Complexidade de Algoritmos

Jorge E. S. Souza

Algoritmos

“Uma seqüência bem definida de procedimentos computacionais (passos) que levam uma entrada a ser transformada em uma saída”

Cormen et al. 1991

Problema

A saída deve corresponder a uma resposta válida para o problema.

O tempo de execução deve ser finito.

Soluções

Um problema pode ser resolvido através de diversos algoritmos;

O fato de um algoritmo resolver um dado problema não significa que seja aceitável na prática.

Ordem	Método de Cramer	Método de Gauss
2	22us	50us
3	102us	159us
4	456us	353us
5	2.35ms	666us
10	1.19mim	4.95ms
20	15255 séculos	38.63ms

Qual a melhor solução?

Na maioria das vezes, a escolha de um algoritmo é feita através de critérios subjetivos como:

1) facilidade de compreensão, codificação e depuração;

2) eficiência na utilização dos recursos do computador e rapidez.

A análise de algoritmo fornece uma medida objetiva de desempenho proporcional ao tempo de execução do algoritmo.

Desempenho

A eficiência de um algoritmo pode ser medida através de seu tempo de execução ou através de funções que descrevem o seu tempo de execução.

Algorithm A

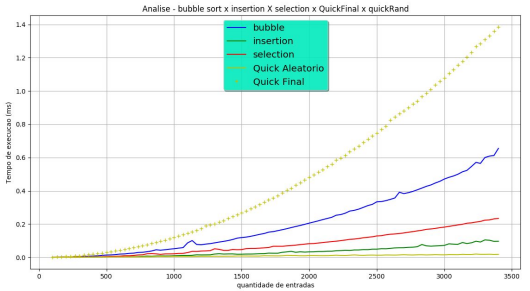
```
sum = 0
for i = 1 to n
  sum = sum + i
```

Algorithm B

```
sum = 0
for i = 1 to n
{
  for j = 1 to i
    sum = sum + 1
}
```

Algorithm C

```
sum = n * (n + 1) / 2
```



Eficiência

É a medida quantitativa inversa da quantidade de recursos (tempo de processamento, memória, etc) requeridos para a execução do algoritmo.

Quanto maior a eficiência menos recursos são gastos.

Como medir a eficiência de um algoritmo?

Métodos

Temos principalmente dois métodos de mensuração:

1) Experimental;

2) Analitico.

Método experimental

- ➔ Implementar diversos algoritmos.
- ➔ Executar um grande número de vezes.
- ➔ Analisar os resultados.

O tempo de execução não depende somente do algoritmo, mas do conjunto de instruções do computador, a qualidade do compilador, a habilidade do programador e etc..

Método experimental

- ➔ Usar variáveis de tempo.
- ➔ Função “*time*” do sistema operacional.

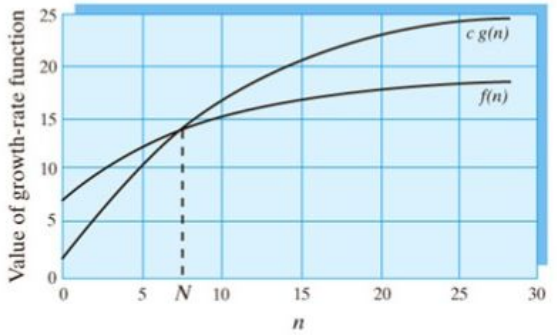
```
File Edit Options Buffers Tools Help
#!/usr/bin/perl
use Time::HiRes qw( time );

sub fib {
    my $n = shift;

    if ($n == 1 or $n == 2) {
        return 1
    }

    return (fib($n-1)+fib($n-2));
}

$inicio =  time();
$seq = fib($ARGV[0]);

print "$seq\n";

printf("Execution Time: %.4f s\n",  time() - $inicio);
```

```
[jorge@edb2]$ ./fib_r.pl 30
832040
Execution Time: 0.8173 s



[jorge@edb2]$ time ./fib_r.pl 30
832040
Execution Time: 0.8077 s

real    0m0,829s
user    0m0,822s
sys     0m0,005s
[jorge@edb2]$
```

Método analitico

➔ A ideia é encontrar funções matemáticas que descrevem o crescimento do tempo de execução dos algoritmos em relação ao tamanho da entrada.

➔ Comparar as funções.



Como expressar a eficiência de um algoritmo?

Através da ordem de crescimento do tempo de execução.

Uma forma simples é levar em conta apenas o termo de mais alta ordem.

Essa é uma forma simples de caracterização da eficiência que 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.
- Analisar um algoritmo significa prever os recursos (tempo) de que o algoritmo necessitará.

Vamos analisar um algoritmo:

	procedimento Busca Sequencial (A[1,..,n],x)
1.	i = 1
2.	Enquanto i ≤ n e A[i] ≠ x
3.	i = i+1
4.	Se i ≤ n retorne (i)
5.	Senão retorne (-1)

O que devemos olhar?

