Estrutura de Dados II

Análise de Algoritmos

Prof. Dr. Marcelo Otone Aguiar

Universidade Federal do Espírito Santo - UFES

14 de Março de 2024

Conteúdo

- Introdução a análise de algoritmos
- Contando instruções
- Comportamento assintótico
- Tipos de análise assintótica
- Classes de problemas

Conceitos Gerais

- **Análise de Algoritmos**
  - Em ciência da computação, é a área de pesquisa cujo foco são os algoritmos
  - Busca responder a seguinte pergunta: **podemos fazer um algoritmo mais eficiente?**

Conceitos Gerais

- Podemos resolver um problema de várias maneiras diferentes, isto é, podemos utilizar algoritmos diferentes para um mesmo problema
  - Algoritmos diferentes, mas capazes de resolver o mesmo problema, não necessariamente o fazem com a mesma eficiência.
- Essas diferenças de eficiência podem ser:
  - **irrelevantes** para um pequeno número de elementos processados
  - **crescer proporcionalmente** com o número de elementos processados

Conceitos Gerais

- Para comparar a eficiência dos algoritmos foi criada uma medida chamada de **complexidade computacional**
- Basicamente, ela indica o **custo** ao se aplicar um algoritmo, sendo:

$$custo = memoria + tempo \tag{1}$$

Em que:

- **memória:** quanto de espaço o algoritmo vai consumir
- **tempo:** a duração de sua execução

Conceitos Gerais

- Para determinar se um algoritmo é o mais eficiente, podemos utilizar duas abordagens
  - **Análise empírica:** comparação entre os programas
  - **Análise matemática:** estudo das propriedades do algoritmo

Análise Empírica

- Avalia o custo (ou **complexidade**) de um algoritmo a partir da avaliação da execução do mesmo quando implementado
  - Ou seja, um algoritmo é analisado pela execução de seu programa correspondente
- **Vantagens:**
  - Avaliar o desempenho em uma determinada configuração de computador / linguagem
  - Considerar custos não aparentes (ex: o custo da alocação de memória)
  - Comparar computadores
  - Comparar linguagens

Análise Empírica

- **Desvantagens:**
  - Necessidade de implementar o algoritmo
    - Isso depende da habilidade do programador
  - Resultado pode ser mascarado pelo hardware (computador utilizado) ou software (eventos ocorridos no momento da avaliação)
  - Qual a natureza dos dados:
    - Dados reais
    - Aleatórios (avaliam o desempenho médio)
    - Perversos (pior caso)

Análise Matemática

- Permite um estudo formal de um algoritmo ao nível **ideia** por trás do algoritmo
- Faz uso de um computador idealizado e simplificações que buscam considerar somente os custos dominantes do algoritmo
- A medição do tempo gasto é feita de maneira independente do **hardware** ou da **linguagem** usada na sua implementação
- **Vantagens:**
  - Detalhes de baixo nível, como a linguagem de programação utilizada, o hardware no qual o algoritmo é executado, ou o conjunto de instruções da CPU, são ignorados
  - Permite entender como um algoritmo se comporta à medida que o conjunto de dados de entrada cresce. Assim, podemos expressar a relação entre o conjunto de dados de entrada

Análise matemática

- Contando instruções de um algoritmo
  - Este algoritmo procura o maior valor presente em um array **A** contendo **n** elementos e o armazena na variável **M**

```
1  int M = A[0];
2  for (i = 0; i < n; i++) {
3     if (A[i] >= M) {
4           M = A[i];
5        }
6  }
```

Análise matemática

- Vamos contar quantas **instruções simples** ele executa
- Uma **instrução simples** é uma instrução que pode ser executada diretamente pelo CPU, ou algo muito perto disso
- Tipos de instruções:
  - Atribuição de um valor a uma variável
  - Acesso ao valor de um determinado elemento do array
  - Comparação de dois valores
  - Incremento de um valor
  - Operações aritméticas básicas, como adição e multiplicação
- Vamos assumir que:
  - As instruções possuem o mesmo **custo**
  - **comandos de seleção** possuem custo zero

Análise matemática

```
1  int M = A[0];
2  for (i = 0; i < n; i++) {
3     if (A[i] >= M) {
4           M = A[i];
5        }
6  }
```

- O custo da **linha 1** é de **1 instrução**
- Na **linha 1**, o valor da primeira posição do array é copiado para a variável **M**
  - Acessar o valor **A[0]** e atribuir a **M**

Análise matemática

```
1  int M = A[0];
2  for (i = 0; i < n; i++) {
3    if (A[i] >= M) {
4        M = A[i];
5      }
6  }
```

- O custo da inicialização do laço **for** (**linha 2**) é de **2 instruções**
- O comando **for** precisa ser inicializado: **1 instrução** ($i = 0$)
- Mesmo que o array tenha tamanho zero, ao menos uma comparação será executada ($i < n$), o que custa mais **1 instrução**

