Análise de algoritmos

1 Introdução

Algoritmos são procedimentos desenvolvidos para resolver uma determinada tarefa. Esta é a ideia principal que está por trás de qualquer programa de computador.

Para que seja útil, um algoritmo deve resolver um problema bem especificado, para todas as entradas possíveis. Um problema algoritmico é especificado pela descrição do conjunto completo de exemplos (instâncias) em que ele precisa funcionar e pelas propriedades que a sua saída deve apresentar como resultado de sua aplicação em cada uma destas instâncias.

Isto quer dizer que um algoritmo que funciona para *quase todos* os casos não é a solução para um problema algoritmico. A solução deve funcionar para *todos* os casos previstos.

Exemplo 1. *O problema algoritmico conhecido como ordenação é definido da seguinte maneira:*

Entrada: *uma sequência de n chaves* $a_1, \ldots, a_n$

Saída: *a permutação (reordenação) da sequência de entrada de modo que* $a_1^r \leq a_2^r \leq \cdots \leq a_n^r$.

Uma ***instância*** *de ordenação poderia ser um vetor de nomes, tal como {Angelina, Davi, Robélia, Flávia, Leandro}, ou uma lista de números como {154, 245, 568, 324, 654, 324}.*

Determinar se você tem de fato um problema geral ou uma instância de um problema é o primeiro passo na resolução deste. No nosso caso, temos:

- ***Problema algoritmico:*** *ordenação;*
- ***Instâncias:***
  - *{Angelina, Davi, Robélia, Flávia, Leandro}*
  - *{154, 245, 568, 324, 654, 324}*

Um algoritmo é um procedimento que toma todas as instâncias de entrada possíveis e transforma-as na saída desejada. Existem vários algoritmos de ordenação diferentes para resolver o problema de ordenação. Um deles é o de inserção direta, que você já deve ter estudado, e é apresentado abaixo na forma de pseudocódigo:

```
InsertionSort(A)
{
  for i = 1 to n-1 do
    for j = i+1 to n do
      if (A[j] < A[j-1]) then
        swap(A[j],A[j-1])
}
```

Note a generalidade deste algoritmo: ele funciona igualmente bem tanto para nomes como para números, dadas uma operação de comparação $<$ *e uma função* `swap()` *adequadas. Dada nossa definição do problema de ordenação, pode-se verificar facilmente que este algoritmo ordena corretamente toda instância possível de entrada.*

Neste capítulo, vamos introduzir as propriedades que os bons algoritmos devem ter, assim como formas de medir se um dado algoritmo alcançou estes objetivos.

A avaliação de desempenho de algoritmos requer uma quantidade modesta de notação matemática, que iremos também apresentar. Embora seja um pouco assustadora no início, esta notação é essencial para que possamos comparar algoritmos e projetar outros mais eficientes.

Embora as pessoas mais "práticas" possam pensar que esta análise teórica seja uma perda de tempo, vamos apresentar este material porque, ao final das contas, é útil. Em particular, este capítulo oferece as seguintes lições:

- Algoritmos que parecem estar certos podem facilmente estar incorretos. A correção de um algoritmo é uma propriedade que precisa ser cuidadosamente demonstrada.
- Os algoritmos devem ser entendidos e estudados de forma independente das máquinas.
- A notação O (*big Oh*) e a análise de pior caso são ferramentas que simplificam grandemente nossa habilidade de comparar a eficiência dos algoritmos.
- Procuramos por algoritmos cujos tempos de execução cresçam de forma logaritmica pois $\log_b(n)$ cresce lentamente com o aumento de n.
- O modelamento de nossa aplicação em termos de estruturas e algoritmos bem definidos é o passo mais importante para uma solução.

2 Estrutura da análise de algoritmos

O que é avaliar um algoritmo? Basicamente, queremos verificar os seguintes pontos:

- O algoritmo está correto ou seja, fornece uma solução para o problema algoritmico para o qual foi desenvolvido?
- Quanta memória é necessária para este algoritmo (eficiência espacial)?
- Quão eficiente é o algoritmo em termos de tempo de processamento (eficiência temporal)?

Antigamente, ambos os recursos - tempo e espaço - eram valiosos. Com o desenvolvimento da tecnologia, a quantidade de memória disponível vem aumentando consideravelmente. Entretanto, a velocidade de processamento ainda é uma questão importante. Embora a lei de Moore ainda esteja valendo, ou seja, a velocidade dos processadores dobre a cada 18 meses, os problemas estão se tornando cada vez mais complexos: tarefas antes inimagináveis passam a ser viáveis, de forma que novos problemas surgem a cada dia. Ainda, é possível obter ganhos extraordinários em termos de desempenho usando algoritmos mais eficientes, da ordem de milhares de vezes em alguns casos, o que é muito mais barato que comprar um computador que funcione milhares de vezes mais rápido.

Com base nesta discussão, vamos nos ater apenas à avaliação da eficiência temporal dos algoritmos neste texto.

3 Métodos para análise de algoritmos

Existem duas abordagens para fazer a análise de algoritmos: a experimental e a teórica.

