Estrutura de Dados I

Ivairton M. Santos

UFMT
Ciência da Computação

November 14, 2016

Outline

Estrutura de Dados I - Introdução

- O que é um Algoritmo?
- O que é uma estrutura de dados?
- Porque pensar sobre isso?

Análise de algoritmos

- Porque analisar um algoritmo?
- O que podemos analisar?
- Um modelo de computador:
  Axioma 1: Tempo de recuperação e armazenamento na memória são constantes: π_{rec}, π_{arm}
  Para o código: $y = x \rightarrow \pi_{rec} + \pi_{arm} = 2$
  $y = 1 \rightarrow \pi_{rec} + \pi_{arm} = 2$
  Axioma 2: Operações aritméticas elementares possuem tempos constantes: $\pi_+, \pi_-, \pi_\times, \pi_\div, \pi_<$, *etc*...
  $y = y + 1 \rightarrow 2\pi_{rec} + \pi_+ + \pi_{arm} = 4$
  Axioma 3: A chamada de uma função e o retornon dela também são constantes: π_{call}, π_{return}

Análise de algoritmos

Código:

```
int resultado = 0;
for (i=1; i<=n; i++) {
            resultado = resultado + 1;
}
return resultado;
```

Resulta em:

$\pi_{rec} + \pi_{arm}$

$\pi_{rec} + \pi_{arm} + \{(2\pi_{rec} + \pi_{<=}) \times (n+1)\} + \{(2\pi_{rec} + \pi_{+} + \pi_{arm}) \times n\}$

$(2\pi_{rec} + \pi_{+} + \pi_{arm}) \times n$

$\pi_{rec} + \pi_{return}$

$= (5\pi_{rec} + 2\pi_{arm} + \pi_{\leq} + \pi_{return}) + (6\pi_{rec} + 2\pi_{arm} + \pi_{\leq} + 2\pi_{+}) \times n$

$T(n)$ é o tempo de processamento do programa, que é $t_1 + t_2 n$

Exercício

- Calcule o tempo para um programa que soma os primeiros n inteiros positivos pares.

Análise de algoritmos

Axioma 4: O tempo para cálculo do endereço de vetor é constante:

$\pi_{[.]}$

$y = a[i] \rightarrow 3\pi_{rec} + \pi_{[.]} + \pi_{arm}$

Calcule $T(n)$ para o algoritmo de Horner ($\Sigma_{i=0}^{n} a_i n^i$):

```
int resultado = a[n];
for (i=n-1; i<=0; i--) {
   resultado = resultado * x + a[i];
}
return resultado;
```

Análise de métodos recursivos

- Analisando a performance do algoritmo recursivo para cálculo do fatorial, considere:

$$n! = \left\{ \begin{array}{ll} 1 & n = 0 \\ \prod_{i=0}^{n} & n > 0 \end{array} \right\}$$

- Recursivamente temos:

$$n! = \left\{ \begin{array}{ll} 1 & n = 0 \\ n \times (n-1)! & n > 0 \end{array} \right\}$$

Análise de métodos recursivos

```
int fatorial(int n) {
        if (n == 0)
                return 1;
        else
                return n * fatorial(n-1);
}
```

Temos para $n = 0$:

$2\pi_{rec} + \pi_{=}$

$\pi_{rec} + \pi_{return}$

Temos para $n > 0$:

$2\pi_{rec} + \pi_{=}$

$3\pi_{rec} + \pi_{-} + \pi_{arm} + \pi_{*} + \pi_{call} + \pi_{return} + T(n-1)$

$$T(n) = \left\{ \begin{array}{ll} t_1 & n = 0 \\ T(n-1) + t_2 & n > 0 \end{array} \right\}$$ (Relação de recorrência)

Análise de métodos recursivos

Resolvendo a relação de recorrência:

$$\begin{aligned} T(n) &= T(n-1) + t_2 \\ &= (T(n-2) + t_2) + t_2 \\ &= T(n-2) + 2t_2 \\ &= (T(n-3) + t_2) + 2t_2 \\ &= T(n-3) + 3t_2 \\ &(...) \end{aligned}$$

Identificando o padrão emergente, neste caso é óbvio:

$T(n) = T(n-k) + kt_2$

Modelo simplificado

- Podemos facilitar a análise, apesar de perder precisão.
- Considere uma constante de tempo para todos os parâmetros, por conta dos ciclos.
- Vamos retomar os exemplos anteriores para ver como fica...

Problemas

Determine o tempo de processamento preditos pelo modelo detalhado e modelo simplificado para os seguintes fragmentos de código:

(a)

```
for (i=0; i<n; i++)
        k++;
```

(b)

```
for (i=n-1; i!=0; i--)
        k++;
```

(c)

```
for (i=0; i<n; i++)
        if (i\%2 == 0)
            k++;
```

(d)

```
for (i=0; i<n; i++)
        for (j=0; j<n; j++)
```

Notação assintótica

- Porque fazer algoritmos eficientes?

