

Capítulo 2

Análise Assintótica da Complexidade de Algoritmos

Elton J. Silva (eltonsilv@gmail.com)

Departamento de Computação
Universidade Federal de Ouro Preto

“Aconselho os meus alunos a terem muito cuidado quando decidirem não estudar mais Matemática. Nesse momento, eles devem estar preparados para ouvirem o som de portas se fechando.” -- James Caballero

1. Complexidade assintótica: *O*, *Omega* e *Theta*

Ao ver uma expressão como $n+10$ ou n^2+1 , a maioria das pessoas pensa automaticamente em valores pequenos de n , valores próximos de zero. A análise de algoritmos faz exatamente o contrário: *ignora os valores pequenos* e concentra-se nos *valores enormes* de n . Para valores enormes de n , as funções

$$n^2, (3/2)n^2, 9999n^2, n^2/1000, n^2+100n, \text{ etc.}$$

crescem todas com a mesma velocidade e portanto são todas "equivalentes". Esse tipo de análise, interessado somente em valores *enormes* de n , é chamado *assintótica*. Nessa matemática, as funções são classificadas em "ordens"; todas as funções de uma mesma ordem são "equivalentes". As cinco funções acima, por exemplo, pertencem à mesma ordem.

Ordem *O* ('big *O*', 'O grande')

Convém restringir a atenção a funções *assintoticamente não-negativas*, ou seja, funções f tais que $f(n) \geq 0$ para todo n suficientemente grande. Mais explicitamente: f é assintoticamente não-negativa se existe n_0 tal que $f(n) \geq 0$ para todo $n \geq n_0$. Agora podemos definir a ordem *O*.

DEFINIÇÃO: Dadas funções assintoticamente não-negativas f e g , dizemos que f *está na ordem *O* de g* , e escrevemos $f = O(g)$ ou $f \in O(g)$, se $f(n) \leq c \cdot g(n)$ para *algum* c positivo e para *toda* n suficientemente grande. Em outras palavras, existe um número positivo c e um número positivo n_0 tais que $f(n) \leq c \cdot g(n)$ para todo $n \geq n_0$. Neste caso, dizemos que $g(n)$ domina assintoticamente $f(n)$, ou que $g(n)$ é um limite assintótico superior para $f(n)$. A Figura 1 apresenta um exemplo gráfico da notação *O*:

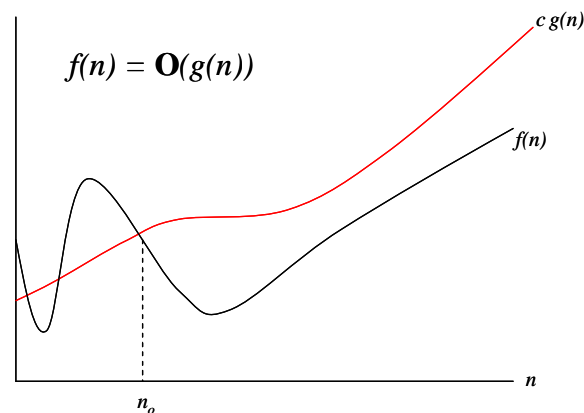


Figura 1 - A notação *O* dá um limite superior para a função $f(n)$. Dizemos que $f(n)$ é $O(g(n))$ se existem constantes positivas c e n_0 , de forma que à direita de n_0 , o valor de $f(n)$ está sempre abaixo ou igual a $c \cdot g(n)$.

EXEMPLO: Suponha que $f(n) = (3/2)n^2 + (7/2)n - 4$ e que $g(n) = n^2$. A tabela abaixo sugere que $f(n) \leq 2g(n)$ para $n \geq 6$ e, portanto, parece que $f(n) = O(g(n))$.

n	f(n)	g(n)
0	-4	0
1	1	1
2	9	4
3	20	9
4	34	16
5	51	25
6	71	36
7	94	49
8	120	64

É fácil verificar que, de fato, $f(n) = O(g(n))$ com um cálculo mais grosseiro: $f(n) \leq 2n^2 + 4n^2 = 6n^2 = 6g(n)$ para todo n .

Uma outra forma de verificar se uma função $g(n)$ domina assintoticamente uma outra função $f(n)$ é a seguinte:

$$\text{Se } 0 \leq \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty, f(n) = O(g(n)).$$

Teorema: Se $A(n) = A_m n^m + \dots + A_1 n + A_0$ é um polinômio de grau m , então $A(n)$ é $O(n^m)$.

Exemplo: A função de complexidade de um algoritmo A é dada por $f(n) = n^3 + 2n^2 + n + 1$. Logo, podemos dizer que este algoritmo tem complexidade $O(n^3)$. Isto significa que para valores muito grandes de n , poderíamos desconsiderar a outra parte do polinômio $(2n^2 + n + 1)$, pois ela não influenciaria tanto no crescimento de $f(n)$.

Ordem o (o pequeno)

DEFINIÇÃO: Dadas funções assintoticamente não-negativas f e g , dizemos que f está na ordem o de g , e escrevemos $f = o(g)$ ou $f \in o(g)$, se $f(n) < c \cdot g(n)$ para *algum* c positivo e para *todo* n suficientemente grande. Em outras palavras, existe um número positivo c e um número positivo n_0 tais que $f(n) < c \cdot g(n)$ para todo $n \geq n_0$.

Intuitivamente, dizemos que "o pequeno" é assintoticamente análogo a "menor que" e "O grande", análogo a "menor ou igual a". Assim, $2n^2$ é $O(n^2)$, mas não é $o(n^2)$. Já $2n$ é $o(n^2)$ e $O(n^2)$.

Algumas propriedades da função O

- (i) $f(n) = O(f(n))$
- (ii) $c \cdot O(f(n)) = O(f(n))$, c constante
- (iii) $O(f(n)) + O(f(n)) = O(f(n))$
- (iv) $O(O(f(n))) = O(f(n))$
- (v) $O(f(n)) + O(g(n)) = O(\text{Max}(f(n), g(n)))$
- (vi) $O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$
- (vii) $O(f(n)) \cdot g(n) = f(n) \cdot O(g(n))$

Ordem Ω (Omega)

A expressão " $f = O(g)$ " tem o significado matemático de " $f \leq g$ ". Agora precisamos de um conceito que tenha o significado de " $f \geq g$ ".

DEFINIÇÃO: Dadas funções assintoticamente não-negativas f e g , dizemos que f está na ordem Omega de g , e escrevemos $f = \Omega(g)$, ou $f \in \Omega(g)$, se $f(n) \geq c \cdot g(n)$ para *algum* c positivo e para *todo* n suficientemente grande. Em outras palavras, existe um número positivo c e um número positivo n_0 tais que $f(n) \geq c \cdot g(n)$ para todo $n \geq n_0$. Neste caso, dizemos que $g(n)$ é um limite assintótico inferior para $f(n)$. Na Figura 2 a seguir, a notação Ω é apresentada graficamente.

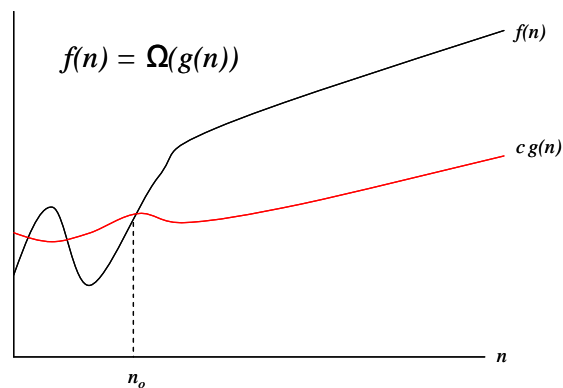


Figura 2 - A notação Ω dá um limite inferior para a função $f(n)$. Dizemos que $f(n)$ é $\Omega(g(n))$ se existem constantes positivas c e n_0 , de forma que à direita de n_0 , o valor de $f(n)$ está sempre acima ou igual a $c \cdot g(n)$.

EXEMPLO: Se $f(n) \geq g(n)/1000000$ para todo $n \geq 888$ então $f = \Omega(g)$. (Mas cuidado: a recíproca não é verdadeira!)