Na abordagem experimental, implementamos o algoritmo em alguma linguagem e executamos o programa resultante para um conjunto de dados de teste, analisando o seu comportamento com o auxílio de programas especiais, que em inglês são chamados de *profilers*. Estes programas medem o tempo de execução de um programa, e mostram quanto tempo foi gasto

em cada uma das suas partes, permitindo uma análise de qual trecho é o mais demorado, de forma a focarmos nossos esforços de otimização nestes pontos.

Entretanto, esta abordagem tem alguns problemas:

- os resultados podem depender de aspectos da implementação;
- não podemos testar todas as entradas possíveis;
- podemos esquecer algum caso em que o algoritmo falha;
- podemos esquecer algum caso em que o desempenho do algoritmo é particularmente bom (ou ruim).

Na abordagem teórica, tentamos prever o comportamento do algoritmo (tempo de execução, requerimentos de memória, etc.) sem implementá-lo em uma plataforma específica. Desta forma, embora alguns não gostem muito do formalismo matemático, optamos por fazer a análise dos algoritmos de forma teórica.

As duas ferramentas mais importantes usadas para este fim são (1) o modelo RAM de computação e (2) a análise assintótica da complexidade, que serão apresentadas nas seções a seguir.

4 O modelo RAM de computação

O modelo RAM (*Random Access Machine*) é um computador hipotético, onde são feitas as seguintes suposições:

- cada operação simples demora exatamente um passo de tempo. As operações simples são:
  - comandos de atribuição (`=`)
  - operações aritméticas (`+`, `-`, `*`, `/`, `%`)
  - operações relacionais (`==`, `!=`, `>`, `<`, `>=`, `<=`)
  - operações lógicas (`&&`, `!`, ||)
  - operações de leitura e escrita (`cin`, `cout`)
- estruturas de repetição, comandos condicionais e subrotinas não são consideradas operações simples; são a composição de várias operações simples. Por exemplo, a rotina `sort` para ordenação de dados não é vista como uma operação simples, pois ordenar 1000000 de itens demora muito mais do que ordenar 10 itens. O tempo de um loop ou subrotina depende do número de iterações do loop ou da natureza expecífica da subrotina.
- Cada acesso de memória leva exatamente um passo de tempo, e temos tanta memória quanto necessitarmos. O modelo RAM não leva em consideração se o ítem armazenado está em memória cache ou no disco, o que simplifica enormemente a análise.

Sob o modelo RAM, medimos o tempo de execução de um algoritmo pela contagem do número de passos que ele dá para uma dada instância do problema. Assumindo que nosso modelo RAM executa um certo número de passos por segundo, podemos facilmente converter esta contagem no tempo real de execução.

O RAM é um modelo simples de como os computadores funcionam. Uma queixa comum é que ele é tão simples que faz com que as conclusões tiradas a partir dele sejam muito grosseiras para serem usadas na prática. Apesar destes defeitos, o modelo RAM é excelente para entender o desempenho que terá um algoritmo em um computador real, pois captura o comportamento essencial dos computadores, ao mesmo tempo em que é fácil de trabalhar.

Na prática, é muito difícil escrever um algoritmo para o qual o modelo RAM forneça resultados substancialmente diferentes do real, ou seja, dê resultados muito melhores ou muito piores do que o previsto. A robustez do RAM nos permite então analisar os algoritmos independentemente da máquina em que ele vai ser implementado. Em outras palavras, usamos o RAM porque ele é útil.

Exemplo 2. *Vamos analisar um caso real com esta técnica. Seja o algoritmo de busca sequencial, mostrado abaixo:*

```
int SequentialSearch(int vector[],int length, int key)
{
  int pos = -1;
  bool found = false;
  int i = 0;

  while (i<length && found==false)
    if (vector[i] == key)
    {
      found = true;
      pos = i;
    }
    else
      i++;

  return pos;
}
```

Vamos supor que o vetor no qual estamos fazendo a busca é vector $= \{1, 2, 3\}$. *Vamos calcular o número de operações necessárias para encontrar as seguintes chaves: 1, 2, 3 e 4.*

a) Chave: key = 1

Comando	número de instruções	total acumulado
`int pos=-1;`	1	1
`bool achou=false;`	1	2
`int i=0;`	1	3
`while (i<length && found==false)`	6	9
`if (vector[i] == key)`	4	13
`found = true;`	1	14
`pos = i;`	2	16

b) Chave: key = 2

Comando	número de instruções	total acumulado
`int pos=-1;`	1	1
`bool achou=false;`	1	2
`int i=0;`	1	3
`while (i<length && found==false)`	6	9
`if (vector[i] == key)`	4	13
`i++;`	3	16
`while (i<length && found==false)`	6	22
`if (vector[i] == key)`	4	26
`found = true;`	1	27
`pos = i;`	1	28

c) Chave: key = 4

Comando	número de instruções	total acumulado
`int pos=-1;`	1	1
`bool achou=false;`	1	2
`int i=0;`	1	3
`while (i<length && found==false)`	6	9
`if (vector[i] == key)`	4	13
`i++;`	3	16
`while (i<length && found==false)`	6	22
`if (vector[i] == key)`	4	26
`i++;`	3	29
`while (i<length && found==false)`	6	35
`if (vector[i] == key)`	4	39
`i++;`	3	42
`while (i<length && found==false)`	6	48

Como você pode ver do exemplo acima, o tempo gasto para efetuar uma busca depende das instâncias que são apresentadas ao algoritmo. É necessário então um passo a mais para que esta técnica realmente se torne útil na avaliação do desempenho dos algoritmos, que veremos na seção a seguir.

