Correção e Eficiência de Algoritmos

Aula 02

Ivone P. Matsuno Yugoshi
ivone.matsuno@ufms.br

Ronaldo Fiorilo dos Santos
ronaldo.santos@ufms.br

Universidade Federal de Mato Grosso do Sul
Câmpus de Três Lagoas
Bacharelado em Sistemas de Informação

Algoritmos e Programação II

Algoritmos e programas

- Algoritmo ou programa é uma sequência bem definida de passos (descritos em uma linguagem de programação específica) que transforma um conjunto de valores – a entrada – em um outro conjunto de valores – a saída

- Algoritmo é uma ferramenta para solucionar um problema computacional

Algoritmos e programas

Problema da busca

Dado um número inteiro n, com $1 \leq n \leq 100$, um conjunto C de n números inteiros e um número inteiro x, verificar se x encontra-se no conjunto C

Algoritmos e programas

```
#include <stdio.h>
#define MAX 100
int main(void)
{
   int n, i, C[MAX], x;

   scanf("%d", &n);
   for (i = 0; i < n; i++)
      scanf("%d", &C[i]);

   scanf("%d", &x);

   i = 0;
   while (i < n && C[i] != x)
      i++;

   if (i < n)
      printf("%d está na posição %d de C\n", x, i);
   else
      printf("%d não pertence ao conjunto C\n", x);
   return 0;
}
```

Algoritmos e programas

- Se um programa para com a resposta correta então dizemos que o programa é correto
  - Um programa correto soluciona o problema computacional associado
- Um programa incorreto pode sequer parar, para alguma entrada, ou pode parar mas com uma resposta indesejada
- Devemos, assim, ser capazes de mostrar que nossos algoritmos são corretos.
  - testes só podem mostrar que o algoritmo está errado
  - somente a análise pode provar que o algoritmo está correto

- Usando **indução matemática** como ferramenta, a correção de uma função recursiva é dada naturalmente, visto que sua estrutura intrínseca nos fornece muitas informações úteis para uma prova de correção.

```
/* Recebe um dois números inteiros x e n
e devolve x a n-ésima potência */

int potR(int x, int n)
{
   if (n == 0)
      return 1;
   else
      return x * potR(x, n-1);
}
```

Correção de funções recursivas

- **Proposição:** A função `potR` recebe dois números inteiros x e n e devolve corretamente o valor de x^n.
- **Demonstração:** Vamos mostrar a proposição por indução em n.
  - Se $n = 0$ então a função devolve $1 = x^0 = x^n$.
  - Suponha que a função esteja correta para todo k, com $0 < k < n$. Ou seja, a função `potR` com parâmetros x e k devolve corretamente o valor x^k para todo k, com $0 < k < n$.
  - Agora, vamos mostrar que a função está correta para $n > 0$.
    - Como $n > 0$ então a última linha do corpo da função é executada: `return x * potR(x, n-1);`
    - Então, como $n - 1 < n$, por hipótese de indução, a chamada `potR(x, n-1)` devolve corretamente o valor x^{n-1}
    - Logo, a chamada de `potR(x, n)` devolve $x * x^{n-1} = x^n$

$\square$

Correção de funções não-recursivas e invariantes

- Funções não-recursivas, em geral, possuem uma ou mais estruturas de repetição (processos iterativos da função).
- Mostrar a correção de uma função não-recursiva é um trabalho mais árduo do que de uma função recursiva.
- Isso porque devemos extrair informações úteis da função que explicam o funcionamento do processo iterativo e que nos permitam usar indução matemática para, por fim, mostrar que o processo está correto.
- Essas informações são denominadas **invariantes** de um processo iterativo.

Correção de funções não-recursivas e invariantes

- Um invariante de um processo iterativo é uma relação entre os valores das variáveis envolvidas neste processo que vale no início de cada iteração do mesmo.
- Devemos provar três elementos sobre um invariante de um processo iterativo:
  - **Inicialização**: é verdadeiro antes da primeira iteração da estrutura de repetição;
  - **Manutenção**: se é verdadeiro antes do início de uma iteração da estrutura de repetição, então permanece verdadeiro antes da próxima iteração;
  - **Término**: quando a estrutura de repetição termina, o invariante nos dá uma propriedade útil que nos ajuda a mostrar que o algoritmo ou programa está correto.
- Como usamos invariantes para mostrar a correção de um algoritmo, a terceira propriedade é a mais importante.

