Complexidade de Algoritmos

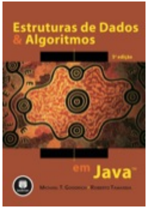
Roland Teodorowitsch

Algoritmos e Estruturas de Dados I - Escola Politécnica - PUCRS

22 de agosto de 2023

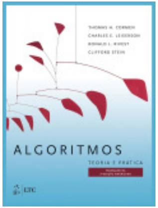
Apresentação

Leitura(s) Recomendada(s)



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
- Medindo o tempo
- Contagem de operações
- Funções
- Exercícios
- Notação O
- Notação Ω e Notação Θ
- Análise assintótica

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

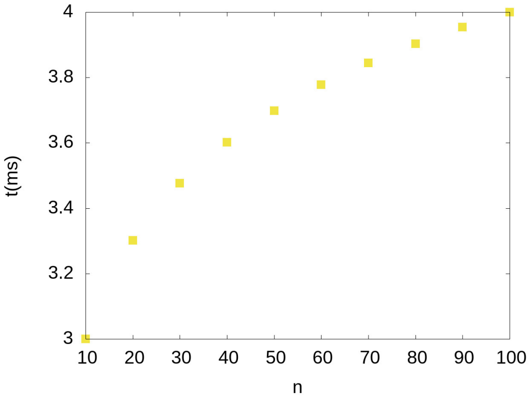
Complexidade de Algoritmos

- 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.

Medindo 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)



GNUPLOT:

```
set terminal jpeg enhanced size 1280,960 font "arial,32.0"
set output "grafico.jpg"
set xlabel "n"
set ylabel "t(ms)"
plot [10:100][3:4] "grafico.txt" with points lw 8 lt 5 ps 2 notitle
```

grafico.txt:

```
10 3.0000
20 3.3010
30 3.4771
40 3.6021
50 3.6990
60 3.7782
70 3.8451
80 3.9031
90 3.9542
100 4.0000
```

Medindo o tempo

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

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

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

```
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>`)

```
struct timeval antes, depois;
gettimeofday(&antes, NULL);
// algoritmo a ser medido...
gettimeofday(&depois, NULL);
unsigned long us = (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)

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

- Melhor caso: o valor procurado é o primeiro elemento do arranjo
- Pior caso: o valor procurado NÃO existe no 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 pesquisa_linear(int *dados, int tam, int valor) {
  for (int i=0; i<tam; i++)
      if (valor == dados[i])
          return i;
  return -1;
}
```

- Melhor caso: o valor procurado é o primeiro elemento do arranjo
- Pior caso: o valor procurado NÃO existe no arranjo
- O arranjo precisa estar ordenado?
  NÃO

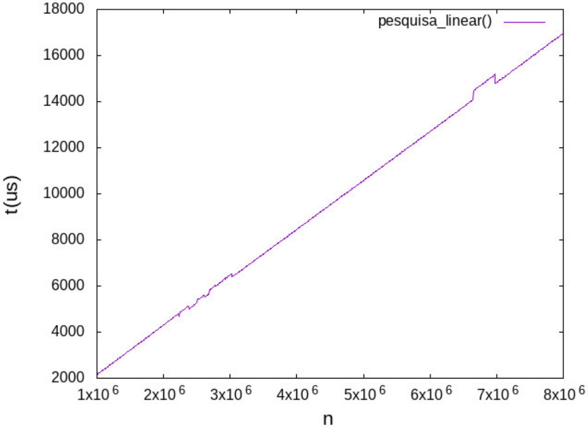
Exemplo: Pesquisa Linear

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

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

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

int pesquisa_linear(int *dados, int tam, int valor);

int main() {
  struct timeval antes, depois;
  int *vetor = malloc( sizeof(int) * FIM );
  if (vetor == NULL) return 1;
  for (int i=0; i<FIM; i++) vetor[i] = i; //preenche o vetor
  for (int total=INI; total<=FIM; total+=INC) {
      unsigned long min;
      for (int j=0; j<10; ++j) {
          gettimeofday(&antes, NULL);
          int loc = pesquisa_linear(vetor, total, total); //pior caso: elemento NAO existe
          gettimeofday(&depois, NULL);
          if (loc != -1) return 1;
          unsigned long microssegundos = (depois.tv_sec - antes.tv_sec) * 1000000 + depois.tv_usec - antes.tv_usec;
          if (j == 0 || microssegundos < min) min = microssegundos;
      }
      printf("%d␣%lu\n", total, min);
  }
  free(vetor);
  return 0;
}
```

Exemplo: Pesquisa Linear (Resultado)

```
1000000 2283
1010000 2178
1020000 2198
1030000 2220
1040000 2239
1050000 2261
1060000 2282
1070000 2303
1080000 2324
1090000 2347
1100000 2368
1110000 2390
1120000 2411
1130000 2432
1140000 2454
1150000 2476
1160000 2498
1170000 2519
1180000 2540
1190000 2562
1200000 2584
1210000 2608
...
```



GNUPLOT:

```
set terminal jpeg
set output "grafico.jpg"
set xlabel "n" font "Arial,16"
set ylabel "t(us)" font "Arial,16"
plot "curva.txt" with lines title "pesquisa\\_linear()"
```

Exemplo: Pesquisa Linear (Considerações)

- Considerando o algoritmo e a sua implementação, que tipo de curva de desempenho seria possível esperar?
- É possível visualizar a curva esperada no gráfico obtido?
- Qual a origem das variações do tempo de execução?

Exercício 1: *Bubble Sort* (Linux)

- Implemente o algoritmo de ordenação *BubbleSort*, por exemplo, a partir do pseudocódigo disponível na Wikipedia (`https://pt.wikipedia.org/wiki/Bubble_sort`) – use o seguinte protótipo como modelo:

```
void bubble_sort(int *dados, int tam);
```

- Acrescente a sua implementação ao código da próxima página
  - Este código trabalha com n variando de 1000 até 10000 com incremento 10, e medindo o tempo em microssegundos
- Gere o gráfico de desempenho a partir da execução
  - Salve seu código em um arquivo chamado `bubble_sort.c` e compile-o usando: `gcc -o bubble_sort bubble_sort.c`
  - Execute o programa direcionando a sua saída para um arquivo chamado `curva1.txt`: `./bubble_sort >curva1.txt`
  - Use o **GNUPLOT** para visualizar o arquivo `curva1.txt` (dicas nas próximas páginas)

Exercício 1: *Bubble Sort* (código da função `main()`)

