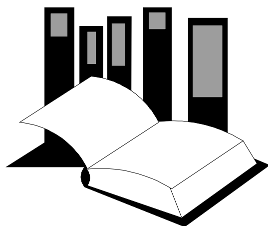
Análise de Algoritmos



- A Análise de Algoritmos é um campo da Ciência da Computação que tem como objetivo o entendimento da complexidade dos algoritmos
- O objetivo desta aula consiste em desenvolver as habilidades de fazer julgamentos elementares da avaliação dos programas

Prof. Dr. José Augusto Baranauskas
DFM-FFCLRP-USP

1

Análise de Algoritmos

- Existem muitos critérios com os quais podemos julgar um programa, por exemplo:
  - Será que ele faz o que se espera que ele faça?
  - Será que ele funciona corretamente de acordo com as especificações?
  - Existe documentação explicando como usá-lo e como ele trabalha?
  - O código está legível?
- Os critérios acima são muito importantes quando se escreve software, especialmente para grandes sistemas.

2

Análise de Algoritmos

- Existem ainda outros critérios diretamente relacionados com o desempenho:
  - tempo de computação e
  - requisitos de memória
- A avaliação de desempenho pode ser liberalmente dividida em 2 partes:
  - (a) estimativas precedentes e
  - (b) testes posteriores

3

Análise de Algoritmos

- Considere inicialmente uma estimativa precedente. Suponha que em algum ponto do seu programa encontra-se a instrução

x = x + 1;

- Gostaríamos de determinar dois valores para esta instrução
  - a duração de tempo para uma única execução;
  - a quantidade de vezes que ela é executada.
- O produto desses valores será o tempo total tomado por esta instrução.
- A segunda estatística é chamada de *contagem de freqüência* e varia de um conjunto de dados para outro.

4

Análise de Algoritmos

- Uma das tarefas mais difíceis, em estimativa de contagem de freqüência, é a seleção adequada de amostras de dados
- Será impossível determinar exatamente quanto tempo levará a execução de qualquer comando, a menos que tenhamos as informações seguintes:
  - máquina onde a instrução será executada;
  - conjunto de instruções da linguagem da máquina;
  - os tempos necessários para cada instrução da máquina;
  - a tradução que um compilador fará do código fonte para a linguagem da máquina.
- Assim, é possível determinar esses valores escolhendo uma máquina real e um compilador existente

5

Análise de Algoritmos

- Outra alternativa consiste em definir um computador hipotético (com um tempo de execução imaginário), porém mantendo os tempos razoavelmente próximos dos equipamentos existentes, para que os valores resultantes sejam representativos
- Nenhuma dessas alternativas mostra-se adequada, pois em ambos os casos o tempo determinado provavelmente não se aplicará a muitos computadores
- Também a variação do compilador de uma máquina para outra representará um problema

6

Análise de Algoritmos

- Todas essas considerações conduzem-nos a limitar nossos objetivos em uma análise *a priori.*
- Vamos nos concentrar apenas no desenvolvimento da contagem de freqüência para todas as instruções
- Considere os seguintes três exemplos:

```
(a)
x = x + 1;

(b)
for(i=1; i <= n; i++)
  x = x + 1;

(c)
for(i=1; i <= n; i++)
  for(j=1; j <= n; j++)
    x = x + 1;
```

7

Análise de Algoritmos

```
(a)
x = x + 1;

(b)
for(i=1; i <= n; i++)
  x = x + 1;

(c)
for(i=1; i <= n; i++)
  for(j=1; j <= n; j++)
    x = x + 1;
```

- No programa (a) assumimos que a instrução **x = x + 1** não está incluída dentro de qualquer laço explícito ou implícito Neste caso, a contagem de freqüência é **1** (um)
- No programa (b) a mesma instrução será executada **n** vezes
- No prograrna (c) $\mathbf{n^2}$ vezes (assumindo **n** > 1)
- Os valores 1, **n** e $\mathbf{n^2}$ são as *ordens de grandeza*