Análise matemática

- O custo para executar o comando de laço **for** (**linha 2**) é de **2n instruções**
- Ao final de cada iteração do laço **for**, precisamos executar uma instrução de
  - Incremento $(i++)$
  - Comparação de continuidade do laço **for** $(i < n)$
- O laço **for** será executado **n** vezes. Assim, essas **2** instruções também serão executadas **n** vezes: **2n** instruções

```
1  int M = A[0];
2  for (i = 0; i < n; i++) {
3    if (A[i] >= M) {
4        M = A[i];
5      }
6  }
```

Custo Dominante ou Pior caso do Algoritmo

- Ignorando os comandos contidos no corpo do laço **for**, teremos que o algoritmo precisa executar $3 + 2n$ instruções:
  - **3** instruções até a inicialização do laço **for**
  - **2** instruções ao final de cada laço **for**, o qual é executado **n** vezes
- Assim, considerando um **laço vazio**, podemos definir uma função matemática que representa o custo do algoritmo em relação ao tamanho do array de entrada: $f(n) = 2n + 3$

Análise matemática

- Contando as instruções restantes do for
  - Comando **if = 1 instrução**: acesso ao valor do array e a sua comparação ($A[i] >= M$)
  - Dentro do **if = 1 instrução**: acessa o valor do array e o atribui a outra variável ($M = A[i]$). Porém, sua execução depende do resultado da comparação feita pelo **if**.

```
1  int M = A[0];
2  for (i = 0; i < n; i++) {
3    if (A[i] >= M) {
4        M = A[i];
5      }
6  }
```

Custo Dominante ou Pior caso do Algoritmo

- Problema:
  - As instruções vistas anteriormente eram sempre executadas
  - As instruções dentro do **for** podem ou não ser executadas
- Antes, bastava saber o tamanho do array, **n**, para definir a função de custo **f(n)**
- Agora temos que considerar também o conteúdo do array
- Exemplo: dois arrays de mesmo tamanho
  - $A1 = 1, 2, 3, 4$
  - $A2 = 4, 3, 2, 1$
- Array **A1**: mais instruções $->$ o comando **if** é sempre **verdadeiro**
- Array **A2**: atribuição nunca é executada pois o comando **if** é sempre **falso**

Custo Dominante ou Pior caso do Algoritmo

- Ao analisarmos um algoritmo, é muito comum considerarmos o **pior caso** possível
- **Pior caso**: maior número de instruções executadas
- No nosso algoritmo o **pior caso** ocorre quando o array possui valores em ordem crescente
  - O valor de **M** é sempre substituído: Maior número de instruções
  - O laço **for** sempre executa as 2 instruções
- Assim, a função custo do algoritmo será

$$f(n) = 3 + 2n + 2n$$

ou

$$f(n) = 4n + 3$$

Comportamento assintótico

- Vimos que o custo para o algoritmo abaixo é dado pela função: $f(n) = 4n + 3$

```
1  int M = A[0];
2  for (i = 0; i < n; i++) {
3     if (A[i] >= M) {
4           M = A[i];
5        }
6  }
```

- Essa é a função de complexidade de tempo
  - Ela nos dá uma ideia do custo de execução do algoritmo para um problema de tamanho **n**

Comportamento assintótico

- Dúvida:
  - Será que todos os termos da função **f** são necessários para termos noção do custo?
- De fato, nem todos os termos são necessários
- Podemos descartar certos termos na função e manter apenas os que nos dizem o que acontece com a função quando o tamanho dos dados de entrada (**n**) cresce muito
- Se um algoritmo é mais rápido do que outro para um grande conjunto de dados de entrada, é muito provável que ele continue sendo também mais rápido em um conjunto de dados menor

Comportamento assintótico

- Podemos descartar todos os termos que crescem lentamente e manter apenas os que crescem mais rápido à medida que o valor de **n** se torna maior
- A função, $f(n) = 4n + 3$, possui dois termos:
  - $4n$
  - 3
- O termo **3** é uma **constante de inicialização**
- Não se altera à medida que **n** aumenta
- Assim, nossa função pode ser reduzida para $f(n) = 4n$

Comportamento assintótico

- Constantes que multiplicam o termo **n** da função também devem ser descartadas
- Isso faz sentido se pensarmos em diferentes linguagens de programação. Por exemplo, a seguinte linha de código em Pascal
  - $M := A[i];$
- Equivale ao seguinte código em linguagem C
  - $if\,(i >= 0 \&\& i < n)$
  - $\quad M = A[i];$
- O acesso a um elemento do array
  - **Pascal:** 3 instruções (etapa de verificação)
  - **C:** 1 instrução (sem etapa de verificação)

