Análise de algoritmos

Crescimento de funções

Notas

Conteúdo

Introdução

Notação assintótica
- Notação O
- Notação Ω
- Notação Θ
- Outras notações
- Propriedades
- Exercícios

Notações padrão e funções comuns

Exercícios

Referências

Notas

Introdução

- Como expressar a eficiência de um algoritmo?
- Através da ordem de crescimento do tempo de execução
  - apenas o termo de mais alta ordem é considerado
  - caracterização simples da eficiência de um algoritmo
  - permite comparar o desempenho relativo entre algoritmos alternativos
- Quando observamos tamanhos de entradas grandes o suficiente, de forma que apenas a ordem de crescimento do tempo de execução seja relevante, estamos estudando a eficiência **assintótica**

Notas

Notação assintótica

Notação O

Para uma dada função $g(n)$, denotamos por $O(g(n))$ o conjunto de funções

$O(g(n)) = \{f(n)$: existem constantes positivas c e n_0 tais que $0 \leq f(n) \leq cg(n)$ para todo $n \geq n_0\}$

A notação O descreve um limite assintótico superior.

Notas

Notação assintótica

Exemplos

Responda utilizando a definição na notação O, se cada expressão a seguir é verdadeira ou falsa.

1. $100n^2 = O(n^2)$
2. $\frac{1}{2}n^2 - 3n = O(n^2)$
3. $3n + 20 = O(n^2)$
4. $6n^3 = O(n^2)$
5. $720 = O(1)$

Notas

Notação assintótica

Notação O

Quando afirmamos que o tempo de execução de um algoritmo é $O(g(n))$, queremos dizer que, independente da entrada específica de tamanho n escolhida para cada valor de n, o tempo de execução é no **máximo** uma constante vezes $g(n)$, para um valor de n suficientemente grande.

Notas

Notação assintótica

Notação Ω

Para uma dada função $g(n)$, denotamos por $\Omega(g(n))$ o conjunto de funções

$\Omega(g(n)) = \{f(n)$: existem constantes positivas c e n_0 tais que $0 \leq cg(n) \leq f(n)$ para todo $n \geq n_0\}$

A notação Ω descreve um limite assintótico inferior.

Notas

Notação assintótica

Exemplos

Responda utilizando a definição na notação Ω, se cada expressão a seguir é verdadeira ou falsa.

1. $100n^2 = \Omega(n^2)$
2. $\frac{1}{2}n^2 - 3n = \Omega(n^2)$
3. $3n^2 + 20 = \Omega(n)$
4. $6n = \Omega(n^2)$
5. $720 = \Omega(1)$

Notas

Notação assintótica

Notação Ω

Quando afirmamos que o tempo de execução de um algoritmo é $\Omega(g(n))$, queremos dizer que, independente da entrada específica de tamanho n escolhida para cada valor de n, o tempo de execução é **pelo menos** uma constante vezes $g(n)$, para um valor de n suficientemente grande.

Notas

Notação assintótica

Notação Θ

Para uma dada função $g(n)$, denotamos por $\Theta(g(n))$ o conjunto de funções

$\Theta(g(n)) = \{f(n)$: existem constantes positivas c_1, c_2 e n_0 tais que $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ para todo $n \geq n_0\}$

A notação Θ descreve um limite assintótico restrito.

Notas

Notação assintótica

Teorema 3.1

Para duas funções quaisquer $f(n)$ e $g(n)$, temos que $f(n) = \Theta(g(n))$ se e somente se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$. □

Notas

Notação assintótica

Exemplos

Responda utilizando a definição na notação Θ, se cada expressão a seguir é verdadeira ou falsa.

1. $100n^2 = \Theta(n^2)$
2. $\frac{1}{2}n^2 - 3n = \Theta(n^2)$
3. $3n^2 + 20 = \Theta(n)$
4. $6n = \Theta(n^2)$
5. $720 = \Theta(1)$

Notas

Notação assintótica em equações de desigualdades

Como interpretamos as seguintes fórmulas?