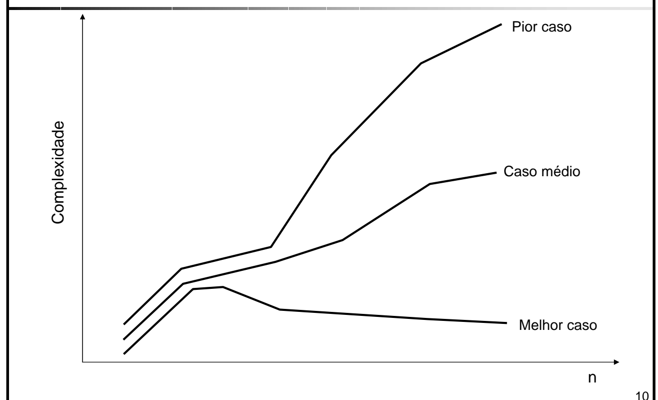
8

Complexidade nos Casos: Pior, Melhor e Médio

- Em geral, na análise de algoritmos são avaliadas as situações (para uma entrada de tamanho **n**) da complexidade do:
  - pior caso do algoritmo (*worst case*) que é uma função definida pelo número máximo de passos utilizados;
  - caso médio do algoritmo (*average case*) que é uma função definida pelo número médio de passos utilizados;
  - melhor caso do algoritmo (*best case*) que é uma função definida pelo número mínimo de passos utilizados;
- Na análise de execução, a preocupação principal consiste na determinação de ordem de grandeza de um algoritmo (pior caso)

9

Complexidade nos Casos: Pior, Melhor e Médio



10

Análise de Algoritmos

- Para determinar a ordem de grandeza, usa-se freqüentemente as fórmulas como:

$$\sum_{i=1}^{n} 1 = n; \qquad \sum_{i=1}^{n} i = \frac{n(n+1)}{2}; \qquad \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

- no segmento do programa (c) anterior a instrução **x = x + 1** é executada

```
for(i=1; i <= n; i++)
  for(j=1; j <= n; j++)
    x = x + 1;
```

$$\sum_{i=1}^{n}\sum_{j=1}^{n} 1 = \sum_{i=1}^{n} n = n^2$$

- Em geral $\sum_{i=1}^{n} i^k = \frac{n^{k+1}}{k} + \text{termos de menor grau}, k \geq 0$

11

Exemplo

- Considere o algoritmo para cálculo do fatorial de um número inteiro **n** dado a seguir.

- Cada instrução é contada uma vez
- O tempo atual tomado por cada instrução naturalmente poderá variar
- A instrução **for** é na verdade uma combinação de diversas instruções, mas aqui vamos contá-la como uma
- Então, a contagem total é **2n + 1**, como é mostrado na tabela seguinte

```
1 int fatorial(int n)
2 { int i, produto;
3
4   produto = 1;
5   for(i = 2; i <= n; i++)
6     produto = produto * i;
7   return produto;
8 }
```

Linhas	n	Freqüência
4, 5, 7	0	3
4, 5, 7	1	3
4, 5, 6, 5, 7	2	3 + 2
4, 5, 6, 5, 6, 5, 7	3	3 + 4
4, 5, 6, 5, 6, 5, 6, 5, 7	4	3 + 6
	n	3 + 2(n-1) = 2n +1

12

Exemplo

- Considere o algoritmo para cálculo do fatorial de um número inteiro **n** dado a seguir.

- Cada instrução é contada uma vez.
- O tempo atual tomado por cada instrução naturalmente poderá variar
- A instrução **for** é na verdade uma combinação de diversas instruções, mas aqui vamos contá-la como uma
- Então, a contagem total é **2n + 1**, como é mostrado na tabela seguinte.

```
1 int fatorial(int n)
2 { int i, produto;
3
4   produto = 1;
5   for(i = 2; i <= n; i++)
6     produto = produto * i;
7   return produto;
8 }
```

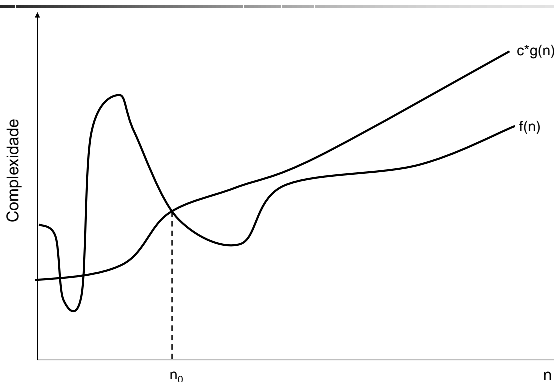
Linhas	n	Freqüência
4, 5, 7	0	3
4, 5, 7	1	3
4, 5, 6, 5, 7	2	3 + 2
4, 5, 6, 5, 6, 5, 7	3	3 + 4
4, 5, 6, 5, 6, 5, 6, 5, 7	4	3 + 6
	n	3 + 2(n-1) = 2n +1

Freqüentemente vamos denotar isso como **O(n)** ignorando as constantes

Notação "O"

