap...
  public static void maxHeapify(int A, int i, int n){ /*... */ }
  public static void buildMaxHeap(int A, int n){
    for(int i = n / 2; i >= 1; i--){
      maxHeapify(A, i, n);
    }
  }
  public static void heapSort(int A){
    int n = A.length - 1;
    buildMaxHeap(A, n);
    for(int i = n; i >= 2; i--){
      swap(A, 1 , i); // Note: swap(A, 1, n) in the original code has a bug, it should be swap(A, 1, i)
      n--;
      maxHeapify(A, 1, n);
    }
  }
}
```

- **A Armadilha da Indexação 1-based**: As fórmulas parent(i)=i/2, left(i)=2*i, e right(i)=2*i+1 são as fórmulas padrão para um heap armazenado em um array com **índice começando em 1**. O código Java, no entanto, usa arrays com índice 0. A implementação do professor contorna isso de uma maneira específica:
    1. int n = A.length - 1;: Ele define n como o último índice válido, assumindo que o array tem um elemento "fantasma" no índice 0 que não será usado, ou que os dados relevantes estão de A a A[n].
    2. Os laços em buildMaxHeap (i de n/2 até 1) e heapSort (i de n até 2) operam estritamente dentro da faixa de índices de 1 a n.

3. **Conclusão Crítica**: Para todos os efeitos práticos, este algoritmo **ignora o elemento no índice 0 do array** e trata o subarray A[1...n] como um heap.

- **maxHeapify(A, i, n)**: Esta é a rotina de "conserto". Ela assume que as sub-árvores cujas raízes são os filhos de i (left(i) e right(i)) **já são Max-Heaps**. Sua única função é pegar o valor em A[i] e "afundá-lo" (sift-down) na árvore até que ele encontre sua posição correta, restaurando a propriedade do Max-Heap para a árvore com raiz em i. O parâmetro n define o tamanho atual do heap, que diminui durante a fase de ordenação.
- **buildMaxHeap(A, n)**: Esta função transforma um array desordenado em um Max-Heap. A lógica de começar o laço em i = n / 2 e ir até 1 é a parte mais importante.
  - O nó no índice n/2 é o **último nó que não é uma folha**.
  - Como todas as folhas (da segunda metade do array) já são, por definição, Max-Heaps de um único elemento, o algoritmo começa a "consertar" os heaps de baixo para cima.
  - Ao chamar maxHeapify para i = n/2, depois n/2 - 1, e assim por diante até 1, ele garante que a pré-condição do maxHeapify (de que as sub-árvores já são heaps) seja sempre satisfeita.
- **heapSort(A)**: Esta é a fase de ordenação propriamente dita. O laço principal executa o seguinte ciclo repetidamente:
  1. **Troca**: swap(A, 1, i) troca o maior elemento do heap atual (que está em A) com o último elemento do heap (A[i]). Isso coloca o maior elemento em sua posição final correta no array.
  2. **Reduz o Heap**: O tamanho do heap é efetivamente reduzido em um (n-- no código original, ou implicitamente pelo limite superior do laço i--). O elemento que acabamos de mover para o final agora é considerado parte da porção ordenada do array.
  3. **Conserta o Heap**: O novo valor na raiz (A, que veio da troca) provavelmente violou a propriedade do Max-Heap. A chamada maxHeapify(A, 1, n) é feita para consertar o heap, garantindo que o novo maior elemento suba para a raiz, pronto para a próxima iteração.

5.2. Teste de Mesa Detalhado para buildMaxHeap (Questão da Prova)

Vamos aplicar buildMaxHeap ao array da **Questão 5**: A = {_, 4, 1, 3, 2, 16, 9, 10, 14, 8, 7}. (O _ denota o índice 0 ignorado).

- n = 10. O laço buildMaxHeap vai de i = n/2 = 5 até 1.

Chamada (i)	Array ANTES de maxHeapify(A, i, 10)	Detalhes da Execução de maxHeapify(i)	Array DEPOIS de maxHeapify(A, i, 10)
i = 5	[_, 4, 1, 3, 2, 16, 9, 10, 14, 8, 7]	A=16. Filhos: A=7. 16 > 7. Nenhuma troca.	[_, 4, 1, 3, 2, 16, 9, 10, 14, 8, 7]

i = 4	[_, 4, 1, 3, 2, 16, 9, 10, 14, 8, 7]	A=2. Filhos: A=14, A=8. largest=8. Troca A com A.	[_, 4, 1, 3, 14, 16, 9, 10, 2, 8, 7]
i = 3	[_, 4, 1, 3, 14, 16, 9, 10, 2, 8, 7]	A=3. Filhos: A=9, A=10. largest=7. Troca A com A.	[_, 4, 1, 10, 14, 16, 9, 3, 2, 8, 7]
i = 2	[_, 4, 1, 10, 14, 16, 9, 3, 2, 8, 7]	A=1. Filhos: A=14, A=16. largest=5. Troca A com A.	[_, 4, 16, 10, 14, 1, 9, 3, 2, 8, 7]
i = 1	[_, 4, 16, 10, 14, 1, 9, 3, 2, 8, 7]	A=4. Filhos: A=16, A=10. largest=2. Troca A com A. maxHeapify(A,2,10) recursivo: A=4. Filhos: A=14, A=1. largest=4. Troca A com A.	[_, 16, 14, 10, 8, 7, 9, 3, 2, 4, 1]

Array após buildMaxHeap: {_, 16, 14, 10, 8, 7, 9, 3, 2, 4, 1}

5.3. Teste de Mesa Detalhado para a fase heapSort

Continuando com o array resultante do passo anterior. O laço de heapSort vai de i = 10 até 2.

