Algoritmos de Busca

Pesquisa, Ordenação e Técnicas de Armazenamento

Prof. Me. Orlando da Silva Junior

Agenda

- Análise de algoritmos
- Busca linear
- Busca binária
- Exercícios

Análise de Algoritmos

ANALISAR UM ALGORITMO

Significa prever os recursos de que o algoritmo necessitará.

Recursos de memória, largura de banda ou hardware também são preocupações....

....mas a maior frequência é com o tempo de computação que desejamos medir.

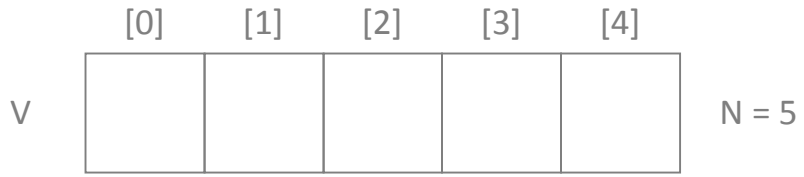
Pela análise de vários algoritmos candidatos para um problema, podemos identificar facilmente um algoritmo mais eficiente.

*Podemos medir o **tempo de execução** de um algoritmo de duas maneiras:*

- ✓ ***Empiricamente:*** *medindo diretamente o tempo que o algoritmo leva para processar uma entrada de tamanho N;*
- ✓ ***Analiticamente:*** *assumindo que cada operação primitiva demanda uma quantia de tempo constante.*

*Em geral, o **tempo de execução** de um algoritmo cresce com o **tamanho da entrada**.*

*Em um problema de ordenação de números em um vetor, o **tamanho da entrada** é dado pelo número de itens na entrada, ou seja, pelo tamanho do vetor a ser ordenado.*



*O **tempo de execução** de um algoritmo em uma determinada entrada é o número de operações primitivas ou "etapas" executadas. Mesmo que cada linha de código demore mais tempo que outra linha, vamos considerar que cada execução da linha leva um tempo constante.*

*O **tempo de execução** do algoritmo é a soma dos tempos de execução para cada instrução executada.*

```
Procedimento fatorial(N: inteiro) : inteiro
Início
      fatorial: inteiro
      fatorial ← 1
      Para i de 1 até N passo 1 Faça
            fatorial ← fatorial * i
      Fim-para
      Devolve fatorial
Fim
```

O tempo de execução será igual a $T(N) = c1 \cdot 1 + c2 \cdot (N + 1) + c3 \cdot N + c4 \cdot 1$

		custo	vezes
	Procedimento fatorial(N: **inteiro**) : **inteiro**		
	Início		
1	fatorial: **inteiro**		
2	fatorial ← 1	c1	1
3	**Para** i **de** 1 **até** N **passo** 1 **Faça**	c2	N+1
4	fatorial ← fatorial * i	c3	N
5	**Fim-para**		
6	**Devolve** fatorial	c4	1
	Fim		

O tempo de execução será igual a $T(N) = c1 \cdot 1 + c2 \cdot (N + 1) + c3 \cdot N + c4 \cdot 1$

Quando N = 1 ⟶ $T(1) = c1 + 2 \cdot c2 + c3 + c4$

Quando N = 10 ⟶ $T(10) = c1 + 11 \cdot c2 + 10 \cdot c3 + c4$

Quando N = 1000 ⟶ $T(1000) = c1 + 1001 \cdot c2 + 1000 \cdot c3 + c4$

Vamos considerar agora um algoritmo recursivo.

		custo	vezes
	Procedimento fatorial(N: **inteiro**) : **inteiro**		
	Início		
1	**Se** N ≤ 1 **Então**	c1	N
2	**Devolve 1**	c2	1
3	**Fim-se**		
4	**Devolve** N * fatorial(N – 1)	c3	?
	Fim		

- *No **melhor caso**, "?" não é executado*
- *No **pior caso**, "?" é executado N vezes*

		custo	vezes
	Procedimento fatorial(N: **inteiro**) : **inteiro**		
	Início		
1	**Se** n ≤ 1 **Então**	c1	N
2	**Devolve 1**	c2	1
3	**Fim-se**		
4	**Devolve** N * fatorial(N – 1)	c3	?
	Fim		

- *No **melhor caso**,* $T(N) = c1 \cdot N + c2$
- *No **pior caso**,* $T(N) = c1 \cdot N + c2 + c3 \cdot N$

