PUC Minas

Lista 8 – Técnicas de projeto de algoritmos

INFORMAÇÕES DOCENTE						
CURSO: ENGENHARIA DE SOFTWARE	**DISCIPLINA:** FUNDAMENTOS DE PROJETO E ANÁLISE DE ALGORITMOS	**TURNO**	**MANHÃ**	**TARDE**	**NOITE**	**PERÍODO/SALA:** 5º
					x	
PROFESSOR (A): João Paulo Carneiro Aramuni						

Lista 8 - Gabarito

Força bruta e pesquisa exaustiva

1) Dado um problema de agendamento de reuniões entre n pessoas com diferentes disponibilidades, por que a aplicação direta de força bruta pode ser inviável em larga escala? Proponha uma alternativa e justifique a escolha.

Resposta: Força bruta exigiria avaliar todas as combinações possíveis de horários para todos os participantes — algo com complexidade exponencial. Isso se torna impraticável conforme n cresce. Uma alternativa é aplicar algoritmos de busca com restrições ou heurísticas, que reduzem o espaço de busca usando informações do domínio. Essas abordagens são mais escaláveis e realistas em ambientes corporativos.

2) Em que cenários o uso de força bruta pode ser preferível mesmo em projetos de engenharia de software profissionais?

Resposta: Quando o espaço de busca é pequeno, o tempo de execução é aceitável e o custo de implementação de um algoritmo sofisticado não se justifica. Ex: protótipos, testes automatizados para validar todos os caminhos possíveis, ou em situações onde a corretude absoluta é mais importante que a eficiência (ex: validação de segurança).

3) Compare a viabilidade de força bruta na resolução de um quebra-cabeça como Sudoku com a abordagem de backtracking. Quais são os trade-offs envolvidos?

Resposta: Força bruta tenta todas as combinações sem usar as regras do Sudoku para filtrar possibilidades, o que é extremamente lento. Backtracking aplica poda com base nas restrições do tabuleiro, o que o torna muito mais eficiente. O trade-off é entre simples implementação vs. eficiência computacional. Backtracking exige mais lógica, mas evita milhões de tentativas inúteis.

Redução e transformação

1) Um engenheiro de software está projetando um sistema de detecção de fraudes em transações financeiras. Ele propõe reduzir o problema a um grafo, onde cada transação é um nó, e arestas representam semelhanças suspeitas (mesmo IP, horário, valor etc.). O objetivo é identificar subconjuntos densos de transações potencialmente fraudulentas.

Explique como essa transformação pode beneficiar a análise e quais técnicas de grafos podem ser aplicadas. Quais cuidados devem ser tomados ao realizar essa modelagem?

Resposta: Essa transformação permite usar algoritmos de teoria dos grafos, como detecção de comunidades, componentes fortemente conexas, ou busca por cliques e subgrafos densos, que são bem estudados em problemas de redes sociais e biológicas. A vantagem está em utilizar ferramentas já otimizadas para detectar padrões estruturais. Entretanto, a modelagem deve garantir que os critérios para criar as arestas sejam bem calibrados — conexões demais podem gerar ruído, enquanto conexões de menos podem omitir fraudes reais. Outro cuidado é a escalabilidade, já que grafos grandes podem gerar alto custo computacional. A redução só é eficaz se o grafo representar fielmente os padrões relevantes de comportamento.

2) Em algoritmos de recomendação, problemas complexos de seleção de conteúdo podem ser reduzidos a problemas de maximização de cobertura ou de mochila. Explique os cuidados e decisões que um engenheiro deve tomar ao escolher a transformação adequada. Cite um possível risco associado.

Resposta: É importante garantir que os critérios de recomendação (ex: diversidade, relevância, custo computacional) estejam preservados na formulação reduzida. Se o modelo de mochila for escolhido, o engenheiro deve definir claramente pesos (ex: tempo de atenção do usuário) e valores (ex: interesse esperado). Um risco é super simplificar a lógica de recomendação, ignorando aspectos importantes como temporalidade ou contexto do usuário. A transformação deve ser representativa, e não apenas conveniente.

3) Você foi encarregado de otimizar um processo de empacotamento de arquivos para backup. Ao analisar o problema, percebe que ele pode ser reduzido ao problema de bin packing. Como essa redução pode orientar suas decisões de implementação, e quais vantagens ela oferece em relação a tentar resolver o problema diretamente?

