Propriedades úteis

1. Séries aritméticas
$$1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2} \tag{1}$$
Se $a_n = a_{n-1} + c$, onde c é uma constante, então:
$$a_1 + a_2 + \cdots + a_n = \frac{n(a_n + a_1)}{2} \tag{2}$$

2. Séries geométricas
$$1 + 2 + 4 + 8 + \cdots + 2^n = 2^{n+1} - 1 \tag{3}$$
A seguinte série é convergente se e somente se $|r| < 1$ e, neste caso, a soma vale:
$$\sum_{n=0}^{\infty} r^n = \frac{1}{1-r} \tag{4}$$

3. Séries harmônicas
$$H_n = \sum_{k=1}^{n} \frac{1}{n} = \ln n + \gamma + \mathrm{O}(1/n) \tag{5}$$
onde $\gamma = 0.577...$ é a constante de Euler.

Crescimento de Funções

4. Critique o seguinte raciocínio: "A derivada de $4n^2 + 2n$ é $8n + 2$. A derivada de n^2 é $2n$. Como $8n+2 > 2n$, podemos concluir que $4n^2+2n$ cresce mais que n^2 e portanto $4n^2+2n$ não é $\mathrm{O}(n^2)$.

5. Critique o seguinte raciocínio: "A derivada de $4n^2 + 2n$ é $8n + 2$. A derivada de $9n^2$ é $18n$. Como $8n + 2 \leq 18n$ para $n \geq 1$, podemos concluir que $4n^2$+2n é $\mathrm{O}(9n^2)$."

6. Disponha as seguintes funções em ordem crescente de complexidade assintótica:
   - $f_1(n) = 500n$
   - $f_2(n) = n \log_2 n$
   - $f_3(n) = n^5$
   - $f_4(n) = 2^n$
   - $f_5(n) = n^2$
   - $f_6(n) = 1$
   - $f_7(n) = \log_2 n$
   - $f_8(n) = \log_2(n^2)$
   - $f_9(n) = n^3$

7. Classifique as funções abaixo do ponto de vista de crescimento assintótico, i.e., se f_i vem antes de f_j na sua ordenação então $f_i = O(f_j)$:
$$2^{\sqrt{\lg n}} \quad 2^n \quad n^{5/4} \quad n \lg^3 n \quad n^{\lg n} \quad 2^{2^n} \quad 2^{n^2}$$

8. Dois algoritmos A e B possuem complexidade n^5 e 2^n, respectivamente. Em qual caso você utilizaria o algoritmo B ao invés do A? Exemplifique.

9. Dadas as funções $f(n) = n^2 + n + 1$ e $g(n) = 2n^3$, apresente a menor constante inteira positiva c tal que $f(n) \leq c \cdot g(n)$, para todo $n \geq 2, n \in \mathbb{N}$.

   Para $c = 1$ já é possível validar a equação solicitada pois $f(n)$ possui um grau inferior ao grau de $g(n)$ e, mesmo com a adição de um fator n, não consegue ser superior a outra equação.

10. Um algoritmo tradicional e muito utilizado possui complexidade de $n^{1,5}$ enquanto um novo proposto é da ordem de $n \log n$. Qual utilizar?

11. Ordene as seguintes funções por ordem de crescimento, ou seja, encontre uma ordenação g_1, g_2, g_3, g_4, g_5, g_6, g_7, g_8 das funções abaixo tal que $g_1 = \mathrm{O}(g_2)$, $g_2 = \mathrm{O}(g_3)$, $g_3 = \mathrm{O}(g_4)$, $g_4 = \mathrm{O}(g_5)$, $g_5 = \mathrm{O}(g_6)$, $g_6 = \mathrm{O}(g_7)$ e $g_7 = \mathrm{O}(g_8)$.

    a) $g_1(n) = n^\pi$
    b) $g_2(n) = \pi^n$
    c) $g_3(n) = \binom{n}{5}$
    d) $g_4(n) = \sqrt{2\sqrt{n}} = 2^{\frac{1}{2}n^{1/2}}$
    e) $g_5(n) = \binom{n}{n-4}$
    f) $g_6(n) = 2^{\log^4 n}$
    g) $g_7(n) = n^{5(\log n)^2}$
    h) $g_8(n) = n^4\binom{n}{4}$

    em que $\binom{n}{k} = \frac{n!}{k!(n-k)!}$, para $0 \leq k \leq n$.

Notação Assintótica

12. Apresente um contra-exemplo para a seguinte afirmação: Sejam $f(n), g(n), r(n)$ e $s(n)$ funções positivas estritamente crescentes. Se $f(n) = \mathrm{O}(s(n))$ e $g(n) = \mathrm{O}(r(n))$ então $f(n) - g(n) = \mathrm{O}(s(n) - r(n))$.

    Contra-exemplo: $f(n) = 2n$, $g(n) = n$, $r(n) = s(n) = n$.

13. Suponha que $f(n)$ e $g(n)$ sejam funções dos inteiros não-negativos nos inteiros não-negativos. Demonstre que as seguintes afirmações são equivalentes:

    a) Existem números reais $a > 1$, $b > 1$, $c > 1$ e existe um número inteiro $n_0 > 0$ tais que $a^{g(n)} \leq b^{f(n)} \leq c^{g(n)}$, para todo $n \geq n_0$;
    b) $f(n) = \Theta(g(n))$.

14. Mostre que a notação ó-grande é transitiva, isto é, se $f(n) = \mathrm{O}(g(n))$ e $g(n) = \mathrm{O}(h(n))$, então $f(n) = \mathrm{O}(h(n))$.

    Resposta: É preciso mostrar que $f(n) = c_3h(n)$ para $n > n_3$. Temos que $f(n) \leq c_1g(n)$ e que $g(n) \leq c_2h(n)$, para $n > n_1$ e para $n > n_2$, respectivamente. Combinando essas desigualdades temos $f(n) \leq c_1g(n) \leq c_1c_2h(n)$ para $n > n_3 = max(n_1, n_2)$. Portanto $f(n) = c_3h(n)$ para $n > n_3$, em que $n_3 \geq max(n_1, n_2)$ e $c_3 \geq c_1c_2$.