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

#define INI  1000
#define FIM 10000
#define INC    10

void bubble_sort(int *dados, int tam);

int esta_ordenado(int *dados, int tam) {
  for (int i=0; i<tam-1; ++i) if (dados[i] > dados[i+1]) return 0;
  return 1;
}

int main() {
  struct timeval antes, depois;
  int *vetor = malloc( sizeof(int) * FIM );
  if (vetor == NULL) return 1;
  for (int total=INI; total<=FIM; total+=INC) {
      unsigned long min;
      for (int j=0; j<10; ++j) {
          for (int pos=0; pos<total; pos++) vetor[pos] = total - pos; //preenche o vetor
          gettimeofday(&antes, NULL);
          bubble_sort(vetor,total);
          gettimeofday(&depois, NULL);
          if (!esta_ordenado(vetor,total)) return 1;
          unsigned long microssegundos = (depois.tv_sec - antes.tv_sec) * 1000000 + depois.tv_usec - antes.tv_usec;
          if (j == 0 || microssegundos < min) min = microssegundos;
      }
      printf("%d %lu\n", total, min);
  }
  free(vetor);
  return 0;
}
```

Exercício 1: *Bubble Sort* (dicas sobre **GNUPLOT**)

- **GNUPLOT** é um aplicativo para gerar gráficos, disponível em várias plataformas
- Para mostrar, por exemplo, o gráfico correspondente aos dados armazenados no arquivo `curva1.txt` em uma janela, executa-se o **GNUPLOT** (`gnuplot`) e digita-se os seguintes comandos:

```
set xlabel "n" font "Arial,16"
set ylabel "t(us)" font "Arial,16"
plot "curva1.txt" with lines title "bubble\\_sort()"
```

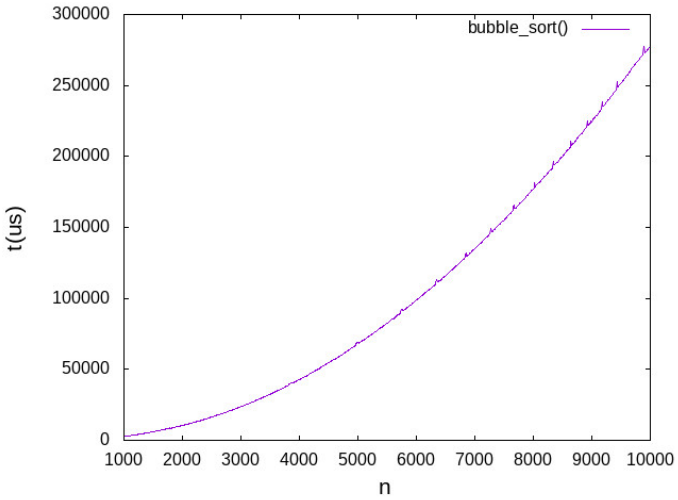
- Para gerar um arquivo JPEG com o conteúdo do gráfico, pode-se usar os seguintes comandos:

```
set terminal jpeg
set output "grafico1.jpg"
set xlabel "n" font "Arial,16"
set ylabel "t(us)" font "Arial,16"
plot "curva1.txt" with lines title "bubble\\_sort()"
```

Solução 1: *Bubble Sort* (primeira implementação)

```
void bubble_sort(int *dados, int tam) {
  int trocou;
  do {
     trocou = 0;
     for (int i=0; i<tam-1; ++i) {
         if (dados[i] > dados[i+1]) {
            int aux = dados[i];
            dados[i] = dados[i+1];
            dados[i+1] = aux;
            trocou = 1;
         }
     }
  } while (trocou);
}
```

Solução 1: *Bubble Sort* (gráfico da primeira implementação)



GNUPLOT:

```
set terminal jpeg
set output "grafico1.jpg"
set xlabel "n" font "Arial,16"
set ylabel "t(us)" font "Arial,16"
plot "curva1.txt" with lines title "bubble\\_sort()"
```

Exercício 2: *Bubble Sort* otimizado

- A versão sugerida para implementação do *Bubble Sort* tem um problema:
  - Depois de executar uma passagem, o maior elemento é "empurrado" para a sua posição
  - E: as passagens seguintes continuam "tentado" empurrar o maior elemento para o final
- A solução consiste em diminuir gradualmente o tamanho até onde cada passagem é executada
- Implemente esta solução e compare o seu desempenho com a versão anterior

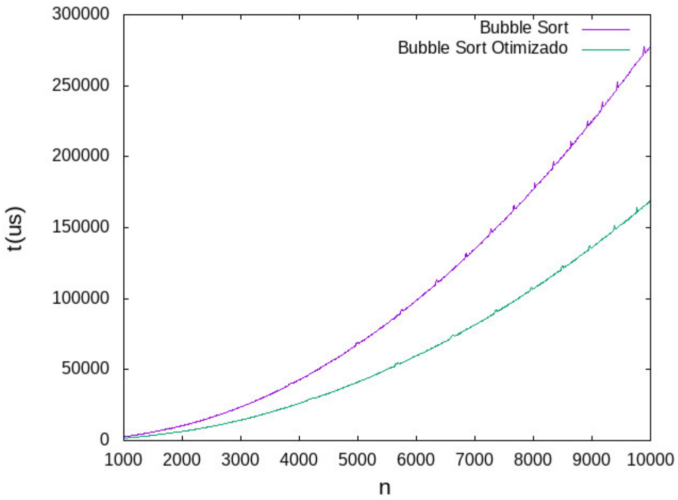
Solução 2: *Bubble Sort* otimizado

```c
void bubble_sort(int *dados, int tam) {
  int trocou;
  do {
      trocou = 0;
      --tam;
      for (int i=0; i<tam; ++i) {
          if (dados[i] > dados[i+1]) {
              int aux = dados[i];
              dados[i] = dados[i+1];
              dados[i+1] = aux;
              trocou = 1;
          }
      }
  }  while (trocou);
}
```

Solução 2: *Bubble Sort* otimizado (comandos do gnuplot)

```
set xlabel "n" font "Arial,16"
set ylabel "t(us)" font "Arial,16"
plot "curva1.txt" with lines title "Bubble Sort",
     "curva2.txt" with lines title "Bubble Sort Otimizado"
```

Solução 2: *Bubble Sort* (gráfico das duas implementações)



GNUPLOT:

```
set terminal jpeg
set output "grafico2.jpg"
set xlabel "n" font "Arial,16"
set ylabel "t(us)" font "Arial,16"
plot "curva1.txt" with lines title "Bubble Sort", \
     "curva2.txt" with lines title "Bubble Sort Otimizado"
```

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 considera o tempo de execução
- Pode 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
```