Correção de funções não-recursivas e invariantes

```
/* Recebe um número inteiro n > 0 e uma sequência de n números
inteiros e mostra o resultado da soma desses números */
int main(void)
{
   int n, i, num, soma;
   printf("Informe n: ");
   scanf("%d", &n);
   soma = 0;
   for (i = 1; i <= n; i++) {
      /* variável soma contém o somatório dos
      primeiros i-1 números fornecidos */
      printf("Informe um número: \n");
      scanf("%d", &num);
      soma = soma + num;
   }
   printf("Soma dos %d números é %d\n", n, soma);
   return 0;
}
```

Correção de funções não-recursivas e invariantes

- **Proposição**: O programa computa corretamente a soma de n números inteiros fornecidos pelo(a) usuário(a).
- **Demonstração**:
  Provar que o programa está correto significa mostrar que para qualquer valor de n e qualquer sequência de n números, a variável *soma* conterá, ao final do processo iterativo, o valor

$$soma = \sum_{i=1}^{n} num_i$$

Correção de funções não-recursivas e invariantes

- No início da primeira iteração, a variável *soma* contém o valor 0 (zero) e i contém 1, é verdade que a variável *soma* contém a soma dos $i - 1$ primeiros números fornecidos pelo(a) usuário(a).
- Suponha agora que o invariante valha no início da i-ésima iteração, com $1 < i < n$.
- Vamos mostrar que o invariante vale no início da última iteração, quando i contém o valor n. Por hipótese de indução, a variável *soma* contém o valor

$$\alpha = \sum_{i=1}^{n-1} num_i$$

Correção de funções não-recursivas e invariantes

- Dessa forma, no decorrer da n-ésima iteração, o(a) usuário(a) deve informar um número que será armazenado na variável num_n e, então, a variável soma conterá o valor

$$soma = \alpha + num_n$$

- Dessa forma, no decorrer da n-ésima iteração, o(a) usuário(a) deve informar um número que será armazenado na variável num_n e, então, a variável soma conterá o valor

$$soma = \alpha + num_n = \left(\sum_{i=1}^{n-1} num_i\right) + num_n$$

Correção de funções não-recursivas e invariantes

- Dessa forma, no decorrer da n-ésima iteração, o(a) usuário(a) deve informar um número que será armazenado na variável num_n e, então, a variável soma conterá o valor

$$soma = \alpha + num_n = \left(\sum_{i=1}^{n-1} num_i\right) + num_n = \sum_{i=1}^{n} num_i$$

- Portanto, isso mostra que o programa de fato realiza a soma dos n números inteiros fornecidos pelo(a) usuário(a).

$\square$

Correção de funções não-recursivas e invariantes

```
#define MAX 100
/* Recebe um número inteiro n > 0 e uma sequência de n
números inteiros e mostra um maior valor da sequência */
int main(void)
{
   int n, vet[MAX], i, max;
   scanf("%d", &n);
   for (i = 0; i < n; i++)
      scanf("%d", &vet[i]);
   max = vet[0];
   for (i = 1; i < n; i++) {
      /* max é um maior elemento em vet[0..i-1] */
      if (vet[i] > max)
         max = vet[i];
   }
   printf("%d\n", max);
   return 0;
}
```

- **Proposição**: O programa encontra um elemento máximo de um conjunto de n números fornecidos pelo(a) usuário(a).
- **Demonstração**:
  Provar que o programa está correto significa mostrar que para qualquer valor de n e qualquer sequência de n números fornecidos pelo(a) usuário(a) e armazenados em um vetor *vet*, a variável *max* conterá, ao final do processo iterativo, o valor do elemento máximo em $vet[0..n-1]$.