15. Verdadeiro ou falso? Prove.

a) $2^{n+1} = \mathrm{O}(2^n)$

Resposta: Verdadeiro. Para mostrar que $2^{n+1} = \mathrm{O}(2^n)$, nós devemos encontrar constantes $c, n_0 > 0$ tal que $0 \leq 2^{n+1} \leq c \cdot 2^n \; \forall n \geq n_0$. Uma vez que $2^{n+1} \leq c \cdot 2^n$ $\forall n$, nós podemos resolver a desigualdade escolhendo $c = 2$ e $n_0 = 1$.

b) $2^{2n} = \mathrm{O}(2^n)$

Resposta: Falso. Por contradição, para mostrar que $2^{2n} \neq \mathrm{O}(2^n)$, suponha que existem constantes $c, n_0 > 0$ tal que $0 \leq 2^{2n} \leq c \cdot 2^n \; \forall n \geq n_0$. Deste modo, $2^{2n} = 2^n \cdot 2^n \leq c \cdot 2^n \Rightarrow 2^n \leq c$, no entanto não existe um valor constante maior que 2^n $\forall n$, e assim chegamos a uma contradição.

c) $10n = \mathrm{O}(n)$

d) $10n^2 = \mathrm{O}(n)$

e) $10n^{55} = \mathrm{O}(2^n)$

f) $n^2 + 200n + 300 = \mathrm{O}(n^2)$

g) $n^2 - 200n - 300 = \mathrm{O}(n)$

h) $\frac{3n^2}{2} + \frac{7n}{2} - 4 = \mathrm{O}(n)$

i) $\frac{3n^2}{2} + \frac{7n}{2} - 4 = \mathrm{O}(n^2)$

j) $n^3 - 999999n^2 - 1000000 = \mathrm{O}(n^2)$

k) $2^{n+1} = \mathrm{O}(2^n)$

l) $3^n = \mathrm{O}(2^n)$

m) $\log_2 n = \mathrm{O}(\log_3 n)$

Resposta: Verdadeiro. Para mostrar que $\log_2 n = \mathrm{O}(\log_3 n)$, nós devemos encontrar constantes $c, n_0 > 0$ tal que $\log_2 n \leq c \cdot \log_3 n \; \forall n \geq n_0$.

$$\begin{aligned}
\log_2 n &\leq c \cdot \log_3 n \\
\frac{\log_3 n}{\log_3 2} &\leq c \cdot \log_3 n \\
\frac{\cancel{\log_3 n}}{\log_3 2} &\leq c \cdot \cancel{\log_3 n} \\
\frac{1}{\log_3 2} &\leq c
\end{aligned}$$

Portanto, podemos escolher $c = \frac{1}{\log_3 2}$ e $n_0 = 1$.

n) $2^{2n} = \mathrm{O}(2^n)$

Resposta: Para provar que a afirmativa é falsa, iremos considerar por aburdo que $2^{2n} = \mathrm{O}(2^n)$. Portanto devem existir constantes c e n_0 tal que $2^{2n} \leq c2^n$, $\forall n \geq n_0$, o que implica em encontrar um valor para c tal que $2^n \leq c$, $\forall n \geq n_0$. Como tal valor não pode ser determinado, a afirmativa é falsa.

o) $\log_3 n = \Theta(\log_2 n)$

Resposta: Verdadeiro. Dividiremos o problema em duas partes para mostrar $\log_3 n = \mathrm{O}(\log_2 n)$ e $\log_3 n = \Omega(\log_2 n)$. Para mostrar que $\log_3 n = \mathrm{O}(\log_2 n)$, nós devemos encontrar constantes $c, n_0 > 0$ tal que $\log_3 n \leq c \cdot \log_2 n \ \forall n \geq n_0$.

$$\log_3 n \leq c \cdot \log_2 n$$
$$\frac{\log_2 n}{\log_2 3} \leq c \cdot \log_2 n$$
$$\frac{\cancel{\log_2 n}}{\log_2 3} \leq c \cdot \cancel{\log_2 n}$$
$$\frac{1}{\log_2 3} \leq c$$

Portanto, podemos escolher $c = \frac{1}{\log_2 3}$ e $n_0 = 1$. Para mostrar que $\log_3 n = \Omega(\log_2 n)$, nós devemos encontrar constantes $c, n_0 > 0$ tal que $\log_3 n \geq c \cdot \log_2 n \ \forall n \geq n_0$. Sem perda de generalidades, podemos utilizar $c = \frac{1}{\log_2 3}$ e $n_0 = 1$. Assim temos que $\log_3 n = \Theta(\log_2 n)$ $\square$

p) $n = \mathrm{O}(2^n)$

q) $\lg n = \mathrm{O}(n)$

r) $n = \mathrm{O}(2^{n/4})$

s) $4 \lg n = \mathrm{O}(n)$

t) $100 \lg n - 10n + 2n \lg n = \mathrm{O}(n \lg n)$

u) $\frac{3n^2}{2} + \frac{7n}{2}n^3 - 4 = \Theta(n^2)$

v) $9999n^2 = \Theta(n^2)$

w) $\frac{n^2}{1000} - 999n = \Theta(n^2)$

x) $log_2 n + 1 = \Theta(log_{10} n)$

y) $n^3 - 3n^2 - n + 1 = \Theta(n^3)$

Resposta[1]: Seja $f(n) = n^3 - 3n^2 - n + 1$ e $g(n) = n^3$. É preciso mostrar que existem constante positivas c_1, c_2 e n_0, tal que $c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n > n_0$. Sendo assim, temos:

$$c_1 n^3 \leq n^3 - 3n^2 - n + 1 \leq c_2 n^3$$

A equação $c_1 n^3 \leq n^3 - 3n^2 - n + 1$ é válida para $c_1 = 1$ e para $n \geq 4$. A equação $n^3 - 3n^2 - n + 1 \leq c_2 n^3$ é válida para $c_2 = 1/16$ e para $n \geq 4$. Portanto, $f(n) = n^3 - 3n^2 - n + 1$ e $g(n) = n^3$ pois existem constantes positivas $c_1 = 1$, $c_2 = 1/4$ e $n_0 = 4$, tal que $c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n > n_0$.