Exercício: Considere o algoritmo a seguir e encontre a função que relaciona o tempo de execução com o tamanho da entrada.

		custo	melhor caso	pior caso
	Procedimento maior(v[0..N]: **inteiro**) : **inteiro**			
	Início			
1	maior: **inteiro**			
2	maior ← v[0]	c1	1	1
3				
4	**Para** i **de** 1 **até** tamanho(v) **passo** 1 **Faça**	c2	N	N
5	**Se** v[i] > maior **Então**	c3	N-1	N-1
6	maior ← v[i]	c4	0	N-1
7	**Fim-se**			
8	**Fim-para**			
9	**Devolve** maior	c5	1	1
	Fim			

- *No **melhor caso**,* $T(N) = (c2 + c3) \cdot N - c3 + c1 + c5$
- *No **pior caso**,* $T(N) = c1 + (c2 + c3 + c4) \cdot N - (c3 + c4) + c5$

Polinômios

Observamos que é possível utilizar um ferramental matemático para descrever o desempenho de um algoritmo por meio de **polinômios** da forma:

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0$$

- a_n são os **coeficientes**
- x é a **variável**
- n é o **grau** do polinômio

Big O

Notação O (lê-se “Big Oh”)

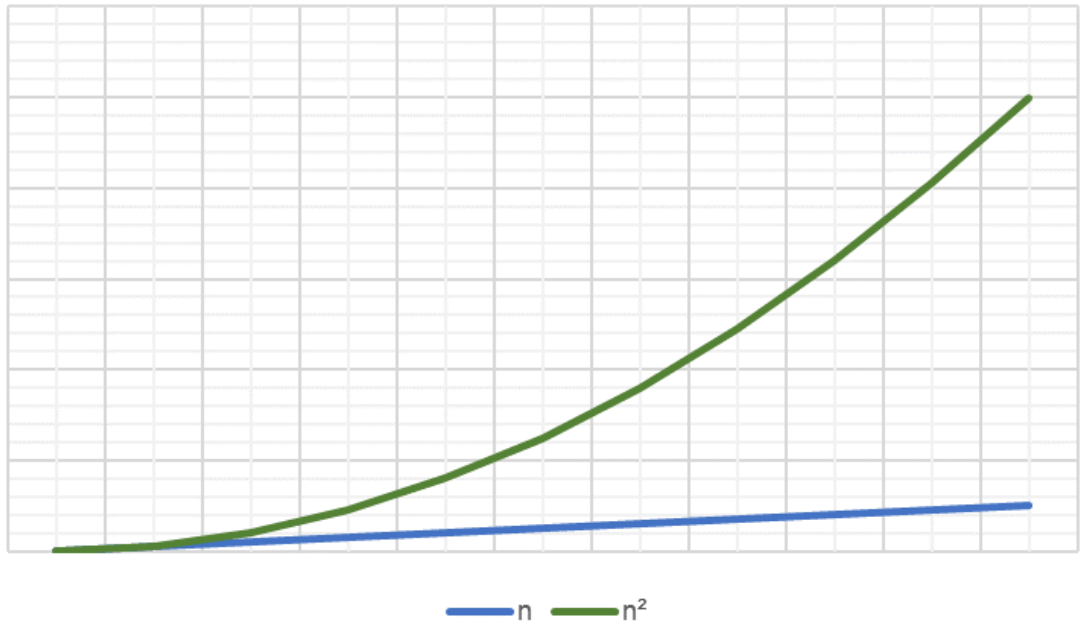
- **Big O** define uma função que cresce mais rapidamente do que outra. Ou seja, uma função está acima de outra a partir de certo ponto.

- $O(f(n))$ caracteriza o número de operações ou uso de memória em função do tamanho da entrada n.

- Quando $T(n) = O(f(n))$, Big O define que $f(n)$ cresce mais rapidamente que $T(n)$.

