ATIVIDADE EM SALA

OKORIE EMERSON EMMANUEL

1. O que significa complexidade de tempo e complexidade de espaço em algoritmos?

A complexidade de tempo de um algoritmo mede o número de operações que ele executa em função do tamanho da entrada. Em outras palavras, ela indica quanto tempo o algoritmo levará para ser executado à medida que a quantidade de dados aumenta. Já a complexidade de espaço está relacionada à quantidade de memória adicional que o algoritmo precisa, além da memória utilizada para armazenar os dados de entrada. Por exemplo, um algoritmo que percorre uma lista de tamanho n tem complexidade de tempo O(n), porque precisa visitar cada elemento, mas pode ter complexidade de espaço O(1) caso não utilize memória extra além da lista original.

2. Por que a análise de complexidade é importante na escolha de algoritmos para grandes volumes de dados?

A análise de complexidade é importante porque permite prever como o desempenho de um algoritmo se comporta quando o volume de dados cresce. Um algoritmo que funciona de forma aceitável com mil elementos pode tornar-se completamente impraticável com um milhão, caso sua complexidade seja alta, como $O(n^2)$ ou pior. Portanto, ao lidar com grandes volumes de dados, a escolha de algoritmos com complexidade mais baixa garante eficiência, menor consumo de recursos e resultados em tempo viável.

3. Diferencie os conceitos de melhor caso, pior caso e caso médio em relação à execução de algoritmos.

O melhor caso ocorre quando o algoritmo encontra a solução rapidamente, geralmente em situações favoráveis. Por exemplo, na busca sequencial, o melhor caso é quando o elemento procurado está logo na primeira posição da lista, o que leva a uma execução de tempo

constante, O(1). O pior caso, por sua vez, representa a situação mais desfavorável, como quando o elemento está na última posição ou nem mesmo está presente, o que obriga o algoritmo a percorrer toda a lista, resultando em O(n). O caso médio considera a probabilidade de todas as entradas possíveis e fornece uma estimativa do desempenho típico esperado, como na busca sequencial, em que o item geralmente será encontrado por volta do meio da lista, correspondendo a O(n) simplificado.

4. O que representa a notação O-grande (Big-O)? Dê um exemplo prático.

A notação Big-O representa a ordem de crescimento da complexidade de um algoritmo, ou seja, descreve como o tempo ou o espaço necessários aumentam conforme o tamanho da entrada cresce. Ela abstrai detalhes específicos e foca no comportamento assintótico do algoritmo. Por exemplo, se uma busca sequencial precisa verificar cada um dos n elementos de uma lista no pior caso, dizemos que sua complexidade é O(n), o que significa que o tempo de execução cresce linearmente com o tamanho da entrada.

5. Compare a complexidade O(1) com O(n). Em que tipo de problema cada uma aparece?

Um algoritmo com complexidade O(1) é executado em tempo constante, independentemente do tamanho da entrada. Um exemplo é o acesso direto a um elemento de um array pelo índice, já que a operação não depende da quantidade de elementos armazenados. Já a complexidade O(n) aparece em algoritmos cujo tempo cresce proporcionalmente ao tamanho da entrada. Um exemplo clássico é a soma de todos os elementos de uma lista: se a lista tiver n elementos, o algoritmo precisará visitar cada um deles, resultando em tempo linear.

6. Dê um exemplo de algoritmo com complexidade $O(n^2)$ e explique por que ele possui esse comportamento.

Um exemplo típico é o BubbleSort, que ordena uma lista comparando cada elemento com todos os outros e realizando trocas quando necessário. O algoritmo utiliza dois laços aninhados: o primeiro percorre todos os elementos e o segundo compara cada elemento com os

seguintes. Isso resulta em aproximadamente $n \times n$ operações, o que explica sua complexidade quadrática $O(n^2)$. Essa característica faz com que o BubbleSort seja ineficiente para listas grandes.

7. Um algoritmo com complexidade O(n log n) pode ser mais eficiente que um com complexidade $O(n^2)$? Justifique com exemplos.

Sim, algoritmos O(n log n) tendem a ser muito mais eficientes que algoritmos $O(n^2)$ à medida que o tamanho da entrada cresce. Por exemplo, o MergeSort, que possui complexidade O(n log n), é significativamente mais rápido que o BubbleSort, de complexidade $O(n^2)$, quando aplicado a listas grandes. Embora para entradas pequenas a diferença possa não ser perceptível, em entradas grandes, a diferença entre n log n e n^2 é enorme, tornando os algoritmos O(n log n) muito mais viáveis.

8. Por que algoritmos de ordenação como o MergeSort e QuickSort são considerados eficientes em relação ao BubbleSort?

O MergeSort e o QuickSort são considerados eficientes porque utilizam a técnica de divisão e conquista. Eles dividem a lista em partes menores, ordenam cada parte separadamente e depois combinam os resultados. Essa estratégia reduz drasticamente o número de comparações necessárias. Já o BubbleSort realiza comparações repetitivas e trocas múltiplas entre elementos, o que o torna muito menos eficiente. Assim, para listas grandes, MergeSort e QuickSort oferecem desempenho superior.

9. Cite um exemplo de problema em que um algoritmo de complexidade linear (O(n)) seria suficiente e eficiente.

Um exemplo simples é verificar se um número específico está presente em uma lista desordenada. Nesse caso, o algoritmo precisa percorrer os elementos um por um até encontrar o número ou chegar ao fim da lista. Como cada elemento precisa ser verificado no pior caso, a complexidade é linear, O(n), e suficiente para resolver o problema de maneira eficiente.