- $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$
- $T(n) = 2T(n/2) + \Theta(n)$
- $2n^2 + \Theta(n) = \Theta(n^2)$
- $2n^2 + 3n + 1 = 2n^2 + \Theta(n) = \Theta(n^2)$

Notas

Notação assintótica

Notação o

Para uma dada função $g(n)$, denotamos por $o(g(n))$ o conjunto de funções

$o(g(n)) = \{f(n)$: para qualquer constante positiva $c > 0$, existe uma constante $n_0 > 0$ tal que $0 \leq f(n) < cg(n)$ para todo $n \geq n_0\}$

A notação o descreve um limite superior que não é assintoticamente restrito.

Intuitivamente, a função $f(n)$ se torna insignificante em relação a $g(n)$ a medida que n se aproxima do infinito, isto é, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

Notas

Notação assintótica

Notação ω

Para uma dada função $g(n)$, denotamos por $\omega(g(n))$ o conjunto de funções

$\omega(g(n)) = \{f(n)$: para qualquer constante positiva $c > 0$, existe uma constante $n_0 > 0$ tal que $0 \leq cg(n) < f(n)$ para todo $n \geq n_0\}$

A notação ω descreve um limite inferior que não é assintoticamente restrito.

Intuitivamente, a função $f(n)$ se torna arbitrariamente grande em relação a $g(n)$ a medida que n se aproxima do infinito, isto é, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

Notas

Propriedades de comparações assintóticas

- Transitividade
  - $f(n) = \Theta(g(n))$ e $g(n) = \Theta(h(n))$ implicam $f(n) = \Theta(h(n))$
  - vale para as outras notações
- Reflexividade
  - $f(n) = \Theta(f(n))$
  - $f(n) = O(f(n))$
  - $f(n) = \Omega(f(n))$
- Simetria
  - $f(n) = \Theta(g(n))$ se e somente se $g(n) = \Theta(f(n))$
- Analogia com comparação de números reais
  - $f(n) = O(g(n))$ semelhante a $a \leq b$
  - $f(n) = \Omega(g(n))$ semelhante a $a \geq b$
  - $f(n) = \Theta(g(n))$ semelhante a $a = b$
  - $f(n) = o(g(n))$ semelhante a $a < b$
  - $f(n) = \omega(g(n))$ semelhante a $a > b$

Notas

Exercícios

2.2-2 Considere a ordenação de n números armazenados no arranjo A, localizando primeiro o menor elemento de A e permutando esse elemento com o elemento contido em $A[1]$. Em seguida, encontre o segundo menor elemento de A e troque pelo elemento $A[2]$. Continue desta maneira para os primeiro $n-1$ elementos de A. Escreva o pseudocódigo para esse algoritmo, conhecido como **ordenação por seleção**. Que invariante de laço esse algoritmo mantém? Por que ele só precisa ser executado para os primeiros $n-1$ elementos, e não para todos os n elementos? Forneça os tempos de execução do melhor caso e do pior caso da ordenação por seleção usando a notação Θ.

Notas

Exercícios - Ordenação por seleção

```
selection-sort(A)
1 for i = 1 to n - 1
2   min = i
3   for j = i + 1 to n
4     if A[min] > A[j]
5       min = j
6   troca(A, min, i)
```

Invariante de laço

- No início da iteração do for externo, o subarranjo $A[1..i-1]$ consiste dos $i-1$ menores elementos do arranjo, e este subarranjo está ordenado.
- Depois dos $n-1$ elementos terem sido processados, o subarranjo $A[1..n-1]$ contém os menores $n-1$ elementos ordenados, desta forma, o elemento $A[n]$ é o maior dos elementos e portanto está no posição correta.

