Algoritmos de Engenharia

Programa de Pós-Graduação em

Engenharia Elétrica e de Computação

Centro de Tecnologia – UFRN

1. Fundamentos - Introdução

- Capítulo 1

O que é um algoritmo?

- **Definição formal:** Um procedimento computacional bem definjdo que recebe entrada e produz saída em um período finito de tempo. É uma sequência de etapas computacionais para transformar entrada em saída.
- **Uma ferramenta para resolução de problemas:** Um algoritmo é um procedimento específico para alcançar uma relação de entrada / saída desejada para todas as instâncias de um problema computacional bem especificado.
- **Principais propriedades de um algoritmo correto:**
- **Interrupções**: Termina em um período finito de tempo para cada instância de entrada.
- **Correção:** gera a solução correta para cada instância de entrada.
- *Ponto de discussão:* algoritmos incorretos podem ser úteis? Sim, se sua taxa de erro puder ser controlada (por exemplo, na localização de grandes números primos).

O problema de ordenação: um exemplo clássico

- **Definição do problema:**
  - **Entrada**: Uma sequência de n números: $\langle a1, a2, \ldots, an \rangle$.
  - **Saída**: Uma permutação (reordenação) da sequência de entrada $\langle a1', a2', \ldots, an' \rangle$ tal que $a1' \leq a2' \leq \cdots \leq an'$.
- **Por que a ordenação é importante?** É uma operação fundamental na ciência da computação e frequentemente usada como uma etapa intermediária em programas mais complexos.
- **Escolhendo o "Melhor" Algoritmo:** A escolha ideal depende de fatores como o número de itens, a ordenação deles e a arquitetura do computador.

Aplicações onipresentes de algoritmos

- **Bioinformática**: O Projeto Genoma Humano usa algoritmos sofisticados para identificação de genes, análise de sequência e armazenamento de dados, muitas vezes empregando técnicas como programação dinâmica.
- **Internet e redes**: Os algoritmos são essenciais para encontrar rotas de dados eficientes e para alimentar os mecanismos de pesquisa.
- **Comércio eletrônico e segurança**: As principais tecnologias dependem de criptografia de chave pública e assinaturas digitais, que são baseadas em algoritmos numéricos.
- **Alocação de recursos**: Resolvido modelando problemas como programas lineares, usados em indústrias de empresas petrolíferas a companhias aéreas para maximizar o lucro ou a eficiência.
- **Problemas de caminho mais curto**: Encontrar a rota mais curta em um mapa é um problema algorítmico clássico com aplicações diretas em transporte e roteamento de rede.

Algoritmos como tecnologia: o argumento da eficiência

- **O argumento central:** Mesmo com computadores infinitamente rápidos, ainda precisaríamos de algoritmos para garantir que nossos métodos terminassem com a resposta correta. Na realidade, o tempo de computação é um recurso precioso e limitado.
- **A eficiência é mais importante do que o hardware**: A escolha do algoritmo pode ter um impacto muito maior no desempenho do que as diferenças na velocidade do hardware.
- **Exemplo ilustrativo:** ordenação de inserção vs. ordenação de mesclagem
  - ordenação de inserção: Leva aproximadamente c1n2 tempo.
  - Mesclar ordenação: Leva aproximadamente c2n lg n tempo.
  - **A lição**: Para uma grande entrada de 10 milhões de números, um computador mais lento executando o algoritmo mais eficiente (Merge Sort) foi mais de 17 vezes mais rápido do que um computador 1000 vezes mais poderoso executando o menos eficiente (Insertion Sort). A escolha do algoritmo é uma tecnologia crítica.