$T(n)$	$1.000s$	$10.000s$	Ganho
$100n$	10	100	$10\times$
$5n^2$	14	45	$3,2\times$
$0,5n^3$	12	27	$2,3\times$
2^n	10	18	$1,3\times$

Notação assintótica

Suponha 2 algoritmos A e B. Se temos a análise dos tempos de processamento $T_A(n)$ e $T_B(n)$, em que n é uma medida do tamanho do problema, então basta comparar os resultados das funções para determinar qual deles é melhor.

Infelizmente não é assim tão simples...

Notação assintótica

Temos a possibilidade de conhecermos, **a priori**, o tamanho do problema. Por exemplo: para n_0 e $T_A(n_0) < T_B(n_0)$, então o algoritmo A é melhor para o tamanho de problema n_0.

No caso geral não temos o conhecimento, a priori, do tamanho do problema. Se pudéssemos demonstrar que $T_A(n) \leq T_B(n)$ para qualquer $n \geq 0$, então o algoritmo A seria melhor que o algoritmo B, independente do tamanho do problema.

Notação assintótica

Infelizmente, no geral, não temos conhecimento anterior do tamanho do problema e também não é verdade que uma das funções seja menor ou iguala outra para qualquer que seja o tamanho do problema.

Para resolver esse impasse, consideramos o comportamento **assintótico** das funções, para tamanho de problemas arbitrariamente grandes.

A notação Θ caracteriza o comportamento assintótico de uma função, estabelecendo um limite superior quanto à taxa de crescimento da função em relação ao crescimento de n.

Notação Θ

Considere uma função $f(n)$ não negativa para todos os inteiros $n \geq 0$. Dizemos que "$f(n)$ é $\Theta(g(n))$" e escrevemos $f(n) = \Theta(g(n))$, se existe um inteiro n_0 e uma constante $c > 0$ tais que para todo inteiro $n > n_0$, $f(n) \leq cg(n)$.

Exemplo: Considere $f(n) = 8n + 128$. Queremos mostrar que $f(n)$ é $\Theta(n^2)$. Portanto é necessário encontrar um inteiro n_0 e uma constante $c > 0$ tais que para todo $n > n_0$, $f(n) \leq cn^2$.

Notação Θ

Suponha escolher $c = 1$, então:

$$f(g) < cn^2 \rightarrow \quad 8n + 128 \leq n^2$$
$$0 \leq n^2 - 8n - 128$$
$$0 \leq (n - 16)(n + 8)$$

Uma vez que $(n + 8) > 0 \forall$ os valores de $n > 0$, concluímos que $(n_0 - 16) \geq 0$, ou seja, $n_0 = 16$.

Assim, para $c = 1$ e $n_0 = 16$, $f(n) \leq cn^2 \forall$ os inteiros $n > n_0$. Portanto, $f(n) = \Theta(n^2)$.

Propriedades de Θ

- Quanto ao comportamento assintótico da soma: Se $f_1(n) = \Theta(g_1(n))$ e $f_2(n) = \Theta(g_2(n))$, então: $f_1(n) + f_2(n) = \Theta(max(g_1(n), g_2(n)))$
- Quanto ao comportamento assintótico da multiplicação: Se $f_1(n) = \Theta(g_1(n))$ e $f_2(n) = \Theta(g_2(n))$, então: $f_1(n) \times f_2(n) = \Theta(g_1(n) \times g_2(n))$

Convenções para as expressões de Θ

- É prática comum escrever a expressão de Θ sem os termos menos significativos. Assim, ao invéz de $\Theta(n^2 + nlogn + n)$, escrevemos simplesmente $\Theta(n^2)$.
- É prática comum desconsiderar os coeficientes constantes. Em lugar de $\Theta(3n^2)$, simplesmente $\Theta(n^2)$. Um caso especial dessa regra é se a função é constante, por exemplo: $\Theta(1024)$, fica simplesmente $\Theta(1)$.

Algumas funções são tão frequentes que recebem denominações:

$\Theta(1)$	Constante	$\Theta(nlogn)$	$nlog$ de n
$\Theta(logn)$	Logarítmica	$\Theta(n^2)$	Quadrática
$\Theta(log^2 n)$	Log quadrado	$\Theta(n^3)$	Cúbica
$\Theta(n)$	Linear	$\Theta(2^n)$	Exponencial

Análise assintótica de algoritmos

Como vimos, a análise detalhada de algoritmos é confusa e cansativa. Podemos então aplciar a análise assintótica. Exemplo:

```
int func_Horner(int a[], int n, int x) {
    int resultado = a[n];
    for (int i = n-1; i >= 0; i--)
        resultado = resultado * x + a[i];
}
```

$$
\begin{aligned}
5 &\rightarrow \Theta(1) \\
4 &\rightarrow \Theta(1) \\
3n+3 &\rightarrow \Theta(n) \\
4n &\rightarrow \Theta(n) \\
9n &\rightarrow \Theta(n) \\
2 &\rightarrow \Theta(1)
\end{aligned}
\qquad 16n + 14 \rightarrow \Theta(n)
$$