Complexidade de Algoritmos

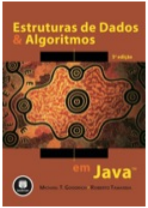
Roland Teodorowitsch

Algoritmos e Estruturas de Dados I - Escola Politécnica - PUCRS

2 de agosto de 2023

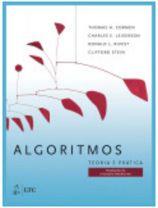
Apresentação

Leituras Recomendadas



Capítulo 4

GOODRICH, Michael T.; TAMASSIA, Roberto. **Estruturas de dados e algoritmos em Java**. Tradução: Bernardo Copstein. 5. ed. Porto Alegre: Bookman, 2013. xxii, 713 p. E-book. ISBN 9788582600191. Tradução de: Data Structures and Algorithms in Java, 5th Edition. Disponível em: <https://integrada.minhabiblioteca.com.br/#/books/9788582600191/>. Acesso em: 01 ago. 2023.



Capítulo 3

CORMEN, Thomas *et al*. **Algoritmos - Teoria e Prática**. Tradução: Arlete Simille Marques. Rio de Janeiro: Grupo GEN, 2012. E-book. ISBN 9788595158092. Tradução de: Introduction to Algorithms, 3rd ed. Disponível em: <https://integrada.minhabiblioteca.com.br/#/books/9788595158092/>. Acesso em: 01 ago. 2023.

Sumário

- Complexidade e análise de algoritmos
- Contando o tempo
- Contagem de operações
- Funções

Complexidade e análise de algoritmos

Complexidade e análise de algoritmos

- No desenvolvimento de uma aplicação tem-se como objetivo projetar "boas" estruturas de dados e "bons" algoritmos
  - Otimizados
  - Simples
- Como saber se um algoritmo é eficiente?

Análise de Algoritmos

- Estudo das características de desempenho de um determinado algoritmo
  - O espaço ocupado é uma característica de desempenho
  - O tempo gasto na execução é outra característica de desempenho

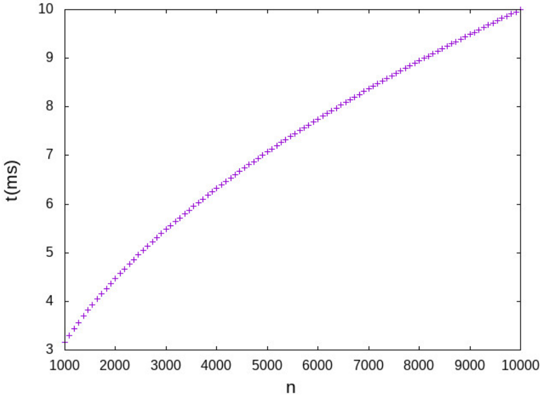
Complexidade de Algoritmos

- A **complexidade de um algoritmo** é a **medida do consumo de recursos** de que o algoritmo necessita durante a sua execução
  - Tempo de processamento
  - Memória ocupada
  - Largura de banda de comunicação
  - Hardware necessário
  - etc.

Contando o tempo

Tempo de processamento

- Depende de uma série de fatores: *hardware*, *software*, tamanho e tipo da entrada de dados
- Algoritmo: tempo de execução (ms) X tamanho da entrada de dados (n)



Medindo o tempo

- Em Java, `System.currentTimeMillis()` retorna o tempo em milissegundos (ms)

```java
long antes = System.currentTimeMillis();
// algoritmo a ser medido...
long ms = System.currentTimeMillis() - antes;
```

- Em Java, `System.nanoTime()` retorna o tempo em nanossegundos (ns)

```java
long antes = System.nanoTime();
// algoritmo a ser medido...
long ns = System.nanoTime() - antes;
```

- Em C, no Unix, pode-se usar `gettimeofday()` (incluir `<sys/time.h>`)

```c
struct timeval antes, depois;
gettimeofday(&antes, NULL);
// algoritmo a ser medido...
gettimeofday(&depois, NULL);
unsigned long ms = (depois.tv_sec  - antes.tv_sec) * 1000000 +
                   depois.tv_usec - antes.tv_usec;
```

Exemplo: Pesquisa Linear

- A função abaixo recebe um arranjo, o seu tamanho e um valor a ser localizado, e retorna a posição do valor no arranjo (ou -1 se não achar)

```
int localiza(int *dados, int tam, int valor) {
  for (int pos=0; pos<tam; pos++)
      if (valor == dados[pos])
         return pos;
  return -1;
}
```

- Melhor caso: o valor procurado é o primeiro elemento do arranjo
- Pior caso: o valor procurado é o último elemento do arranjo
- O arranjo precisa estar ordenado?