16. Existem duas funções f e g, dos naturais nos reais positivos, tais que $f(n) \notin O(g(n))$ e $g(n) \notin O(f(n))$? Em caso afirmativo, apresente um exemplo. Em caso negativo, prove que tais funções não existem.

17. Prove ou apresente contra-exemplo: se $f(n) = O(g(n))$ e $g(n) = O(f(n))$, então segue que $f(n) = g(n)$ para todo n.

    Contra-exemplo: $f(n) = 2n$ e $g(n) = n$.

18. Prove ou apresente um contra-exemplo para a seguinte afirmação:
    Sejam $f(n), g(n), r(n)$ e $s(n)$ funções positivas crescentes. Se $f(n) = \mathrm{O}(s(n))$ e $g(n) = \mathrm{O}(r(n))$ então $f(n) - g(n) = \mathrm{O}(s(n) - r(n))$.
    Contra-exemplo: $f(n) = 2n$, $g(n) = n$, $r(n) = s(n) = n$.

[1] Esta questão possui infinitas soluções válidas

19. Considere as afirmativas abaixo onde $f(n)$ e $g(n)$ são funções assintoticamente positivas. Para cada uma delas, indique se a afirmativa é verdadeira ou falsa.

    a) Considere um algoritmo que faz 2^{2n} operações. Pode-se dizer que esse algoritmo é $\mathrm{O}(2^n)$.

    b) $f(n) = \Theta(f(n/2))$.

    c) Considere um algoritmo cuja função de complexidade é $f(n) = \lg(n)$. É correto afirmar que esse algoritmo é $\mathrm{O}(n)$ mas não é $\Theta(n)$.

    d) $f(n) + g(n) = \Theta(min(f(n), g(n)))$.

20. Mostre que, para quaisquer constantes reais a e b, onde $b > 0$, $(n + a)^b = \Theta(n^b)$.

21. Nós podemos extender a notação vista em sala de aula para o caso de funções com dois parâmetros n e m que tendem ao infinito em proporções independentes. Para uma dada função $g(n, m)$, nós denotamos por $\mathrm{O}(n, m)$ o conjunto de funções

$$\begin{aligned}\mathrm{O}(g(n,m)) = \{f(n,m) :\ & \text{existem constantes positivas } c, n_0 \text{ e } m_0 \text{ tal que} \\ & 0 \leq f(n,m) \leq c \cdot g(n,m) \text{ para todo } n \geq n_0 \text{ ou } m \geq m_0\}\end{aligned}$$

    Dê as definições correspondentes para $\Omega(g(n, m))$ e $\Theta(g(n, m))$.

    Resposta:

$$\begin{aligned}\Omega(g(n,m)) = \{f(n,m) :\ & \text{existem constantes positivas } c, n_0 \text{ e } m_0 \text{ tal que} \\ & 0 \leq c \cdot g(n,m) \leq f(n,m) \text{ para todo } n \geq n_0 \text{ ou } m \geq m_0\}\end{aligned}$$

$$\begin{aligned}\Theta(g(n,m)) = \{f(n,m) :\ & \text{existem constantes positivas } c_1, c_2, n_0 \text{ e } m_0 \text{ tal que} \\ & 0 \leq c_1 \cdot g(n,m) \leq f(n,m) \leq c_2 \cdot g(n,m) \text{ para todo } n \geq n_0 \text{ ou } m \geq m_0\}\end{aligned}$$

22. Usando a definição formal de Θ prove que $6n^3 \neq \Theta(n^2)$.

23. Sejam f e g duas sequências de números reais positivos. Prove que, se $\lim_{n\to\infty} \frac{f(n)}{g(n)} \in \mathbb{R}$, então $f(n) = O(g(n))$.

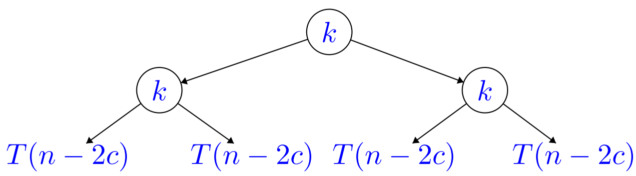
Recorrências

24. Prove que o comportamento assintótico para a relação de recorrência

$$T(n) = 2T(n - c) + k,$$

    onde c e k são constantes, é $T(n) = \mathrm{O}(2^d)$, para algum valor d.

    Observe a seguinte árvore de recursão nos seus 3 primeiros níveis. Observamos que o custo



do nível i é $2^i k$ e a árvore pára de crescer quando $n - ic = 0 \to i = n/c$. Podemos calcular o custo da árvore a partir do seguinte somatório:

$$\sum_{i=0}^{n/c} 2^i k = k2^{n/c+1} - k = \mathrm{O}(2^d)$$

para $d = n/c$.

25. Dê a solução exata da recorrência:

$$T(n) = \begin{cases} 1, & \text{se } n = 1. \\ T(n-1) + \frac{n}{2}, & \text{se } n > 1 \end{cases}$$

26. Dê a solução exata da recorrência:

$$T(n) = \begin{cases} 1, & \text{se } n = 1. \\ 4, & \text{se } n = 2 \\ 8T(n-1) + 15T(n-2), & \text{se } n > 2 \end{cases}$$

27. Se $T(0) = T(1) = 1$, cada uma das seguintes recorrências define uma função T nos inteiros não negativos.

    a) $T(n) = 3T(\lfloor n/2 \rfloor) + n^2$.
    b) $T(n) = 2T(n-2) + 1$.
    c) $T(n) = T(n-1) + n^2$.

    Qual delas não pode ser limitada por uma função polinomial? Justifique a sua resposta.