Notas

Exercícios - Ordenação por seleção

Análise do tempo de execução

- A linha 1, 2 e 6 são executadas $\Theta(n)$ vezes
- A quantidade de vezes que a linha 3 é executada depende dos valores de i:

i	vezes (j)
1	2..n + 1 = n
2	3..n + 1 = n - 1
3	4..n + 1 = n - 2
...	...
i	n - i + 1

Portanto, a quantidade de vezes que a linha 3 é executa é

$$\sum_{i=1}^{n-1} n - i + 1$$

Notas

Exercícios - Ordenação por seleção

Análise do tempo de execução

- Portanto, a quantidade de vezes que a linha 3 é executa é

$$\begin{aligned}
\sum_{i=1}^{n-1} n - i &= \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} 1 \\
&= n(n-1) - \frac{(n-1)((n-1)+1)}{2} + (n-1) \\
&= n(n-1) - \frac{(n-1)n}{2} + (n-1) \\
&= \frac{n(n-1)}{2} + (n-1) \\
&= \frac{1}{2}n^2 + \frac{1}{2}n - 1 \\
&= \Theta(n^2)
\end{aligned}$$

Notas

Exercícios - Ordenação por seleção

Análise do tempo de execução

- A linha 4 é executada $n-1$ vezes a menos que a linha 3, e portanto também é executada $\Theta(n^2)$ vezes.
- A quantidade de vezes que a linha 5 é executada depende dos valores de entrada. No melhor caso o vetor está ordenado e a linha 5 é executada 0 vezes. No pior caso o vetor está na ordem inversa e a linha 5 é executada o mesmo número de vezes da linha 4, ou seja, $\Theta(n^2)$ vezes.

Notas

Exercícios - Ordenação por seleção

Análise do tempo de execução

- Considerando que a execução de cada linha isolada demora um tempo constante, somaremos a quantidade de vezes que cada linha é executada para obter o tempo total de execução do algoritmo:
  - Melhor caso:
    $\Theta(n) + \Theta(n) + \Theta(n^2) + \Theta(n^2) + 0 + \Theta(n) = \Theta(n^2)$
  - Pior caso:
    $\Theta(n) + \Theta(n) + \Theta(n^2) + \Theta(n^2) + \Theta(n^2) + \Theta(n) = \Theta(n^2)$
  - Não existe diferença entre os tempos de execução do melhor e do pior caso.

Notas

Exercícios

2.2-3 Considere mais uma vez a pesquisa linear (ver exercício 2.1-3). Quantos elementos da sequência de entrada precisam ser verificados em média, supondo-se que o elemento que está sendo procurado tenha a mesma probabilidade de ser qualquer elemento do arranjo? E no pior caso? Quais são os tempos de execução do caso médio e do pior caso da pesquisa linear em notação Θ? Justifique suas respostas.

2.1-3 Considere o **problema de pesquisa**:
Entrada: Uma sequência de n números $A = \langle a_1, a_2, \ldots, a_n \rangle$ e um valor v.
Saída: Um índice i tal que $v = A[i]$ ou o valor especial `NIL`, se v não aparecer em A.

Notas

Exercícios

```
pesquisa-linear(A, v)
1 i = 1
2 while i <= n
3   if A[i] == v
4     return i
5   i = i + 1
6 return NIL
```

Análise do tempo de execução - caso médio

- A probabilidade do elemento estar em cada posição é $\frac{1}{n}$
- Se o elemento está na posição 1, um elemento precisa ser verificado. Se o elemento está na posição 2, dois elementos precisam ser verificados. Se o elemento está na posição i, i elementos precisam ser verificados.
- Temos que fazer a média ponderada da quantidade de verificações:
  $\frac{1}{n}1 + \frac{1}{n}2 + \frac{1}{n}3 + \cdots + \frac{1}{n}n = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n}\frac{n(n+1)}{2} = \frac{n+1}{2} = \Theta(n)$

Notas

Exercícios

```
pesquisa-linear(A, v)
1 i = 1
2 while i <= n
3   if A[i] == v
4     return i
5   i = i + 1
6 return NIL
```

Análise do tempo de execução - pior caso

- O elemento é o último
- Neste caso, são necessárias n comparações
- O tempo de execução é $\Theta(n)$

Notas

Exercícios

2.2-4 Como podemos modificar praticamente qualquer algoritmo para ter um bom tempo de execução no melhor caso?

2.3-5 Voltando ao problema de pesquisa (ver exercício 2.1-3) observe que, se a sequência A estiver ordenada, podemos comparar o ponto médio da sequência com v e eliminar metade da sequência de consideração posterior. A **pesquisa binária** é um algoritmo que repete este procedimento, dividindo ao meio o tamanho da porção restante da sequência a cada vez. Escreva o pseudocódigo, sendo ele iterativo ou recursivo, para a pesquisa binária. Demostre que o tempo de execução do pior caso da busca binária é $\Theta(\ln n)$.