4.1 Complexidade do melhor caso, pior caso e caso médio

Vimos que usando o modelo RAM de computação, podemos contar quantos passos nosso algoritmo irá executar para qualquer instância de entrada. Entretanto, para verificar realmente o quão bom ou ruim é um algoritmo, precisamos saber como ele funciona para todas as instâncias.

Para entender melhor as noções de complexidade de melhor, pior e caso médio, pode-se pensar em rodar um algoritmo em todas as possíveis instâncias de entrada possíveis. Para o problema de ordenação, o conjunto de entradas possíveis consiste de todos os possíveis arranjos de todos os números possíveis de chaves. Podemos representar cada instância de entrada como um ponto em um gráfico, onde o eixo x é o tamanho do problema (para o problema de ordenação, o número de itens a serem ordenados) e o eixo y é o número de passos realizados pelo algoritmo nesta instância. Assumimos aqui que não importa quais são os

valores das chaves, apenas quantas são e como estão ordenadas. A ordenação de 1000 nomes em inglês não deve demorar mais que a ordenação de 10000 nomes em fracês, por exemplo.

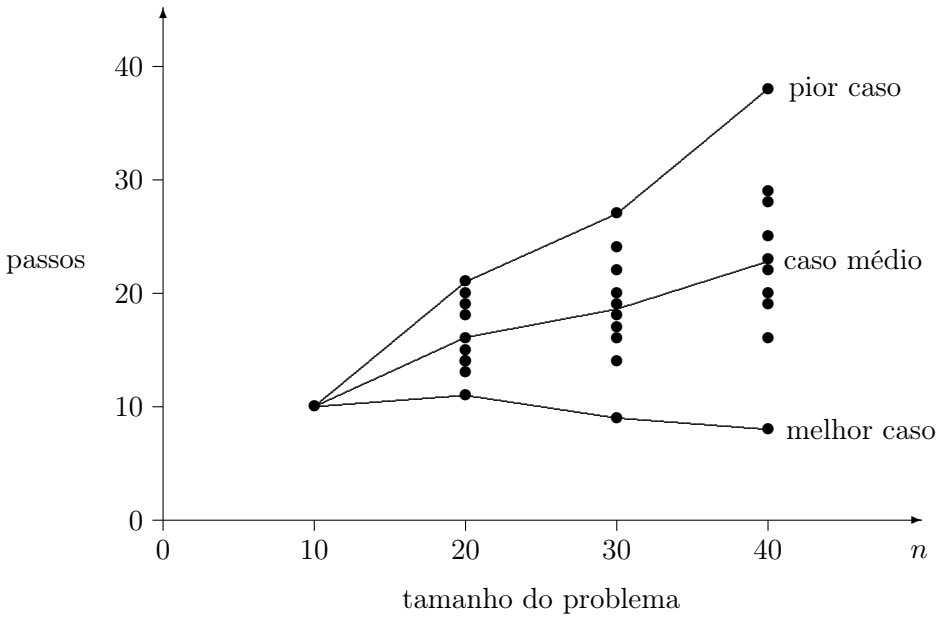

Figura 1: Complexidades de melhor caso, caso médio e pior caso.

Como mostrado na Figura 1, estes pontos se alinham naturalmente em colunas, pois os tamanhos das entradas são inteiros. Uma vez que temos estes pontos, podemos definir três funções sobre eles:

- A **complexidade de pior caso** do algoritmo é a função definida pelo número máximo de passos tomados em qualquer instância de tamanho n. Ela representa a curva que passa pelos pontos mais altos de cada coluna.
- A **complexidade de melhor caso** do algoritmo é a função definida pelo número mínimo de passos tomados em qualquer instância de tamanho n. Ela representa a curva que passa pelos pontos mais baixos de cada coluna.
- Finalmente, a **complexidade de caso médio** do algoritmo que é a função definida pelo número médio de passos tomados em qualquer instância de tamanho n.

Na prática, a medida mais útil destas três é a complexidade de pior caso, o que muitas pessoas podem achar estranho, pois intuitivamente podemos pensar que a melhor medida seria a complexidade média. Entretanto, se refletirmos um pouco mais sobre o assunto, podemos ver que a complexidade média pode dar margens a indefinições, como por exemplo: o que significa exatamente a média? Em alguns casos, a média pode estar bem abaixo do pior caso, e assim não iremos ter uma idéia realista de quanto realmente nosso algoritmo vai demorar, se topar com o pior caso. Ainda, quantas vezes iremos topar com casos melhores ou piores do que a média?