28. Resolva as seguintes recorrências:

    a) $T(n) = T(\frac{n}{3}) + T(\frac{2n}{3}) + \Theta(n)$
    b) $T(n) = \log n + T(\sqrt{n})$
    c) $T(n) = T(n-1) + 3n + 2$
    d) $T(n) = T(n-1) + n$
    e) $T(n) = 2T(n/2) + \lg(n)$
    f) $T(n) = 2T(n/2) + n\lg(n)$
    g) $T(n) = 9T(n/3) + n$
    h) $T(n) = 2T(n/3) + 1$
    i) $T(n) = 2T(n/2) + n\lg^2(n)$
    j) $T(n) = 4T(n/2) + 1$
    k) $T(n) = 4T(n/2) + n$
    l)

    $$T(n) = \begin{cases} 1, & \text{se } n = 1 \\ T(n/2) + \sqrt{n}, & \text{se } n > 1 \end{cases}$$

    Pelo Teorema Mestre, seja $a = 1$, $b = 2$, $f(n) = \sqrt{(}n)$ e $n^{\log_2 1} = n^0$. Considerando o caso 3, temos $\sqrt{n} = n^{1/2} = \Omega(n^{0+\epsilon})$ e $\sqrt{n/2} \leq c\sqrt{n}$, para $c = \frac{1}{\sqrt{2}}$, portanto $T(n) = \Theta(\sqrt{n})$.

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

n) $T(n) = 2T(n/4) + \sqrt{n}$

Resposta: Recorrência do tipo divisão e conquista que pode ser revolvida diretamente com o uso do Teorema Mestre, onde $a = 2$, $b = 4$, $f(n) = \sqrt(n)$, e $n^{\log_b a} = n^{\log_4 2} = \sqrt(n)$. Uma vez que $\sqrt(n) = \Theta(n^{\log_4 2})$, podemos aplicar o caso 2 do Teorema Mestre e assim temos que $T(n) = \Theta(\sqrt{n} \lg n)$.

o) $T(n) = T(\sqrt{n}) + 1$

Resposta: Seja $m = \lg n$ e $S(m) = T(2^m)$. $T(2^m) = T(2^{m/2}) + 1$, tal que $S(m) = S(m/2)+1$. Utilizando o Teorema Mestre, temos $n^{\lg_b a} = n^{\log_2 1} = n^0 = 1$ e $f(n) = 1$. Uma vez que $f(n) = \Theta(1)$, podemos aplicar o caso 2 e $S(m) = \Theta(\lg m)$. Finalmente, $T(n) = \Theta(\lg \lg n)$.

29. Seja T uma função que leva números naturais em números reais. Suponha que T satisfaz a recorrência $T(n) = 2T(\frac{n}{2}) + 7n + 2$. Mostre que $T(n) = \Omega(n \log n)$.

30. Considere a recorrência

$$T(n) = \begin{cases} 1, & \text{se } n = 1. \\ T(n-1) + 1, & \text{se } n > 1 \text{ é ímpar} \\ 3T(n/2), & \text{se } n > 1 \text{ é par} \end{cases}$$

(a) Prove que, sempre que n é uma potência de dois, $T(n) = n^{\log_2 3}$.

(b) Prove que T é crescente.

(c) Prove que existe $k \geq 1$ tal que $k \cdot n^{\log_2 3} \geq (2n)^{\log_2 3}$ para todo $n \in \mathbf{N}$.

31. Considere a recorrência

$$U(n) = \begin{cases} 1, & \text{se } n = 1. \\ 2U(n-1), & \text{se } n > 1 \text{ é ímpar} \\ 3U(n/2), & \text{se } n > 1 \text{ é par} \end{cases}$$

Prove que, sempre que n é uma potência de dois, $U(n) = n^{\log_2 3}$.

32. Considere os três algoritmos abaixo. Qual a complexidade de cada um desses algoritmos?

- **Algoritmo A**: resolve um problema de tamanho n dividindo-o em 5 subproblemas de tamanho $\frac{n}{2}$, recursivamente resolve cada subproblema, e então combina as soluções em tempo linear.

  A complexidade pode ser calculada a partir da recorrência $\mathrm{T}(n) = 5\mathrm{T}(\frac{n}{2}) + \mathrm{O}(n)$. Sabendo disso, utilizamos o caso 1 do teorema mestre - já que $a = 5$, $b = 2$ e $n^{\log_b a} = n^{\log_2 5}$, tomamos $\epsilon = 1$ e ficamos, para todo $n > 0$, com $n^{\log_2 5-\epsilon} = n^{\log_2 5-1} = n^{\log_2 (5/2)} > n^{\log_2 2} = n$. Disso, segue por definição que $O(n) = O(n^{\log_2 5-\epsilon})$. O segundo caso do teorema mestre, então, garante que $T(n) = O(n^{\lg 5})$.

- **Algoritmo B**: resolve um problema de tamanho n dividindo-o em 2 subproblemas de tamanho $n-1$, recursivamente resolve cada subproblema, e então combina as soluções em tempo constante.

  A complexidade pode ser calculada a partir da recorrência $\mathrm{T}(n) = 2\mathrm{T}(n-1) + \mathrm{O}(1)$. Tomemos $n = \lg m$ e $T(\lg m) = S(m)$. Podemos escrever então que $T(\lg m) = 2T(\lg$

$m-1)+O(1) \Rightarrow T(lg\ m) = 2T(lg\ m - lg\ 2)+O(1) \Rightarrow T(lg\ m) = 2T(\lg \frac{m}{2})+O(1) \Rightarrow S(m) = 2S(\frac{m}{2}) + O(1)$. Podemos resolver essa recorrência pelo caso 1 do teorema mestre, tomando $\epsilon = 1$. Logo teremos que $O(1) = O(n^{\lg 1}) \Rightarrow O(1) = O(n^0) = O(1)$. Logo $S(m) = \Theta(m)$. Fazendo a substituição, como $m = 2^n$, $T(n) = \Theta(2^n)$.

- **Algoritmo C**: resolve um problema de tamanho n dividindo-o em 9 subproblemas de tamanho $\frac{n}{3}$, recursivamente resolve cada subproblema, e então combina as soluções em tempo $\mathrm{O}(n^2)$.