Notas

Exercícios

```
pesquisa-binaria(A, v)
 1 inicio = 1
 2 fim = n
 3 meio = (inicio + fim) / 2
 4 while v != A[meio] and fim >= inicio
 5   if v < A[meio]
 6     fim = meio - 1
 7   else
 8     inicio = meio + 1
 9   meio = (inicio + fim) / 2
10 if v == A[meio]
11   return meio
12 return NIL
```


Notas

Exercícios

2.3-6 Observe que o laço while das linhas 5 a 7 do procedimento `insertion-sort` na seção 2.1 utiliza uma pesquisa linear para varrer (no sentido inverso) o subarranjo ordenado $A[1..j-1]$. Podemos usar em vez disso uma pesquisa binária (ver exercício 2.3-5) para melhorar o tempo de execução global do pior caso da ordenação por inserção para $\Theta(n \lg n)$?

```
insertion-sort(A)
1 for j = 2 to A.length
2     chave = A[j]
3     # insere A[j] na sequência ordenada A[1..j-1]
4     i = j - 1
5     while i > 0 and A[i] > chave
6         A[i + 1] = A[i]
7         i = i - 1
8     A[i + 1] = chave
```


Notas

Exercícios

2.3-7 Descreva um algoritmo de tempo $\Theta(n \lg n)$ que, dado um conjunto S de n inteiros e outro inteiro x, determine se existem ou não dois elementos em S cuja soma seja exatamente x.

Notas

Monotonicidade

- Uma função $f(n)$ é **monotonicamente crescente** se $m \leq n$ implica $f(m) \leq f(n)$.
- Uma função $f(n)$ é **monotonicamente decrescente** se $m \leq n$ implica $f(m) \geq f(n)$.
- Uma função $f(n)$ é **estritamente crescente** se $m < n$ implica $f(m) < f(n)$.
- Uma função $f(n)$ é **estritamente decrescente** se $m < n$ implica $f(m) > f(n)$.

Notas

Pisos e tetos

Para qualquer número real x,

- denotamos o maior inteiro menor que ou igual a x por $\lfloor x \rfloor$ (lê-se "o piso de x")
- denotamos o menor inteiro maior que ou igual a x por $\lceil x \rceil$ (lê-se "o teto de x")

A função piso e a função teto são monotonicamente crescente.
Para todo real x,

- $x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$

Para qualquer inteiro n,

- $\lfloor n/2 \rfloor + \lceil n/2 \rceil = n$

Notas

Aritmética modular

Para qualquer inteiro a e qualquer inteiro positivo n, o valor a mod n é o **resto** do quociente a/n:

- $a \bmod n = a - \lfloor a/n \rfloor n$.

Se $(a \bmod n) = (b \bmod n)$, escrevemos $a \equiv b \pmod{n}$ e dizemos que a é equivalente a b módulo n. $a \equiv b \pmod{n}$ se e somente se n é divisor de $b - a$.
Escrevemos $a \not\equiv b \pmod{n}$ se a não é equivalente a b, módulo n.

Notas

Polinômios

Dado um inteiro não negativo d, um **polinômio em n de grau d** é uma função $p(n)$ da forma:

- $p(n) = \sum_{i=0}^{d} a_i n^i$

onde as constantes $a_0, a_1, \ldots, a_d$ são os **coeficientes** do polinômio e $a_d \neq 0$.
Para um polinômio assintoticamente positivo $p(n)$ de grau d, temos $p(n) = \Theta(n^d)$.
Dizemos que uma função $f(n)$ é **polinomialmente limitada** se $f(n) = O(n^k)$ para alguma constante k.

Notas

Exponenciais

Para todos os valores $a \neq 0$, m e n reais, temos as seguintes identidades:

- $a^0 = 1$
- $a^1 = a$
- $a^{-1} = 1/a$
- $(a^m)^n = a^{mn}$
- $(a^m)^n = (a^n)^m$
- $a^m a^n = a^{m+n}$

Para todo n e $a \geq 1$, a função a^n é monotonicamente crescente em n.