Uma outra forma de verificar se uma função $f(n)$ domina assintoticamente uma outra função $g(n)$ é a seguinte:

$$\text{Se } 0 < \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \leq \infty, f(n) = \Omega(g(n)).$$

Qual a relação entre O e Ω ? Não é difícil verificar que $f = O(g)$ se e somente se $g = \Omega(f)$.

Ordem ω

DEFINIÇÃO: Dadas funções assintoticamente não-negativas f e g , dizemos que f está na ordem ω de g , e escrevemos $f = \omega(g)$ ou $f \in \omega(g)$, se $f(n) > c \cdot g(n)$ para *algum* c positivo e para *todo* n suficientemente grande. Em outras palavras, existe um número positivo c e um número positivo n_0 tais que $f(n) > c \cdot g(n)$ para todo $n \geq n_0$.

Intuitivamente, dizemos que ω é assintoticamente análogo a "maior que" e Ω análogo a "maior ou igual a". Assim, $n^2/2$ é $\Omega(n^2)$, mas não é $\omega(n^2)$. Já $n^2/2$ é $\Omega(n)$ e $\omega(n)$.

Ordem Θ (Theta)

Além dos conceitos de " $f \leq g$ " e de " $f \geq g$ ", precisamos de um que tenha o significado de " $f = g$ ".

DEFINIÇÃO: Dizemos que f e g estão na mesma ordem *Theta* e escrevemos $f = \Theta(g)$ ou $f \in \Theta(g)$ se $f = O(g)$ e $f = \Omega(g)$. Trocando em miúdos, $f = \Theta(g)$ significa que existem números positivos c_1 e c_2 tais que $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ para todo n suficientemente grande ($n \geq n_0$). Na Figura 3 a notação Θ é apresentada graficamente.

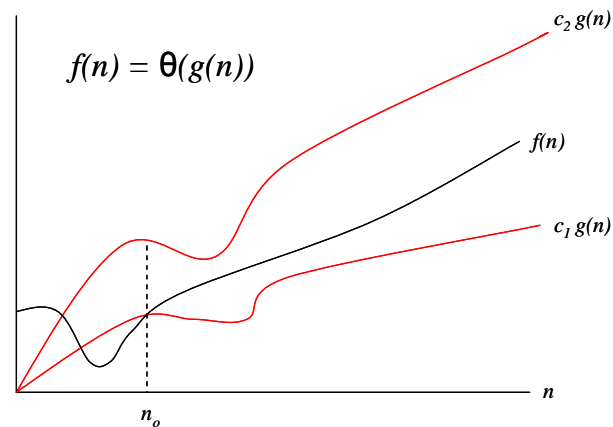


Figura 3 - Dizemos que $f(n)$ é $\Theta(g(n))$ se existem constantes positivas c_1 , c_2 e n_0 , de forma que à direita de n_0 , o valor de $f(n)$ está sempre entre (inclusive) $c_1 \cdot g(n)$ e $c_2 \cdot g(n)$.

EXEMPLO: As funções abaixo pertencem todas à ordem $\Theta(n^2)$:

$$n^2, \quad (3/2)n^2, \quad 9999n^2, \quad n^2/1000, \quad n^2+100n.$$

Uma outra forma de verificar se uma função $f(n) = \Theta(g(n))$ é a seguinte:

$$\text{Se } 0 < \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty, f(n) = \Theta(g(n)).$$

Propriedades de O , Ω e Θ

(i) **Reflexividade** : $f(n) = O(f(n))$, $f(n) = \Omega(f(n))$, $f(n) = \Theta(f(n))$

(ii) **Transitividade**:

Se $f(n) = O(g(n))$ e $g(n) = O(h(n))$, então $f(n) = O(h(n))$

Se $f(n) = \Omega(g(n))$ e $g(n) = \Omega(h(n))$, então $f(n) = \Omega(h(n))$

Se $f(n) = \Theta(g(n))$ e $g(n) = \Theta(h(n))$, então $f(n) = \Theta(h(n))$

(iii) $f(n) = \Theta(g(n))$ se e somente se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$

(iv) **Simetria**: $f(n) = \Theta(g(n))$ se e somente se $g(n) = \Theta(f(n))$

(v) **Simetria Transposta**

$f(n) = O(g(n))$ se e somente se $g(n) = \Omega(f(n))$

$f(n) = \Omega(g(n))$ se e somente se $g(n) = O(f(n))$

Essas propriedades das notações assintóticas sugerem uma analogia entre a comparação assintótica de duas funções f e g e a comparação de dois números reais a e b :

$$f(n) = O(g(n)) \approx a \leq b,$$

$$f(n) = \Omega(g(n)) \approx a \geq b,$$

$$f(n) = \Theta(g(n)) \approx a = b,$$

$$f(n) = o(g(n)) \approx a < b,$$

$$f(n) = \omega(g(n)) \approx a > b.$$

Entretanto, uma propriedade dos números reais não se aplica à notação assintótica. **Tricotomia**: Para quaisquer dois números reais a e b , $a < b$ ou $a = b$ ou $a > b$.

Embora quaisquer dois números reais possam ser comparados, nem todas as funções são assintoticamente comparáveis. Isto é, para duas funções $f(n)$ e $g(n)$, pode não acontecer $f(n) = O(g(n))$ nem $f(n) = \Omega(g(n))$. Por exemplo, as funções n e $n^{1+\sin n}$ não podem ser comparadas usando notação assintótica, uma vez que o valor do expoente $1+\sin n$ oscila entre 0 and 2.

3. Algoritmos e Classes de Comportamento Assintótico

Algoritmos podem ser classificados segundo as suas ordens de complexidade. As principais classes de comportamento assintótico de algoritmos são as seguintes:

- **$O(1)$** : constante – mais rápido, impossível. O uso do algoritmo independe do tamanho de n . Neste caso as instruções do algoritmo são executadas um número fixo de vezes.
- **$O(\log n)$** : muito bom, ocorre tipicamente em algoritmos que resolvem um problema transformando-o em problemas menores.
- **$O(n)$** : linear – Em geral um pequeno trabalho é realizado sobre cada elemento de entrada. Esta é a melhor situação possível para um algoritmo que tem que processar n elementos de entrada ou produzir n elementos de saída. Cada vez que n dobra de tamanho o tempo de execução dobra.
- **$O(n \log n)$** : limite de muitos problemas práticos, ex.: ordenar uma coleção de números. Este tempo de execução ocorre tipicamente em algoritmos que resolvem um problema quebrando-o em problemas menores, resolvendo cada um deles independentemente e depois juntando as soluções.
- **$O(n^2)$** : quadrático. Algoritmos desta ordem de complexidade ocorrem quando os itens de dados são processados aos pares, muitas vezes em um anel (loop) dentro de outro. Algoritmos deste tipo são úteis para resolver problemas de tamanhos relativamente pequenos.
- **$O(n^k)$** : polinomial – OK para k pequeno.
- **$O(k^n)$, $O(n!)$, $O(n^n)$** : exponencial – evite! Algoritmos desta ordem de complexidade geralmente não são úteis sob o ponto de vista prático. Eles ocorrem na solução de problemas quando se usa força bruta para resolvê-los.

Para ilustrar melhor a diferença entre as classes de comportamento assintótico, vamos observar a Tabela 1 a seguir. Nela é mostrada a razão de crescimento de várias funções de complexidade para tamanhos diferentes de n , em que cada função expressa o tempo de execução em microssegundos. Nesta tabela, um algoritmo linear executa em um segundo um milhão de operações.

Função de custo	$n=10$	$n=20$	$n=30$	$n=40$	$n=50$	$n=60$
n	0,00001 s	0,00002 s	0,00003 s	0,00004 s	0,00005 s	0,00006 s
n^2	0,0001 s	0,0004 s	0,0009 s	0,0016 s	0,0025 s	0,0036 s
n^3	0,001 s	0,008 s	0,027 s	0,064 s	0,125 s	0,316 s
n^5	0,1 s	3,2 s	24,3 s	1,7 min	5,2 min	13 min
2^n	0,001 s	1 s	17,9 min	12,7 dias	35,7 anos	366 séc.
3^n	0,59 s	58 min	6,5 anos	3855 séc.	10^8 séc.	10^{13} séc.