  A complexidade pode ser calculada a partir da recorrência $\mathrm{T}(n) = 9\mathrm{T}(\frac{n}{3}) + \mathrm{O}(n^2)$. Caso a combinação de soluções fosse feita em tempo $\Theta(n^2)$, usando o caso 2 do teorema mestre, tomando $k = 0$, teríamos que $T(n) = \Theta(n^2 \lg n)$. Infelizmente, a combinação de soluções é feita em tempo $O(n^2)$. Assim, ajustamos nossa estimativa de $T(n)$ para $O(n^2 \lg n)$ e tentamos verificar pelo método da substituição:

$$\begin{aligned}
T(n) &= 9T\left(\frac{n}{3}\right) + O(n^2) \\
&\leq 9c\left(\frac{n}{3}\right)^2 \lg\left(\frac{n}{3}\right) + O(n^2) \\
&= cn^2 \lg\left(\frac{n}{3}\right) + O(n^2) \\
&= cn^2 \lg n - cn^2 \lg 3 + O(n^2) \\
&\leq cn^2 \lg n - cn^2 \lg 3 + dn^2 \\
&= cn^2 \lg n + n^2(d - c\lg 3)
\end{aligned}$$

  Resolvendo, então, a inequação $cn^2 \lg n + n^2(d - c\lg 3) \leq cn^2 \lg n$:

$$\begin{aligned}
cn^2 \lg n + n^2(d - c\lg 3) \leq& cn^2 \lg n \\
c\lg n + d - c\lg 3 \leq& c\lg n \\
d - c\lg 3 \leq& 0 \\
d \leq& c\lg 3
\end{aligned}$$

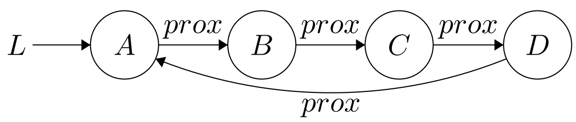
  Tomando $d \leq c\lg 3$, temos:

$$\begin{aligned}
T(n) &= 9T\left(\frac{n}{3}\right) + O(n^2) \\
&\leq cn^2 \lg n - cn^2 \lg 3 + O(n^2) \\
&\leq cn^2 \lg n - cn^2 \lg 3 + dn^2 \\
&\leq cn^2 \lg n
\end{aligned}$$

  o que mostra a validade da substituição. Logo, $T(n) = O(n^2 \lg n)$.

Projeto de algoritmos

33. A imagem a seguir apresentar um exemplo de uma lista encadeada circular `L`, em que a última célula aponta para a primeira da lista.



Escreva uma função, que recebe um ponteiro[2] para o início de uma lista como parâmetro, e que conte o número de células de uma lista encadeada circular.

Solução do Professor:

```
int conta(node* L){
    if ( L == NULL ) return 0;
    node* inicio=L, *i;
    int contador = 1;
    for ( i = L->prox; i != inicio; i = i->prox )
        contador++;
    return contador;
}
```

34. Apresente um algoritmo que receba a pilha A com n elementos e a pilha B com m verifique se elas são iguais, retornando verdadeiro ou falso. Qual a complexidade deste algoritmo?

35. (5%) Suponha que as chaves 50 30 70 20 40 60 80 15 25 35 45 36 são inseridas, nesta ordem, numa árvore de busca vazia. Em seguida remova o nó que contém o valor 30. Desenhe a árvore imediatamente antes e imediatamente após a remoção.

36. Escreva uma função que troque de posição dois nós de uma mesma lista duplamente encadeada.

```
// Parametros: 2 ponteiros para os nos que serao trocados
void troca(ponteiro A,ponteiro B){
   // insira seu codigo aqui!
}
```

37. Escreva uma função que insira em uma lista encadeada um novo nó com conteúdo `x` imediatamente após o `k`-ésimo[3] nó. Descreva a estrutura[4] de representação do nó.

```
void insere(node** L,int k, int x){
   // insira seu codigo aqui!
}
```

38. Dada um fila com n elemenos implementada por meio de uma lista simplesmente encadeada, implemente uma função `inverte(node **f)` que inverta a ordem dos elementos da fila. Qual a complexidade da função?

39. Um palíndromo é uma palavra que possui a propriedade de poder ser lida tanto da direita para a esquerda como da esquerda para a direita. Exemplos de palíndromo são: esse, mussum e osso. Utilizando não mais que as funções de inserção e remoção em listas e pilhas, implemente um algoritmo de reconhecimento de palíndromos de tamanho n, tal que n é par.

---

[2] O ponteiro pode ser do tipo node* ou node**.

[3] $1 \leq k < n$.

[4] Uma *struct* na linguagem C.

```c
#include <stdio.h>
#include "fila.h"
#include "pilha.h"
// Cria uma fila com capacidade para n elementos
//fila_cria(Fila* p, int n);
/* Cria uma pilha com capacidade para n elementos */
/* pilha_cria(Pilha* p, int n); */
// Insere o caracter c na pilha
// void pilha_insere(Pilha* p, char c);
/* Insere o caracter c na fila */
/* void fila_insere(Fila* p, char c); */
// Remove um caracter da pilha
//char pilha_remove(Pilha* p);
/* Remove um caracter da fila */
/* char fila_remove(Fila* p); */
int main(){
    int i,n;
    int palindromo;
    // Armazena a palavra de entrada
    char* entrada;
    // Declaracao das estruturas de dados
    Fila* f;
    Pilha* p;
    // Leitura da palavra de entrada
    scanf("%s",entrada);
    n = strlen(entrada);
    // Criacao das estruturas
    fila_cria(f,n);
    pilha_cria(p,n);
    for(i=0; i < n; i++)
      fila_insere(f,entrada[i]);

    /* Insira o seu codigo a partir deste ponto. Nao eh permitido fazer
   uso da variavel "entrada", nem mesmo declarar estruturas de dados
   auxiliares. Utilize apenas as estruturas fornecidas e suas funcoes.
   Implemente o seu algoritmo com complexidade O(n) */

    if(palindromo == 0)
        printf("%s nao eh palindromo \n",entrada);
    else
        printf("%s eh palindromo\n",entrada);
    return 0;
}
```

40. Dado um inteiro `x` e um vetor ordenado `int a[ ]` com n números inteiros distintos, descreva um algoritmo de complexidade assintótica $O(n)$ para determinar se existem índices `i` e `j` tais que `(a[i] + a[j] == x)`.