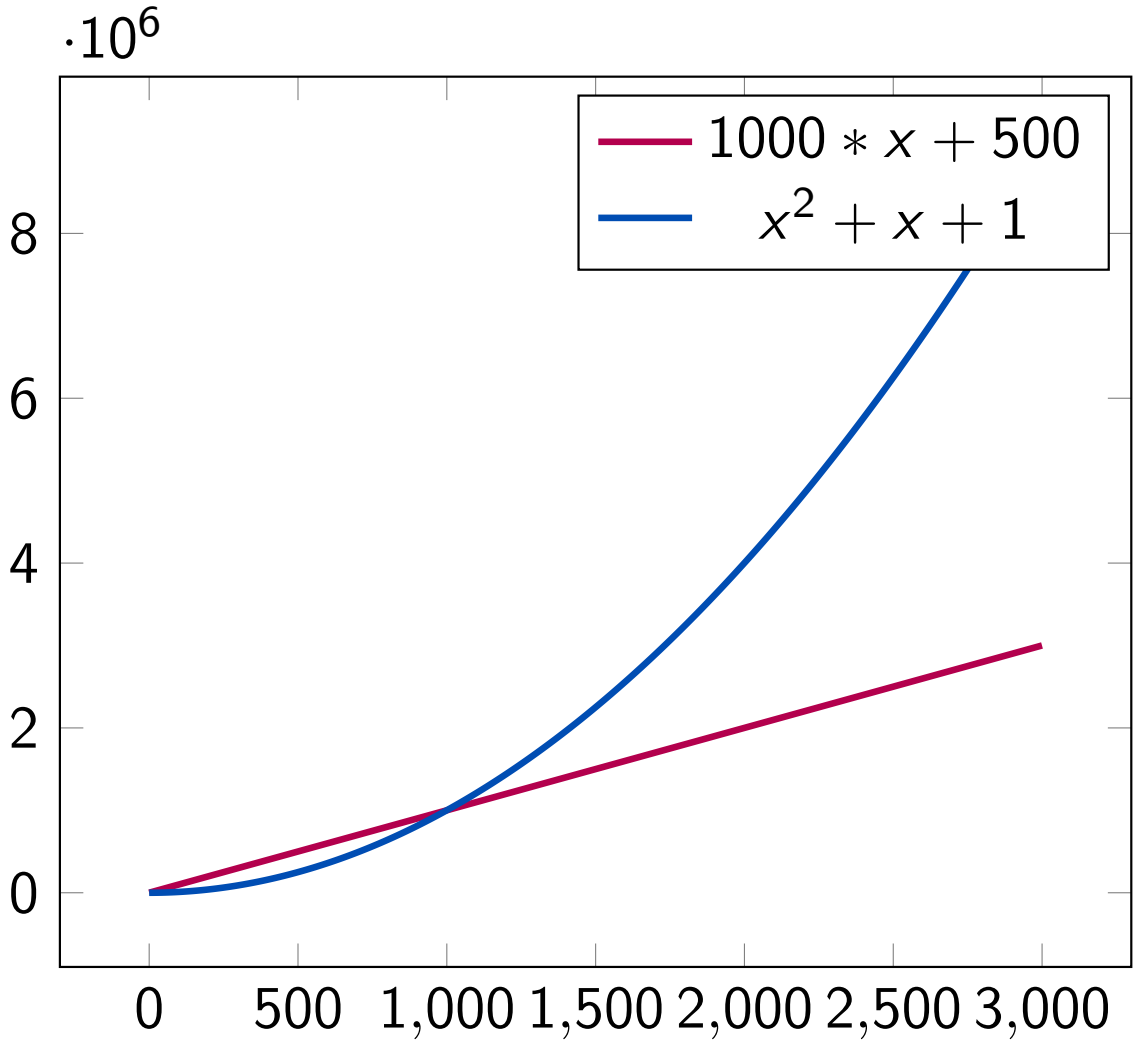
Comportamento assintótico

- Ignorar essas constantes de multiplicação equivale a ignorar as particularidades de cada linguagem e compilador e analisar apenas a ideia do algoritmo
- Assim, nossa função pode ser reduzida para $f(n) = n$
- Descartando todos os termos constantes e mantendo apenas o de maior crescimento, obtemos o **comportamento assintótico**
- Trata-se do comportamento de uma função **f(n)** quando **n** tende ao infinito
- Isso acontece porque o termo que possui o maior expoente domina o comportamento da função **f(n)** quando **n** tende ao infinito

Comportamento assintótico

- Para entender melhor, considere duas funções:
  - $g(n) = 1000n + 500$
  - $h(n) = n^2 + n + 1$
- Apesar da função **g(n)** possuir constantes maiores multiplicando seus termos, existe um valor de **n** a partir do qual o resultado de **h(n)** é sempre maior do que **g(n)**, tornando os demais termos e constantes pouco importantes