- *Definição*: **O(g(n)) = {f(n) :** existem duas constantes **c** e $\mathbf{n_0}$ tais que **|f(n)|** <= **|c*g(n)|**, para todo **n** > $\mathbf{n_0}$}
- Como, normalmente, é difícil determinar com exatidão f(n), a notação "O" é utilizada
- Assim, a notação "O" fornece um limite superior para uma função dentro de um fator constante

Notação "O"



Notação "O"

- Dado um algoritmo, analisamos a contagem de freqüência para cada instrução e somamos todas
- Isto normalmente resulta em um polinômio do tipo (c_k != 0)
  $$P(n) = c_k n^k + c_{k-1} n^{k-1} + ... + c_1 n + c_0$$
- Usando a notação "**O**"
  $$P(n) = c_k n^k + c_{k-1} n^{k-1} + ... + c_1 n + c_0 \equiv O(n^k)$$
- Por outro lado, se qualquer passo for executado 2^n vezes ou mais a expressão será (c != 0)
  $$c2^n + P(n) = c2^n + c_k n^k + c_{k-1} n^{k-1} + ... + c_1 n + c_0 \equiv O(2^n)$$
- Assim, na notação "O", considera-se apenas o termo de maior ordem

Notação "O"

- A notação
  - O(1) indica que o tempo de computação é constante, independentemente de qualquer fator
  - O(n) chama-se linear
  - O(n^2) chama-se quadrática
  - O(n^3) chama-se cúbica
  - O(2^n) chama-se exponencial
- Os tempos de computação O(1), O(log n), O(n), O(n log n), O(n^2), O(n^3) e O(2^n) são aqueles comumente encontrados e os quais vamos trabalhar no decorrer do curso, e log **n** é, normalmente, o logaritmo de **n** na base 2.

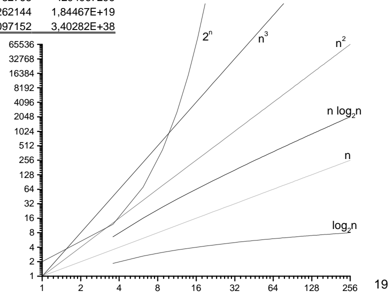
Notação "O"

- Possuindo dois algoritmos para desempenhar a mesma tarefa, sendo o primeiro com um tempo de computação O(n) e o segundo O(n^2), geralmente considera-se o primeiro como superior
- Isso porque à medida que **n** aumenta, o tempo de processamento do segundo algoritmo vai piorar muito comparando com o tempo do primeiro

Notação "O"

$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	4294967296
6	64	384	4096	262144	1,84467E+19
7	128	896	16384	2097152	3,40282E+38

Observe como os tempos O(log n) e O(n log n) crescem muito mais devagar do que os outros.
São geralmente impraticáveis, para grandes conjuntos de dados, algoritmos com uma complexidade superior a O(n log n).
Um algoritmo que é exponencial vai funcionar apenas com entradas Muito pequenas.



Exercício

- A Regra de Horner é um método que fornece os meios para a avaliação de um polinômio

$$A(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

- no ponto $\mathbf{x_0}$ usando uma quantidade mínima de multiplicações. A regra é:

$$A(x) = (\cdots((a_n x_0 + a_{n-1}) x_0 + \cdots + a_1) x_0 + a_0$$

- Escreva uma função para avaliar um polinômio, usando a Regra de Horner. Determine quantas vezes cada instrução será executada. Qual a ordem de grandeza da complexidade na notação "**O**"?

Solução

```
1 y = 0;
2 for(i=N; i>=0; i--)
3   y = a[i] + x*y;
```


Solução

```
1 y = 0;
2 for(i=N; i>=0; i--)
3   y = a[i] + x*y;
```

Linhas	n	Freqüência
1,2,3,2	0	1 + 2 + 1
1,2,3,2,3,2	1	1 + 4 + 1
1,2,3,2,3,2,3,2	2	1 + 6 + 1
	n	1 + 2(n+1) + 1 = 2n + 4 = O(n)

Solução

```
1 float horner(int N, float a[], float x)
2 { int i;
3   float y;
4
5   y = 0;
6   for(i=N; i>=0; i--)
7     y = a[i] + x*y;
8   return y;
9 }
```

Linhas	n	Freqüência
5,6,7,6,8	0	1 + 2 + 2
5,6,7,6,7,6,8	1	1 + 4 + 2
5,6,7,6,7,6,7,6,8	2	1 + 6 + 2
	n	1 + 2(n+1) + 2 = 2n + 5 = O(n)

Exercício

- Qual a freqüência das instruções e a ordem de grandeza da complexidade na notação "**O**" de um procedimento que multiplica duas matrizes quadradas $A_{n,n}$ e $B_{n,n}$? E para duas matrizes $A_{n,m}$ e $B_{m,r}$?