- No início da primeira iteração do processo iterativo, *max* contém o valor armazenado em $vet[0]$ e, em seguida, a variável i é inicializada com o valor 1, então é verdade que a variável *max* contém o elemento máximo em $vet[0..i-1]$.
- Suponha agora que o invariante valha no início da i-ésima iteração, com $1 < i < n - 1$.
- Vamos mostrar que o invariante vale no início da última iteração, quando i contém o valor $n-1$. Por hipótese de indução, no início desta iteração a variável *max* contém o valor do elemento máximo de $vet[0..n-2]$. Então, no decorrer dessa iteração, o valor $vet[n-1]$ é comparado com *max* e dois casos devem ser avaliados:

- $vet[n-1] > max$
  Isso significa que o valor $vet[n-1]$ é maior que qualquer valor armazenado em $vet[0..n-2]$. Assim, a variável *max* é atualizada com $vet[n-1]$ e portanto a variável *max* conterá, ao final desta última iteração, o elemento máximo de $vet[0..n-1]$.
- $vet[n-1] \leq max$
  Isso significa que existe pelo menos um valor em $vet[0..n-2]$ que é maior ou igual a $vet[n-1]$. Por hipótese de indução, esse valor está armazenado em *max*. Assim, ao final desta última iteração, a variável *max* conterá o elemento máximo de $vet[0..n-1]$.
- Portanto, isso mostra que o programa de fato encontra o elemento máximo em uma sequência de n números inteiros armazenados em um vetor.

$\square$

Exercícios

1. Mostre que os algoritmos escritos para os exercícios 1 e 2, do início da aula, estão corretos.
2. Escreva duas versões (recursiva e iterativa) de uma função que calcule a soma dos elementos positivos de um vetor $A[0..n-1]$.
    - Mostre que suas funções estão corretas.

Eficiência de algoritmos

Motivação

- Tenho mais de um algoritmo/programa para solucionar um problema computacional. Qual devo escolher?
  - Aquele que gasta *menos* tempo?
  - Aquele que gasta *menos* memória?
  - Aquele que faz *menos* comunicação entre processos?
  - Aquele que usa/acessa *menos* portas lógicas?

Análise de algoritmos

- Antes de analisar um algoritmo ou programa, devemos conhecer o modelo da tecnologia de computação usada na máquina em que implementamos o programa, para estabelecer os custos associados aos recursos que o programa usa.
- Na análise de algoritmos, consideramos regularmente um modelo de computação genérico chamado de máquina de acesso aleatório (do inglês *random acess machine* – RAM) com um processador.
- Nesse modelo, as instruções são executadas uma após outra, sem concorrência.

Análise de algoritmos

- A análise de um programa pode ser uma tarefa desafiadora envolvendo diversas ferramentas matemáticas tais como combinatória discreta, teoria das probabilidades, álgebra e etc.
- A análise de um programa também depende do número de elementos fornecidos na entrada
  - Procurar um elemento x em um conjunto C com milhares de elementos certamente gasta mais tempo que em um conjunto C com 3 elementos
  - Mesmo para dois conjuntos diferentes mas com a mesma quantidade de elementos, uma busca pode ser mais demorada que a outra

Análise de algoritmos

- O tempo gasto por um algoritmo cresce com o tamanho da entrada, isto é, o número de itens na entrada
- O tempo de execução de um algoritmo sobre uma entrada particular é o número de passos realizados por ele
- Uma linha i de um algoritmo gasta uma quantidade constante de tempo $c_i > 0$

Análise de algoritmos

	Custo	Vezes
`#include <stdio.h>`	c_1	1
`#define MAX 100`	c_2	1
`int main(void)`	c_3	1
`{`	0	1
`int n, i, C[MAX], x;`	c_4	1
`scanf("%d", &n);`	c_5	1
`for (i = 0; i < n; i++)`	c_6	$n+1$
`scanf("%d", &C[i]);`	c_7	n
`scanf("%d", &x);`	c_8	1
`for (i = 0; i < n && C[i] != x; i++)`	c_9	t_i
`;`	0	$t_i - 1$
`if (i < n)`	c_{10}	1
`printf("%d está na posição %d de C\n", x, i);`	c_{11}	1
`else`	c_{12}	1
`printf("%d não pertence ao conjunto C\n", x);`	c_{13}	1
`return 0;`	c_{14}	1
`}`	0	1

- O tempo de execução $T(n)$ do algoritmo é dado pela soma dos tempos para cada sentença executada
- Ou seja, devemos somar os produtos das colunas ***Custo*** e ***Vezes***

$$T(n) = c_1 + c_2 + c_3 + c_4 + c_5 + c_6(n+1) + c_7 n + c_8 + c_9 t_i + c_{10} + c_{11} + c_{12} + c_{13} + c_{14} \, .$$