Notas

Exponenciais

As taxas de crescimento de polinômios e exponenciais podem ser relacionadas:

- Temos que $\lim_{n\to\infty} \frac{n^b}{a^n} = 0$, onde $a > 1$ e b são constantes reais

e portanto, concluímos que $n^b = o(a^n)$

Notas

Logaritmos

Notações utilizadas para logaritmos:

- $\lg n = \log_2 n$
- $\ln n = \log_e n$
- $\lg^k n = (\lg n)^k$
- $\lg \lg n = \lg(\lg n)$

Convenção: $\lg n + k$ significa $(\lg n) + k$ e não $\lg(n + k)$.
Se $b > 1$ constante, então para $n > 0$, a função $\log_b n$ é estritamente crescente.

Notas

Logaritmos

Para todo $a > 0$, $b > 0$, $c > 0$ e n real,

- $a = b^{\log_b a}$
- $\log_c(ab) = \log_c a + \log_c b$
- $\log_b a^n = n \log_b a$
- $\log_b a = \frac{\log_c a}{\log_c b}$
- $\log_b(1/a) = -\log_b a$
- $\log_c(a/b) = \log_c a - \log_c b$
- $\log_b a = \frac{1}{\log_a b}$
- $a^{\log_b c} = c^{\log_b a}$

onde as bases dos logaritmos não são iguais a 1.

Notas

Logaritmos

Dizemos que uma função $f(n)$ é **polilogaritimicamente limitada** se $f(n) = O(\lg^k n)$ para alguma constante k.
Qualquer função polinomial positiva cresce mais rapidamente que qualquer função polilogaritmica, ou seja:

- $\lg^b n = o(n^a)$, para constantes reais $a > 0$ e b

Notas

Fatorial

A notação $n!$ (lê-se "n fatorial") é definida para inteiros $n \geq 0$ como

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n \cdot (n-1)! & \text{se } n > 0 \end{cases}$$

Um limite superior fraco para a função fatorial é $n! \leq n^n$.
A aproximação de Stirling $n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1 + \Theta\left(\frac{1}{n}\right)\right)$, onde e é a base do logaritmo natural, nós dá um limite superior e inferior mais restrito:

- $n! = o(n^n)$
- $n! = \omega(2^n)$
- $\lg(n!) = \Theta(n \lg n)$

Notas

Exercícios

3.1-1 Sejam $f(n)$ e $g(n)$ funções assintoticamente não negativas. Usando a definição básica da notação Θ, prove que $\max(f(n), g(n)) = \Theta(f(n) + g(n))$.

Prova

Pela definição da notação Θ, para provar que $\max(f(n), g(n)) = \Theta(f(n) + g(n))$, temos que encontrar constantes c_1, c_2 e $n_0 > 0$ tal que , $0 \leq c_1(f(n) + g(n)) \leq \max(f(n), g(n)) \leq c_2(f(n) + g(n))$ para todo $n \geq n_0$.
Como as funções $f(n)$ e $g(n)$ são assintoticamente não negativas, para $n \geq n_0$ (para algum n_0), temos $f(n) + g(n) \geq f(n) \geq 0$ e $f(n) + g(n) \geq g(n) \geq 0$. Como para um valor qualquer de n, $\max(f(n), g(n)) = f(n)$ ou $\max(f(n), g(n)) = g(n)$, temos que $f(n) + g(n) \geq \max(f(n), g(n)) \geq 0$. O que mostra que $\max(f(n), g(n)) \leq c_2(f(n) + g(n))$, para $c_2 = 1$.