Lembre-se de que estamos querendo verificar o desempenho de um algoritmo para todos os casos e não para alguns apenas. Usando a medida de pior caso, evitamos todas estas indefinições.

5 Análise assintótica da complexidade

O segundo método para avaliar o desempenho dos algoritmos quanto ao tempo de processamento é a análise assintótica da complexidade. Vamos inicialmente aprender como determinar

a complexidade de um algoritmo. Depois veremos como usar esta complexidade para comparar diferentes algoritmos através da notação O.

5.1 Metodologia para determinação da complexidade de algoritmos

Como vimos, a eficiência de um algoritmo pode ser medida em termos do tamanho da instância de entrada. Vamos verificar na prática como determinar esta função para os casos mais comuns. A idéia básica aqui é mostrar alguns exemplos de códigos simples e verificar a sua complexidade a partir da análise de seu funcionamento.

5.1.1 Complexidade linear

Vamos analisar o seguinte trecho de código:

```
for (i=0;i<N;i++)
{
  {conjunto de instruções;}
}
```

Neste caso, podemos ver que o `conjunto de instruções` será executado N vezes. Se N dobra, este trecho é executado $2N$ vezes, se N cai pela metade, este trecho é executado $N/2$ vezes, e assim por diante. Desta forma, podemos concluir que a dependência do número de repetições com N é linear, e então escrevemos:

$$f(n) = n$$

Seja a agora o trecho de código abaixo:

```
i=1;
while (i<N)
{
  {conjunto de instruções;}
  i = i+2;
}
```

Neste caso, o `conjunto de instruções` será executado $N/2$ vezes, e podemos então escrever:

$$f(n) = \frac{n}{2}$$

5.1.2 Complexidade logaritmica

Vamos estudar agora o caso em que a variável de controle é multiplicada ou dividida por um fator constante. Sejam os códigos abaixo:

```
i=1;
while (i<N)
{
  {conjunto de instruções;}
  i=i*2;
}
```

```
i=N;
while (i>=1)
{
  {conjunto de instruções;}
  i=i/2;
}
```

Neste caso fica um pouco mais difícil determinar uma expressão analítica de quantas vezes o `conjunto de instruções` irá ser executado. Vamos fazer uma tabela para poder entender melhor o que está acontecendo:

Tabela 1: Número de repetições versus tamanho da entrada.

Multiplicação			Divisão	
repetições	n	$\log_2(n)$	repetições	n
0	1	1	1	1000
1	2	2	2	500
2	4	3	3	250
3	8	4	4	125
4	16	5	5	62
5	32	6	6	31
6	64	7	7	15
7	128	8	8	7
8	256	9	9	3
9	512	10	10	1

Observando a tabela acima, podemos ver que o número de repetições cresce de acordo com $\log_2(n)$ (compare a primeira e a terceira colunas da Tabela 1). Então, de forma genérica, podemos escrever:

$$f(n) = \log_2(n)$$

5.1.3 Complexidade de estruturas de repetição aninhadas

Se o trecho a ser analisado contiver estruturas de repetição aninhadas, é necessário analisar o número de vezes que cada uma se repete. O total de repetições pode ser determinado multiplicando-se o número de iterações da estrutura externa pelo número de repetições da estrutura interna:

$$n_t = n_e \times n_i \tag{1}$$

A seguir são apresentadas as estruturas aninhadas mais encontradas na prática.

5.1.4 Logaritmo linear

```
n=1;
while (n<=10)
{
  k=1;
  while (k<=10)
  {
    {conjunto de instruções;}
    k=k*2;
  }
  n=n+1;
}
```

Das análises anteriores, identificamos que a complexidade da estrutura interna é $f_1(n) = \log_2(n)$, e a da estrutura externa é $f_2(n) = n$. Portanto, usando a Equação 1, temos que a complexidade da estrutura aninhada é:

$$f(n) = n \log_2(n)$$

5.1.5 Dependência quadrática

```
n=1;
while (n<=10)
{
  k=1;
  while (k<=n)
  {
    {conjunto de instruções;}
    k=k+1;
  }
  n=n+1;
}
```

Neste caso, a estrutura externa é idêntica ao do caso anterior, mas a estrutura interna depende de um parâmetro da estrutura externa (n). A estrutura interna é executada uma vez na primeira repetição, duas vezes na segunda, e assim por diante. Assim, o número total de vezes que a estrutura interna vai ser executada pode ser calculado como:

$$1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 = 55$$

O número médio de vezes que a estrutura interna será executada é $55/10 = 5,5$. Observe que podemos chegar ao mesmo resultado fazendo o número de iterações mais um, dividido por 2: $(10 + 1)/2 = 5,5$. Para o caso geral, temos que o número médio de iterações da estrutura interna é: $(n + 1)/2$.