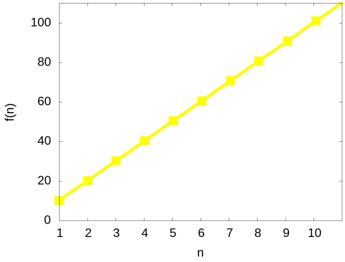
- Operação (multiplicação, atribuição e acesso às posições do arranjo): n

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 às posições do arranjo): n



GNUPLOT:

```
set terminal jpeg enhanced size 1280,960 font "arial,32.0"
set output "10n.jpg"
set xlabel "n"
set xtics 1,1,10
set ylabel "f(n)"
set style function linespoints
set style line 1 lw 8 lc rgb 'yellow' ps 4 pt 5 pi 10
plot [n=1:11][0:110] 10*n ls 1 notitle
```

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
```

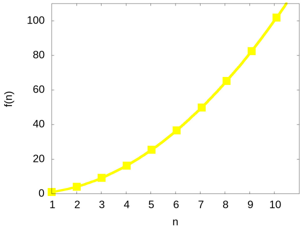
- Operação (multiplicação, atribuição e acesso às posições do arranjo): $n \times n$

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 às posições do arranjo): $n \times n$



GNUPLOT:

```
set terminal jpeg enhanced size 1280,960 font "arial,32.0"
set output "nxn.jpg"
set xlabel "n"
set xtics 1,1,10
set ylabel "f(n)"
set style function linespoints
set style line 1 lw 8 lc rgb 'yellow' ps 4 pt 5 pi 10
plot [n=1:11][0:110] n*n ls 1 notitle
```

Exercício (1/2)

Conte o número de operações executadas pelas seguintes funções.

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

2.
```
int funcao(int n) {
  int op = 0;
  for (int i=0; i<n; ++i)
      for (int j=0; j<n; ++j)
          op++;
  return op;
}
```

3.
```
int funcao(int n) {
  int op = 0;
  for (int i=0; i<n; ++i)
      for (int j=i+1; j<n; ++j)
          op++;
  return op;
}
```

4.
```
int funcao(int n) {
  int op = 0;
  for (int i=0; i<n; ++i)
      for (int j=0; j<2*i; ++j)
          op++;
  return op;
}
```

Exercício (2/2)

5.
```
int funcao(int n) {
  int op = 0;
  for (int i=0; i<n; i++)
      for (int j=i; j<i+3; j++)
          for (int k=i; k<j; k++)
              op++;
  return op;
}
```

6.
```
int funcao(int n) {
  int op = 1;
  if (n == 1) return op;
  return op + funcao(n-1) + funcao(n-1);
}
```

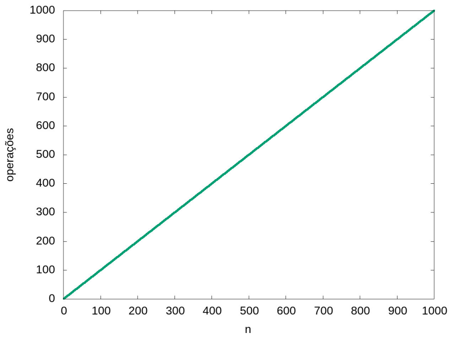
Exercício: soluções (1/6)

1.

```
int funcao(int n) {
  int op = 0;
  for (int i=0; i<n; ++i)
      op++;
  return op;
}
```

```
n = 10   /  i = 0..9    [op = 10]

n = 20   /  i = 0..19   [op = 20]

n = 50   /  i = 0..49   [op = 50]

n = 100  /  i = 0..99   [op = 100]

n = 1000 /  i = 0..999  [op = 1000]
```

Número de operações: $n \therefore O(n)$



GNUPLOT:

```
set terminal jpeg enhanced size 1280,960 font "arial,24.0"
set output "contagem01.jpg"
set xlabel "n"
set ylabel "operações"
plot "contagem01.txt" with lines lw 5 lt 2 notitle
```

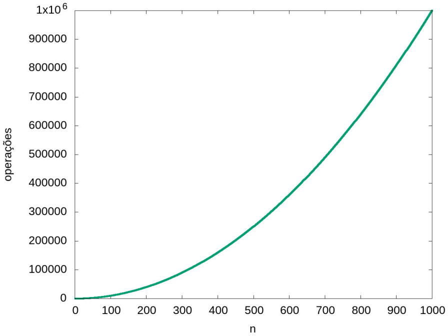
Exercício: soluções (2/6)

2

```
int funcao(int n) {
  int op = 0;
  for (int i=0; i<n; ++i)
      for (int j=0; j<n; ++j)
          op++;
  return op;
}
```

```
n = 10  /  i = 0  /  j = 0..9
           i = 1  /  j = 0..9
           i = 2  /  j = 0..9
           i = 3  /  j = 0..9
           i = 4  /  j = 0..9
           i = 5  /  j = 0..9
           i = 6  /  j = 0..9
           i = 7  /  j = 0..9
           i = 8  /  j = 0..9
           i = 9  /  j = 0..9  [op = 100]
```

Número de operações: $n \times n = n^2 \therefore O(n^2)$



GNUPLOT:

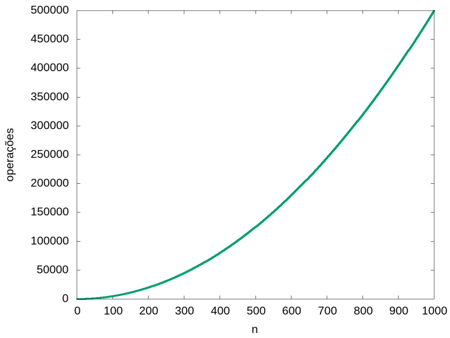
```
set terminal jpeg enhanced size 1280,960 font "arial,24.0"
set output "contagem02.jpg"
set xlabel "n"
set ylabel "operações"
plot "contagem02.txt" with lines lw 5 lt 2 notitle
```

Exercício: soluções (3/6)

3.
```
int funcao(int n) {
  int op = 0;
  for (int i=0; i<n; ++i)
      for (int j=i+1; j<n; ++j)
          op++;
  return op;
}
```