- Tempo de execução $T(n)$ do algoritmo da busca:

$$T(n) = c_1 + c_2 + c_3 + c_4 + c_5 + c_6(n+1) + c_7 n + c_8 + c_9 t_i + c_{10} + c_{11} + c_{12} + c_{13} + c_{14}\,.$$

- Melhor caso: ocorre se x encontra-se na primeira posição do vetor C; então, $t_i = 1$ e assim

$$\begin{aligned} T(n) &= c_1 + c_2 + c_3 + c_4 + c_5 + c_6(n+1) + c_7 n + c_8 + \\ &\quad c_9 + c_{10} + c_{11} + c_{12} + c_{13} + c_{14} \\ &= (c_6 + c_7)n + c_1 + \ldots + c_6 + c_8 + \ldots + c_{14} \\ &= an + b\,. \end{aligned}$$

- Tempo de execução $T(n)$ do algoritmo da busca:

$$T(n) = c_1 + c_2 + c_3 + c_4 + c_5 + c_6(n+1) + c_7 n + c_8 + c_9 t_i + c_{10} + c_{11} + c_{12} + c_{13} + c_{14}\,.$$

- Pior caso: ocorre se x não se encontra no vetor C; então, $t_i = n+1$ e assim

$$\begin{aligned} T(n) &= c_1 + c_2 + c_3 + c_4 + c_5 + c_6(n+1) + c_7 n + c_8 + \\ &\quad c_9(n+1) + c_{10} + c_{11} + c_{12} + c_{13} + c_{14} \\ &= (c_6 + c_7 + c_9)n + c_1 + \ldots + c_6 + c_8 + \ldots + c_{14} \\ &= an + b\,. \end{aligned}$$

Análise de algoritmos

- Geralmente estamos interessados no tempo de execução de pior caso de um algoritmo
- Como o tempo de execução de pior caso de um programa é um limitante superior para seu tempo de execução para qualquer entrada, temos então uma garantia que o programa nunca vai gastar mais tempo que esse estabelecido.
- Além disso, o pior caso ocorre muito frequentemente nos programas em geral, como no caso do problema da busca.

- Quando falamos em tempo de execução de um algoritmo, estamos interessados na taxa de crescimento ou ordem de crescimento de uma função que descreve esse tempo
  - Consideramos apenas o *maior* termo da fórmula e ignoramos constantes
  - Dizemos assim que o algoritmo da busca tem tempo de execução de pior caso $O(n)$
- Um algoritmo é mais eficiente que outro se seu tempo de execução de pior caso tem ordem de crescimento menor
  - Essa avaliação pode ser errônea para pequenas entradas mas, para entradas suficientemente grandes, um programa com tempo de execução de pior caso $O(n)$ executará mais rapidamente no pior caso que um programa com tempo de execução de pior caso $O(n^2)$.

Ordem de crescimento de funções

- Quando olhamos para entradas grandes o suficiente para fazer com que somente a taxa de crescimento da função que descreve o tempo de execução de um algoritmo seja relevante, estamos estudando na verdade a eficiência assintótica de um algoritmo.
  - Isto é, concentramo-nos em saber como o tempo de execução de um programa cresce com o tamanho da entrada no *limite*, quando o tamanho da entrada cresce ilimitadamente.
- Usualmente, um programa que é assintoticamente mais eficiente será a melhor escolha para todas as entradas, excluindo talvez algumas entradas pequenas

Ordem de crescimento de funções

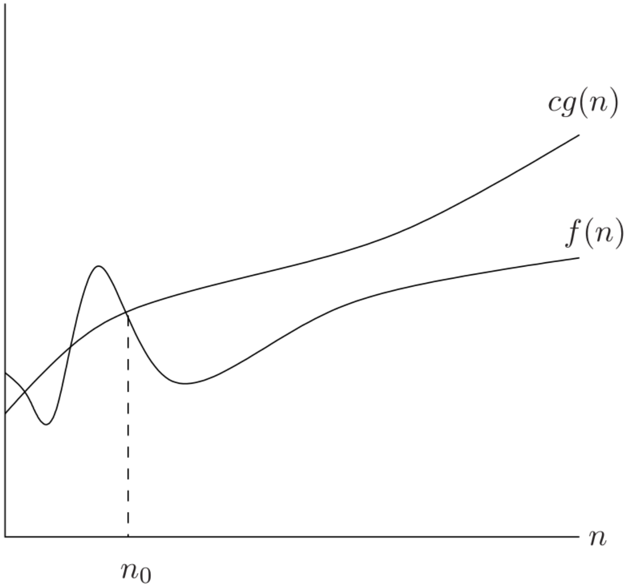
Definição

Para uma dada função $g(n)$, denotamos por $O(g(n))$ o conjunto de funções

$$O(g(n)) = \{f(n) \colon \text{existem constantes positivas } c \text{ e } n_0 \text{ tais que } 0 \leq f(n) \leq cg(n) \text{ para todo } n \geq n_0\} \,.$$