Tabela 1 Diferença entre algumas classes de comportamento assintótico

4. Complexidade e suas cotas

Suponha que A é um algoritmo a ser analisado, I_n é o conjunto de todas as possíveis entradas de A , cada uma com tamanho n . Seja f_A , uma função que expressa o custo de A , ou seja, se I é uma entrada em I_n , $f_A(I)$ é o custo de A para a entrada I . Então:

$$\text{Pior caso: } W(A) = \max_{I \in I_n} f_A(I)$$

Melhor caso: $B(A) = \min_{I \in I_n} f_A(I)$

Caso médio: $E(A) = \sum_{i=0}^n i \cdot p_i$, distribuição de probabilidade, onde p_i é a probabilidade do evento i ocorrer.

Graficamente, podemos representar o pior caso, melhor caso e caso médio conforme a figura a seguir:

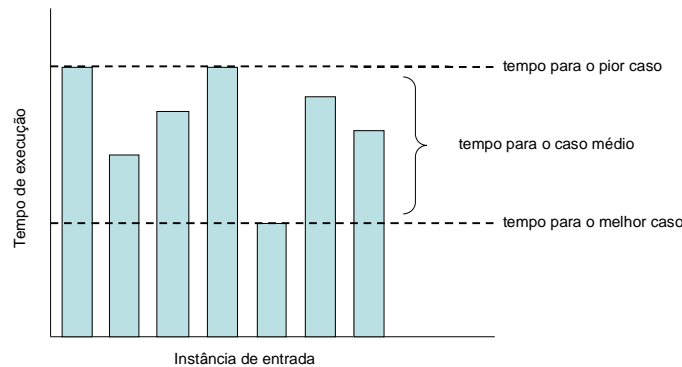


Figura 4 – A diferença entre o tempo para o pior caso e o tempo para o melhor caso. Cada barra representa o tempo de execução de um algoritmo sobre uma entrada diferente.

Exemplo:

Seja o seguinte trecho de algoritmo:

```
(1) para i de 1 até n faça
(2)   se (A[i] <= x)
(3)     A[i] = 2*A[i]
```

Encontre a função de complexidade $f(n)$ do trecho dado, em relação ao **número de multiplicações realizadas** (linha 3), para o **melhor caso**, o **pior caso** e o **caso médio**. Para o caso médio utilize o conceito de distribuição de probabilidade.

No **melhor caso**, a condição da linha (2), $A[i] \leq x$, nunca é satisfeita, logo $f(n)=0$.

No **pior caso**, a condição da linha (2) é sempre satisfeita, logo $f(n)=n$, dado pelo número de vezes que o comando 'para' (linha 1) é executado. Logo, no pior caso, o algoritmo é $O(n)$.

No **caso médio**, vamos considerar a seguinte distribuição de probabilidade:

$$\sum_{i=0}^n i \cdot p_i = 0 \cdot \frac{1}{n+1} + 1 \cdot \frac{1}{n+1} + 2 \cdot \frac{1}{n+1} + \dots + n \cdot \frac{1}{n+1} = (1 + 2 + \dots + n) \cdot \frac{1}{n+1} = \left(\frac{1+n}{2}\right) \cdot n \cdot \frac{1}{n+1} = \frac{n}{2}$$

Assim, no caso médio o algoritmo também é $O(n)$, pois $n/2$ é $O(n)$.

Cota Superior e Cota Inferior de um Problema

Cota Superior de um problema P : é a menor das complexidades de pior caso dos algoritmos existentes (conhecidos) para resolver P . Indica que podemos sempre resolver, para instâncias arbitrárias de tamanho n , o problema P , com tempo menor ou igual à cota superior. Em outras palavras, não devemos ficar satisfeitos com um algoritmo de complexidade de pior caso maior que a cota superior, pois um outro algoritmo de complexidade de pior caso igual à cota superior já existiria. A cota superior é o mínimo sobre todos os algoritmos existentes.

A cota superior de um problema é parecida com o *record mundial* de uma modalidade de atletismo. Ela é estabelecida pelo melhor atleta (algoritmo) do momento. Assim como o *record mundial*, a cota superior pode ser melhorada por um algoritmo (atleta) mais veloz. Na tabela a seguir, temos a cota superior dos 100 metros rasos no atletismo:

Ano	Atleta (Algoritmo)	Tempo
1988	Carl Lewis	9s92
1993	Linford Christie	9s87
1993	Carl Lewis	9s86
1994	Leroy Burrell	9s84
1996	Donovan Bailey	9s84
1999	Maurice Greene	9s79
2002	Tim Montgomery	9s78
2008	Usain Bolt ¹	9s69

Tabela 2 Cota Superior dos 100 metros rasos

Cota Inferior de um problema P : é a complexidade intrínseca ou inerente de um problema P . Isto é, nenhum algoritmo pode resolver o problema com complexidade de pior caso menor que a cota inferior, para entradas arbitrárias de tamanho n . A cota inferior é o mínimo sobre todos os algoritmos possíveis. Na analogia anterior, uma cota inferior de uma modalidade de atletismo não dependeria mais do atleta. Seria algum tempo mínimo que a modalidade exige, qualquer que seja o atleta. Uma cota inferior trivial para os 100 metros rasos seria o tempo que a velocidade da luz leva para percorrer 100 metros no vácuo.

Quando a cota superior de um problema P é igual a sua cota inferior, dizemos que P tem um **algoritmo ótimo** que o resolve.

¹ Estima-se que o homem poderá vencer a distância de 100m em até 9s48, ou seja, correr 0s21 mais rápido do que o atual recorde mundial de 9s69, de Usain Bolt. Estudos anteriores calcularam que, caso Bolt não tivesse "tirado o pé do acelerador" nos últimos 15 metros em Pequim, quando abriu os braços e passou a comemorar o ouro, o jamaicano poderia ter completado os 100 metros em 9s55 ou até menos.

Exemplo:

Suponha que um determinado problema P tem cota inferior igual a n e cota superior igual a n^2 .

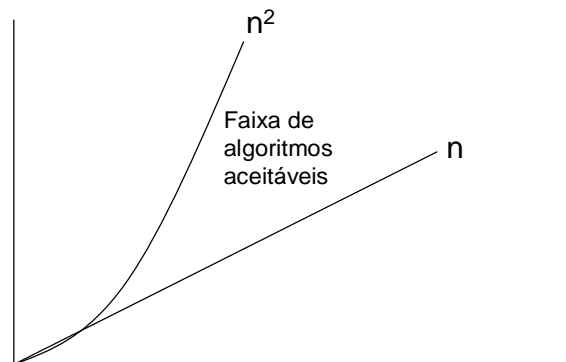


Figura 3 – Entendendo os conceitos de Cota Inferior e Cota Superior

Qualquer algoritmo que resolva P com complexidade de pior caso maior que n^2 (por exemplo, n^3 , n^4) não deveria ser aceito, pois já existe algoritmo $O(n^2)$ que o resolve.

De forma semelhante, também devemos desconfiar de qualquer algoritmo que diz resolver P com complexidade de pior caso menor que n (por exemplo, $\lg n$, $n^{1/2}$), pois essas complexidades estão abaixo da cota inferior n , que representa um limite inferior para as complexidades dos algoritmos que resolvem P.

Qualquer algoritmo com complexidade de pior caso entre a cota superior e a cota inferior (por exemplo, $n \lg n$, $n^{3/2}$) pode ser aceito.

No caso do problema de ordenação, é sabido que a cota inferior e a cota superior são iguais a $n \lg n$. Nesse caso, dizemos que o problema de ordenação tem um algoritmo ótimo que o resolve, pois não tem como esse algoritmo ser melhor (menor que a cota inferior).