```
n = 10  /  i = 0  /  j = 1,2,3,4,5,6,7,8,9
           i = 1  /  j = 2,3,4,5,6,7,8,9
           i = 2  /  j = 3,4,5,6,7,8,9
           i = 3  /  j = 4,5,6,7,8,9
           i = 4  /  j = 5,6,7,8,9
           i = 5  /  j = 6,7,8,9
           i = 6  /  j = 7,8,9
           i = 7  /  j = 8,9
           i = 8  /  j = 9
           i = 9  /  j =        [op = 45]
```

Número de operações: $\frac{n\times(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} \therefore O(n^2)$



GNUPLOT:

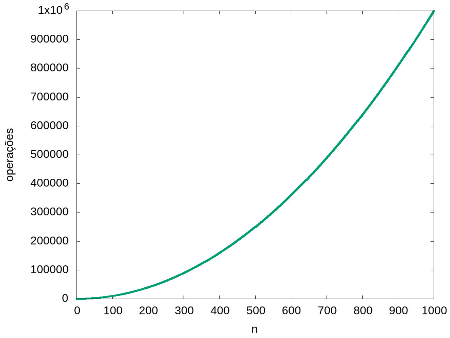
```
set terminal jpeg enhanced size 1280,960 font "arial,24.0"
set output "contagem03.jpg"
set xlabel "n"
set ylabel "operações"
plot "contagem03.txt" with lines lw 5 lt 2 notitle
```

Exercício: soluções (4/6)

4.
```
int funcao(int n) {
  int op = 0;
  for (int i=0; i<n; ++i)
      for (int j=0; j<2*i; ++j)
          op++;
  return op;
}
```

```
n = 10  /  i = 0  /  j =
           i = 1  /  j = 0..1
           i = 2  /  j = 0..3
           i = 3  /  j = 0..5
           i = 4  /  j = 0..7
           i = 5  /  j = 0..9
           i = 6  /  j = 0..11
           i = 7  /  j = 0..13
           i = 8  /  j = 0..15
           i = 9  /  j = 0..17  [op = 90]
```

Número de operações: $n \times (n-1) = n^2 - n \therefore O(n^2)$



GNUPLOT:

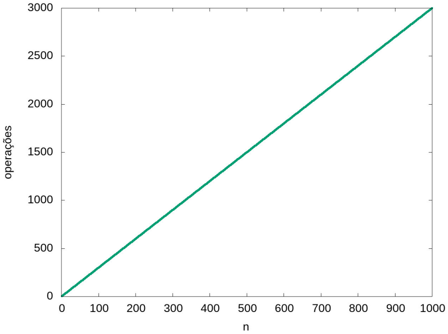
```
set terminal jpeg enhanced size 1280,960 font "arial,24.0"
set output "contagem04.jpg"
set xlabel "n"
set ylabel "operações"
plot "contagem04.txt" with lines lw 5 lt 2 notitle
```

Exercício: soluções (5/6)

5.
```
int funcao(int n) {
  int op = 0;
  for (int i=0; i<n; i++)
      for (int j=i; j<i+3; j++)
          for (int k=i; k<j; k++)
              op++;
  return op;
}
```

```
n = 10  / i = 0  /  j = 0  /  k =
                    j = 1  /  k = 0
                    j = 2  /  k = 0..1
          i = 1  /  j = 1  /  k =
                    j = 2  /  k = 1
                    j = 3  /  k = 1..2
          ...
          i = 9  /  j = 9  /  k =
                    j = 10 /  k = 9
                    j = 11 /  k = 9..10   [op = 30]
```

Número de operações: $n \times 3 = 3n \therefore O(n)$



GNUPLOT:

```
set terminal jpeg enhanced size 1280,960 font "arial,24.0"
set output "contagem05.jpg"
set xlabel "n"
set ylabel "operações"
plot "contagem05.txt" with lines lw 5 lt 2 notitle
```

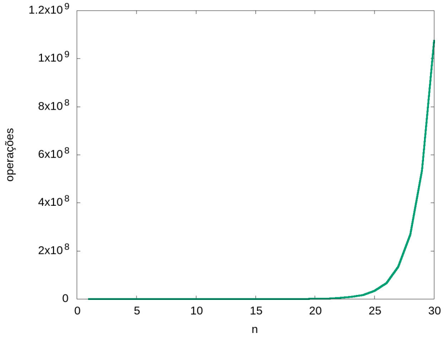
Exercício: soluções (6/6)

6

```
int funcao(int n) {
  int op = 1;
  if (n == 1) return op;
  return op + funcao(n-1) + funcao(n-1);
}
```

```
n = 1   [op = 1]
n = 2   [op = 3]
n = 3   [op = 7]
n = 4   [op = 15]
n = 5   [op = 31]
n = 6   [op = 63]
n = 7   [op = 127]
n = 8   [op = 255]
n = 9   [op = 511]
```

Número de operações: $2^n - 1 \therefore O(2^n)$



GNUPLOT:

```
set terminal jpeg enhanced size 1280,960 font "arial,24.0"
set output "contagem06.jpg"
set xlabel "n"
set ylabel "operações"
plot "contagem06.txt" with lines lw 5 lt 2 notitle
```

Funções

Funções

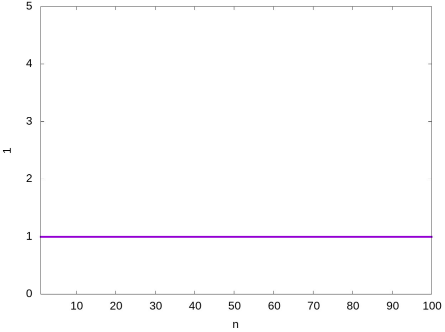
- Uma **classe de complexidade** é uma forma de agrupar algoritmos que apresentam complexidade similar. Por exemplo:
  - Complexidade **constante**: o algoritmo sempre ocupa a mesma quantidade de recursos
  - Complexidade **linear**: o algoritmo consome recursos de forma diretamente proporcional ao tamanho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

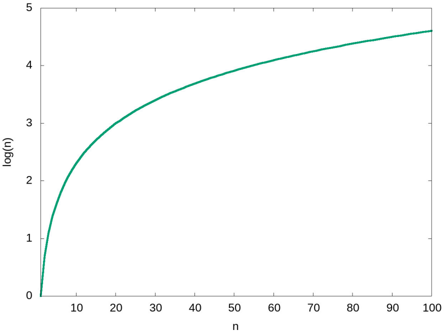


GNUPLOT:

```
set terminal jpeg enhanced size 1280,960 font "arial,24.0"
set output "1.jpg"
set xlabel "n"
set ylabel "1"
plot [n=1:100][0:5] 1 with lines lw 5 lt 1 notitle
```

Função Logaritmo: $\log n$

- $f(n) = \log n$ ou $f(n) = \log_2 n$
- 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*)

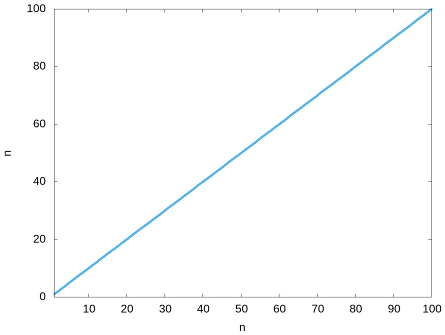


GNUPLOT:

```
set terminal jpeg enhanced size 1280,960 font "arial,24.0"
set output "log_n.jpg"
set xlabel "n"
set ylabel "log(n)"
plot [n=1:100][0:5] log(n) with lines lw 5 lt 2 notitle
```

Função Linear: n

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

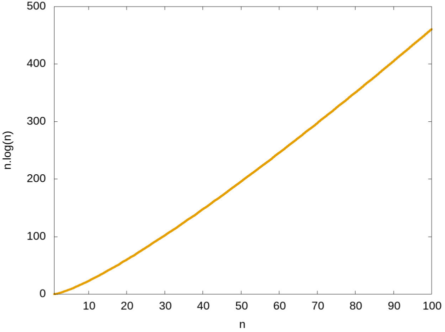


GNUPLOT:

```
set terminal jpeg enhanced size 1280,960 font "arial,24.0"
set output "n.jpg"
set xlabel "n"
set ylabel "n"
plot [n=1:100][0:100] n with lines lw 5 lt 3 notitle
```

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/`

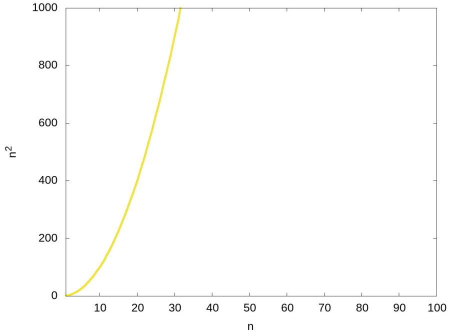


GNUPLOT:

```
set terminal jpeg enhanced size 1280,960 font "arial,24.0"
set output "n_log_n.jpg"
set xlabel "n"
set ylabel "n.log(n)"
plot [n=1:100][0:500] n*log(n) with lines lw 5 lt 4 notitle
```

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*

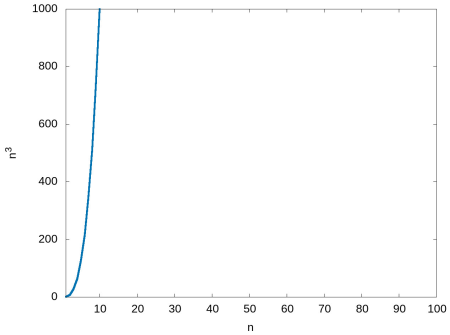


GNUPLOT:

```
set terminal jpeg enhanced size 1280,960 font "arial,24.0"
set output "n^2.jpg"
set xlabel "n"
set ylabel "n^2"
plot [n=1:100][0:1000] n**2 with lines lw 5 lt 5 notitle
```

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

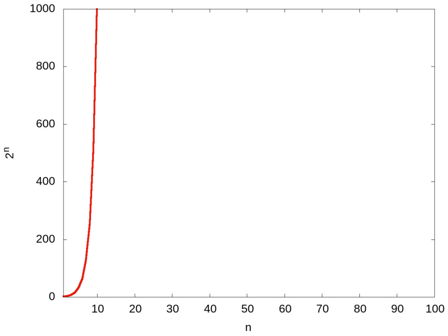


GNUPLOT:

```
set terminal jpeg enhanced size 1280,960 font "arial,24.0"
set output "n^3.jpg"
set xlabel "n"
set ylabel "n^3"
plot [n=1:100][0:1000] n**3 with lines lw 5 lt 6 notitle
```

Função Exponencial: a^n

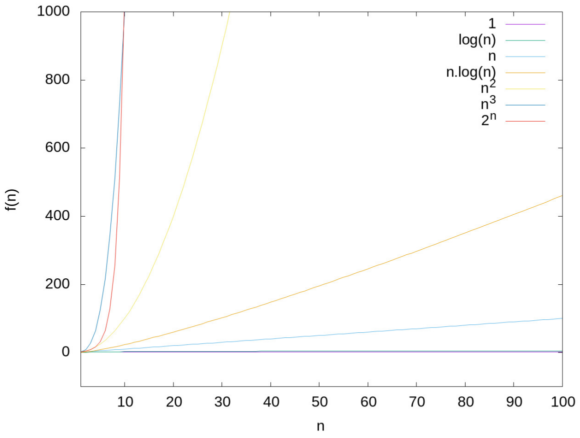
- Expoente é variável
- $f(n) = a^n$
- Algoritmos “ruins”, crescem abruptamente
- Aplicável apenas em problemas pequenos.
- Exemplos: quebrar senhas com força bruta e listar todos os subconjuntos de um conjunto S



GNUPLOT:

```
set terminal jpeg enhanced size 1280,960 font "arial,24.0"
set output "n^2.jpg"
set xlabel "n"
set ylabel "n^2"
plot [n=1:100][0:1000] n**2 with lines lw 5 lt 5 notitle
```

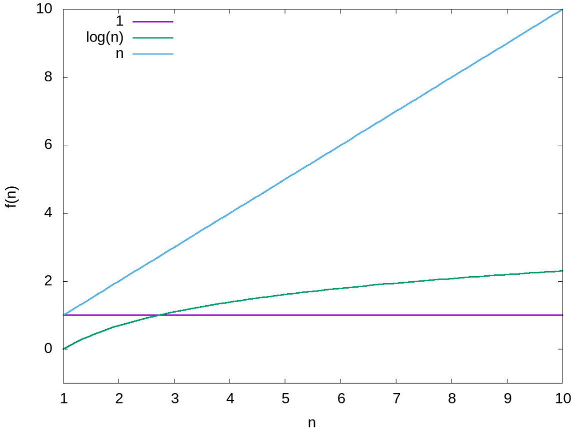
Comparativo entre as taxas de crescimento das funções (todas)



GNUPLOT:

```
set terminal jpeg enhanced size 1280,960 font "arial,24.0"
set output "todas.jpg"
set xlabel "n"
set ylabel "f(n)"
plot [n=1:100][-100:1000] 1        with lines lw 1 lt 1 title "1", \
                          log(n)   with lines lw 1 lt 2 title "log(n)", \
                          n        with lines lw 1 lt 3 title "n", \
                          n*log(n) with lines lw 1 lt 4 title "n.log(n)", \
                          n**2     with lines lw 1 lt 5 title "n^2", \
                          n**3     with lines lw 1 lt 6 title "n^3", \
                          2**n     with lines lw 1 lt 7 title "2^n"
```

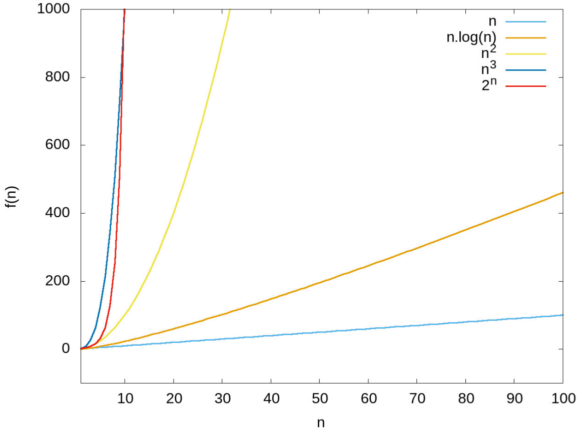
Comparativo entre as taxas de crescimento das funções (grupo 1)



GNUPLOT:

```
set terminal jpeg enhanced size 1280,960 font "arial,24.0"
set output "grupo1.jpg"
set key top left
set xlabel "n"
set ylabel "f(n)"
plot [n=1:10][-1:10] 1      with lines lw 3 lt 1 title "1", \
                     log(n) with lines lw 3 lt 2 title "log(n)", \
                     n      with lines lw 3 lt 3 title "n"
```

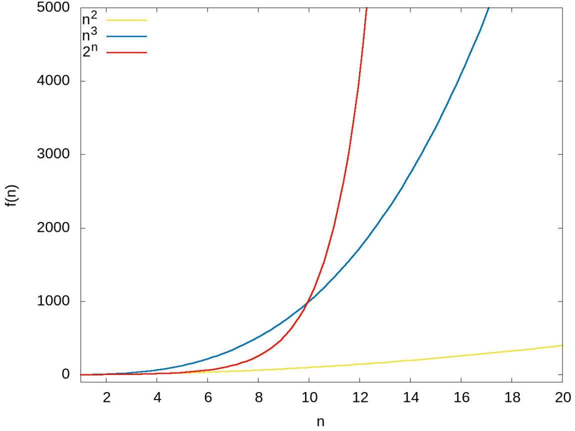
Comparativo entre as taxas de crescimento das funções (grupo 2)



GNUPLOT:

```
set terminal jpeg enhanced size 1280,960 font "arial,24.0"
set output "grupo2.jpg"
set xlabel "n"
set ylabel "f(n)"
plot [n=1:100][-100:1000] n        with lines lw 3 lt 3 title "n", \
                          n*log(n) with lines lw 3 lt 4 title "n.log(n)", \
                          n**2     with lines lw 3 lt 5 title "n^2", \
                          n**3     with lines lw 3 lt 6 title "n^3", \
                          2**n     with lines lw 3 lt 7 title "2^n"
```

Comparativo entre as taxas de crescimento das funções (grupo 3)



GNUPLOT:

```
set terminal jpeg enhanced size 1280,960 font "arial,24.0"
set output "grupo3.jpg"
set key top left
set xlabel "n"
set ylabel "f(n)"
plot [n=1:20][-100:5000] n**2     with lines lw 3 lt 5 title "n^2", \
                         n**3     with lines lw 3 lt 6 title "n^3", \
                         2**n     with lines lw 3 lt 7 title "2^n"
```

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

Exercícios

Exercício 1

Dois algoritmos para resolver o mesmo problema foram analisados e concluiu-se que eles executam os seguintes números de operações:

- Algoritmo A: $f_A(n) = 2n^2 + 5n$ operações
- Algoritmo B: $f_B(n) = 500n + 4000$ operações

Qual desses algoritmos consiste na melhor solução? Analise o número de operações de cada algoritmo para diferentes valores de n (por exemplo, $n = 10$, $n = 100$, $n = 1000$, etc.)

Exercício 2

Analise os algoritmos abaixo e identifique a sua classe de complexidade. Para verificar se a sua resposta está correta, implemente cada algoritmo, teste com diferentes valores de n e gere um gráfico usando o GNUPLOT.

1.
```
int funcao(int n) {
  int op = 0;
  for (int i=0; i<n; i++)
      for (int j=i; j<i+3; j++)
          for (int k=j; k<j+2; k++)
              op++;
  return op;
}
```

2.
```
int funcao(int n) {
  int op = 0;
  for (int i=0; i<n; i++)
      for (int j=i; j <2*i; j++)
          op++;
  return op;
}
```

3.
```
int funcao(int n) {
  int op = 0;
  for (int i=1; i<n; i=i+i)
      op++;
  return op;
}
```

4.
```
int funcao(int n) {
  int op = 0;
  for (int i=1; i<n; i++)
      for (int j=1; j<n; j=j+j)
          op++;
  return op;
}
```

Exercício 2

5.
```
int funcao(int n) {
  int op = 0;
  for (int i=0; i<n; i++)
      for (int j=0; j<n; j++)
          for (int k=0; k<n; k++)
              op++;
  return op;
}
```

6.
```
int funcao(int n) {
  int op = 0;
  for (int i=0; i<n; i++)
      for (int j=i; j <i+2; j++)
          for (int k=0; k<n; k++)
              op++;
  return op;
}
```

Exercício 3

Faça um algoritmo para o problema abaixo e analise a sua complexidade.

Problema:

- Dado um intervalo [1;V]
- Determinar quantas sequencias de valores entre 1 e V somam exatamente V

Exemplo:

- V=15
- Sequências
  - 1+2+3+4+5
  - 4+5+6
  - 7+8
  - 15
- Resposta: 4 sequências de valores somam exatamente 15

Notação O

Notação O

- O número de passos durante a execução de um algoritmo pode variar
- Exemplo: ordenar um vetor
  $\boxed{2}\ \boxed{1}\ \boxed{3}\ \boxed{4}\ \boxed{5}\ \boxed{6}$ x $\boxed{6}\ \boxed{5}\ \boxed{4}\ \boxed{3}\ \boxed{2}\ \boxed{1}$
- Existe o limite superior e o limite inferior para cada algoritmo
- Exemplo:
  `https://www.toptal.com/developers/sorting-algorithms/bubble-sort`

Notação O

- Limite inferior
  - Menor tempo de execução sobre todas as entradas de tamanho n
  - Exemplo: *Bubble Sort* recebe vetor de entrada quase ordenado
- Limite superior
  - Maior tempo de execução sobre todas as entradas de tamanho n
  - Exemplo: *Bubble Sort* recebe vetor de entrada ordenado de trás para frente
- Caso "médio"
  - Média dos tempos de execução sobre todas as entradas de tamanho n

Notação O

- Tempo cresce conforme a entrada de dados cresce
- Determinar o "tempo médio" é difícil
- Portanto, para análise de algoritmos quase sempre se quer saber o limite superior
  - Mais fácil de analisar
  - Crucial para determinadas aplicações

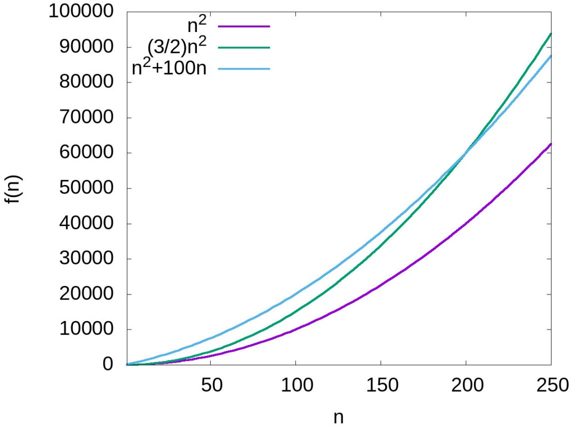
Notação O

- Importante:
  - Concentrar-se na taxa de crescimento do tempo de execução como uma função do tamanho da entrada n
- Por exemplo:
  - No algoritmo que verifica maior número de um arranjo de inteiros, o tempo de execução cresce proporcionalmente a n
  - Tempo de execução: n vezes um fator constante

Notação O

- Ao ver uma expressão $f(n) = a^n$, normalmente se pensa em valores pequenos
  - É o que se consegue resolver
- Análise de algoritmos
  - Ignora os valores pequenos e concentra-se nos valores enormes de n
- Para grandes valores de n, as funções n^2 , $(3/2)n^2$, $n^2 + 100n$, por exemplo, crescem com a mesma velocidade, portanto são equivalentes

Notação O



GNUPLOT:

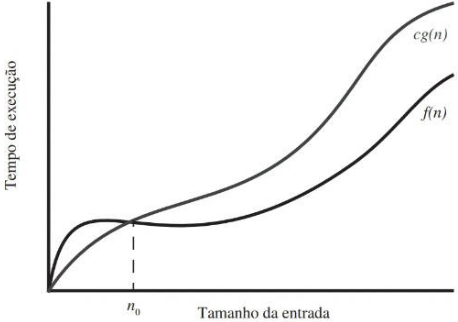
```
set terminal jpeg enhanced size 1280,960 font "arial,32.0"
set output "grafico2.jpg"
set key top left
set xlabel "n"
set ylabel "f(n)"
plot [n=1:250] n*n with lines title "n^2" lw 4, \
               1.5*n*n with lines title "(3/2)n^2" lw 4, \
               n*n+100*n with lines title "n^2+100n" lw 4
```

Notação Assintótica

- Assintótico (Adj.)
  - Quando se quer descrever o comportamento quando se aproxima de um limite, ou seja, quando n tende ao infinito
  - Aquilo que está relativamente próximo
  - Para todos os valores suficientemente grandes
  - O comportamento a ser observado em uma função $f(n)$, quando n tende ao infinito
  - Relativo à Assintota
    - (Mat.) linha reta relacionada com uma curva, cuja distância entre elas se torna infinitamente pequena, a partir de determinado ponto
    - (Fig.) Caminho que se aproxima continuamente de um ideal sem jamais o atingir
- Notações
  - Notação O
  - Notação Ω (ômega)
  - Notação Θ (*theta*)

Notação O

- Sejam $f(n)$ e $g(n)$ funções mapeando inteiros não negativos em números reais
- Diz-se que $f(n)$ é $O(g(n))$ se existe uma constante real $c > 0$ e uma constante inteira $n_0 \geq 1$ tais que $f(n) \leq cg(n)$, para todo inteiro $n \geq n_0$



Notação O

- Notação O:
  - $f(n)$ é **O** de $g(n)$
  - $f(n)$ é **da ordem** de $g(n)$
- $f(n) = O(g(n))$
  - Significa que $g(n)$ cresce mais (ou da mesma forma que) a função $f(n)$ à medida que n cresce para o infinito
  - Para isso, pode ser preciso multiplicar $g(n)$ por uma constante
- Notação O é usada para
  - **Caracterizar o tempo de execução** e limites espaciais em função de um parâmetro n que varia de problema para problema

Notação O

- Exemplo: algoritmo para encontrar o maior elemento de um arranjo de inteiros
  - n representa o número de elementos no arranjo
  - Usando a notação O se pode afirmar (independente do computador que será usado):
    - Este algoritmo executa em tempo $O(n)$
  - Justificativa:
    - O número de operações primitivas executadas pelo algoritmo é constante em cada iteração
    - Pode-se dizer que o tempo de execução do algoritmo para uma entrada de tamanho n é no máximo uma constante, vezes n

Propriedades da notação O

- Permite ignorar os fatores constantes e os termos de menor ordem, focando nos principais componentes da função que afetam seu crescimento
- Se $f(n)$ é um polinômio de grau d, isto é,

$$f(n) = a_0 + a_1 n + ... + a_d n^d$$

  e $a_d > 0$, então $f(n)$ é $O(n^d)$
- Deve-se descrever a função O em termos simples

Propriedades da notação O

- Portanto, o termo de mais alto grau em um polinômio é o termo que determina a taxa de crescimento assintótico do polinômio
- Exemplos:
  - $5n^2 + 3n\log n + 2n + 5$ é $O(n^2)$
  - $20n^3 + 10n\log n + 5$ é $O(n^3)$
  - $3\log n + 2$ é $O(\log n)$
  - 2^{n+2} é $O(2^n)$
  - $2n + 100\log n$ é $O(n)$

Notação O

- As sete funções apresentadas são as mais usadas em conjunto com a notação O
  $1,\ \log n,\ n,\ n \log n,\ n^2,\ n^3,\ a^n$
- Caracterizam os tempos de execução e consumo de memória dos algoritmos
- Os nomes destas funções são usados para referenciar o tempo de execução
- Exemplos:
  - Algoritmo que executa no limite superior em tempo $4n^2 + n \log n$ é um algoritmo de tempo quadrático, pois executa em tempo $O(n^2)$
  - Algoritmo que executa no limite superior em $5n + 20 \log n + 4$ é um algoritmo de tempo linear, ou seja, $O(n)$

Notação Ω e Notação Θ

Notação Ω e Notação Θ

- Notação O
  Maneira assintótica de dizer que uma função é "menor que ou igual a" outra função
- Notação Ω
  Maneira assintótica de dizer que uma função cresce a uma taxa que é "maior ou igual a" outra função
- Notação Θ
  Permite dizer que duas funções crescem à mesma taxa

Notação Ω

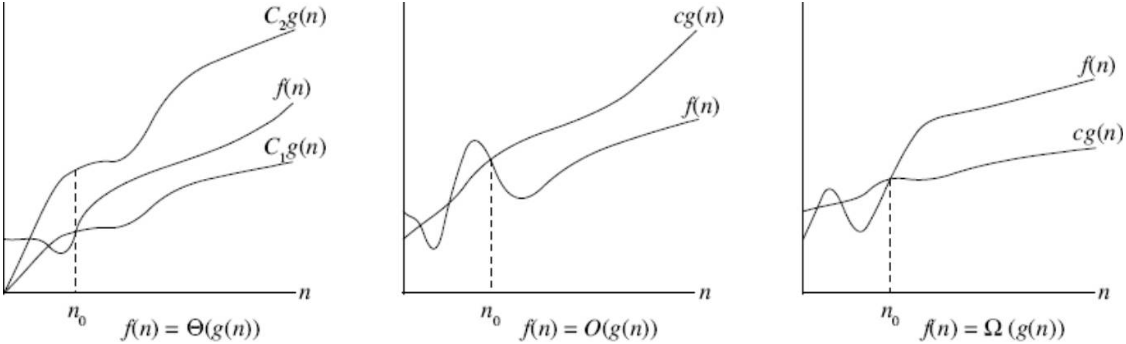
- Sejam $f(n)$ e $g(n)$ funções mapeando inteiros não negativos em números reais
- Diz-se que $f(n)$ é $\Omega(g(n))$ se $g(n)$ é $O(f(n))$, ou seja, se existe uma constante real $c > 0$ e uma constante inteira $n_0 \geq 1$ tais que $f(n) \geq cg(n)$, para todo inteiro $n \geq n_0$
- Permite dizer que uma função é assintoticamente maior que ou igual a outra, exceto por um fator constante
- Se diz que $f(n)$ **é ômega de** $g(n)$
- Exemplo: $3n \log n + 2n$ é $\Omega(n \log n)$

Notação Θ

- Diz-se que $f(n)$ é $\Theta(g(n))$, se $f(n)$ é $O(g(n))$ e $f(n)$ é $\Omega(g(n))$, ou seja, existem constantes reais $c' > 0$ e $c'' > 0$ e uma constante inteira $n_0 \geq 1$ tais que $c'g(n) \leq f(n) \leq c''g(n)$, para $n \geq n_0$
- Se diz que $f(n)$ **é theta** de $g(n)$
- Exemplo: $3n \log n + 4n + 5 \log n$ é $\Theta(n \log n)$

Exemplo de gráficos das notações O, Ω e Θ

- Notação Θ limita uma função entre valores constantes
- Notação O dá um limite superior para uma função dentro de um valor constante
- Notação Ω dá um limite inferior para uma função dentro de um valor constante



Fonte: Cormen *et al.* (2012)

Análise assintótica

Análise assintótica

- Suponha dois algoritmos, A e B, para resolver um mesmo problema
  - A tem um tempo de execução $O(n)$
  - B tem um tempo de execução $O(n^2)$
  - Qual deles é melhor?
  - A é assintoticamente melhor que B

Análise assintótica

- Pode-se usar a notação O para ordenar classes de funções por seu crescimento assintótico
  $1 \quad \log n \quad n \quad n \log n \quad n^2 \quad n^3 \quad 2^n$

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4.096	65.536
32	5	32	160	1.024	32.768	4.294.967.296
64	6	64	384	4.096	262.144	$1{,}84 \times 10^{19}$
128	7	128	896	16.384	2.097.152	$3{,}40 \times 10^{38}$
256	8	256	2.048	65.536	16.777.216	$1{,}15 \times 10^{77}$
512	9	512	4.608	262.144	134.217.728	$1{,}34 \times 10^{154}$

Análise assintótica

- Para analisar algoritmos com a notação O, o mais apropriado é dizer:
  $f(n)$ é $O(g(n))$,
  ou
  $f(n) \in O(g(n))$

O que é um algoritmo rápido?

- Em geral, um algoritmo executando em tempo $O(n \log n)$, com um fator constante razoável, pode ser considerado eficiente
- Até um método $O(n^2)$ pode ser suficientemente rápido em alguns contextos (n pequeno)
- Algoritmo executando em tempo $O(2^n)$ dificilmente pode ser considerado eficiente

Análise assintótica

- Notações O, Ω e Θ fornecem uma linguagem conveniente para análise de estruturas de dados e algoritmos
  - Permitem a concentração nos aspectos gerais, em vez dos detalhes

Exercício 1

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 2

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 $O(2n)$

Exercício 3

Analise os trechos de algoritmos abaixo e identifique suas respectivas classes de complexidade.

1. **Algoritmo I:**