Usando agora a Equação 1 para determinar a complexidade da estrutura aninhada, temos:

$$f(n) = \frac{n(n+1)}{2}$$

5.1.6 Quadrática

```
n=1;
while (n<=10)
{
  k=1;
  while (k<=10)
  {
    {conjunto de instruções;}
    k=k+1;
  }
  n=n+1;
}
```

A estrutura externa é executada 10 vezes. Para cada iteração da estrutura externa, a estrutura interna é executada também 10 vezes. Desta forma, o número total de repetições é $10 \times 10 = 100$. A fórmula genérica neste caso é simples:

$$f(n) = n^2$$

5.2 As notações O, Ω e Θ.

Vimos na Seção 4.1 que as complexidades de pior caso, melhor caso e caso médio de um dado algoritmo são funções numéricas do tamanho das instâncias. Entretanto, é difícil trabalhar

com estas funções exatas porque elas são em geral bastante complicadas, com muitas subidas e descidas pequenas, como mostrado na Figura 1. Assim, é realmente mais fácil falar sobre limitantes superiores e inferiores destas funções, como mostrado na Figura 2. É aqui que entra a notação 'O' (lê-se big Oh).

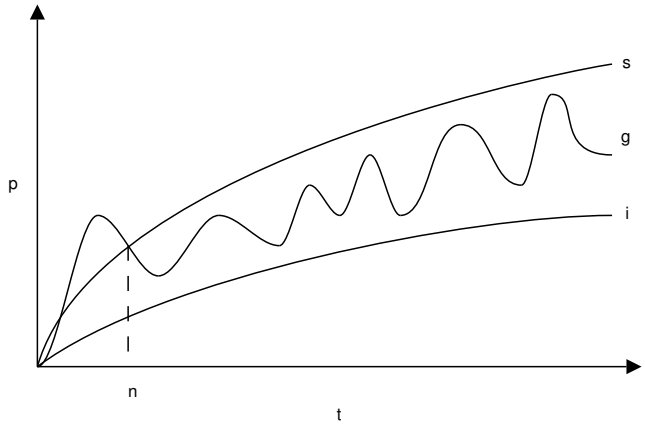

Figura 2: Limitantes superiores e inferiores suavizam o comportamento de funções complexas.

Usamos esta suavização para ignorar níveis de detalhe que não têm impacto em nossa comparação de algoritmos. Vamos usar a seguinte notação:

- $f(n) = O(g(n))$ significa que existem constantes c e n_0 de modo que $cg(n) \geq f(n)$ para $n > n_0$.
- $f(n) = \Omega(g(n))$ significa que existem constantes c e n_0 de modo que $cg(n) \leq f(n)$ para $n > n_0$.
- $f(n) = \Theta(g(n))$ significa que existem constantes c_1, c_2 e n_0 de modo que $c_1 g(n) \leq f(n)$ e $c_2 g(n) \geq f(n)$ para $n > n_0$.

Em outras palavras, $O(g(n))$ é um limitante superior para $f(n)$, $\Omega(g(n))$ é um limitante inferior para $f(n)$, e $\Theta(g(n))$ é um limitante tanto inferior como superior para $f(n)$. Veja a Figura abaixo para entender melhor.

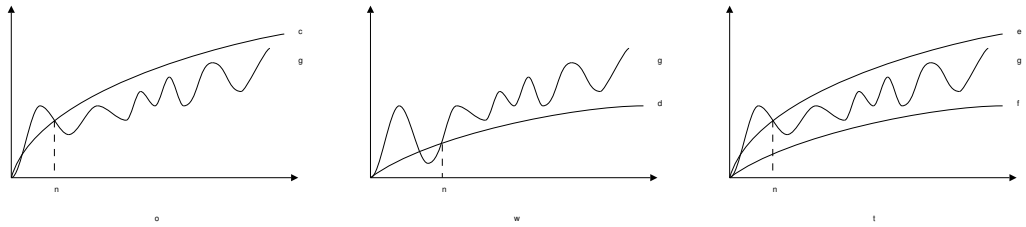

Figura 3: Ilustração das notações O, Ω e Θ.

Observe como todas estas definições implicam em uma constante n_0 acima da qual elas são sempre satisfeitas. Não estamos preocupados com valores pequenos de n, isto é, qualquer coisa abaixo de n_0. De qualquer forma, não importa se um dado algoritmo de ordenação é mais eficiente que outro para ordenar 6 elementos, por exemplo. Para este tamanho, todos operam rápido. O problema é quando temos que ordenar um milhão, dezenas de centenas de milhões de dados. Aí é que o bicho pega. Desta forma, a notação 'O' é útil pois nos permite ignorar detalhes sem importância e focar nos aspectos gerais.

Exemplo 3. *Abaixo tem-se alguns exemplos. Certifique-se de ter entendido todos eles, pois serão importantes para as análises que faremos a seguir. Escolhemos algumas constantes na explicação pois elas funcionam, mas você é livre para escolher qualquer constante que mantenha a mesma desigualdade.*

$$\begin{aligned}
3n^2 - 100n + 6 &= O(n^2) \text{ pois } 3n^2 > 3n^2 - 100n + 6 \\
3n^2 - 100n + 6 &= O(n^3) \text{ pois } .00001n^3 > 3n^2 - 100n + 6 \\
3n^2 - 100n + 6 &\neq O(n) \text{ pois } cn < 3n^2 \text{ quando } n > c
\end{aligned}$$