Operações:

1. Operações de Entrada e Saída;

2. Operações Aritméticas;

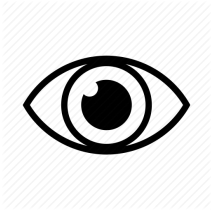
4. Movimentação de Dados entre os Componentes;

3. Operações Lógicas e Relacionais;

4. Atribuições, declarações de métodos e variáveis;

5. Estruturas de decisão e repetição;

6. Chamadas de métodos e etc..



Quanto custa?

- O esforço realizado por um algoritmo é calculado a partir da quantidade de vezes que a operação fundamental é executada.
- Para o algoritmo de busca, uma operação fundamental é a comparação entre elementos quando executada à busca.

Então:

- ➔ Vamos chamar o tamanho do array de n;
- ➔ Vamos contabilizar o custo de cada linha;
- ➔ Cada operação primitiva tem custo 1 (demora uma unidade de tempo);
- ➔ Contamos quantas vezes (no máximo) cada linha é executada;
- ➔ Somamos o custo total de cada linha.

Vamos mensurar:

			custo	vezes	total
1.	i = 1	.....	1		
2.	Enquanto i ≤ n e A[i] ≠ x	.....	1		
3.	i = i+1	.....	1		
4.	Se i ≤ n retorne (i)	.....	1		
5.	Senão retorne (-1)	.....	1		

Vamos mensurar:

			custo	vezes	total
1.	i = 1	.....	1	1	
2.	Enquanto i ≤ n e A[i] ≠ x	.....	1	n + 1	
3.	i = i+1	.....	1	n	
4.	Se i ≤ n retorne (i)	.....	1	1	
5.	Senão retorne (-1)	.....	1	1	

Vamos mensurar:

			custo	vezes	total
1.	i = 1	.....	1	1	1
2.	Enquanto i ≤ n e A[i] ≠ x	.....	1	n + 1	n + 1
3.	i = i+1	.....	1	n	n
4.	Se i ≤ n retorne (i)	.....	1	1	1
5.	Senão retorne (-1)	.....	1	1	1

Vamos mensurar:

			custo	vezes	total
1.	i = 1	.....	1	1	1
2.	Enquanto i ≤ n e A[i] ≠ x	.....	1	n + 1	n + 1
3.	i = i+1	.....	1	n	n
4.	Se i ≤ n retorne (i)	.....	1	1	1
5.	Senão retorne (-1)	.....	1	1	1

Somando: $1 + n + 1 + n + 1 + 1 = 2n + 4$

Complexidade em Tempo

É dita complexidade do algoritmo A se, para todo x, dado x como entrada de A, A termina em exatamente $t_A(x)$ passos.

ps.: $t_A:\{0,1\}^* \rightarrow N$

Pode a análise ser realizada em três casos:

- ➔ **Melhor Caso**
- ➔ **Pior Caso**
- ➔ **Caso Médio**

Melhor Caso

Estamos interessados na dependência da complexidade como função do tamanho da entrada, tomando o valor mínimo de passos para entradas de tamanho relevante.

	procedimento Busca Sequencial (A[1,..,n],x)		custo	vezes	total
1.	i = 1	.....	1	1	1
2.	Enquanto i ≤ n e A[i] ≠ x	.....	1	1	1
3.	i = i+1	.....	1	0	0
4.	Se i ≤ n retorne (i)	.....	1	1	1
5.	Senão retorne (-1)	.....	1	1	1

Exemplo: No melhor caso a chave de busca sempre se encontra na primeira posição da lista.

Então mesmo com: $n \to \omega$, sempre temos $t_A(n) = 4$.

Pior Caso

Estamos interessados na dependência da complexidade como função do tamanho da entrada, tomando o valor máximo de passos para entradas de tamanho relevante.

	procedimento Busca Sequencial (A[1,..,n],x)		custo	vezes	total
1.	i = 1	.....	1	1	1
2.	Enquanto i ≤ n e A[i] ≠ x	.....	1	n + 1	n + 1
3.	i = i+1	.....	1	n	n
4.	Se i ≤ n retorne (i)	.....	1	1	1
5.	Senão retorne (-1)	.....	1	1	1

Exemplo: No pior caso a chave de busca sempre se encontra na última posição da lista.

Então com: $n \to \omega$, sempre temos $t_A(n) = 2n + 4$.

Caso Médio

Depende de probabilidades associadas às possíveis entradas para o algoritmo.