```
for (i=0; i < n; i++)
    for (j=0; j < n; j++)
        a += i*3 + aa[i][j];
```

2. **Algoritmo II:**

```
for (i=1; i < n; i=i+i)
    b = b >> i;
```

3. **Algoritmo III:**

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

4. **Algoritmo IV:**

```
for (i=0; i < n; i++)
    for (j=i; j < i+5; j++)
        for (k=0; k < n; k++)
            d++;
```

5. **Algoritmo V:**

```
for (i=0; i < 127; i++)
    for (j = 0; j < 127; j++)
        e[i][j] = 0;
```

Exercício 4

Considere as seguintes funções:

$$f_1(n) = O(n) \qquad f_2(n) = O(\log n) \qquad f_3(n) = O(2^n) \qquad f_4(n) = O(n^2)$$

A sequência que apresenta as funções acima ordenadas, pela sua taxa de crescimento, de forma crescente é:

A. $f_2 - f_1 - f_4 - f_3$
B. $f_3 - f_4 - f_1 - f_2$
C. $f_1 - f_3 - f_2 - f_4$
D. $f_1 - f_2 - f_3 - f_4$
E. $f_2 - f_1 - f_3 - f_4$

Exercício 5

Qual a notação O dos algoritmos que têm as seguintes taxas de crescimento assintóticas?

a. $3n \log n + 2n + 5$
b. $1000n \log n + 15n^3$
c. $8 \log n + n$
d. $500n^5 + 2n$
e. $10 \log n + 1521$
f. $3n + 4500n$
g. $3n + 753 \log n + 4$

Créditos

Créditos

- Estas lâminas contêm trechos de materiais criados e disponibilizados pelos professores Isabe Harb Manssour e Iaçanã Ianiski Weber.