$$\begin{aligned}
3n^2 - 100n + 6 &= \Omega(n^2) \text{ pois } 2.99n^2 < 3n^2 - 100n + 6 \\
3n^2 - 100n + 6 &\neq \Omega(n^3) \text{ pois } n^3 > 3n^2 - 100n + 6 \\
3n^2 - 100n + 6 &= \Omega(n) \text{ pois } 10^{10^{10}} n < 3n^2 - 100n + 6
\end{aligned}$$

$$\begin{aligned}
3n^2 - 100n + 6 &= \Theta(n^2) \text{ pois ambos } O \text{ e } \Omega \\
3n^2 - 100n + 6 &\neq \Theta(n^3) \text{ pois somente } O \\
3n^2 - 100n + 6 &\neq \Theta(n) \text{ pois somente } \Omega
\end{aligned}$$

5.2.1 Procedimento para calcular a notação 'O'

Vimos no exemplo acima que para uma dada função existem várias funções que são limitantes superiores. Entretanto, estamos interessados no limitante mais "apertado", ou seja, a menor função que seja maior que $f(n)$. Para encontrar tal função, podemos seguir os seguintes passos:

1. para cada um dos termos da função, deixar seu coeficiente igual a 1.
2. Mantenha o maior termo da função e descarte os termos restantes. Os termos são classificados da importância mais baixa para a mais alta, como é mostrado a seguir:

   $\log(n)$, n, $n\log(n)$, n^2, n^3, $\ldots$, n^k, 2^n, $n!$

Exemplo 4. *Calcule a notação 'O' da seguinte função:*

$$f(n) = \frac{n(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n$$

Solução. *Primeiramente, fazemos todos os coeficientes iguais a 1:*

$n^2 + n$

Em seguida, removemos os fatores de menor importância, deixando apenas o maior:

n^2

Finalmente, a função original, na notação 'O' fica:

$O(f(n)) = O(n^2)$

Exemplo 5. *Calcule a notação 'O' da seguinte expressão:*

$$f(n) = \sum_{j=0}^{k} a_j n^j$$

Solução. *Primeiro, vamos expandir o somatório para esta expressão ficar num formato mais legível:*

$$f(n) = \sum_{j=0}^{k} a_j n^j = a_0 + a_1 n + a_2 n^2 + \cdots + a_{k-1} n^{k-1} + a_k n^k$$

Fazendo todos os coeficientes $a_j = 1$, temos:

$$f(n) = 1 + n + n^2 + \cdots + n^{k-1} + n^k$$

Desconsiderando os termos de mais baixa ordem, finalmente chegamos a:

$$O(f(n)) = O(n^k)$$

Exemplo 6. *Vamos agora a um exemplo mais completo: determine a notação 'O' do seguinte trecho de código:*

```
i=1;
while (i<N)
{
  {conjunto de instruções;}
  i = i+2;
}
n=1;
while (n<=N)
{
  k=1;
  while (k<=N)
  {
    {conjunto de instruções;}
    k=k*2;
  }
  n=n+1;
}
```

Solução. *Podemos ver que este trecho pode ser dividido em duas partes:*

- *a primeira é uma repetição simples, onde o contador é incrementado de 2 a cada passo. Desta forma, sua complexidade é $f_1(n) = n/2$.*
- *a segunda parte contém duas estruturas de repetição aninhadas, que vimos ter complexidade $f_2(n) = n \log_2(n)$.*

Como o tempo total de execução deste trecho é igual aos tempos de cada uma destas partes, podemos escrever

$$f_T(n) = f_1(n) + f_2(n) = \frac{n}{2} + n \log_2(n) = \frac{1}{2} n + n \log_2(n)$$

Finalmente, para obter a notação 'O' deste trecho de código, fazemos todos os coeficientes iguais a 1 e descartamos os termos de mais baixa ordem. Desta forma, ficamos com:

$$O(f_T(n)) = O(n \log_2(n))$$

5.3 Crescimento de Funções e Notação assintótica

Quando trabalhamos com a notação 'O', descartamos as constantes multiplicativas. As funções $f(n) = 0,001n^2$ e $g(n) = 1000n^2$ são tratadas de forma idêntica, mesmo que $g(n)$ seja um milhão de vezes maior do que $f(n)$ para todos os valores de n. O método por trás desta aparente loucura é ilustrado pela Tabela 2, que tabula a taxa de crescimento de várias funções que aparecem na análise de algoritmos, em instâncias de problemas de tamanho razoável. Na Figura 4 têm-se estas informações em forma de gráficos.