Exemplo: Pesquisa Linear

- A função abaixo recebe um arranjo, o seu tamanho e um valor a ser localizado, e retorna a posição do valor no arranjo (ou -1 se não achar)

```
int localiza(int *dados, int tam, int valor) {
  for (int pos=0; pos<tam; pos++)
      if (valor == dados[pos])
         return pos;
  return -1;
}
```

- Melhor caso: o valor procurado é o primeiro elemento do arranjo
- Pior caso: o valor procurado é o último elemento do arranjo
- O arranjo precisa estar ordenado?
  NÃO

Exemplo: Pesquisa Linear

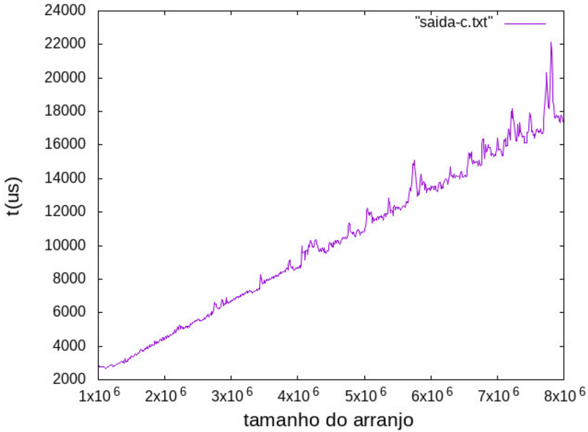
- Para visualizar como o algoritmo se comporta (“entender” a sua complexidade), executa-se a função com valores crescentes de arranjo

```
#define INI 1000000
#define FIM 8000000
#define INC 10000

int main() {
  struct timeval antes, depois;
  int *vetor = malloc( sizeof(int) * FIM );
  if (vetor == NULL) return 1;
  for (int pos=0; pos<FIM; pos++) //preenche o vetor
      vetor[pos] = pos;
  for (int total=INI; total<=FIM; total+=INC) {
      gettimeofday(&antes, NULL);
      int loc = localiza(vetor, total, total-1); //pior caso: ultimo elem
      gettimeofday(&depois, NULL);
      if (loc != total-1) return 1;
      unsigned long microssegundos = (depois.tv_sec - antes.tv_sec) * 1000000 +
                                     depois.tv_usec - antes.tv_usec;
      printf("%d␣%lu\n", total, microssegundos );
  }
  free(vetor);
  return 0;
}
```

Exemplo: Pesquisa Linear (Resultado)

```
1000000  2752
1010000  2794
1020000  2809
1030000  2796
1040000  2744
1050000  2753
1060000  2774
1070000  2752
1080000  2755
1090000  2780
1100000  2703
1110000  2685
1120000  2707
1130000  2732
1140000  2757
1150000  2786
1160000  2805
1170000  2837
1180000  2855
1190000  2876
1200000  2900
1210000  2892
...
```



- Qual a origem das variações do tempo de execução?

Exercício: *BubbleSort*

- Considere o algoritmo de ordenação *BubbleSort*, incluindo eventuais variações
- Implemente-o com eventuais variações e após a sua aplicação acrescente um código para verificar se a ordenação funcionou
- Escolha uma faixa de tamanhos para avaliar cada implementação e também uma unidade de tempo (ms, us ou ns)
- Para cada tamanho de arranjo, gere um array com elementos diferentes, em ordem decrescente do maior valor para o menor
- Gere os gráficos de desempenho de cada implementação
- Gere gráficos também para arranjo já ordenado e arranjo com valores aleatórios

Análise da eficiência de um algoritmo

- Medir o tempo depende de hardware e software (sistema operacional, por exemplo)
- Alternativa?

Análise da eficiência de um algoritmo

- Medir o tempo depende de hardware e software (sistema operacional, por exemplo)
- Alternativa?
  Contar o número de operações (atribuição, operação aritmética, comparação, etc.)