5. Alguns Princípios da Análise de Algoritmos

1. O tempo de execução de um **comando de atribuição**, de entrada ou de saída é $O(1)$ ou $O(k)$, k constante.
2. O tempo de execução de uma **seqüência de comandos** é o maior tempo de execução de qualquer comando da seqüência. $O(f(n)) + O(g(n)) = O(\text{Max}(f(n), g(n)))$

Exemplo: Suponha 3 trechos seqüenciais de um programa com tempos de execução $O(n)$, $O(n^2)$ e $O(n \lg n)$. O tempo de execução dos 2 primeiros trechos é $O(\text{max}(n, n^2))$, que é $O(n^2)$. O tempo de execução de todos os 3 trechos é, então, $O(\text{max}(n^2, n \lg n))$, que é $O(n^2)$

3. O tempo de execução de um **comando de decisão simples** é composto pelo tempo para avaliar a condição mais o tempo dos comandos executados dentro da condição.

Obs.: comandos if-then-else

```

if (cond) {
    seqüência de comandos 1
}
else {
    seqüência de comandos 2
}

```

Neste caso, será executada a seqüência de comandos 1 ou a seqüência de comandos 2. Assim, a complexidade de pior caso é dada por $\text{Max}(\text{seq1}, \text{seq2})$. Por exemplo, se a seqüência 1 é $O(N)$ e a seqüência 2 é $O(1)$, a complexidade de pior caso para a declaração inteira do if-then-else é $O(N)$.

4. O tempo para executar um **comando de repetição** é a soma do tempo de execução do corpo do laço mais o tempo de avaliar a condição de parada multiplicado pelo número de iterações do laço.

Ex.: loop FOR

```

for (i = 0; i < N; i++) {
    seqüência de comandos
}

```

O loop acima é executado N vezes, logo a seqüência de comandos é executada também N vezes. Assumindo que a seqüência de comandos é $O(1)$, o tempo total para o loop é $N * O(1)$, ou seja, $O(N)$.

Ex.: loop FOR aninhado

```

for (i = 0; i < N; i++) {
    for (j = 0; j < M; j++) {
        seqüência de comandos
    }
}

```

O FOR mais externo executa N vezes. Cada vez que ele é executado, o FOR mais interno executa M vezes. Como resultado a seqüência de comandos do FOR mais interno é executada $N * M$ vezes. Assim, a complexidade é $O(N * M)$. Se a condição de parada do FOR mais interno fosse N , ao invés de M , a complexidade do trecho acima seria $O(N^2)$, ou seja, complexidade quadrática.

Outro exemplo de cálculo de complexidade do loop FOR é apresentado a seguir:

```

for (i = 2; i <= N-2; i++) {
    for (j = 1; j < N+2; j++) {
        c1
    }
}

```

No trecho acima, o comando **c1** é executado $\sum_{i=2}^{N-2} \sum_{j=1}^{N+1} 1$ vezes, ou seja

$$\sum_{i=2}^{N-2} (N+1) = (N+1)(N-3) = N^2 - 2N - 3 \text{ vezes.}$$

Podemos também ter situações com um loop aninhado onde o número de vezes do loop mais interno depende do valor do índice no loop mais externo, como no trecho a seguir:

```

for (i = 0; i < N; i++) {
    for (j = i; j < N; j++) {
        c1
    }
}

```

No trecho acima, o comando **c1** é executado $\sum_{i=0}^{N-1} \sum_{j=i}^{N-1} 1$ vezes, ou seja

$$\sum_{i=0}^{N-1} (N-i) = (N) + (N-1) + (N-2) + \dots + 1 = \frac{N^2 + N}{2} \text{ vezes.}$$

5. Programas com rotinas não recursivas

- (i) Cada rotina é tratada separadamente iniciando por aquelas que não chamam outra rotina.
- (ii) A seguir, computar o tempo das rotinas que chamam as rotinas que não chamam outras rotinas, utilizando os tempos já avaliados em (i).
- (iii) Repetir o processo até chegar na rotina principal.

Obs.: Comandos com chamadas de rotinas:

Quando um comando envolve a chamada de uma rotina, a complexidade do comando deve incluir a complexidade da rotina chamada. Assuma que é sabido que uma rotina *f* tem tempo constante, e uma rotina *g* tem um tempo linear proporcional a um parâmetro *k*. Então, os comandos a seguir possuem as complexidades indicadas:

```

f(k); // O(1)
g(k); // O(k)

```

Quando essas rotinas são chamadas em um loop FOR, por exemplo, as mesmas regras continuam sendo aplicadas. Por exemplo:

```

for (j = 0; j < N; j++) g(N);

```

tem complexidade $O(N^2)$. O loop é executado *N* vezes e cada chamada da rotina *g(N)* tem complexidade $O(N)$.

6. Programas com rotinas recursivas

Para analisar uma rotina recursiva é necessário primeiro encontrar uma relação de recorrência que descreve a rotina. Em seguida, devemos "resolver" a relação de recorrência. Na próxima seção, veremos com mais detalhes como encontrar a complexidade de algoritmos recursivos.

6. Análise de Complexidade de Algoritmos Recursivos

Para analisar um algoritmo recursivo é necessário "resolver" uma relação de recorrência. Uma *relação de recorrência* é uma "fórmula" que define uma função (em geral sobre os números naturais). Por exemplo, eis uma relação de recorrência que define uma função *T* nos pontos 1, 2, etc.

$$\begin{aligned}
 T(1) &= 1 \\
 T(n) &= T(n-1) + 3n + 2 \text{ para } n = 2, 3, 4, \text{ etc.}
 \end{aligned}$$

Os valores de $T(n)$ para valores pequenos de n são:

n 1 2 3 4 5 6 ...

$T(n)$ 1 9 20 34 51 71 ...

Para a relação de recorrência:

$$T(1) = 2$$

$$T(n) = 2T(n-1) \text{ para } n \geq 2,$$

temos:

$$T(1) = 2$$

$$T(2) = 2T(1) = 2 \cdot 2 = 2^2$$

$$T(3) = 2T(2) = 2 \cdot 2^2 = 2^3$$

$$T(4) = 2T(3) = 2 \cdot 2^3 = 2^4$$

$$T(5) = 2T(4) = 2 \cdot 2^4 = 2^5$$

Podemos concluir que $T(n)=2^n$. Esta equação é denominada **solução em forma fechada** para a relação de recorrência $T(n)$ sujeita à condição básica $T(1)$.

Denomina-se RESOLVER uma relação de recorrência ao processo de se encontrar uma solução em forma fechada para a recorrência. Sempre que possível é bom encontrar uma solução em forma fechada para a recorrência.

Resolver recorrências nem sempre é fácil. Existem vários métodos de resolução de relações de recorrência. Veremos a seguir alguns desses métodos, suficientes para resolver as relações de recorrência da maioria dos algoritmos recursivos que iremos estudar.

6.1. Método da Iteração

Nem sempre temos intuição suficiente sobre a forma geral da recorrência para dar um palpite correto. O método da iteração permite que se reconheça um padrão sem necessidade de chutar ou adivinhar, como fizemos na recorrência $T(n)=2T(n-1)$. A solução do problema da recorrência é obtida através de manipulação algébrica da expressão, geralmente resolvendo-se um somatório.

O método da iteração consiste esquematicamente de:

- Algumas iterações do caso geral são expandidas até se encontrar uma lei de formação.
- O somatório resultante é resolvido substituindo-se os termos recorrentes por fórmulas envolvendo apenas o(s) caso(s) base.

Exemplo:

$$T(n) = 2T(n/2) + n$$

$$T(1) = 1$$

Resolvendo esta recorrência pelo método da iteração, temos:

$$T(n) = 2T(n/2) + n$$

$$= 2T(n/4) + n/2 + n = 4T(n/4) + 2n$$

$$= 4(2T(n/8) + n/4) + 2n = 8T(n/8) + 3n$$

$$= 8(2T(n/16) + n/8) + 3n = 16T(n/16) + 4n$$

= ...

$$= 2^k T(n/(2^k)) + kn$$

Lembramos que, no limite, temos que chegar no caso base da recursão, ou seja, $T(1)$. Para termos a fórmula acima em termos de $T(1)$, $n/(2^k)$ tem que convergir para 1, e isso só acontece se $2^k = n$, ou seja, $k = \lg n$. Temos então:

$$T(n) = 2^{\lg n} T(n/2^{\lg n}) + (\lg n)n$$

$$= n^{\lg 2} T(1) + n \lg n$$

$$= n + n \lg n$$

6.2. Teorema Mestre

Utilizado para resolver recorrências cujo caso geral é da forma $T(n) = a.T(n/b) + f(n)$, onde $a \geq 1$ e $b > 1$ são constantes, n/b significa $\lceil n/b \rceil$ ou $\lfloor n/b \rfloor$, e $f(n)$ é uma função assintoticamente positiva.