Tabela 2: Taxas de crescimento de algumas funções.

n	$\log_2(n)$	n	$n \log_2(n)$	n^2	n^3	2^n	$n!$
10	3,3	10	33	10^2	$1 \cdot 10^3$	10^3	$3 \cdot 10^6$
20	4,3	20	86	$4 \cdot 10^2$	$8 \cdot 10^3$	10^6	$2 \cdot 10^{18}$
30	4,9	30	147	$9 \cdot 10^2$	$3 \cdot 10^4$	10^9	$2 \cdot 10^{18}$
40	5,3	40	212	$1 \cdot 10^3$	$6 \cdot 10^4$	10^{12}	$3 \cdot 10^{32}$
50	5,6	50	282	$2 \cdot 10^3$	$1 \cdot 10^5$	10^{15}	$8 \cdot 10^{47}$
60	5,9	60	354	$4 \cdot 10^3$	$2 \cdot 10^5$	10^{18}	$3 \cdot 10^{64}$
70	6,1	70	429	$5 \cdot 10^3$	$3 \cdot 10^5$	10^{21}	$8 \cdot 10^{81}$
80	6,3	80	505	$6 \cdot 10^3$	$5 \cdot 10^5$	10^{24}	$1 \cdot 10^{100}$
90	6,5	90	584	$8 \cdot 10^3$	$7 \cdot 10^5$	10^{27}	$7 \cdot 10^{118}$
10^2	6,6	10^2	664	10^4	10^6	10^{30}	$1 \cdot 10^{138}$
10^3	9,9	10^3	996	10^6	10^9	10^{301}	$9 \cdot 10^{157}$
10^4	13,2	10^4	10^5	10^8	10^{12}		
10^5	16,6	10^5	$2 \cdot 10^6$	10^{10}	10^{15}		
10^6	19,9	10^6	$2 \cdot 10^7$	10^{12}	10^{18}		
10^7	23,2	10^7	$2 \cdot 10^8$	10^{14}	10^{21}		
10^8	26,5	10^8	$3 \cdot 10^9$	10^{16}	10^{24}		
10^9	29,9	10^9	$3 \cdot 10^{10}$	10^{18}	10^{27}		
10^{10}	33,2	10^{10}	$3 \cdot 10^{11}$	10^{20}	10^{30}		

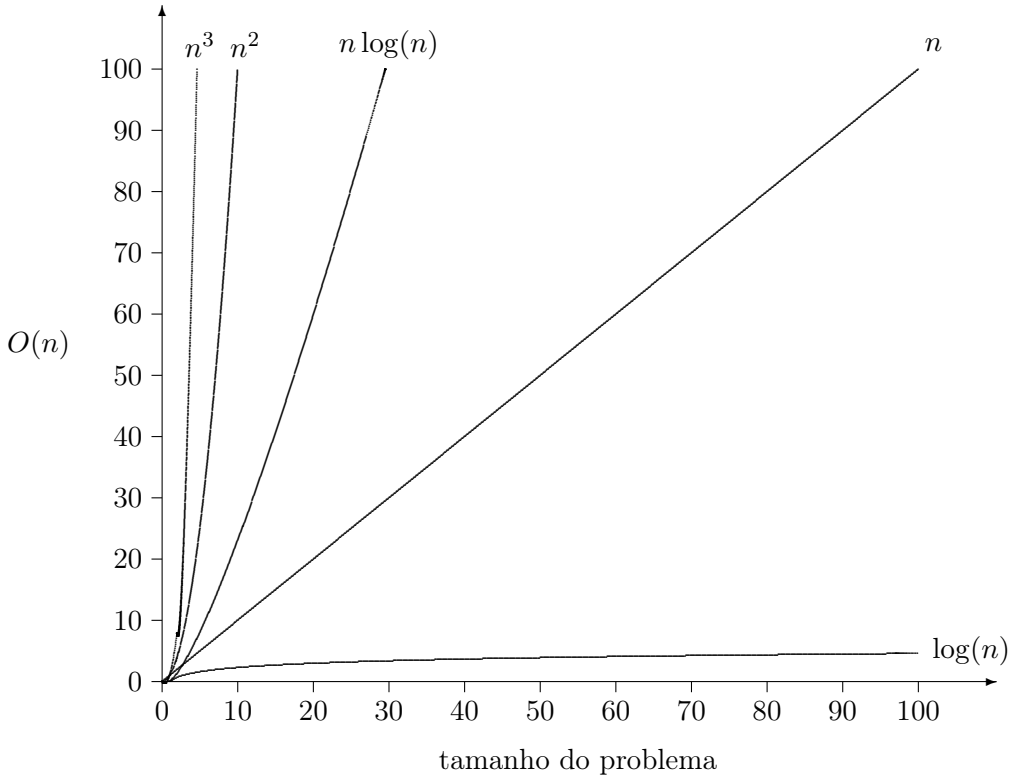

Figura 4: Taxas de crescimento para várias funções.

Uma outra forma de ver estes dados que talvez seja mais intuitiva é convertê-los para unidades de tempo. Vamos supor que cada repetição leva 1 nanossegundo (10^{-9}s)para ser executada. Convertendo então dos dados da Tabela 2 para segundos, geramos os dados da Tabela 3.