Tópicos avançados e contexto mais amplo

- **Problemas difíceis (NP-Completude):**
  - Uma classe de problemas para os quais não existe nenhum algoritmo eficiente conhecido.
  - **Uma propriedade chave**: Se um algoritmo eficiente é encontrado para um problema NP-completo, existem algoritmos eficientes para todos eles.
  - **Exemplo**: O "problema do caixeiro viajante".
  - **Abordagem prática**: Quando um problema é mostrado como NP-completo, podemos nos concentrar em encontrar um algoritmo de aproximação eficiente - um que forneça uma solução boa, mas não necessariamente perfeita.
- **Estruturas de dados:**
  - Uma maneira de armazenar e organizar dados para facilitar o acesso e as modificações.
  - A escolha apropriada da estrutura de dados é uma parte crucial do design do algoritmo.
- **Modelos de computação alternativos:**
  - **Algoritmos paralelos:** Projetado para processadores *multi-core* para realizar mais cálculos por segundo.
  - **Algoritmos online:** processa a entrada que chega ao longo do tempo, sem saber quais serão os dados futuros (por exemplo, agendamento de trabalhos em um data center ou roteamento do tráfego da Internet).

1. Fundamentos - Introdução

- Capítulo 2

Ordenaçao de inserção: uma abordagem incremental

- **A analogia:** Ordenando uma mão de cartas de baralho. Você pega uma carta de cada vez e a insere na posição correta em sua mão já ordenada.
- **O algoritmo:**
- Itere do segundo elemento até o final da vetor.
- Para cada elemento (`chave`), compare-o com os elementos no subvetor ordenado à sua esquerda.
- Desloque todos os elementos maiores para a direita para criar um espaço.
- Insira a `chave` na posição correta.
- **Característica principal**: É um algoritmo *in-place*, eficiente para ordenar um pequeno número de elementos.

Provendo a corretude com invariantes de laço

- **O que é um invariante de laço?** Uma propriedade que é verdadeira antes e depois de cada iteração de um laço. É uma maneira formal de raciocinar sobre a correção de um algoritmo.
- **Três propriedades essenciais a serem comprovadas**:
  - **Inicialização:** A invariante é verdadeira antes da primeira iteração do loop.
  - **Manutenção:** Se a invariante for verdadeira antes de uma iteração, ela permanecerá verdadeira antes da próxima.
  - **Terminação**: Quando o loop termina, a invariante (junto com a condição de término) fornece uma propriedade útil que ajuda a mostrar que o algoritmo está correto.
- **Para ordenação de inserção**: A invariante de laço é que, no início de cada iteração, o subvetor à esquerda do elemento atual consiste nos elementos originais, mas em ordem.

Analisando o desempenho de algoritmos

- **Objetivo**: Prever os recursos (especialmente o tempo) que um algoritmo requer.
- **Modelo de Computação**: Usamos o modelo de **Máquina de Acesso Aleatório** (RAM).
  - As instruções são executadas uma após a outra (sem simultaneidade).
  - Cada instrução "simples" (aritmética, movimentação de dados, controle) leva um tempo constante.
- **Métricas-chave:**
  - **Tamanho da entrada (n):** o número de itens na entrada.
  - **Tempo de execução T(n):** O número de instruções executadas em função do tamanho da entrada.

Análise da ordenação de inserção

- **Melhor caso**: A matriz já está ordenada.
  - A condição do laço `while` interno sempre falha imediatamente.
  - O tempo de execução é uma função linear de n. **$T(n) = \Theta(n)$**.
- **Pior caso**: A matriz é ordenada na ordem inversa.
  - Cada elemento deve ser comparado com todos os outros elementos no subvetor ordenado.
  - O tempo de execução é uma função quadrática de **n**. **$T(n) = \Theta(n^2)$.**
- **Caso médio**: Supondo que todas as permutações sejam igualmente prováveis, o tempo de execução também é quadrático. **$T(n) = \Theta(n^2)$.**
- **Concentre-se no pior caso**: fornece um limite superior no tempo de execução, o que é uma garantia crucial.