Podemos interpretar esta relação de recorrência associada a um algoritmo recursivo como: o algoritmo divide o problema em a partes iguais, cada uma de tamanho b vezes menor que o problema original. O trabalho executado em cada instância da recursão é determinado pela função $f(n)$.

O método consiste no teste de três casos, tornando mais simples a solução de muitas recorrências:

1. Se $f(n) = O(n^{\log_b a - \epsilon})$ para uma constante $\epsilon > 0$,
então $T(n) = \Theta(n^{\log_b a})$
2. Se $f(n) = \Theta(n^{\log_b a})$ para uma constante $\epsilon > 0$,
então $T(n) = \Theta(n^{\log_b a} \lg n)$
3. Se $f(n) = \Omega(n^{\log_b a + \epsilon})$ para uma constante $\epsilon > 0$,
e se $af(n/b) \leq cf(n)$, para alguma constante $c < 1$ e n suficientemente grande
então $T(n) = \Theta(f(n))$

Nos 3 casos estamos comparando a função $f(n)$ com a função $n^{\log_b a}$. A solução da recorrência é dada pela maior das duas funções.

- No caso 1, a função $n^{\log_b a}$ é maior, então a solução é $T(n) = \Theta(n^{\log_b a})$
- No caso 3, a função $f(n)$ é maior, então a solução é $T(n) = \Theta(f(n))$
- No caso 2, as funções são de mesma dimensão, sendo introduzido um fator $\lg n$ e a solução é $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(f(n) \lg n)$

É importante ressaltar que o teorema não cobre todos os casos possíveis. No caso 1, $f(n)$ deve ser menor que $n^{\log_b a}$ por um fator polinomial n^ϵ , para alguma constante $\epsilon > 0$. No caso 3, $f(n)$ deve ser maior que $n^{\log_b a}$ por um fator polinomial n^ϵ , para alguma constante $\epsilon > 0$.

Além disso, para o caso 3, deve também ser satisfeita a condição de regularidade onde $af(n/b) \leq cf(n)$, constante $c < 1$.

Exemplo: (usando a mesma relação de recorrência utilizada no método da iteração)

$$T(n) = 2T(n/2) + n$$

$$T(1) = 1$$

A relação de recorrência se encaixa no caso geral $T(n) = a.T(n/b) + f(n)$, onde $a(1 < a)$ e $b(b > 1)$ são constantes e $f(n)$ é uma função assintoticamente positiva.

Na recorrência $T(n) = 2T(n/2) + n$, temos $a=2$, $b=2$, $f(n)=n$

Vamos testar o caso 1 do Teorema Mestre:

1. Se $f(n) = O(n^{\log_b a - \epsilon})$ para uma constante $\epsilon > 0$,
então $T(n) = \Theta(n^{\log_b a})$

$n = O(n^{\log_2 2 - \epsilon})$ para uma constante $\epsilon > 0$?

$n = O(n^{1-\epsilon})$ para uma constante $\epsilon > 0$? É fácil verificar que não!

Vamos então testar o caso 2 do Teorema Mestre:

2. Se $f(n) = \Theta(n^{\log_b a})$ para uma constante $\epsilon > 0$,
então $T(n) = \Theta(n^{\log_b a} \lg n)$

$n = \Theta(n^{\log_2 2})$ para uma constante $\epsilon > 0$?

$n = \Theta(n)$ para uma constante $\epsilon > 0$? É fácil verificar que sim!

Logo, pela conclusão do caso 2, temos que $T(n) = \Theta(n^{\log_2 2} \lg n) = \Theta(n \lg n)$

6.3. Complexidade de relações-padrão

- $T(n) = T(n/c) + k$, k, c constantes, $c > 1$, $T(n) = O(\log_c n)$
- $T(n) = cT(n/c) + k$, k, c constantes, $c > 1$, $T(n) = O(n)$
- $T(n) = aT(n/c) + k$, k, a, c constantes, $a > c$, $c > 1$, $T(n) = O(n^{\log_c a})$
- $T(n) = aT(n/c) + kn$, k, a, c constantes, $c > 1$,
 $a < c$, $T(n) = O(n)$
 $a = c$, $T(n) = O(n \lg n)$
 $a > c$, $T(n) = O(n^{\log_c a})$

7. Medidas Empíricas de Desempenho de Algoritmos

A análise de algoritmos pode ter um enfoque teórico e/ou prático.

A **análise teórica** de algoritmos de ordenação, por exemplo, é uma análise matemática geralmente feita em relação às seguintes funções de complexidade **C(n)**, número de comparações realizadas na estrutura sendo ordenada, e **M(n)**, número de movimentações realizadas na estrutura, considerando o **pior caso**, **melhor caso** e **caso médio**.

Já a **análise empírica (prática)**, também chamada de análise de desempenho ou bateria de testes, leva em consideração o tipo de computador onde o programa foi executado, a linguagem em que foi implementado, tipo da

chave (grande ou pequena), estruturas a serem ordenadas (leves ou pesadas), variação no tamanho das entradas, variação na forma das entradas (ascendente, descendente e aleatória). Os resultados de uma avaliação empírica de algoritmos de ordenação consistem nos tempos coletados para se ordenar um conjunto de elementos, números de comparações realizadas, número de movimentações e gráficos ilustrativos. O objetivo deste tipo de análise é verificar, na prática, o comportamento assintótico de um algoritmo. No caso de algoritmos de ordenação, deve-se coletar dados quantitativos para o tempo em função do número de elementos a serem ordenados, e em seguida traçar gráficos com os valores obtidos. Os passos seguidos em uma bateria de testes empíricos de algoritmos de ordenação são basicamente os seguintes:

- i) Implementação do algoritmo de ordenação em uma linguagem de programação.
- ii) Para a ordenação de registros com chaves grandes, constrói-se um procedimento que gere valores randômicos para gerar as várias chaves a serem inseridas no vetor. Para isto, usa-se a função *random* (ou uma similar na linguagem escolhida) para gerar os números. Cada chave deve ser formada pela concatenação de n caracteres obtidos da conversão de cada número gerado. Para que as várias chaves tenham valores distintos usa-se o procedimento *randomize* (ou correspondente) antes de iniciar a geração de uma chave.

```
Tipo Item = estrutura
           chave: string 200
           outros campos dependentes da aplicação
fim-estrutura
```

obs.: o tamanho da estrutura dos itens a serem ordenados influencia diretamente no tempo em relação ao número de movimentações realizadas na estrutura, uma vez que em muitas linguagens ocorre a chamada *deep copy* nas atribuições.

- iii) Coleta do tempo gasto para ordenar conjuntos com números crescentes de elementos, por exemplo 100, 500, 1000, 2000, 3000, etc. Para obter o tempo, insere-se no programa comandos de interface com o sistema operacional e obtém-se, por subtração, o valor desejado.

- iv) Medida do tempo para ordenar estruturas já ordenadas, inversamente ordenadas e aleatoriamente ordenadas.

- v) Para cada caso-teste são traçados gráficos do tipo $n \times t$ (onde n é o número de elementos a ser ordenado e t , o tempo gasto na ordenação). Também podem ser traçados gráficos do tipo $c \times t$ e $m \times t$, relativos ao número de comparações na estrutura e número de movimentações realizadas, respectivamente.

Exercícios de Fixação

1) Usando a definição da ordem O , demonstre a seguinte propriedade:

$$O(f(n)) + O(g(n)) = O(\text{Max}(f(n), g(n)))$$

2) Prove que $an^2 + bn + c = \theta(n^2)$, para quaisquer constantes a, b, c e $a > 0$.