41. Escreva um algoritmo de complexidade assintótica $O(n)$ que, dados dois vetores de números ordenados `int a[ ]` e `int b[ ]`, cada um com n elementos, imprima todos os elementos

dos dois vetores de forma ordenada.

42. Modifique a solução do exercício 41, mantendo sua complexidade assintótica, de forma a garantir que somente os elementos presentes em ambos os vetores sejam impressos.

43. Descreva algoritmos polinomiais para cada um dos seguintes problemas:

    a) Quadrado perfeito

    b) Equação do segundo grau

44. Considere um conjunto contento n elementos, cada um com as cores vermelho, branco e azul. Nós queremos ordenar esse conjunto de elementos de tal forma que todos os vermelhos apareçam antes dos brancos e que todos os brancos apareçam antes dos azuis. Apresente um algoritmo de complexidade $O(n)$ para o problema de ordenação vermelho-branco-azul.

    Resposta:

```
/* Assumir que os elementos sao objetos */
#define vermelho 0
#define branco 1
#define azul 2
void ordena( elemento entrada[], elemento saida [], int n ){
   int i = 0;
    for ( int j=0; j < n; j++)
        if( entrada[j].cor == vermelho)
            saida[i++] = entrada[j];
    for ( int j=0; j < n; j++)
        if( entrada[j].cor == branco)
            saida[i++] = entrada[j];
    for ( int j=0; j < n; j++)
        if( entrada[j].cor == azul)
            saida[i++] = entrada[j];
}
```

45. Um problema muito comum em compiladores e em editores de texto é o de determinar se a parentização em uma cadeia de caracteres encontra-se balanceada e aninhada. Por exemplo, a cadeia ((())())() contém seus parênteses corretamente balanceados e aninhados, enquanto a cadeia)()(não se encontra aninhada e a cadeia ()) não se encontra balanceada. Apresente um algoritmo de complexidade $O(n)$, em que n é o tamanho da cadeia de parênteses, que retorne o valor booleano verdadeiro se a cadeia contém uma parentização que seja corretamente balanceada e aninhada e retorne o valor booleano falso caso contrário.

    Resposta:

```
bool check( vector<char> &v ){
    int contador = 0;
    for ( int i = 1; i < v.size(); i++ ){
        if ( v[i] == '(' )
            contador++;
        else if ( contador > 0 )
            contador--;
```

```
        else
            return false;
    }
    return (contador == 0);
}
```

46. Considere o seguinte pseudocódigo:

```
//Pseudocodigo o algoritmo ingenuo para
//casamento de cadeias de caracteres
NAIVE-STRING-MATCHER(texto T, padrao P){
   n = T.lenght;
   m = P.lenght;
   for s = 0 to n-m do
      if P[1 .. m] == T[s+1 .. s+m]
      for( int j = 1; j <=i; j++)
         print "Padrao encontrado em" s-m
}
```

a) Conte as comparações que o algoritmo NAIVE-STRING-MATCHER realiza com o parão $P = 0001$ dentro do texto $T = 000010001010001$.

b) Suponha que sabemos que todos os caracteres em um padrão P são diferentes. Mostre como acelerar o algoritmo NAIVE-STRING-MATCHER para executar com tempo $\mathrm{O}(n)$ em um texto de tamanho n.

47. Descreva um algoritmo que, dados n inteiros com valores entre 1 e k, pré-processe esses inteiros e então (descontado o tempo do pré-processamento) responda perguntas da forma "dados a, b, quantos dos n inteiros estão no intervalo $[a\ ,\ b]$" em tempo $\mathrm{O}(1)$. O pré-processamento deve ser feito em tempo $\mathrm{O}(n + k)$.

48. Descreva um algoritmo que, dados n inteiros com valores de 1 a 10, pré-processe esses inteiros e então responda a pergunta da seguinte forma: "dados três inteiros (i, j, x) tais que $1 \leq i \leq j \leq n$ e $1 \leq x \leq 10$, quantos dos $j - i + 1$ inteiros no intervalo $[i, j]$ têm valor igual a x?". O pré-processamento deve ser feito em tempo $O(n)$ e cada consulta deve ser respondida em tempo $O(1)$.

49. Realize a menor quantidade de modificações necessárias no procedimento INSERTION-SORT para que o mesmo seja capaz de ordenar elementos em ordem decrescente.

```
INSERTION-SORT(A)
   for j = 2 to A.length
      key = A[j]
      i = j - 1
      while i > 0 and A[i] > key
         A[i+1] = A[i]
         i = i - 1
      A[i+1] = key
}
```

50. Considere o seguinte problema de busca:

Entrada Uma sequencia de n números em um vetor $A = (a_1, a_2, ..., a_n)$ e um valor v.

Saida O índice i tal que $v = A[i]$ ou um valor especial $NULL$ caso v não esteja presente na sequência A.

Escreve um algoritmo de busca linear que varre a sequência a procura de v. Prove que o seu algoritmo está correto por meio de uma invariante. Certifique-se que a invariante atende às três propriedades fundamentais (inicialização, manutenção e terminação).

51. Considere o seguinte problema de adição de números binários.

    Entrada Dois números inteiros A e B representados na forma de vetores de tamanhos n onde cada uma de suas posições contém os bits 0 ou 1.

    Saida Um número C representado na forma de um vetor de tamanho $n+1$ que armazena a soma dos números A e B.

    Escreve um algoritmo linear que resolva este problema. Prove a sua complexidade.

52. Considere o problema de ordenação de n elementos armazenados em um vetor A. Primeiramente, encontramos o menor elemento de A e o trocamos de posição com o elemento $A[1]$. Em seguida, encontramos o segundo menor elemento de A e trocamos de posição com o elemento $A[2]$. Continuamos desta maneira para os primeiros $n-1$ elementos de A. Esta técnica de ordenação é conhecida como Selection Sort. Escreva um algoritmo para o Selection Sort. Qual a invariante que este algoritmo mantém? Por que precisamos executar este algoritmo apenas para os primeiros $n-1$ elementos? Apresente as análises de melhor e pior caso?