Resposta: A redução ao bin packing permite aplicar algoritmos bem estudados (First Fit, Best Fit, FFD) e heurísticas que têm garantias de desempenho. Isso acelera o desenvolvimento, pois se evita “reinventar a roda” e ainda se ganha acesso a algoritmos já otimizados e analisados. Além disso, a literatura sobre bin packing ajuda a estimar limites inferiores de eficiência e avaliar a qualidade da solução. A principal vantagem está na reusabilidade de soluções existentes para problemas equivalentes.

Divisão e conquista

1) Considere o problema de multiplicação de duas matrizes muito grandes. Por que a abordagem de divisão e conquista (como Strassen) pode ser mais eficiente do que a multiplicação tradicional?

Resposta: A multiplicação tradicional tem complexidade $O(n^3)$, enquanto o algoritmo de Strassen reduz esse tempo para aproximadamente $O(n^{2.81})$ ao dividir as matrizes em

submatrizes e combinar os resultados com menos multiplicações. Isso é vantajoso em matrizes grandes, onde o custo computacional se torna significativo.

2) Um aluno propôs usar divisão e conquista para resolver a contagem de palavras em um texto massivo (100GB). Isso faz sentido? Justifique.

Resposta: Sim, desde que a divisão seja feita de forma que não quebre palavras no meio, pode-se contar palavras em blocos separados e somar os resultados. Isso é paralelizável e melhora o desempenho em sistemas distribuídos. Porém, cuidado deve ser tomado nos limites das divisões para manter a integridade semântica do texto.

3) Ao implementar um algoritmo baseado em divisão e conquista, quais aspectos de desempenho devem ser analisados além da complexidade assintótica?

Resposta: Deve-se considerar o overhead das chamadas recursivas, custos de junção das subsoluções, uso de memória adicional e até caches da CPU (localidade de referência). Em alguns casos, algoritmos iterativos ou híbridos podem superar divisão e conquista por conta desses fatores.

Decrementar para conquistar

1) Considere um algoritmo recursivo que resolve um problema de n elementos sempre chamando a si mesmo para n - 1. Em que condições essa estratégia pode ser mais cara que outras abordagens?

Resposta: Se o algoritmo não reutiliza subresultados (como em Fibonacci ingênuo), ele pode recalcular os mesmos valores várias vezes. Isso resulta em complexidade exponencial, enquanto outras abordagens como programação dinâmica evitariam essas recomputações, sendo muito mais eficientes.

2) Compare a abordagem "decrementar para conquistar" com a técnica iterativa de cauda (*tail recursion*) em termos de eficiência e legibilidade de código.

Resposta: Ambas podem ter complexidade similar, mas tail recursion permite que algumas linguagens otimizem a chamada recursiva, reduzindo uso de pilha. No entanto, em linguagens sem otimização de cauda (como Python), a versão iterativa costuma ser mais eficiente. A recursiva pode ser mais legível para problemas naturalmente recursivos.

3) Como um engenheiro de software pode avaliar quando não vale a pena aplicar o paradigma de "decrementar para conquistar"?

Resposta: Quando os subproblemas não são independentes ou há repetição significativa de cálculos, a estratégia perde eficiência. Ex: em problemas com subproblemas sobrepostos, como no cálculo de combinações, é melhor usar memoização ou tabulação. O engenheiro deve considerar se há ganhos reais em clareza ou eficiência.

Retrocesso e poda

1) Em um sistema de recomendação de produtos com várias regras e preferências, por que um algoritmo de retrocesso com poda pode ser mais eficiente do que força bruta?

Resposta: Porque ele usa as restrições do domínio para eliminar cedo caminhos que não podem levar a uma recomendação válida. Isso evita o custo de explorar todas as combinações possíveis. Com poda inteligente, é possível explorar somente o subconjunto viável das possibilidades.

2) Quais estratégias você adotaria para implementar a poda de forma eficiente em um algoritmo de busca por solução de quebra-cabeças?

Resposta: Usaria heurísticas, como estimativas de distância para o objetivo (ex: A*), e validaria parcialmente as soluções à medida que elas se constroem. Também manteria um histórico de estados visitados para evitar ciclos e repetição. A poda deve ser implementada sem comprometer a completude do algoritmo.

3) Retrocesso sempre encontra a solução ótima? Justifique sua resposta com base em algum cenário prático.