Contagem de operações

Análise de algoritmos

- Não pode considerar o tempo de execução
- Deve ser feita diretamente sobre o pseudocódigo de alto nível
- Consiste em contar quantas **operações primitivas** são executadas
  - Operação primitiva: instrução de baixo nível com um tempo de execução constante
- Assume-se que os tempos de execução de operações primitivas diferentes são similares

Operações primitivas

- Atribuição de valores a variáveis
- Chamadas de métodos
- Operações aritméticas (por exemplo, adição de dois números)
- Comparação de dois números
- Acesso a um arranjo
- Retorno de um método

Exemplo 1

- Contar o número de operações para atribuir para cada posição v[i] de um arranjo unidimensional o resultado de i*2

```
v[0..10] : inteiro
for (i = 0; i < v.comprimento; i++)
    v[i] = i * 2
```

Exemplo 1

- Contar o número de operações para atribuir para cada posição v[i] de um arranjo unidimensional o resultado de i*2

```
v[0..10] : inteiro
for (i = 0; i < v.comprimento; i++)
    v[i] = i * 2
```

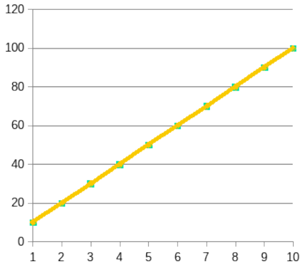
- Operação (multiplicação, atribuição e acesso as posições do arranjo): **n vezes (n = 10)**

Exemplo 1

- Contar o número de operações para atribuir para cada posição v[i] de um arranjo unidimensional o resultado de i*2

```
v[0..10] : inteiro
for (i = 0; i < v.comprimento; i++)
    v[i] = i * 2
```

- Operação (multiplicação, atribuição e acesso as posições do arranjo): **n vezes ($n = 10$)**



Exemplo 2

- Contar o número de operações para atribuir para cada posição m[i,j] de um arranjo bidimensional o resultado de i*j

```
m[0..10][0..10] : inteiro
for (i=0; i<m.comprimento; i++)
    for (j=0; j<m[i].comprimento; j++)
        m[i][j] = i * j
```

Exemplo 2

- Contar o número de operações para atribuir para cada posição m[i,j] de um arranjo bidimensional o resultado de i*j

```
m[0..10][0..10] : inteiro
for (i=0; i<m.comprimento; i++)
    for (j=0; j<m[i].comprimento; j++)
        m[i][j] = i * j
```

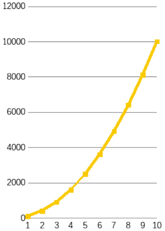
- Operação (multiplicação, atribuição e acesso as posições do arranjo): **n*n vezes (n = 10)**

Exemplo 2

- Contar o número de operações para atribuir para cada posição m[i,j] de um arranjo bidimensional o resultado de i*j

```
m[0..10][0..10] : inteiro
for (i=0; i<m.comprimento; i++)
    for (j=0; j<m[i].comprimento; j++)
        m[i][j] = i * j
```

- Operação (multiplicação, atribuição e acesso as posições do arranjo): **n*n vezes (n = 10)**



Exercícios (1/2)

Implementar e contar o número de operações das funções listadas a seguir.

1.
```
int f1(n)
    r=0
    for (i=1; i<n; i++)
        r = r + 1
    return r
```
2.
```
int f2(n)
    r=0
    for (i=1; i<n; i++)
        for (j=i+1; j<n; j++)
            r = r + 2
    return r
```
3.
```
int f3(n)
    cont=0
    for (i=1; i<n; i++)
        for (j=1; j<n; j++)
            print i*j
            cont++
     return cont
```

Exercícios (2/2)

4.
```
int f4(n)
    r=0
    for (i=1; i<n; i++)
        for (j=i; j<2*i; j++)
            for (k=i; k<j; k++)
                r = r + 1
     return r
```

5.
```
int f5(n)
    r=0
    for (i=1; i<n; i++)
        for (j=i; j<i+3; j++)
            for (k=i; k<j; k++)
                r = r + 1
    return r
```

6.
```
int f6(n)
    if (n==0)
        return 1
    else
        return f6(n-1) + f6(n-1)
```

Funções

Funções

- Uma **classe de complexidade** é uma forma de agrupar algoritmos que apresentam complexidade similar. Por exemplo:
  - Complexidade **constante**: o algoritmo sempre ocupa a mesma quantidade de recursos
  - Complexidade **linear**: o algoritmo consome recursos de forma diretamente proporcional ao tamanho do problema

Funções

- Sete funções mais comuns usadas em análise de algoritmos:
  - Constante: 1
  - Logaritmo: $\log n$
  - Linear: n
  - n-log-n: $n \log n$
  - Quadrática: n^2
  - Cúbica: n^3
  - Exponencial: a^n

Função Constante: 1

- Função mais simples
- $f(n) = c$
- Não importa o valor de n, sempre será igual ao valor da constante c
- Exemplo: função que recebe um arranjo de inteiros e retorna o valor do primeiro elemento multiplicado por 2

Função Logaritmo: $\log n$

- $f(n) = \log_2 n$
- O número de operações realizadas para solução do problema não cresce da mesma forma que n
  - Se dobra o valor de n, o incremento do consumo é bem menor
- Exemplo: conversão de número decimal para binário e pesquisa binária (*binary search*)

Função Linear: n

- Se dobra o valor de n, dobra o consumo de recursos
- $f(n) = n$
- Exemplo: localizar um elemento em uma lista.