53. Podemos expressar o algoritmo Insertion Sort recursivamente da seguinte forma: Para ordenar o vetor $A[1, ..., n]$, recursivamente ordenamos o vetor $A[1, ..., n-1]$ e então inserimos o elemento $A[n]$ no vetor ordenado $A[1, .., n-1]$. Escreva a recorrência que calcula o tempo de execução para esta versão recursiva do Insertion Sort.

54. Observe que no problema 50, se a o vetor $A = (a_1, a_2, .., a_n)$ estiver ordenado, nós podemos comparar o ponto médio da sequência com o valor v e eliminar pelo menos cerca da metade dos elementos da sequência. O algoritmo chamado **busca binária** repete esse procedimento, dividindo pela metade o tamanho da sequência restante a cada iteração. Escreva dois algoritmos, iterativo e recursivo, para a busca binária. Argumente que o pior caso deste algoritmo é $O(\lg n)$.

55. Seja o elemento da posição k em um vetor de n números não ordenados. Apresente um algoritmo eficiente para determinar apenas a posição que esse elemento ocupará no vetor ordenado em ordem ascendente. Apresente a ordem de complexidade do seu algoritmo.

56. Um elemento majoritário em um vetor de inteiros de tamanho n é um inteiro que ocorre em pelo menos $n/2$ posições do vetor. Nem todos os vetores têm um elemento majoritário, mas o elemento deve ser claramente único se ele existe.

    a) Projete um algoritmo de custo linear que identifica o elemento majoritário, se ele existir.

    b) Assuma agora que o vetor não contenha inteiros, mas contenha outro tipo de dados que permita apenas testes de igualdade. Qual é a complexidade para determinar se o vetor contem um elemento majoritário?

57. Dada uma sequência (não necessariamente ordenada) $a[1], ..., a[n]$ de inteiros, queremos determinar se há dois índices i e j tais que $a[i] = 2 \times a[j]$. Descreva um algoritmo que resolva esse problema de forma eficiente.

58. Suponha que você deseja ordenar n números, sendo que cada um deles é 0 ou 1. Descreva um algoritmo eficiente para resolver este problema e apresente sua complexidade.

59. Discuta quais as situações que levam ao melhor caso e ao pior caso do Quicksort e determine a sua ordem de complexidade nesses casos. Para o cálculo da complexidade, determine e resolva a equação de recorrência correspondente a cada caso.

60. Suponha que seja dada uma sequencia S de n elementos, tal que cada elemento em S represente um voto distinto em uma eleição e cada voto seja representado por um número inteiro que representa a identificação do candidato escolhido. Suponha que você conheça o número k ($k < n$) de candidatos que concorrem às eleições. Descreva um algoritmo $O(n \lg k)$ que determine o vencedor das eleições.

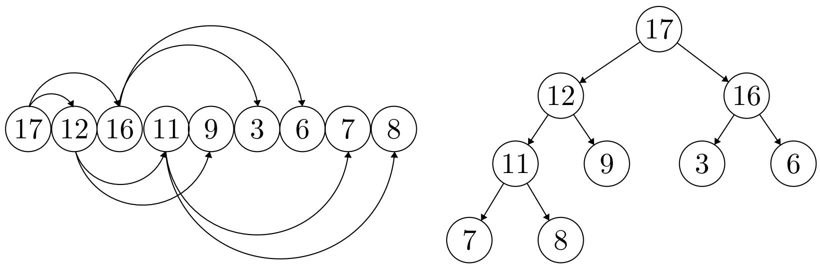
61. Suponha que você tenha 8 esferas do mesmo tamanho. 7 esferas possuem o mesmo peso e apenas 1 delas é ligeiramente mais leve. Como você poderá determinar a esfera mais leve, usando uma balança de pratos (veja a Figura 1) e apenas 2 medições?

Figura 1: Balança de pratos.

62. Dado um vetor com n números inteiros deseja-se reorganizar os elementos de forma que todos os números negativos precedam os não negativos e os zeros fiquem entre eles. Desenvolva um algoritmo que reorganize esse vetor com ordem de complexidade $\Theta(n)$. Forneça um exemplo para ilustrar seu algoritmo.

63. Apresente a visualização do *max-heap* correspondente ao vetor `(16, 7, 3, 9, 12, 6, 17, 8, 11)`, juntamente com a árvore binária associada, em que os elementos foram inseridos na ordem apresentada.

    Solução do Professor:



64. Escreva um algoritmo que decida se um vetor $v[1, ..., n]$ é ou não um heap. Discuta a complexidade deste algoritmo.

65. Escreva a função `CorrigeSubindo` que organiza um vetor `A[1..n]`, a partir do elemento na posição `n`, de modo a transforma-lo em um *max-heap*.

66. Suponha que $v[1,..,n]$ é um heap e i é um índice menor que $\frac{n}{2}$. É verdade que $v[2i] \geq v[2i+1]$? É verdade que $v[2i] \leq v[2i+1]$?

67. Escreva um programa que cronometre suas implementações do Quicksort e Heapsort (na linguagem C/C++ use a função clock da biblioteca time). Divida os tempos por $n \log n$ para comparar com a previsão teórica.

68. Considere a seguinte sequência de entrada. É solicitada a realização de uma classificação em ordem crescente sobre a sequência dada usando o algoritmo de ordenação Heapsort. Mostre como cada passo é executado.