Projeto de algoritmo: dividir e conquistar

- Um poderoso paradigma de projeto recursivo.
- **Três etapas:**
  - **Dividir**: Divida o problema em vários subproblemas menores e independentes que são instâncias menores do problema original.
  - **Conquistar**: Resolva os subproblemas recursivamente. Se eles forem pequenos o suficiente (caso base), resolva-os diretamente.
  - **Combinar**: Mescle as soluções dos subproblemas para criar uma solução para o problema original.

Merge Sort: um exemplo de dividir e conquistar

- O algoritmo:
  - **Dividir:** Divida a matriz de **n** elementos em dois subvetores de **n/2** elementos cada.
  - **Conquistar**: Ordene os dois subvetores recursivamente usando merge sort.
  - **Combinar**: mescle os dois subvetores ordenados para produzir o vetor ordenado final.
- **O procedimento** `MERGE`: O núcleo do algoritmo. Ele pega dois subvetores ordenados e os combina em um único vetor ordenado em tempo linear, **Θ(n)**.

Analysis of Merge Sort

- O tempo de execução pode ser descrito por uma equação de recorrência.
- **Recorrência para o Merge Sort:**
- **Dividir**: Leva tempo constante, $\Theta(1)$.
- **Conquistar**: Duas chamadas recursivas em problemas de tamanho $n/2$, contribuindo com $2T(n/2)$.
- **Combinar**: O procedimento `MERGE` leva tempo linear, $\Theta(n)$.
- **Total**: $T(n) = 2T(n/2) + \Theta(n)$
- **Solução**: O pior caso de tempo de execução para o Merge Sort é $\Theta(n \log n)$. Esta é uma melhoria significativa em relação ao $\Theta(n^2)$ do Insertion Sort para entradas grandes.

1. Fundamentos - Introdução

- Capítulo 3

Além do benchmarking: por que precisamos de análise assintótica

- **O problema com os tempos brutos**: Executar um algoritmo e cronometrá-lo com um cronômetro não é confiável. O resultado depende do hardware específico, da linguagem de programação, do compilador e até mesmo de outros processos em execução na máquina.
- **O objetivo**: Precisamos de uma maneira de falar sobre a "escalabilidade" e a eficiência de um algoritmo que seja independente dos detalhes da implementação. Queremos responder: "Como o esforço necessário para resolver um problema cresce à medida que o problema aumenta?"
- **Eficiência assintótica**: Nós nos concentramos na ordem de crescimento do tempo de execução. Isso nos informa a tendência de desempenho para grandes entradas, o que é crítico para aplicações de engenharia do mundo real.

A Linguagem do Crescimento: Notações Assintóticas

- Usamos uma notação especial para descrever a taxa de crescimento das funções. Pense neles como classificações para o desempenho do algoritmo.
- **As três classificações principais**:
  - **Notação O (Big-O)**: Fornece um **limite superior** (um teto). "O tempo de execução não cresce mais rápido do que isso."
  - **Notação Ω (Big-Omega):** Fornece um **limite inferior** (um piso). "O tempo de execução cresce pelo menos tão rápido quanto isso."
  - **Notação Θ (Big-Theta):** Fornece um limite apertado. "O tempo de execução cresce exatamente nessa taxa."

O-Notation (Big-O): a garantia do "pior caso"

- **Analogia**: Um limite de tempo. Se o limite de tempo for de 100 h, é garantido que você não levará mais tempo, mas poderá levar menos.
- **Significado**: $T(n)=O(n^2)$ significa que, para uma entrada n grande o suficiente, o tempo de execução do algoritmo não excederá algum múltiplo constante de n^2.
- **Uso prático**: Dá-nos um piso de desempenho. Sabemos que o algoritmo nunca terá um desempenho pior do que esse limite. É a notação mais comumente usada porque geralmente nos preocupamos com o desempenho do pior caso.

Notação Ω (Big-Omega): A garantia do "melhor caso"

- **Analogia**: Um tempo mínimo necessário. Se você precisar levar pelo menos 50 h, é garantido que não levará menos, mas poderá levar muito mais tempo.
- **Significado**: T (n) = Ω (n) significa que, para qualquer entrada grande, o algoritmo levará pelo menos um tempo linear. Isso não pode ser feito mais rápido.
- **Uso prático**: Diz-nos a complexidade mínima inerente de uma tarefa.