⇒ Caso médio da busca sequencial:

Assumindo que x está em A, $t_A(n) = 1*p_1 + 2*p_2 + \ldots + n*p_n$

Onde p_i é a probabilidade de x estar na posição i;

⇒ Na busca sequencial as probabilidades são iguais: $p_i = 1/n$;

Então: $t_A(n) = 1/n\ (1 + 2 + 3 + \ldots + n) = 1/n\ (\ n(n+1) / 2\ ) = (\ n + 1\ ) / 2$

Ou seja, complexidade de tem: (n + 1) / 2.

Uma pesquisa bem-sucedida examina aproximadamente metade dos registros.

Comparação entre Complexidades

- ➔ A complexidade exata possui muitos detalhes.
- ➔ A escolha de um algoritmo é feita através de sua taxa de crescimento
- ➔ Esta taxa é representada através de cotas que são funções mais simples.
- ➔ A ordem de crescimento do tempo de execução de um algoritmo fornece uma caracterização simples de eficiência do algoritmo.

Termo de maior ordem

Imagine um algoritmo com complexidade:

$$an^2 + bn + c$$

1 - Desprezamos os termos de baixa ordem;

2 - Ignoramos o coeficiente constante;

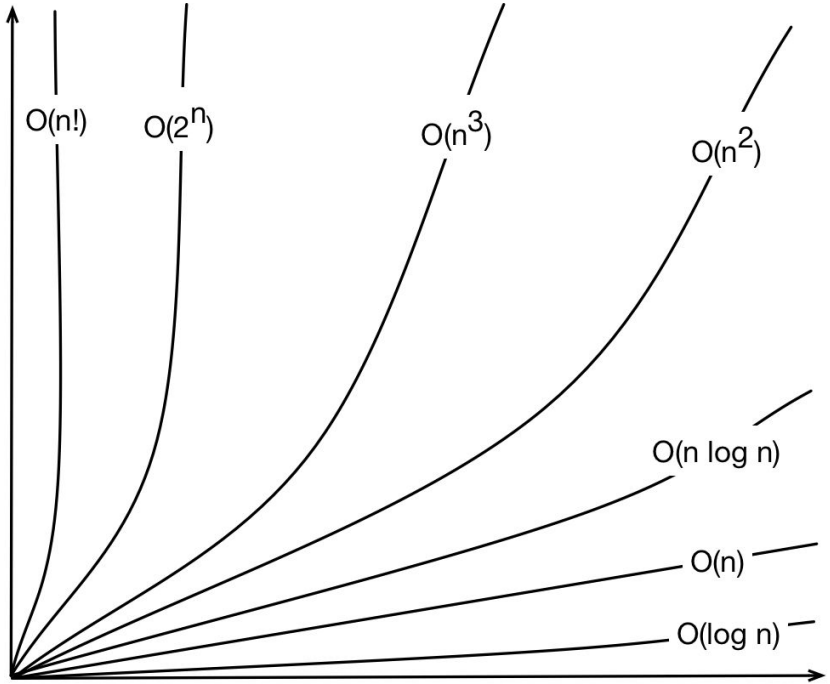
Logo o tempo de execução do algoritmo tem cota igual a: n^2

Ou seja, $O(n^2)$ → veremos isso mais a diante.

Funções comuns encontradas quando analisamos o tempo de execução de algoritmos

Notação	Nome	Exemplos
$O(1)$	constante	Determinar se um número é par ou ímpar, encontrar o valor máximo em um arranjo ordenado
$O(\log n)$	logarítmico	Encontrar um valor em um arranjo ordenado usando busca binária
$O(n)$	linear	Encontrar um valor em um arranjo não ordenado usando busca linear
$O(n \log n)$	loglinear	quicksort
$O(n^2)$	quadrático	bubblesort
$O(c^n)$, $c > 1$	exponencial	Encontrar a solução exata para o problema do caixeiro viajante usando programação dinâmica
$O(n!)$	fatorial	Encontrar a solução exata para o problema do caixeiro viajante usando força bruta

Funções comuns encontradas quando analisamos o tempo de execução de algoritmos



Alguns padrões (para identificar)

⇒ Uma sequência sem laço ou recursão conta passo constante (1):

```
/* bloco com número de passos constante */
```

⇒ Um único laço com n passos internos constante: linear (n):

```
for(i=0; i < n; i++)
    /* bloco com número de passos constante */
```

⇒ Dois laços de tamanho n alinhados: quadrático (n^2):

```
for(i=0; i < n; i++)
    for(j=0; j < n; j++)
        /* bloco com número de passos constante */
```

Alguns padrões (para identificar)

⇒ Um laço interno dependente de um externo: quadrático (n^2):

```
for(i=0; i < n; i++)
      for(j=0; j < i; j++)
          /* bloco com número de passos constante */
```

⇒ Quando divide o problema pela metade: logarítmico ($\log_2 n$):

```
subprob(i,n)
    if (test)
        subprob(0,n/2)
    else
        subprob(n/2+1, n)
```

Obrigado

jorge@imd.ufrn.br