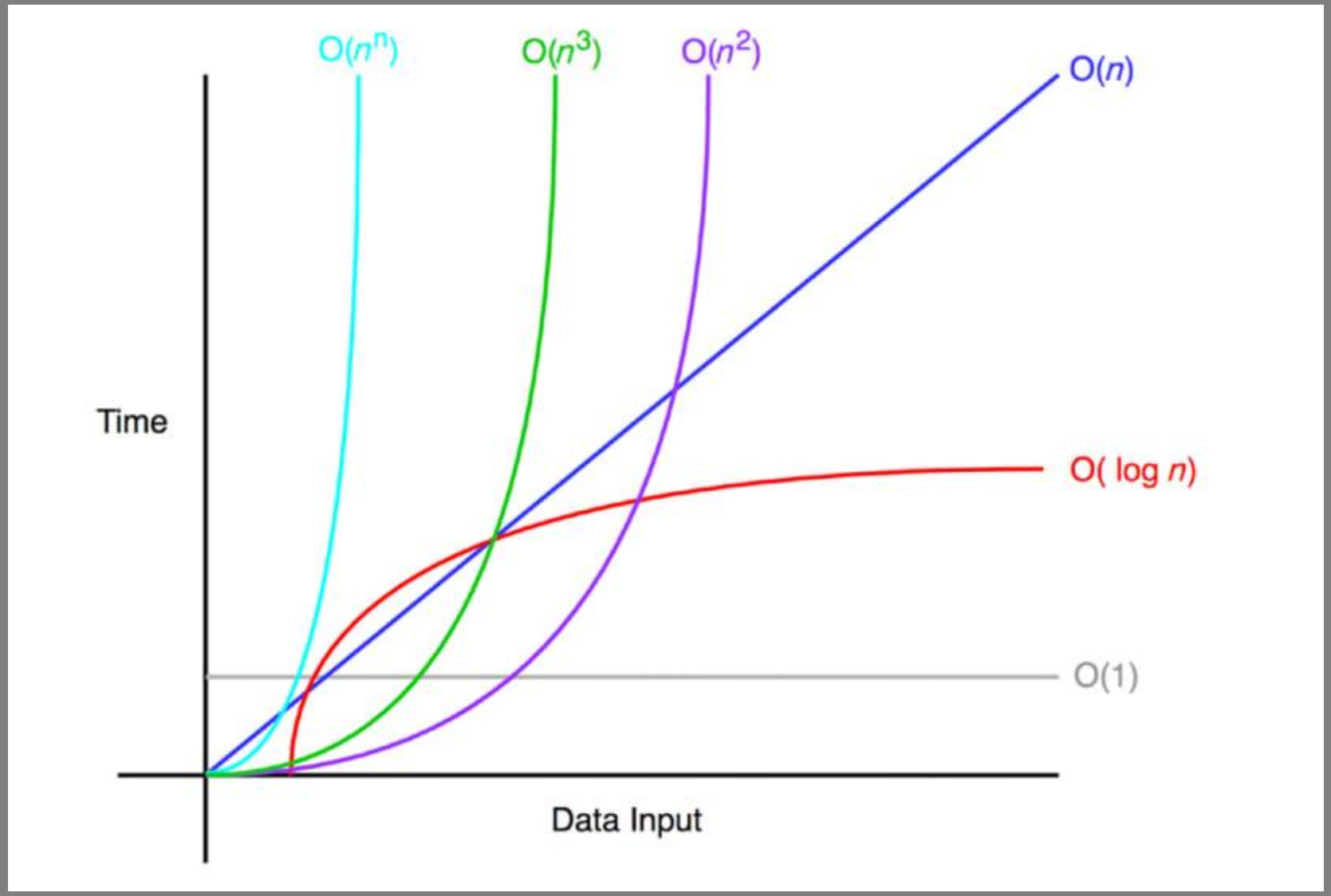
<u>Por exemplo:</u> uma função quadrática n^2 cresce mais rapidamente do que uma função linear n.

Podemos dizer que n^2 é uma cota assintótica superior para n.



*A **complexidade assintótica** é definida pelo crescimento da complexidade para entradas suficientemente grandes.*

O algoritmo assintoticamente mais eficiente é melhor para todas as entradas, exceto para entradas relativamente pequenas.


O(n^n)
O(n^3)
O(n^2)
O(n)
Time
O(log n)
O(1)
Data Input

Algoritmos de Busca

O problema da busca é um problema clássico da computação.

Em um processo de busca, estamos interessados em encontrar um valor particular, chamado de **alvo** (*target*), em uma estrutura de dados que armazene uma quantidade volumosa de informações.