10. Descreva uma situação prática em que a complexidade O(log n) aparece.

A complexidade O(log n) aparece, por exemplo, na busca binária em uma lista ordenada. Se uma empresa possui uma base de dados de clientes ordenada alfabeticamente, localizar um cliente pelo nome pode ser feito cortando a busca ao meio a cada passo. Isso faz com que, em vez de analisar todos os elementos, apenas um número muito pequeno de comparações seja necessário, proporcional ao logaritmo do tamanho da lista.

11. Um algoritmo de busca binária tem complexidade O(log n). Explique por que ele é mais eficiente que a busca sequencial O(n).

A busca binária é mais eficiente porque reduz o espaço de busca pela metade a cada passo, enquanto a busca sequencial precisa verificar elemento por elemento. Por exemplo, em uma lista de um milhão de elementos, a busca sequencial pode precisar realizar até um milhão de comparações no pior caso, enquanto a busca binária encontrará o elemento em cerca de 20 comparações. Essa diferença torna a busca binária muito mais eficiente para grandes volumes de dados.

12. Compare a eficiência entre algoritmos de força bruta e algoritmos que utilizam técnicas de divisão e conquista.

Os algoritmos de força bruta testam todas as possibilidades possíveis até encontrar uma solução, o que costuma resultar em alto custo computacional. Em contrapartida, os algoritmos baseados em divisão e conquista resolvem problemas grandes dividindo-os em subproblemas menores e combinando os resultados. Por exemplo, o BubbleSort (força bruta) é muito menos eficiente que o MergeSort (divisão e conquista), especialmente para entradas grandes. A diferença de eficiência está justamente em evitar cálculos repetitivos e reduzir o número de operações.

13. Por que o estudo de algoritmos com complexidade exponencial (O(2^n)) é importante, mesmo sendo inviáveis na prática?

O estudo desses algoritmos é importante porque muitos problemas reais, como o Caixeiro Viajante ou a satisfatibilidade booleana (SAT), pertencem a essa categoria. Embora sejam inviáveis para grandes entradas, compreender sua complexidade ajuda a definir os limites do que é computacionalmente possível, além de estimular a busca por soluções aproximadas, heurísticas ou algoritmos mais eficientes. Assim, mesmo que não sejam usados diretamente, eles têm papel fundamental na teoria da computação.

14. Cite um exemplo de problema de complexidade $O(n^3)$ e explique por que seu desempenho é crítico em grandes entradas.

A multiplicação de matrizes pelo método tradicional é um exemplo clássico de algoritmo $O(n^3)$. Nesse processo, cada elemento da matriz resultante exige a multiplicação e soma de uma linha por uma coluna, o que envolve três laços aninhados. Para matrizes pequenas, isso pode ser aceitável, mas para matrizes de grande dimensão, como em aplicações científicas ou gráficas, o custo computacional torna-se extremamente alto, exigindo algoritmos mais sofisticados.

15. O que significa dizer que um algoritmo é escalável em termos de complexidade?

Dizer que um algoritmo é escalável significa que ele mantém um desempenho aceitável à medida que o tamanho da entrada cresce. Em outras palavras, o tempo de execução não cresce de forma descontrolada. Por exemplo, um algoritmo de complexidade $O(n \log n)$, como o MergeSort, pode lidar com milhões de dados de forma eficiente, enquanto um algoritmo $O(n^2)$, como o BubbleSort, rapidamente se torna inviável para o mesmo volume.

16. Como a escolha de uma estrutura de dados pode influenciar na complexidade de um algoritmo? Dê exemplos.

A escolha da estrutura de dados influencia diretamente no desempenho de operações. Por exemplo, procurar um elemento em um array não ordenado tem complexidade O(n), enquanto procurar em uma tabela

hash pode ser feito em tempo O(1), ou seja, constante. Da mesma forma, a inserção em uma árvore binária balanceada, como uma árvore AVL, tem complexidade O(log n), o que é muito mais eficiente do que inserir em uma lista encadeada ordenada, que pode custar O(n). Portanto, a estrutura de dados escolhida pode transformar um algoritmo ineficiente em um muito mais rápido.

17. Explique como a análise assintótica ajuda a comparar algoritmos sem depender do hardware utilizado.

A análise assintótica foca apenas no crescimento da função de complexidade, abstraindo fatores como velocidade do processador ou quantidade de memória da máquina. Dessa forma, conseguimos comparar algoritmos de maneira independente do hardware. Por exemplo, mesmo que um computador seja mais rápido que outro, um algoritmo O(n log n) sempre será mais eficiente que um $O(n^2)$ para entradas grandes, independentemente da máquina utilizada.

18. Dê um exemplo de algoritmo que possui baixa complexidade no melhor caso, mas alta no pior caso.

Um exemplo é o QuickSort. No melhor caso, quando os pivôs são escolhidos de forma equilibrada, o QuickSort funciona em tempo O(n log n). No entanto, no pior caso, quando os pivôs são sempre os piores possíveis (por exemplo, sempre o maior ou o menor elemento), ele se degrada para $O(n^2)$. Essa diferença mostra a importância de considerar os diferentes cenários de execução.

19. Qual a importância de equilibrar legibilidade/manutenção do código e eficiência algorítmica em projetos reais?

Em projetos reais, não basta que um algoritmo seja eficiente do ponto de vista teórico; ele também precisa ser legível, de fácil manutenção e compreensível pela equipe de desenvolvimento. Um código extremamente otimizado, mas complexo demais, pode se tornar difícil de corrigir, adaptar ou expandir no futuro. Por outro lado, um código simples mas ineficiente pode comprometer o desempenho do sistema. Portanto, o equilíbrio entre clareza e eficiência é essencial para garantir tanto a qualidade técnica quanto a sustentabilidade do projeto no longo prazo.