Notas

Exercícios

Prova
Como para um valor qualquer de n, $\max(f(n), g(n)) = f(n)$ ou $\max(f(n), g(n)) = g(n)$, temos que $0 \leq f(n) \leq \max(f(n), g(n))$e $0 \leq g(n) \leq \max(f(n), g(n))$, somando as duas inequações, obtemos $0 \leq f(n) + g(n) \leq 2\max(f(n), g(n))$, que pode ser escrito como $0 \leq \frac{f(n)+g(n)}{2} \leq \max(f(n), g(n))$. O que mostra que $0 \leq c_1(f(n) + g(n)) \leq \max(f(n), g(n))$ para $c_1 = \frac{1}{2}$.
Como existem as constantes c_1, c_2 e n_0, mostramos que $\max(f(n), g(n)) = \Theta(f(n) + g(n))$. □

Notas

Exercícios

3.1-2 Mostre que, para quaisquer constantes reais a e b, onde $b > 0$, $(n + a)^b = \Theta(n^b)$

Prova
Pela definição da notação Θ, para provar que $(n + a)^b = \Theta(n^b)$, temos que encontrar constantes c_1, c_2 e $n_0 > 0$ tal que , $0 \leq c_1 n^b \leq (n + a)^b \leq c_2 n^b$ para todo $n \geq n_0$.
Vamos achar um limite superior para $n + a$. Temos que

$$\begin{aligned} n + a &\leq n + |a| \\ &\leq 2n \qquad \text{quando } |a| \leq n \end{aligned}$$

De forma semelhante, vamos achar um limite inferior para $n + a$. Temos que

$$\begin{aligned} n + a &\geq n - |a| \\ &\geq \frac{1}{2}n \qquad \text{quando } |a| \leq \frac{1}{2}n \end{aligned}$$

Portanto, quando $n \geq 2|a|$, $0 \leq \frac{1}{2}n \leq n + a \leq 2n$.

Notas

Exercícios

Como $b > 0$, a inequação se mantém verdadeira quando todas as partes são elevadas a b:

$$0 \leq \left(\frac{1}{2}n\right)^b \leq (n + a)^b \leq (2n)^b$$

$$0 \leq \left(\frac{1}{2}\right)^b n^b \leq (n + a)^b \leq 2^b n^b$$

Tomando as contantes $c_1 = \left(\frac{1}{2}\right)^b$ e $c_2 = 2^b$, e $n_0 = 2|a|$ mostramos que $(n + a)^b = \Theta(n^b)$. □

Notas

Exercícios

3.1-3 Explique por que a declaração "o tempo de execução do algoritmo A é no mínimo $O(n^2)$" é isenta de significado.

Notas

Exercícios

3.1-4 É verdade que $2^{n+1} = O(2^n)$? É verdade que $2^{2n} = O(2^n)$?

Vamos tentar mostrar que $2^{n+1} = O(2^n)$

Pela definição da notação O, para mostrar que $2^{n+1} = O(2^n)$, temos que encontrar constantes positivas c e n_0 tais que $0 \leq 2^{n+1} \leq c2^n$ para todo $n \geq n_0$. Temos

$$\begin{aligned} 2^{n+1} &\leq c2^n \\ 2^n \cdot 2^1 &\leq c2^n \\ 2 &\leq \frac{c2^n}{2^n} \\ c &\geq 2 \end{aligned}$$

Tomando $c = 2$ e $n_0 = 1$, temos que $0 \leq 2^{n+1} \leq c2^n$ para todo $n \geq n_0$, portanto $2^{n+1} = O(2^n)$. □

Notas

Exercícios

3.1-4 É verdade que $2^{n+1} = O(2^n)$? É verdade que $2^{2n} = O(2^n)$?

Vamos tentar mostrar que $2^{2n} = O(2^n)$

Pela definição da notação O, para mostrar que $2^{2n} = O(2^n)$, temos que encontrar constantes positivas c e n_0 tais que $0 \leq 2^{2n} \leq c2^n$ para todo $n \geq n_0$. Temos

$$\begin{aligned} 2^{2n} &\leq c2^n \\ 2^n \cdot 2^n &\leq c2^n \\ 2^n &\leq c \end{aligned}$$

Como não podemos definir um valor de c tal que $2^n \leq c$ para $n \geq n_0$, concluímos que $2^{2n} \neq O(2^n)$. □

Notas

Referências

- Thomas H. Cormen et al. Introdução a Algoritmos. 2ª edição em português. Capítulo 3.

Notas

Notas