Comportamento assintótico



Comportamento assintótico

- Podemos então suprimir os termos menos importantes da função e considerar apenas o termo de maior grau
- Assim, podemos descrever a complexidade usando somente o seu custo dominante:
  - n para a função $g(n)$
  - n^2 para a função $h(n)$
- Se a função não possui nenhum termo multiplicativo por n, seu comportamento assintótico é constante (1)

Função de custo	Comportamento Assintótico
$f(n) = 105$	$f(n) = 1$
$f(n) = 15n + 2$	$f(n) = n$
$f(n) = n^2 + 5n + 2$	$f(n) = n^2$
$f(n) = 5n^3 + 200n^2 + 112$	$f(n) = n^3$

Comportamento assintótico

- De modo geral, podemos obter a função de custo de um programa simples apenas contando os comandos de laços aninhados
- Exemplos:
  - Algoritmo sem laço: número constante de instruções (exceto se houver recursão), ou seja, $f(n) = 1$
  - Com um laço indo de 1 a n será $f(n) = n$ (ou seja, um conjunto de instruções constantes antes e/ou depois do laço e um conjunto de instruções constantes dentro do laço)
  - Dois comandos de laço aninhados será $f(n) = n^2$, e assim por diante

Tipos de Análise Assintótica

- A seguir, são matematicamente descritas outras formas de análise assintótica
  - Notação Grande-Omega
  - Notação Grande-O
  - Notação Grande-Theta
  - Notação Pequeno-o
  - Notação Pequeno-omega

Notação Grande-Omega

- Descreve o **limite assintótico inferior** de um algoritmo
- É a notação utilizada para analisar o **melhor caso** do algoritmo
- A notação $\Omega(n^2)$ nos diz que o custo do algoritmo é, assintoticamente, maior ou igual a n^2
- Em outras palavras, o custo do algoritmo original é no **mínimo** tão ruim quanto n^2

Notação Grande-O

- Descreve o **limite assintótico superior** de um algoritmo
- É a notação utilizada para analisar o **pior caso** do algoritmo
- A notação $O(n^2)$ nos diz que o custo do algoritmo é, assintoticamente, menor ou igual a n^2
- Em outras palavras, o custo do algoritmo original é no **máximo** tão ruim quanto n^2

Notação Grande-Theta

- Descreve o **limite assintótico firme** ou **estreito** de um algoritmo
- É a notação utilizada para analisar o limite **inferior** e **superior** do algoritmo
- A notação $\Theta(n^2)$ nos diz que o custo do algoritmo é, assintoticamente, igual a n^2
- Em outras palavras, o custo do algoritmo original é n^2 dentro de um fator constante **acima** e **abaixo**

Notação Pequeno-o e Pequeno-omega

- Parecidas com as notações **Grande-O** e **Grande-Omega**
- As notações **Grande-O** e **Grande-Omega** possuem uma relação de **menor ou igual** e **maior ou igual**
- As notações **Pequeno-o** e **Pequeno-omega** possuem uma relação de **menor** e **maior**
- Ou seja, não representam limites próximos da função, mas apenas estritamente
  - **Superiores:** sempre maior
  - **Inferiores:** sempre menor

Classes de Problemas

- Algumas classes de complexidade de problemas são mais comumente usadas
- $O(1)$: ordem constante
  - As instruções são executadas um número fixo de vezes. Não depende do tamanho dos dados de entrada
- $O(log(n))$: ordem logarítmica
  - Típica de algoritmos que resolvem um problema transformando-o em problemas menores
- $O(n)$: ordem linear
  - Em geral, uma certa quantidade de operações é realizada sobre cada um dos elementos de entrada

Classes de Problemas

- $O(nlog(n))$: ordem log linear
  - Típica de algoritmos que trabalham com particionamento dos dados. Esses algoritmos resolvem um problema transformando-o em problemas menores, que são resolvidos de forma independente e depois unidos (ex: **Quicksort**)
- $O(n^2)$: ordem quadrática
  - Normalmente ocorre quando os dados são processados aos pares. Uma característica deste tipo de algoritmos é a presença de um aninhamento de dois comandos de repetição
- $O(n^3)$: ordem cúbica
  - É caracterizado pela presença de três estruturas de repetição aninhadas

Classes de Problemas

- $O(2^n)$: ordem exponencial
  - Geralmente ocorre quando se usa uma solução de **força bruta**. Não são úteis do ponto de vista prático
- $O(n!)$: ordem fatorial
  - Geralmente ocorre quando se usa uma solução de **força bruta**. Não são úteis do ponto de vista prático. Possui um comportamento muito pior que o exponencial