3) Dê um exemplo de duas funções $f(n)$ e $g(n)$, $f(n) \neq g(n)$, onde $f(n) = O(g(n))$ e $g(n) = O(f(n))$. Justifique.

4) Marque **V** ou **F**. Justifique.

	Sempre que $f=O(h)$ e $g=O(h)$, $f=O(g)$
	Se $f \neq g$ e $f=O(g)$ então $g=O(f)$
	As funções $n \cdot \log(n)$ e $n \cdot \log(n \cdot n)$ possuem a mesma ordem de complexidade
	$\log(n^c)$ é $\Theta(\log(n))$ para qualquer constante $c > 0$
	2^{100} é $O(1)$
	$2^{n-1} = O(2^n)$
	$2^n = \omega(2^{n-1})$

5) Em um futuro próximo, você precisa resolver um determinado problema. João V. Lozz lhe oferece dois algoritmos, A1 e A2, que resolvem o problema com funções de complexidade $n^2 + n$ e $10^3 n \lg n$, respectivamente. Qual desses algoritmos você escolheria? Justifique cuidadosamente a sua resposta.

6) Qual o menor valor de n de forma que um algoritmo cujo tempo de execução é $100n^2$ roda mais rápido que um algoritmo cujo tempo de execução é 2^n , rodando na mesma máquina?

7) Classifique as seguintes funções de acordo com a ordem de crescimento de cada uma, ou seja, encontre uma ordem $g_1, g_2, g_3, \dots, g_{10}$, de funções que satisfaçam $g_1 = \Omega(g_2)$, $g_2 = \Omega(g_3)$, $g_3 = \Omega(g_4)$, \dots , $g_9 = \Omega(g_{10})$.

$$\lg(\lg n), n^{1/2}, n^2, n^2 \lg n, n \lg n, \sqrt{\lg n}, n^n, n^3, 5!, 2^{\lg n}$$

8) João V. Lozz, na análise de quatro algoritmos para criptografia de dados, A₁, A₂, A₃ e A₄, encontrou as seguintes expressões para o número de operações matemáticas realizadas:

$$\begin{array}{ll} A_1: f(n) = \log n^{10} & A_2: f(n) = n^{2/3} \\ A_3: f(n) = n^{3/2} & A_4: f(n) = \sqrt[4]{n} \end{array}$$

Ajude-o a demonstrar o seguinte:

- a) A₁ é $\theta(\log n)$
- b) A₂ é $o(n \cdot \lg n)$
- c) A₃ é $\omega(n \cdot \lg n)$
- d) A₄ é $\Omega(\ln^2 n)$

9) Marque V ou F. Justifique .

	Um algoritmo com complexidade $O(\lg n)$ é adequado para resolver um problema cuja cota inferior é n .
	Um algoritmo com complexidade $O(n^{3/2})$ não deveria ser usado para resolver um problema cuja cota superior é n^2 e cota inferior n .
	Um algoritmo com complexidade $\Omega(n^3)$ não deveria ser usado para resolver um problema cuja cota superior é n^3 .

10) João V. Lozz diz ter desenvolvido um algoritmo de complexidade $O(n^{2/3})$ para ordenar um conjunto de n elementos. Você compraria este algoritmo? Justifique cuidadosamente a sua resposta. (obs.: sabe-se que a **cota inferior** e a **cota superior** para o problema de ordenação é $n \cdot \lg n$).

11) José \$á Bido diz ter desenvolvido um algoritmo de complexidade $O(n^{3/2})$ para ordenar um conjunto de n elementos. Você compraria este algoritmo? Justifique cuidadosamente a sua resposta.

12) Considere o algoritmo a seguir para encontrar o maior elemento e o menor elemento de uma sequência $A[1..n]$, $n \geq 1$.

```

Procedure MaxMin (var A: Vetor; var Max, Min: integer);
var i: integer;
begin
    Max := A[1];
    Min := A[1];
    for i := 2 to n do
    begin
        if A[i] > Max then Max := A[i];
        if A[i] < Min then Min := A[i];
    end
end;

```

Seja $C(n)$ o número de comparações entre os elementos de A . a) Calcule $C(n)$ para o melhor caso, pior caso e caso médio. b) Proponha um algoritmo mais eficiente para resolver este problema. c) Calcule $C(n)$ para esse novo algoritmo.

13) Encontre o número de comparações no caso médio para o seguinte trecho de programa:

```

for i:= 2 to n do
    if A[i] < x then x := A[i];

```

14) Dado o seguinte trecho do algoritmo:

```

(1) i := 1;
(2) repetir
(3)   se (A[i] <= 10) então x := A[i]
(4)   senão se (A[i] <= 100) então y := A[i]
(5)   senão se (A[i] <= 1000) então z := A[i];
(6)   i := i + 1
(7) até (i > n);

```

Encontre a **função de complexidade $C(n)$** , do **caso médio**, em relação ao número de **comparações** realizadas no vetor A .

15) Encontre a complexidade de cada um dos seguintes trechos de programa:

a) Dois loops em seqüência:

```
for (i = 0; i < N; i++) {  
    seqüência de comandos  
}  
for (j = 0; j < M; j++) {  
    seqüência de comandos  
}
```

O que acontece se trocarmos a complexidade do segundo loop por N ao invés de M?

b) Um loop aninhado seguido por um loop não aninhado:

```
for (i = 0; i < N; i++) {  
    for (j = 0; j < N; j++) {  
        seqüência de comandos;  
    }  
}  
for (k = 0; k < N; k++) {  
    seqüência de comandos  
}
```

c) Um loop aninhado onde o número de vezes do loop mais interno depende do valor do índice no loop mais externo:

```
for (i = 0; i < N; i++) {  
    for (j = i; j < N; j++) {  
        seqüência de comandos  
    }  
}
```

16) Encontre a complexidade de **melhor caso** e de **pior caso** do seguinte trecho de programa:

```
if (x==y) {  
    for (i = 0; i < N; i++) {  
        for (j = 0; j < N; j++) {  
            seqüência de comandos;  
        }  
    }  
    for (i = 0; i < N; i++) {  
        seqüência de comandos  
    }  
}  
else {  
    for (i = 0; i < N; i++) {  
        for (j = 0; j < N; j++) {  
            for (k = 0; k < N; k++) {  
                seqüência de comandos;  
            }  
        }  
    }  
}
```

17) Nos trechos de programa a seguir, encontre o número de vezes que o comando c1 é executado:

a)

```
(1) for (i=1; i<=N-1; i++)
(2)   for (j=i+1; j<N; j++)
(3)     c1;
```

b)

```
(1) for (i=N; i>0; i--)
(2)   for (j=N-1; j>1; j--)
(3)     c1;
```

c)

```
(1) for (i=2; i<=N-3; i++)
(2)   for (j=N; j>i; j--)
(3)     c1;
```

18) O tempo de execução de um algoritmo recursivo A é descrito pela relação de recorrência $T(n) = 7T(n/2) + n^2$. Um outro algoritmo recursivo B para resolver o mesmo problema tem um tempo de execução dado pela relação $T'(n) = \alpha T'(n/4) + n^2$. Usando o *Teorema Master*, determine qual o maior valor inteiro de α tal que o algoritmo B seja assintoticamente mais rápido do que o algoritmo A.

19) Resolva a seguinte relação de recorrência pelo método da iteração e pelo método Master:

$$T(n) = 27, \text{ para } n = 1$$

$$T(n) = 2T(n/4) + n, \text{ para } n > 1$$

20) Seja o seguinte programa em Pascal para resolver o problema clássico de recursividade para as *Torres de Hanoi*:

```
Program Hanoi;
var discos: byte;
  procedure Move (num: byte; origem, destino, temp: char);
  begin
    if num > 0 then begin
      Move (num-1, origem, temp, destino);
      writeln (origem, '->', destino);
      Move (num-1, temp, destino, origem)
    end;
  begin
    write ('Numero de discos: ');
    readln (discos);
    Move (discos, 'A', 'C', 'B')
  end.
```

a) Defina uma relação de recorrência relacionada ao número de chamadas recursivas, para o procedimento Move.

b) Calcule a ordem de complexidade O do programa *Hanoi*.

c) Na prática, o que significa o resultado obtido em b?