- $f(n)$ pertence ao conjunto $O(g(n))$ se existe uma constante positiva c tal que $f(n)$ "não seja maior que" $cg(n)$, para n suficientemente grande
- Usamos a notação O para fornecer um limitante assintótico superior sobre uma função, dentro de um fator constante
- Usamos $f(n) = O(g(n))$ para $f(n) \in O(g(n))$

Ordem de crescimento de funções



Será que $4n + 1 = O(n)$?

Existem constantes positivas c e n_0 tais que

$$4n + 1 \leq cn$$

para todo $n \geq n_0$. Tome $c = 5$ e então

$$\begin{aligned} 4n + 1 &\leq 5n \\ 1 &\leq n\,, \end{aligned}$$

ou seja, para $n_0 = 1$, a desigualdade $4n + 1 \leq 5n$ é satisfeita para todo $n \geq n_0$ e assim, $4n + 1 = O(n)$

Análise da ordenação por trocas sucessivas

```
void trocas_sucessivas(int n, int v[MAX])
{
   int i, j, aux;

   for (i = n-1; i > 0; i--)
      for (j = 0; j < i; j++)
         if (v[j] > v[j+1]) {
            aux = v[j];
            v[j] = v[j+1];
            v[j+1] = aux;
         }
}
```

Análise da ordenação por trocas sucessivas

	Custo	Vezes
`void trocas_sucessivas(int n, int v[MAX]`	c_1	1
`{`	0	1
`   int i, j, aux;`	c_2	1
`   for (i = n-1; i > 0; i--)`	c_3	n
`      for (j = 0; j < i; j++)`	c_4	$\sum_{i=1}^{n-1}(i+1)$
`         if (v[j] > v[j+1]) {`	c_5	$\sum_{i=1}^{n-1} i$
`            aux = v[j];`	c_6	$\sum_{i=1}^{n-1} t_i$
`            v[j] = v[j+1];`	c_7	$\sum_{i=1}^{n-1} t_i$
`            v[j+1] = aux;`	c_8	$\sum_{i=1}^{n-1} t_i$
`         }`	0	$\sum_{i=1}^{n-1} i$
`}`	0	1

Análise da ordenação por trocas sucessivas

- Melhor caso: quando a sequência de entrada com n números inteiros é fornecida em ordem crescente ($t_i = 0$ para todo i)

$$
\begin{aligned}
T(n) &= n + \sum_{i=1}^{n-1}(i+1) + \sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} t_i + \sum_{i=1}^{n-1} t_i + \sum_{i=1}^{n-1} t_i \\
&= n + 2\sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} 1 + 3\sum_{i=1}^{n-1} t_i \\
&= n + 2\sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} 1 + 3\sum_{i=1}^{n-1} 0 \\
&= n + 2\,\frac{n(n-1)}{2} + n - 1 \\
&= n^2 + n - 1
\end{aligned}
$$

Análise da ordenação por trocas sucessivas

- Melhor caso: $T(n) = n^2 + n - 1 = O(n^2)$
- Devemos encontrar duas constantes positivas c e n_0 tais que

$$n^2 + n - 1 \leq cn^2 \,, \quad \text{para todo } n \geq n_0$$

- Então, escolhendo $c = 2$ temos:

$$n^2 + n - 1 \leq 2n^2$$
$$n - 1 \leq n^2$$

ou seja,

$$n^2 - n + 1 \geq 0 \,.$$

- A inequação $n^2 - n + 1 \geq 0$ é sempre válida para todo $n \geq 1$

Análise da ordenação por trocas sucessivas

- Assim, escolhendo $c = 2$ e $n_0 = 1$, temos que

$$n^2 + n - 1 \leq cn^2$$

  para todo $n \geq n_0$, onde $c = 2$ e $n_0 = 1$
- Portanto, $T(n) = O(n^2)$

- Pior caso: quando a sequência de entrada com n números inteiros é fornecida em ordem decrescente ($t_i = i$ para todo i)