Iter. (i)	Array ANTES da Troca	swap(A, A[i])	Array APÓS a Troca	Tamanho do Heap (n)	maxHeapify(A, 1, n) (Resultado)
i = 10	[_, 16, 14, 10, 8, 7, 9, 3, 2, 4, 1]	swap(16, 1)	[_, 1, 14, 10, 8, 7, 9, 3, 2, 4, **16**]	9	[_, 14, 8, 10, 4, 7, 9, 3, 2, 1, **16**]
i = 9	[_, 14, 8, 10, 4, 7, 9, 3, 2, 1, **16**]	swap(14, 1)	[_, 1, 8, 10, 4, 7, 9, 3, 2, **14**, **16**]	8	[_, 10, 8, 9, 4, 7, 1, 3, 2, **14**, **16**]
i = 8	[_, 10, 8, 9, 4, 7, 1, 3, 2, **14**, **16**]	swap(10, 2)	[_, 2, 8, 9, 4, 7, 1, 3, **10**, **14**, **16**]	7	[_, 9, 8, 3, 4, 7, 1, 2, **10**, **14**, **16**]
...	...	...	...	...	...

O processo continua até i = 2. Ao final, o array (ignorando o índice 0) estará ordenado: {_, 1, 2, 3, 4, 7, 8, 9, 10, 14, 16}.

Parte VI: Conclusão - Estratégias Finais para a Prova

Resumo Rápido

Algoritmo	Complexidade de Tempo (Pior Caso)	Lida com Pesos Negativos?	Caso de Uso Principal	Estrutura de Dados Chave
DFS	O(V+E)	N/A	Travessia, detecção de ciclo, topologia	Stack (Pilha)
BFS	O(V+E)	N/A	Caminho mais curto (não ponderado)	Queue (Fila)
Kruskal	O(ElogE) ou O(ElogV)	Sim	Árvore Geradora Mínima (MST)	UnionFind
Dijkstra	O(ElogV)	**Não**	Caminho mais curto (pesos não-neg.)	PriorityQueue
Bellman-Ford	O(V·E)	**Sim**	Caminho mais curto (detecta ciclos neg.)	Array de distâncias
HeapSort	O(nlogn)	N/A	Ordenação "in-place" garantida	Array (como um Heap)

Guia de Decisão: "Qual algoritmo usar quando...?"

- **"Preciso apenas visitar todos os nós de um grafo, e a ordem não importa tanto?"**
  - Use **DFS** ou **BFS**. DFS é geralmente mais simples de implementar recursivamente.
- **"Preciso encontrar o caminho mais curto entre duas cidades em um mapa, onde a distância é o número de paradas?"**
  - Grafo não ponderado. Use **BFS**.
- **"Preciso encontrar a rota de carro mais rápida (menor tempo/distância) entre duas cidades em um mapa de estradas?"**
  - Grafo ponderado com pesos não-negativos. Use **Dijkstra**.
- **"Preciso encontrar o caminho mais lucrativo em uma rede de transações financeiras, onde algumas transações podem representar um custo (peso negativo)?"**
  - Grafo ponderado com possíveis pesos negativos. Use **Bellman-Ford**. Ele também dirá se existe uma oportunidade de lucro infinito (ciclo negativo).
- **"Preciso planejar uma rede de fibra ótica para conectar todos os bairros de uma**

cidade usando a menor quantidade total de cabo?"
    - Problema de custo mínimo para conectar todos os pontos. Use **Kruskal** (ou Prim) para encontrar a MST.
- **"Preciso ordenar uma grande lista de números no próprio local (sem usar memória extra significativa) e preciso de uma garantia de que a ordenação não será quadrática no pior caso?"**
    - Ordenação in-place com complexidade garantida. Use **HeapSort**.

Palavras Finais de Encorajamento

Você chegou ao final deste guia detalhado. A chave para a prova não é apenas memorizar os algoritmos, mas entender a **lógica** por trás deles, o **porquê** de cada estrutura de dados e como **simular sua execução** passo a passo. Os testes de mesa são sua ferramenta mais poderosa para solidificar esse conhecimento. Respire fundo, refaça os testes de mesa com os exemplos da prova e confie no processo. Você está mais preparado do que imagina. Boa sorte!

Works cited

1. Breadth First Search or BFS for a Graph - GeeksforGeeks, accessed June 15, 2025, https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/
2. Minimum Spanning Tree - Kruskal - Algorithms for Competitive ..., accessed June 15, 2025, https://cp-algorithms.com/graph/mst_kruskal.html
3. Union By Rank and Path Compression in Union-Find Algorithm ..., accessed June 15, 2025, https://www.geeksforgeeks.org/union-by-rank-and-path-compression-in-union-find-algorithm/
4. Java Program to Implement Bellman Ford Algorithm | GeeksforGeeks, accessed June 15, 2025, https://www.geeksforgeeks.org/java-program-to-implement-bellman-ford-algorithm/