BUSCA LINEAR

X

BUSCA BINÁRIA

Busca Sequencial (Linear)

Considere:

- Um vetor *v* de *n* elementos não necessariamente ordenados;
- Um valor *k* a ser procurado;

Faça:

- Percorra o vetor *v* até encontrar o valor *k*;
  - Inicie da posição mais à esquerda e compare *k* com cada valor *v[i]*
- Retorne a posição *i* do valor *k*, se ele for encontrado;
- Caso não seja encontrado, retorne um valor especial (nulo, -1, etc.)

```
   Função buscaLinear(V[0, ..., N]: vetor, T: inteiro) : inteiro
   Início
1          Para i de 0 até N passo 1 Faça
2                 Se V[i] = T Então
3                        Devolve i
4                 Fim-se
5          Fim-para
6          Devolve -1
   Fim
```

Busca Linear – Análise de Complexidade

```
     Função buscaLinear(V[0, ..., N]: vetor, T: inteiro) : inteiro
     Início
1            Para i de 0 até N passo 1 Faça
2                  Se A[i] = T Então
3                        Devolve i
4                  Fim-se
5            Fim-para
6            Devolve -1
     Fim
```

	custo	melhor caso	pior caso
1	c1	1	N
2	c2	1	N
3	c3	1	0
4			
5			
6	c4	0	1

- *No **melhor caso**,* $T(N) = c1 + c2 + c3 \rightarrow O(1)$
- *No **pior caso**,* $T(N) = (c1 + c2) \cdot N + c4 \rightarrow O(N)$

Busca Binária

Considere:

- Um vetor v de n elementos necessariamente ordenados;
- Um valor k a ser procurado;

- **Repita:**
  - Encontre o elemento central do vetor, chamado **pivô**;
  - Divida o vetor em 2 novos vetores:
    - Um à esquerda do pivô;
    - Outro à direita do pivô;
  - Se o elemento a ser encontrado for menor que o pivô, busque à esquerda;
  - Senão, busque à direita;
  - Caso o valor não seja encontrado, retorne um valor especial (nulo, -1, etc.)

```
     Função buscaBinaria(V[0, ..., N]: vetor, T: inteiro) : inteiro
     Início
1            ini, fim, pivo: inteiro
2            ini ← 0
3            fim ← tamanho{V}
4            Enquanto ini ≤ fim Então
5                    pivo ← ⌊ (ini + fim) / 2 ⌋
6                    Se V[pivo] = T Então
7                            Devolve pivo
8                    Senão
9                            Se V[pivo] < T Então
10                                   ini = pivo + 1
11                           Senão
12                                   fim = pivo - 1
13                           Fim-se
14                   Fim-se
15           Fim-enquanto
16           Devolve -1
     Fim
```

Busca Binária – Análise de Complexidade

```
     Função buscaBinaria(V[0, ..., N]: vetor, T: inteiro) : inteiro
     Início
1            ini, fim, pivo: inteiro
2            ini ← 0
3            fim ← tamanho{V}
4            Enquanto ini ≤ fim Então
5                    pivo ← ⌊ (ini + fim) / 2 ⌋
6                    Se V[pivo] = T Então
7                            Devolve pivo
8                    Senão
9                            Se V[pivo] < T Então
10                                   ini = pivo + 1
11                           Senão
12                                   fim = pivo - 1
13                           Fim-se
14                   Fim-se
15           Fim-se
16           Devolve -1
     Fim
```

	custo	melhor caso	pior caso
1			
2	c1	1	1
3	c2	1	1
4	c3	1	$log_2 N$
5	c4	1	$log_2 N$
6	c5	1	$log_2 N$
7	c6	1	1
8			
9	c7	0	$log_2 N$
10	c8	0	$log_2 N$
11			
12	c9	0	0
13			
14			
15			
16	c10	0	0

- *No **melhor caso**,* $T(N) = c1 + c2 + c3 + c4 + c5 + c6 = O(1)$
- *No **pior caso**,* $T(N) = O(\log_2 N)$

Questões para discutir

1. Uma vez que a busca binária é mais rápida que a busca linear, é verdade que a busca linear nunca deve ser utilizada? Explique.
2. Quando a busca linear deve ser utilizada?
3. Como a busca linear pode ser melhorada?
4. Quais são as desvantagens da busca binária?

Exercícios

1. Implemente em Java a **busca linear** para a estrutura de dados vetor.
2. Implemente em Java a **busca binária** para a estrutura de dados vetor.
3. Implemente em Java uma **versão recursiva** para a **busca linear**.
4. Implemente em Java uma **versão recursiva** da **busca binária**.