21) João V. Lozz diz ter desenvolvido um algoritmo de pesquisa recursivo cuja relação de recorrência é dada por:

$$T(n) = 3 T(n/3) + n$$

$$T(1) = 1$$

Encontre a ordem de complexidade desse algoritmo.

22) Encontre a ordem de complexidade da seguinte função recursiva que encontra o n-ésimo número da sequência de Fibonacci:

```
int fib (n: int);
{
    if (n==0) or (n==1) return 1
    else fib = fib(n-1)+fib(n-2)
}
```

23) Um certo algoritmo A tem uma estrutura recursiva que permite descrever seu tempo de computação por:

$$T(n) = T(\lfloor n/3 \rfloor) + T(\lfloor n/5 \rfloor) + \sqrt{n}$$

$$T(1) = 1,$$

onde n representa o tamanho da entrada. Qual a **ordem de complexidade de pior caso** desse algoritmo?

24) Encontre a **ordem de complexidade** do programa XPTO, no **melhor caso** e no **pior caso**.

```
(1) PROGRAM XPTO;
(2)   PROCEDURE P(N: INT);
(3)   {
(4)     IF (N > 1) THEN {
(5)       //DIVIDE N EM DUAS INSTÂNCIAS N1 E N2, DE IGUAL TAMANHO, COM CUSTO N2
(6)       P(N1);
(7)       P(N2)
(8)     }
(9)   };
(10)  FUNCTION F(N: INT): INT;
(11)  {
(12)    IF (N = 0) THEN F := 1
(13)    ELSE F := 3 * F(N-2)
(14)  };
(15)  {
(16)    read(x,y); //x e y são inteiros positivos, x é par e y>=2
(17)    IF (x < Y) THEN P(x)
(18)    ELSE Z := F(y);
(19)  }
```

25) Defina uma relação de recorrência e, a partir dela, encontre a ordem de complexidade de cada um dos algoritmos abaixo:

```

a) Procedure Divide (n: int);
{
  if n > 1 then {
    <divide n por três, com custo  $n^2$ >;
    Divide(n1);
    Divide(n2);
    Divide(n3)
  }
}

```

```

b) Procedure Doodle (n, m: int);
{
  if n > 0 then {
    DrawLine(n, n, m, m);
    DrawLine(n, m, n, m);
    Doodle(n-1, m)
  }
}

```

b1) suponha custos iguais para desenhar linhas de tamanhos diferentes.
 b2) suponha custo proporcional ao tamanho das linhas desenhadas.

26) Apresentamos a seguir, um algoritmo recursivo para encontrar o maior e menor elemento em um conjunto de elementos não repetidos. Encontre a complexidade desse algoritmo.

```

Método MaxMin (i, j, fmax, fmin);
{
  case
    i = j: fmax = fmin = A[i];
    i = j-1: if A[i] < A[j] { fmax = A[j];
                          fmin = A[i] }

    else meio =  $\lfloor (i+j)/2 \rfloor$  ;
    MaxMin (i, meio, gmax, gmin);
    MaxMin (meio+1, j, hmax, hmin);
    fmax = max(gmax, hmax);
    fmin = min(gmin, hmin)
  }
}

```

27) Para os algoritmos de ordenação a seguir, apresente a análise de complexidade de melhor caso, pior caso e caso médio, em relação a $C(n)$, número de comparações nas chaves e $M(n)$, número de movimentações nas chaves .

a)

```

(1) MÉTODO BubbleSort (A: vetor);
(2) {
(3)   for (i=2; i<=n; i++)
(4)     for (j=n; j>=i; j--)
(5)       if (A[j].chave < A[j-1].chave)
(6)         { aux=A[j-1]; A[j-1]=A[j]; A[j]=aux }
(7) };

```

b)

MÉTODO SelectionSort(A: vetor);

```
(1) PARA i = 1 até (n - 1) FAÇA
(2) {
(3)     k = i;
(4)     PARA j = (i + 1) até n FAÇA
(5)         SE (A[j].CHAVE < A[k].CHAVE) ENTÃO k = j;
(6)     aux = A[k];
(7)     A[k] = A[i];
(8)     A[i] = aux
(9) }
```

c)

MÉTODO InsertionSort(A: vetor);

```
(1) PARA i = 2 até n FAÇA
(2) {
(3)     x = A[i];
(4)     A[0] = x; // sentinela
(5)     j = i-1;
(6)     ENQUANTO (A[j].CHAVE > x.CHAVE) FAÇA
(7)     {
(8)         A[j+1] = A[j];
(9)         j = j-1
(10)    }
(11)    A[j+1] = x
(12) }
```

28) Para o algoritmo de ordenação a seguir, apresente a análise de complexidade de melhor caso e pior caso, em relação ao número de chamadas recursivas do método Sort:

MÉTODO QuickSort (A: vetor);**MÉTODO Sort(int Esq, Dir);**

```
(1) {
(2)     i = Esq; j = Dir;
(3)     x = A[(i+j) DIV 2]; // obtém o pivô x
(4)     repetir
(5)         enquanto (A[i].CHAVE < x.CHAVE) i++;
(6)         enquanto (A[j].CHAVE > x.CHAVE) j--;
(7)         se (i <= j)
(8)         {
(9)             aux = A[i]; A[i] = A[j]; A[j] = aux;
(10)            i++; j--;
(11)        }
(12)     até (i>j);
(13)     se (Esq < j) Sort(Esq, j); // ordena partição esquerda
(14)     se (i < Dir) Sort(i, Dir) // ordena partição direita
(15) };
```

```
(1) {
(2)   Sort(1,n)
(3) };
```

29) Desenvolva um algoritmo de busca binária que divide, a cada passo, o conjunto de elementos em dois conjuntos, um de tamanho aproximadamente duas vezes maior que o tamanho do outro. Compare a complexidade desse algoritmo com o algoritmo de busca binária visto em sala.

30) Seja $A = (a_1, a_2, \dots, a_n)$ um vetor ordenado em ordem crescente, que armazena n números inteiros. Considere o algoritmo de busca binária que, dado um valor inteiro x , determina se existe $i \in \{1, 2, \dots, n\}$ tal que $a_i = x$. Especifique um algoritmo recursivo de **busca ternária**, baseado na partição do intervalo de busca em três partes "iguais". Faça uma análise comparativa dos algoritmos de busca binária e de busca ternária em termos de suas complexidades relacionadas ao número de comparações e chamadas recursivas.

31) É possível desenvolver um algoritmo de busca em um vetor ordenado de n elementos, baseado em comparação de chaves, que seja mais eficiente que o algoritmo de busca binária? Justifique.

32) Elabore um algoritmo de complexidade $\theta(n \cdot \lg n)$ que, dado um conjunto **não ordenado** C com n números reais e um real x , determine se existem dois números a e b pertencentes a C , tais que $a + b = x$.

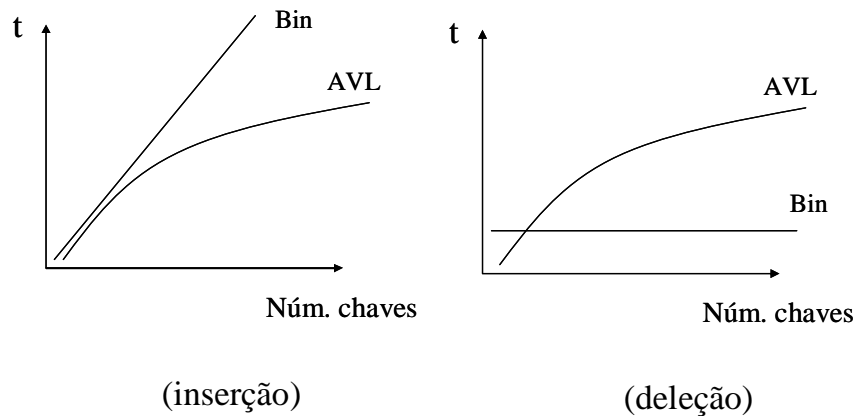
33) Dada a seguinte implementação em Pascal de uma busca recursiva em árvore binária de pesquisa, encontre a ordem de complexidade de melhor caso e pior caso.

```
Function Busca (x: Tchave; raiz: Tree):Tree;
begin
  if raiz=nil then
    Busca = nil
  else
    if raiz^.chave = x then
      busca:=raiz
    else
      if raiz^.chave < x then
        Busca:=Busca(x, raiz^.dir)
      else
        Busca:=Busca(x,raiz^.esq);
    end;
  end;
```

34) Encontre a complexidade do seguinte procedimento para percorrer uma árvore binária com n nodos, imprimindo os seus elementos em ordem:

```
(1) Procedure InOrdem (t: Ptr);
(2) begin
(3)   if t <> nil then begin
(4)     InOrdem(t^.esq);
(5)     writeln (t^.elem);
(6)     InOrdem(t^.dir)
(7)   end
(8) end;
```


35) Em uma bateria de testes de **desempenho** realizada com árvores binárias e árvores AVL foram obtidos os gráficos da figura a seguir. Com base nos gráficos, dê uma interpretação para o que aconteceu nos testes.