Notação Θ (Big-Theta): O "Desempenho Típico"

- **Analogia**: Dirigir a uma velocidade fixa. Se o seu controle de cruzeiro estiver ajustado para 80 km/h, você está indo exatamente nessa velocidade (dentro de uma margem muito pequena).
- **Significado**: T (n) = Θ (nlogn) significa que a taxa de crescimento do algoritmo está com limites apertados. Seu melhor e pior caso crescem na mesma proporção.
- **Uso prático**: Esta é a caracterização mais precisa e desejável. Ela informa exatamente como um algoritmo deve ser dimensionado. Por exemplo, Merge Sort é sempre Θ(nlogn), tornando seu desempenho altamente previsível.

Regras práticas

- Ao examinar uma função de tempo de execução, você pode simplificá-la para encontrar seu limite assintótico.
  1. **Mantenha o termo de crescimento mais rápido**: Em uma expressão como $4n^2 + 100n + 500$, o termo $4n^2$ dominará para n grande. Os outros termos tornam-se insignificantes.
  2. **Descarte os coeficientes constantes**: A diferença entre $4n^2$ e n^2 é um fator constante. A notação assintótica ignora isso, concentrando-se puramente na ordem de crescimento.
- **Exemplo**: A função $4n^2+100n+500$ é simplificada para Θ(n2).

A hierarquia das taxas de crescimento comuns

- É crucial reconhecer quais funções crescem mais rápido. Aqui estão eles, da melhor para a pior escalabilidade:
  - **Θ(1) - Constante**: Não afetado pelo tamanho da entrada. (Excelente)
  - **Θ(logn) - Logarítmico**: Crescimento muito lento. (Excelente)
  - **Θ(n) - Linear**: Escala diretamente com o tamanho de entrada. (Muito bom)
  - **Θ(nlogn) - Log-linear**: Muito comum para um processamento eficiente. (Bom)
  - **Θ(n^2) - Quadrático**: Torna-se lento para entradas grandes. (Ok para pequenas entradas)
  - **Θ(n^3) - Cúbico**: O desempenho se degrada rapidamente. (Use com cuidado)
  - **Θ(2^n) - Exponencial**: Torna-se inutilizável mesmo para entradas de tamanho moderado. (Evite se possível)

A conclusão prática

- A análise assintótica é uma ferramenta crítica para **prever escalabilidade**.
- Ele ajuda você a escolher o algoritmo certo antes de investir tempo na implementação.
- Para problemas pequenos e de tamanho fixo, um algoritmo com um tempo de execução assintótico "pior" (por exemplo, $\Theta(n^2)$) pode ser mais rápido devido a fatores constantes menores.
- Para sistemas de grande escala (big data, simulações complexas, processamento de alta frequência), o algoritmo com a melhor taxa de crescimento assintótico **sempre** vencerá.

Lista de exercícios de Fundamentos

1. (Exercício 2.1-3 do livro do Cormen) Escreva um pseudocódigo para a busca linear e mostre, usando invariância de laço, que o seu algoritmo está correto.
2. Implemente o algoritmo de ordenação por inserção e crie uma cópia anotada dele que mede o número de operações no modelo da *Random Access Machine* (RAM, seção 2.2 livro do Cormen). Usando entradas de tamanho crescente, mostre em um gráfico quando o tempo de execução no modelo RAM diverge de medições feitas em uma máquina real.
3. Mostre numericamente com suas implementações dos algoritmos de *insertion-sort* e *merge-sort* como se comporta o desempenho de cada algoritmo utilizando entradas de tamanho crescente, considerando entradas de pior caso, melhor caso e caso médio. Análise, para cada tipo de entrada, se existe algum ponto a partir do qual um algoritmo passa a ser mais rápido que o outro.