Função n-log-n: $n \log n$

- $f(n) = n \log n$
- Atribui para uma entrada n o valor de n multiplicado pelo logaritmo de base 2 de n
- Cresce mais rápido que a função linear e mais devagar que a função quadrática
- Exemplo: Algoritmos de ordenação mergesort e heapsort
  `http://www.sorting-algorithms.com/`

Função Quadrática: n^2

- Função polinomial com expoente 2
- $f(n) = n^2$
- Não cresce de forma abrupta,mas dificultam o uso em problemas grandes.
- Exemplo: Ordenação com o algoritmo $bubblesort$

Função Cúbica: n^3

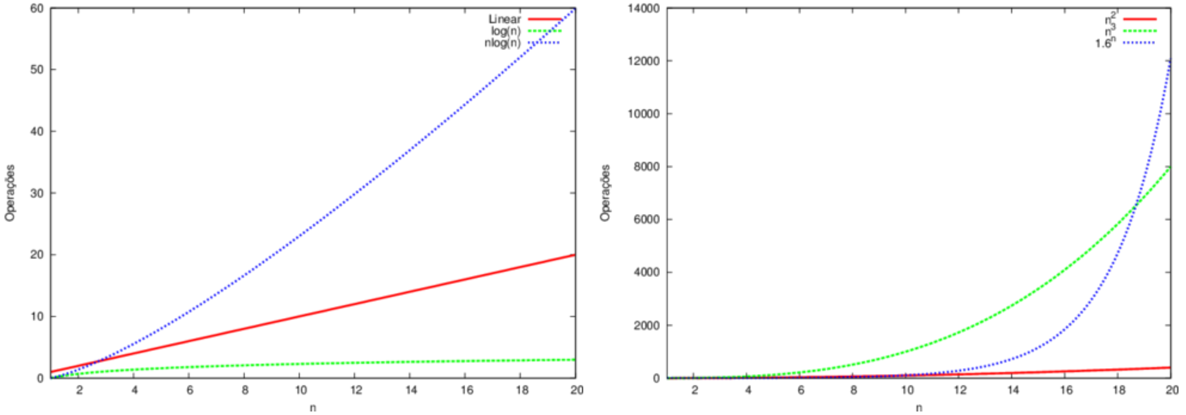
- Função polinomial com expoente 3
- $f(n) = n^3$
- Aparece com menos frequência na análise de algoritmos do que as funções constante, linear ou quadrática
- Exemplo: multiplicar duas matrizes

Função Exponencial: a^n

- Expoente é variável
- $f(n) = a^n$
- Algoritmos “ruins”, crescem abruptamente
- Aplicável apenas em problemas pequenos.
- Exemplos: quebrar senhas com força bruta e listar todos os subconjuntos de um conjunto S

Funções

- Taxas de crescimento para as funções usadas em análise de algoritmos



Funções

- Um problema é dividido em funções/métodos
  - Cada função/método tem um "custo" diferente
  - Estes custos são somados para determinar o custo total para solução do problema

Exercício 1

- Dois algoritmos para resolver o mesmo problema
  $f_1(n) = 2n^2 + 5n$ operações
  $f_2(n) = 500n + 4000$ operações
- Considere o número de operações de cada um para diferentes valores de n (por exemplo, $n = 10$ e $n = 1000$)
- Qual é a melhor solução?

Exercício 2

Qual a complexidade do algoritmo abaixo?

```
v[1..N] : inteiro
maximo = 0
for (i=1; i<=n; ++i)
    sum = 0    // sum = somatorio de x[i..j]
    for (j = i; j <= n; ++j)
        sum += x[j]
    maximo = max(maximo,sum)
```

- A $O(n)$
- B $O(n^2)$
- C $O(n \log 2n)$
- D $O(2n)$

Exercício 3

Qual a complexidade do algoritmo abaixo?

```
int busca(v[1..N] : inteiro, elem : inteiro)
    for (i = 1; i <= n; i++)
        if (elem == v[i])
           return i // elemento encontrado no indice i
    return -1  // elemento NAO encontrado
```

- A $O(n)$
- B $O(n^2)$
- C $O(n \log 2n)$
- D $O(2n)$

Créditos

Créditos

- Estas lâminas contêm trechos de materiais criados e disponibilizados pela professora Isabe Harb Manssour.