Solução $A_{n,n}$ e $B_{n,n}$

```
const int Max = 1000;
  .
  .
  .
1 void multMatriz(int n, float a[][Max], float b[][Max])
2 { int i,j,k;
3
4  for(i=1; i<=n; i++)
5    for(j=1; j<=n; j++)
6    { c[i][j] = 0;
7      for(k=1; k<=n; k++)
8        c[i][j] = c[i][j]+a[i][k]*b[k][j];
9    }
10}
```


Solução $A_{n,n}$ e $B_{n,n}$

```
const int Max = 1000;
  .
  .
  .
1 void multMatriz(int n, float a[][
2 { int i,j,k;
3
4  for(i=1; i<=n; i++)
5    for(j=1; j<=n; j++)
6    { c[i][j] = 0;
7      for(k=1; k<=n; k++)
8        c[i][j] = c[i][j]+a[i][k]*b[k][j];
9    }
10}
```

Linha	Freqüência
4	n+1
5	n*(n+1)
6	n*n
7	n*n*(n+1)
8	n*n*n
Total	$2n^3+3n^2+2n+1=O(n^3)$

Solução $A_{n,m}$ e $B_{m,r}$

```
const int Max = 1000;
  .
  .
  .
1 void multMatriz(int n, float a[][Max], float b[][Max])
2 { int i,j,k;
3
4  for(i=1; i<=n; i++)
5    for(j=1; j<=r; j++)
6    { c[i][j] = 0;
7      for(k=1; k<=m; k++)
8        c[i][j] = c[i][j]+a[i][k]*b[k][j];
9    }
10}
```


Solução $A_{n,m}$ e $B_{m,r}$

```
const int Max = 1000;
  .
  .
  .
1 void multMatriz(int n, float a
2 { int i,j,k;
3
4  for(i=1; i<=n; i++)
5    for(j=1; j<=r; j++)
6    { c[i][j] = 0;
7      for(k=1; k<=m; k++)
8        c[i][j] = c[i][j]+a[i][k]*b[k][j];
9    }
10}
```

Linha	Freqüência
4	n+1
5	n*(r+1)
6	n*r
7	n*r*(m+1)
8	n*r*m
Total	2nrm+3nr+2n+1=O(nrm)

Exercícios

- Dado um vetor **a** com **n** elementos, é possível encontrar um algoritmo com O(**n**) que:
  - encontra o maior valor de **a**?
  - encontra o menor valor de **a**?
- Qual a ordem de grandeza de um algoritmo que encontra a diferença entre o maior e o menor elementos de um vetor de **n** elementos?

Solução (maior de **a**)

```
1 float maior(int n, float a[])
2 { int i;
3   float x;
4
5   x = a[1];
6   for(i=2; i<=n; i++)
7     if(x < a[i])
8       x = a[i];
9   return x;
10}
```

Pior Caso	
Linha	Freqüência
5	1
6	n
7	(n-1)
8	(n-1)
9	1
Total	3n = O(n)

Solução (maior de **a**)

```
1 float maior(int n, float a[])
2 { int i;
3   float x;
4
5   x = a[1];
6   for(i=2; i<=n; i++)
7     if(x < a[i])
8       x = a[i];
9   return x;
10}
```

Melhor Caso	
Linha	Freqüência
5	1
6	n
7	(n-1)
8	0
9	1
Total	2n+1 = O(n)

Solução (maior de **a**)

```
1 float maior(int n, float a[])
2 { int i;
3   float x;
4
5   x = a[1];
6   for(i=2; i<=n; i++)
7     if(x < a[i])
8       x = a[i];
9   return x;
10}
```

Caso Médio	
Linha	Freqüência
5	1
6	n
7	(n-1)
8	(n-1)/2
9	1
Total	(5n+1)/2=O(n)

Solução (maior de **a**)

```
1 float maior(int n, float a[])
2 { int i;
3   float x;
4
5   x = a[1];
6   for(i=2; i<=n; i++)
7     if(x < a[i])
8       x = a[i];
9   return x;
10}
```

Resumo	
Pior Caso	3n
Caso Médio	(5n+1)/2
Melhor Caso	2n+1

Resumo

- A Análise de Algoritmos permite fazer julgamentos sobre a complexidade dos programas
- Dado um algoritmo, podemos estimar complexidades do pior caso, o melhor caso e o caso médio
- Nem sempre a análise de um algoritmo pode ser efetuada de forma exata => uso da notação “O”