Resposta: Não necessariamente. Retrocesso garante encontrar uma solução válida (ou todas), mas não necessariamente a melhor, a menos que seja combinado com verificação de custo. Ex: em labirintos, pode encontrar o caminho mais curto se o critério de custo estiver embutido — caso contrário, encontra apenas o primeiro caminho viável.

Algoritmos gulosos

1) Você está construindo um serviço de alocação de anúncios em tempo real, onde cada anúncio tem um valor e uma janela de tempo disponível. Um colega sugere um algoritmo guloso para sempre escolher o anúncio mais valioso que “cabe” na agenda. Quais critérios você usaria para avaliar se essa abordagem resultará em uma solução próxima da ideal? E se não for o caso, qual alternativa proporia?

Resposta: A avaliação deve considerar se o problema tem subestrutura ótima (ou seja, decisões locais levam à melhor solução global) e escolha segura (não há arrependimento futuro por uma escolha atual). Se os anúncios tiverem conflitos complexos de agendamento ou se a janela de tempo for interdependente, o algoritmo guloso pode falhar. Nesses casos, a alternativa é usar programação dinâmica ou algoritmos aproximativos baseados em backtracking com poda, que consideram mais contextos ao tomar decisões.

2) Compare a eficácia de algoritmos gulosos em três contextos distintos:
(a) compactação de arquivos,
(b) roteamento de pacotes em redes,
(c) agendamento de aulas em uma universidade.

Quais são as limitações e vantagens em cada um?

Resposta:
(a) Compactação de arquivos: Algoritmos gulosos como Huffman funcionam muito bem, pois as decisões locais (escolher os símbolos mais frequentes) levam à menor codificação global — é um caso clássico de subestrutura ótima.
(b) Roteamento de pacotes: Gulosos funcionam em redes sem congestionamento, mas podem levar a gargalos se usados isoladamente. A falta de visão global pode causar colisões ou atrasos.
(c) Agendamento de aulas: Gulosos podem ajudar a alocar horários rapidamente, mas falham ao lidar com múltiplas restrições (ex: professor, sala, disciplina), onde decisões locais (alocar a próxima vaga disponível) podem impedir melhores configurações futuras.

3) Você desenvolve um sistema de logística para entregas com janelas de tempo. O cliente quer minimizar o tempo total de deslocamento. Por que um algoritmo guloso que sempre escolhe o destino mais próximo pode não gerar a melhor rota? Qual seria uma abordagem mais robusta?

Resposta: Esse algoritmo corresponde ao algoritmo do vizinho mais próximo, que frequentemente leva a caminhos subótimos (ex: ele pode fechar uma rota curta agora e criar um trajeto longo no final). Uma abordagem mais robusta seria aplicar algoritmos aproximativos para o problema do caixeiro viajante (TSP), como algoritmos genéticos, simulated annealing ou branch-and-bound, que consideram a rota como um todo, e não apenas o próximo passo.

Programação dinâmica

1) Explique por que a programação dinâmica é adequada para problemas com subproblemas sobrepostos. Dê um exemplo real em engenharia de software.

Resposta: Porque evita recomputações desnecessárias, melhorando a eficiência. Exemplo real: cálculo de métricas de similaridade em múltiplos arquivos (como Levenshtein). Reusar resultados de edições anteriores entre strings similares economiza tempo e recursos, essencial em sistemas de análise de código em larga escala.

2) Um colega propõe resolver o problema de caminhos mínimos em grafo com programação dinâmica. Em que condições essa abordagem é vantajosa sobre Dijkstra?

Resposta: Se o grafo tiver pesos negativos, Dijkstra falha, enquanto algoritmos como Bellman-Ford (baseado em programação dinâmica) funcionam. Além disso, se houver necessidade de computar todos os pares de caminhos mínimos (como em Floyd-Warshall), a abordagem de PD é preferível.

3) Você precisa projetar um sistema de sugestão de texto com base em correções anteriores (autocorreção). Como a programação dinâmica pode ser usada nesse contexto?

Resposta: Aplicaria PD para calcular a distância de edição entre palavras digitadas e palavras conhecidas, reutilizando subproblemas para acelerar o cálculo. Isso permite sugestões rápidas e inteligentes, essenciais em sistemas responsivos. Armazenar os resultados em cache melhora o desempenho.