36) Seja o seguinte algoritmo em alto nível de busca em uma árvore-B:

```
(1) algoritmo Busca (t, x) //procura por x na árvore t
(2) início
(3)   achou := false;
(4)   enquanto (t ≠ null) e (not achou) faça
(5)     início
(6)       lê página do disco //a primeira página lida é a raiz;
(7)       busca x na página t;
(8)       se encontrar, achou := true, senão segue pelo novo t
(9)     fim-enquanto
(10)  fim
```

Encontre a **função de complexidade** $A(n)$, para o **caso médio**, em relação ao **número de acessos a disco** (obs.: n é a altura da árvore).

37) O algoritmo de ordenação por inserção consta essencialmente de duas etapas disjuntas: a busca do ponto de inserção de cada novo elemento e a inserção propriamente dita deste elemento no vetor ordenado até então.

a) Discuta as vantagens e desvantagens de se usar:

i) busca binária, ao invés de busca seqüencial, no vetor ordenado, durante a primeira etapa.

ii) lista ligada para facilitar a inserção do novo elemento.

b) Usando-se (i) ou (ii) consegue-se melhorar a complexidade no pior caso? Justifique.

38) Faça uma análise de complexidade de melhor caso e pior caso relacionada ao número de comparações $C(n)$ e trocas $T(n)$ da seguinte implementação do *BubbleSort*.

```
// Este programa ordena um vetor de forma ascendente
// Fonte: Deitel, , H. M. C++ Como Programar

#include <iostream>
using std::cout;
using std::endl;
#include <iomanip>
using std::setw;
int main()
{
    const int arraySize = 10;
    int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
    int i, hold;
    cout << "Data items in original order\n";
    for ( i = 0; i < arraySize; i++ )
        cout << setw( 4 ) << a[ i ];
    for ( int pass = 0; pass < arraySize - 1; pass++ ) // passes
        for ( i = 0; i < arraySize - 1; i++ )          // one pass
            if ( a[ i ] > a[ i + 1 ] ) {                // one comparison
                hold = a[ i ];                          // one swap
                a[ i ] = a[ i + 1 ];
                a[ i + 1 ] = hold;
            }
    cout << "\nData items in ascending order\n";
    for ( i = 0; i < arraySize; i++ )
        cout << setw( 4 ) << a[ i ];

    cout << endl;
    return 0;
}
```

Bibliografia

- *Algoritmos: Teoria e Prática*, T. H. Cormen, C. E. Leiserson, R. L. Rivest & C. Stein, Editora Campus, 1991
- *Computer Algorithms*, E. Horowitz, S. Sahni & S. Rajasekaran, Computer Science Press, 1998.
- *Data Structures and Algorithms*, Aho, A., Hopcroft, J. F. & Ullman, J. D.
- *Fundamentals of Computer Algorithms*, Horowitz, E. & Sahni, S.
- Projeto de Algoritmos: fundamentos, análise e exemplos da Internet, Michael Goodrich, Bookman, 2004.
- Projeto e Análise de Algoritmos, Elton Silva, 2006 (anotações do professor)

Anexo A - Fundamentos Matemáticos

Somatório

$\sum_{i=k}^n a_i = a_k + a_{k+1} + a_{k+2} + \dots + a_n$ <p>$(n - k + 1)$ termos</p>	$\sum_{i=1}^n 1 = n$
$\sum_{i=k}^n c a_i = c \sum_{i=k}^n a_i$	
$\sum_{i=k}^n a_i \pm b_i = \sum_{i=k}^n a_i \pm \sum_{i=k}^n b_i$	Atenção: $\sum_{i=k}^n a_i * b_i \neq \sum_{i=k}^n a_i * \sum_{i=k}^n b_i$ $\sum_{i=k}^n a_i / b_i \neq \sum_{i=k}^n a_i / \sum_{i=k}^n b_i$
Soma Aritmética (P.A) $\sum_{i=1}^n a_i = a_1 + a_2 + \dots + a_n = \frac{(a_1 + a_n)n}{2}$	Soma de Quadrados $\sum_{i=0}^n i^2 = \frac{n(n+1)(2n+1)}{6}$ Soma de Cubos $\sum_{i=0}^n i^3 = \frac{n^2(n+1)^2}{4}$
Soma Geométrica (P.G) $\sum_{i=0}^n x^i = x^0 \cdot \frac{x^{n+1} - 1}{x - 1}, x \neq 1$	

Exponencial e Logaritmos

$x^0 = 1$ $x^{-a} = 1 / x^a$ $x^{a+b} = x^a \cdot x^b$ $x^{a-b} = x^a / x^b$ $x^{a \cdot b} = (x^a)^b = (x^b)^a$	$\log_b 1 = 0$ $a = b^{\log_b a}$ $\log_c (a \cdot b) = \log_c a + \log_c b$ $\log_c (a / b) = \log_c a - \log_c b$ $\log_b a^n = n \cdot \log_b a$ $a^{\log_b n} = n^{\log_b a}$ $\log_b a = (\log_c a) / (\log_c b)$ $\log_b a = 1 / (\log_a b)$	Convenções: $\lg a = \log_2 a$ $\ln a = \log_e a \quad e = 2,71\dots$ $\log a = \log_{10} a$ $\log a + b = (\log a) + b$ $\log^k n = (\log n)^k$ $\log \log n = \log(\log n)$
--	---	--

Piso e Teto

$x \in \mathbb{R}$ $\lfloor x \rfloor$: piso de x, maior inteiro menor ou igual a x, ex.: $\lfloor 2,5 \rfloor = 2$ $\lceil x \rceil$: teto de x, menor inteiro maior ou igual a x, ex.: $\lceil 2,5 \rceil = 3$	$x-1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x+1$ Para $x \in \mathbb{Z}$, $\lfloor x/2 \rfloor + \lceil x/2 \rceil = x$ $\lceil x/2 \rceil - 1 < x \leq \lceil x/2 \rceil$
--	--

Regra de L'Hôpital

O limite da razão de duas funções é indeterminado se ambas as funções tendem a zero, ou ambas tendem a infinito. Se $f(x)$ e $g(x)$ são diferenciáveis, com limites indeterminados, então $\lim_{x \rightarrow \infty} (f(x)/g(x)) = \lim_{x \rightarrow \infty} (f'(x)/g'(x))$.

Derivadas:

$$f(x) = g(x) \pm h(x), \quad f'(x) = g'(x) \pm h'(x)$$

$$f(x) = g(x).h(x), \quad f'(x) = g'(x).h(x) + g(x).h'(x)$$

$$f(x) = g(x)/h(x), \quad f'(x) = (g'(x).h(x) - g(x).h'(x))/h(x)^2$$

$$f(x) = g(x)^n, \quad f'(x) = n.g(x)^{n-1}.g'(x)$$

$$f(x) = e^x, \quad f'(x) = e^x$$

$$f(x) = \log_a x, \quad f'(x) = 1/(x.\ln a)$$

$$y = f(g(x)), \quad y' = f'(g(x)).g'(x) \quad (\text{regra da cadeia})$$