$$
\begin{aligned}
T(n) &= n + \sum_{i=1}^{n-1}(i+1) + \sum_{i=1}^{n-1} i + 3\sum_{i=1}^{n-1} t_i \\
&= n + \sum_{i=1}^{n-1}(i+1) + \sum_{i=1}^{n-1} i + 3\sum_{i=1}^{n-1} i \\
&= n + 5\sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} 1 \\
&= n + 5\,\frac{n(n-1)}{2} + n - 1 \\
&= \frac{5}{2}\,n^2 - \frac{5}{2}\,n + 2n - 1 = \frac{5}{2}\,n^2 - \frac{1}{2}\,n - 1
\end{aligned}
$$

- Pior caso: $T(n) = (5/2)n^2 - (1/2)n - 1 = O(n^2)$
- Devemos encontrar duas constantes positivas c e n_0 tais que

$$\frac{5}{2}\,n^2 - \frac{1}{2}\,n - 1 \leq cn^2\,, \quad \text{para todo } n \geq n_0$$

- Então, escolhendo $c = 5/2$ temos:

$$\begin{aligned} \frac{5}{2}\,n^2 - \frac{1}{2}\,n - 1 &\leq \frac{5}{2}\,n^2 \\ -\frac{1}{2}\,n - 1 &\leq 0 \end{aligned}$$

ou seja,

$$\frac{1}{2}\,n + 1 \geq 0$$

Análise da ordenação por trocas sucessivas

- A inequação $(1/2)n + 1 \geq 0$ é sempre válida para todo $n \geq 1$
- Assim, escolhendo $c = 5/2$ e $n_0 = 1$, temos que

$$\frac{5}{2}\,n^2 - \frac{1}{2}\,n - 1 \leq cn^2$$

  para todo $n \geq n_0$, onde $c = 5/2$ e $n_0 = 1$
- Portanto, $T(n) = O(n^2)$

Moral da história

- Algoritmos diferentes para resolver um mesmo problema em geral diferem dramaticamente em eficiência
- Essas diferenças podem ser muito mais significativas que a diferença de tempos de execução em um supercomputador e em um computador pessoal

Moral da história

Problema da ordenação

- Algoritmo A
  - Tempo de execução de pior caso $O(n^2)$
  - Supercomputador: 100 milhões de operações por segundo
  - Programador: hacker, codificação com tempo de execução $2n^2$
- Algoritmo B
  - Tempo de execução de pior caso $O(n \log_2 n)$
  - Computador pessoal: 1 milhão de operações por segundo
  - Programador: mediano, codificação com tempo de exec. $50n \log_2 n$
- Ordenar 1 milhão de números ($n = 10^6$)

Moral da história

Problema da ordenação

- Algoritmo A

$$\frac{2 \cdot (10^6)^2 \text{ operações}}{10^8 \text{ operações/segundo}} = 20.000 \text{ segundos} \approx 5{,}56 \text{ horas}$$

- Algoritmo B

$$\frac{50 \cdot 10^6 \log_2 10^6 \text{ operações}}{10^6 \text{ operações/segundo}} \approx 1.000 \text{ segundos} \approx 16{,}67 \text{ minutos}$$

Moral da história

- "*Um bom algoritmo, mesmo rodando em uma máquina lenta, sempre acaba derrotando (para instâncias grandes do problema) um algoritmo pior rodando em uma máquina rápida. Sempre.*"
  \- **S. S. Skiena**, The Algorithm Design Manual

Exercícios

1. É verdade que $2^{n+1} = O(2^n)$? E é verdade que $2^{2n} = O(2^n)$?

2. Rearranje a seguinte lista de funções em ordem crescente de taxa de crescimento. Isto é, se a função $g(n)$ sucede imediatamente a função $f(n)$ na sua lista, então é verdade que $f(n) = O(g(n))$.

$$
\begin{aligned}
f_1(n) &= n^2 \log_2 n \\
f_2(n) &= n + 10 \\
f_3(n) &= \sqrt{2n} \\
f_4(n) &= 10^n \\
f_5(n) &= 100^n
\end{aligned}
$$