1	2	3	4	5	6	7	8	9	10
26	34	9	0	4	89	6	15	27	44

Análise de Algoritmos

69. (Valor: 2 pontos)Apresente a complexidade dos seguintes algoritmos em função de n. Justifique sua resposta:

a)
```
sum = 0; \\ 1 operacao
for (int i = 0; i < n*n; i++) \\ N*N operacoes
    sum++; \\ 1 operacao
```

Temos $1 + 1 \times N^2$ operações, o que nos leva a um algoritmo com complexidade assintótica $O(n^2)$.

b)
```
sum = 0;
for (int i = 0; i < n; i++)
    for (int j = 0; j < n*n; j++)
        sum++;
```

c)
```
sum = 0;
for (int i = 1; i < n; i*=2)
    sum++;
```

d)
```
sum = 0;
for (int i = 0; i < n; i++)
    for (int j = n; j > 0; j/=2)
       sum++;
```

70. O que é um algoritmo polinomial?

71. Por muitas vezes damos atenção apenas ao pior caso dos algoritmos? Explique o porque.

72. Dos algoritmo de ordenação baseados em comparação mais utilizados, o Insertion sort possui complexidade $\Omega(n)$. Podemos dizer que nenhum outro algoritmo poderá atingir uma complexidade menor que esta? Justifique.

73. Determine a ordem de complexidade para cada função e justifique a sua resposta:

```
char* f(int n){
    char s[n];
    char x = 'x';
    for (int i = 0; i < n; i++)
        s[i] = x;
    return s;
}

char* g(int n){
    int i=-1;
    char s[n];
    char x = 'x';
    while (n != 0){
        if (n % 2 == 1)
            s[++i] = x;
        n = n/2;
    }
    return s;
}
```

74. Seja o seguinte código do algoritmo INSERTION-SORT.

```
INSERTION-SORT(A)
   for j = 2 to A.length
      key = A[j]
      i = j - 1
      while i > 0 and A[i] > key
         A[i+1] = A[i]
         i = i - 1
      A[i+1] = key
}
```

É possível analisar a corretude de um trecho de código por meio de uma invariante. Encontrar uma invariante que avalie de forma completa e adequada a corretude de um trecho de código pode ser uma tarefa difícil, mas uma vez bem definida, basta verificar se essa se mantém verdadeira antes, durante e após a execução do código.

A seguinte invariante foi definida para o procedimento INSERTION-SORT:

> Os elementos em $A[1, ..., j-1]$ estão ordenados.

Justifique a validade dessa invariante.

75. Prove que o seguinte algoritmo para determinar o valor máximo de um vetor com n elementos está correto.[5]

[5]Qual invariante este algoritmo mantém?

```
int max( int* a, int n ){
    int m = a[0];
    for ( int i = 1; i < n; i++ )
      if ( a[i] > m )
         m = a[i];
    return m;
}
```

Resposta:
Invariante: "m é maior ou igual a todo elemento em $a[0, ..., i-1]$"
Inicialização: Com $i = 1$, $a[0, ..., i-1]$ implica em $a[0]$. Como $m = a[0]$, a invariante é mantida.
Manutenção: Comparamos $a[i]$ com m. Caso $a[i] \leq m$, ao final do loop a invariante se mantém. Caso $a[i] > m$, o valor de m será atualizado para que a invariante seja mantida no final da execução do loop.
Terminação: Quando $i = n$, m é maior ou igual a qualquer elemento em $a[0, ..., n-1]$, portanto m determina o valor máximo do vetor com n elementos.

76. Dos algoritmo de ordenação baseados em comparação mais utilizados, o que possui menor complexidade é o inserção, considerando o seu melhor caso, $\Omega(n)$. Podemos dizer que nenhum outro algoritmo poderá atingir uma complexidade melhor do que esta? Justifique.

Divisão e conquista

77. Separe um bando de $2n$ Orcs em dois times com n Orcs cada. Cada Orc tem um número marcado em suas costas que indica o quão habilidoso o Orc é em combate. Divida o grupo da forma mais injusta possível para que o combate entre os dois times de Orcs seja extremamente sanguinário. Justifique sua escolha de divisão e explique como esta tarefa pode ser feita utilizando divisão e conquista em tempo $O(n \log n)$.

    Eu posso simplesmente executar a montagem de uma árvore binária simples, inserindo cada novo orc do grupo a partir de uma ordenação semelhante ao Merge Sort, cujo tempo é O(n log n) e utiliza da estratégia 'Dividir para Conquistar'. Após ordenado, basta selecionar a primeira metade da ordenação como time 1 (os mais fracos) e a segunda como time 2 (os mais fortes) e deixar o sangue jorrar.

78. O tamanho das instâncias de um certo problema é medido por um parâmetro n. Tenho três algoritmos — A, B e C — para o problema:

    - A divide cada instância do problema em cinco subinstâncias de tamanho [n/2], resolve as subinstâncias e então combina as soluções em tempo O(n)
    - B divide cada instância do problema em duas subinstâncias de tamanho n - 1, resolve as subinstâncias e então combina as soluções em tempo O(1)
    - C divide cada instância do problema em nove subinstâncias de tamanho [n/3], resolve as subinstâncias e então combina as soluções em tempo $O(n^2)$

    Qual o consumo de tempo de cada um dos algoritmos? Qual dos algoritmo é assintoticamente mais eficiente no pior caso?

79. Cormen et. al., Capítulo 4, Exercício 4.3-1

80. Cormen et. al., Capítulo 4, Exercício 4.3-2
81. Cormen et. al., Capítulo 4, Exercício 4.3-3
82. Cormen et. al., Capítulo 4, Exercício 4.3-6
83. Cormen et. al., Capítulo 4, Exercício 4.5-1
84. Cormen et. al., Capítulo 4, Exercício 4.5-3
85. Cormen et. al., Capítulo 4, Exercício 4.5-4
86. Cormen et. al., Capítulo 4, Exercício 4-1 (final do capítulo)
87. Cormen et. al., Capítulo 4, Exercício 4-3 (final do capítulo)

Teorema Mestre

Sejam as constantes $a \geq 1$ e $b > 1$, seja $f(n)$ uma função, e seja $T(n)$ definida nos inteiros não-negativos pela recorrência:

$$T(n) = aT(n/b) + f(n)$$

A equação de recorrência $T(n)$ pode ser limitada assintoticamente da seguinte forma:

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

(b) Se $f(n) = \Theta(n^{\log_b a} \lg^k n)$ onde $k \geq 0$, então $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$. Na versão simplificada, temos: $k = 0 \Rightarrow f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a} \lg n)$

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