Tabela 3: Dados da Tabela 2 convertidos para segundos.

n	$\log_2(n)$	n	$n\log_2(n)$	n^2	n^3	2^n	$n!$
10	3ns	10ns	33ns	100ns	1μs	1μs	3ms
20	4,3ns	20ns	86ns	400ns	8μs	1ms	77 anos
30	4,9ns	30ns	147ns	900ns	27μs	1s	
40	5,3ns	40ns	212ns	1μs	64μs	18 min	
50	5,6ns	50ns	282ns	2μs	125μs	13 dias	
60	5,9ns	60ns	354ns	4μs	216μs	36 anos	
70	6,1ns	70ns	429ns	5μs	343μs		
80	6,3ns	80ns	505ns	6μs	512μs		
90	6,5ns	90ns	584ns	8μs	729μs		
10^2	6,6ns	100ns	664ns	10μs	1ms		
10^3	10ns	1μs	9,96μs	1ms	1s		
10^4	13ns	10μs	132,88μs	100ms	16 min		
10^5	16ns	100μs	1,66ms	10s	11 dias		
10^6	20ns	1ms	19,93ms	16 min	31 anos		
10^7	23ns	10ms	232,53ms	1 dia			
10^8	26ns	100ms	2,66s	115 dias			
10^9	30ns	1s	29,90s	31 anos			
10^{10}	33ns	10s	5min30s				

Ao analisarmos os dados desta Tabela, podemos chegar às seguintes conclusões:

- Todos estes algoritmos são rápidos para $n = 10$.
- O algoritmo cujo tempo de execução é $O(n!)$ torna-se inviável bem antes de $n = 20$.
- O algoritmo cujo tempo de execução é $O(2^n)$ tem uma faixa de operação um pouco maior, mas torna-se impraticável para $n > 40$.
- O algoritmo cujo tempo de execução é $O(n^2)$ é perfeitamente razoável até aproximadamente $n = 10^4$, mas se deteriora rapidamente para entradas maiores. Para $n > 10^6$ parece ser inviável.
- Os algoritmos com tempo de execução $O(n)$ e $O(n \log(n))$ são viáveis para entradas de até um bilhão de itens.
- Para algoritmos com tempos de execução $O(\log_2(n))$ será difícil encontrar um problema prático para o qual este se torne muito lento.

A lição a ser aprendida desta análise é a de que mesmo ignorando os fatores constantes, podemos ter uma excelente idéia se um dado algoritmo poderá executar uma tarefa de determinado tamanho em um espaço de tempo razoável.

De fato, um algoritmo cujo tempo de processamento é $f(n) = n^3$ segundos irá ser mais rápido que um algoritmo cujo tempo de execução é $g(n) = 1000000n^2$ somente para $n < 1000000$. Outro ponto a ser notado que constantes enormes como estas ocorrem muito menos frequentemente do que problemas com n grande.

6 Exercícios

1. Coloque em ordem crescente as seguintes complexidades: 2^n, $n!$, n^5, 10000, $n \log_2(n)$.
2. Faça o mesmo para a seguinte sequência: $n \log_2(n)$, $n + n^2 + n^3$, 2^4, $\sqrt{n}$.
3. Obtenha o valor da notação 'O' das seguintes expressões de complexidade:
   $5n^{5/2} + n^{2/5}$ $\quad$ $6 \log_2(n) + 9n$ $\quad$ $3n^4 + n \log_2(n)$ $\quad$ $5n^2 + n^{3/2}$
4. Suponha que o algoritmo presente na parte `<...>` tenha complexidade $5n$. Calcule então a complexidade de tempo de execução do trecho de algoritmo apresentado a seguir:

```
k=1;
while (k<=n)
{
  <...>
  k=k+1;
}
```

5. Suponha que o algoritmo presente na parte `<...>` tenha complexidade $O(n^2)$. Calcule então a complexidade de tempo de execução do trecho de algoritmo apresentado a seguir:

```
k=1;
while (k<n)
{
  <...>
  k=k*2;
}
```

6. Suponha que a complexidade de um algoritmo seja $5n \log_2(n)$. Um passo neste algoritmo leva algo em torno de 1 nanossegundo (10^{-9}s). Quanto tempo este algoritmo levará para processar uma entrada de tamanho $n = 1000$ elementos?

7. Um programador deseja verificar o desempenho de deteminado algoritmo. Para isto ele fez alguns testes:

   - primeiro aplicou uma entrada de $n = 4096$ elementos e verificou que este algoritmo processava esta entrada em 512ms;
   - depois aplicou uma entrada de $n = 16384$ elementos, e o algoritmo demorou 1024ms.

   Determine a complexidade e a notação 'O' deste algoritmo.

8. Determine a complexidade dos seguintes algoritmos de ordenação:

   (a) inserção direta

   (b) seleção direta

   (c) bubble sort

   (d) shell sort

9. Suponha que uma exigência de tempo seja dada pela fórmula $an^2 + bn \log_2(n)$, onde a e b são constantes. Responda às seguintes perguntas, provando seus resultados matematicamente e escrevendo um programa para validar os resultados empiricamente.

   (a) Para que valores de n (expressos em termos de a e b) o primeiro termo domina o segundo?

   (b) Para que valores de n (expressos em termos de a e b) os dois termos são iguais?

   (c) Para que valores de n (expressos em termos de a e b) o segundo termo domina o primeiro?

10. Quantas comparações são necessárias para achar o maior e o menor elemento de um conjunto de n elementos distintos? Justifique sua resposta.