Complexidade Assintótica

Prof. Jussara M. Almeida

Adaptado do material de Prof. Luiz Chaimowicz

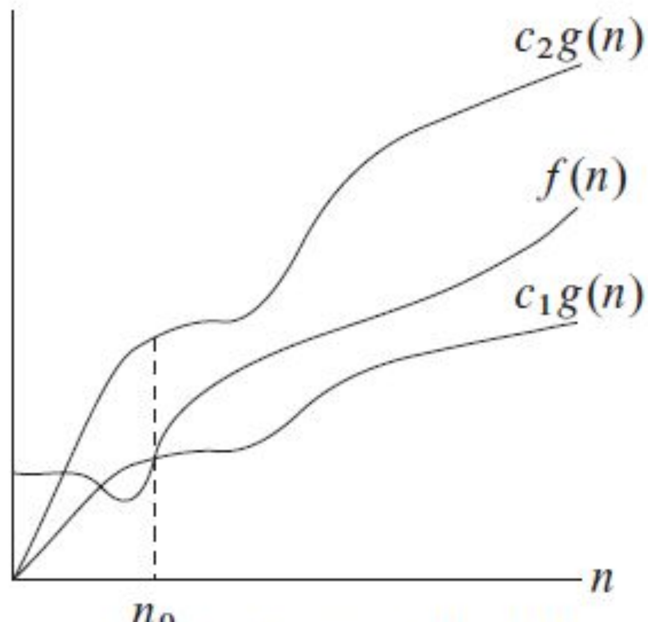
Complexidade Assintótica

- Na análise de algoritmos, na maioria das vezes usa-se o estudo da complexidade assintótica, ou seja, **analisa-se o algoritmo quando o valor de *n* tende a infinito**
- Nesse caso, não é necessário se preocupar com as constantes e termos de menor crescimento
- Usa-se notações especiais para representar a complexidade assintótica

Notação Θ

- Limite assintótico firme

$$\Theta(g(n)) = \{ f(n) : \text{ there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0 \} .^1$$



$f(n) = \Theta(g(n))$

Notação Θ

- Exemplo: $$\frac{1}{2}n^2 - 3n = \Theta(n^2)$$

- É necessário encontrar constantes c_1, c_2 e n_0 tais que:

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2 \;\; \forall n \geq n_0$$

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

O lado direito é satisfeito com $c_2 = 1/2$ para n maior ou igual a 1

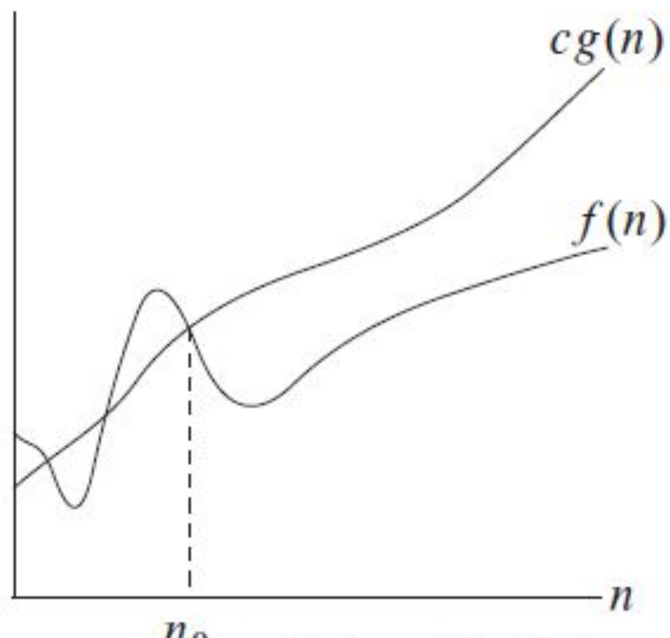
O lado esquerdo é satisfeito com $c_1 = 1/14$ para n maior ou igual a 7

Logo, a equação é satisfeita com $c_1 = 1/2$, $c_2 = 1/14$ e n_0 igual a 7

Notação O

- Limite assintótico superior

$$O(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \} .$$



$f(n) = O(g(n))$

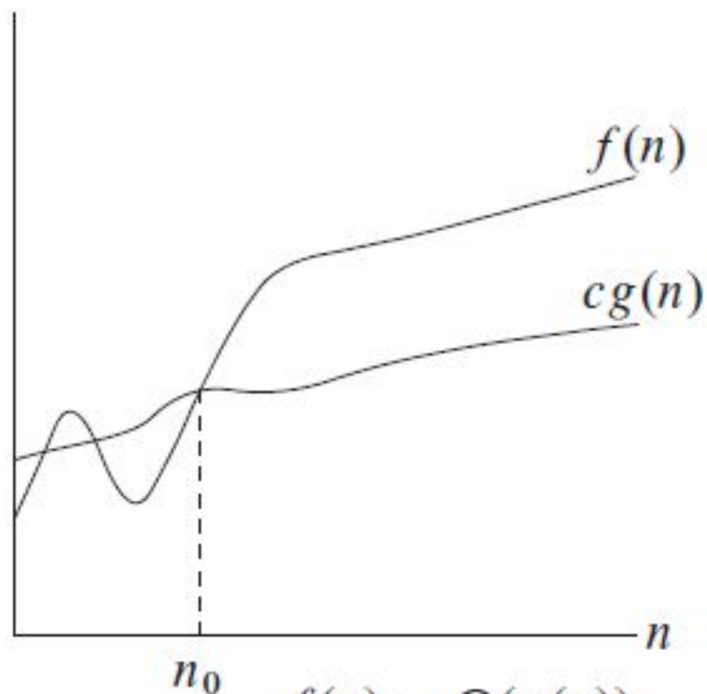
Notação O

- Apesar de muito usada na literatura como limite firme, a notação O é mais fraca que a notação Θ
  - **$f(n) = \Theta(g(n))$ implica em $f(n) = O(g(n))$**, mas não o contrário
  - $\Theta(g(n)) \subset O(g(n))$
- Como é um limite superior, é muito usado para o pior caso. Ex.
  - O inserção é $O(n^2)$, para qualquer entrada.

Notação Ω

- Limite assintótico inferior

$$\Omega(g(n)) = \{ f(n) : \text{ there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0 \} .$$



$f(n) = \Omega(g(n))$

Notação Ω

- Também mais fraca que Θ
- Limite inferior, portanto pode ser usado para o melhor caso
- Inserção é Ω(n), para qualquer entrada

- **Teorema:**

  Para quaisquer duas funções $f(n)$ e $g(n)$, teremos $f(n) = \Theta(g(n))$ se e somente se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$

Notações *o* e *ω*

- Representam, respectivamente limites superiores e inferiores estritos
- Notação *o:*

$$o(g(n)) = \{ f(n) : \text{ for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \le f(n) < cg(n) \text{ for all } n \ge n_0 \} .$$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

- Notação *ω:*

$$\omega(g(n)) = \{ f(n) : \text{ for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \le cg(n) < f(n) \text{ for all } n \ge n_0 \} .$$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

Propriedades

- Transitividade

$$
\begin{array}{lllll}
f(n) = \Theta(g(n)) & \text{and} & g(n) = \Theta(h(n)) & \text{imply} & f(n) = \Theta(h(n)) \\
f(n) = O(g(n)) & \text{and} & g(n) = O(h(n)) & \text{imply} & f(n) = O(h(n)) \\
f(n) = \Omega(g(n)) & \text{and} & g(n) = \Omega(h(n)) & \text{imply} & f(n) = \Omega(h(n)) \\
f(n) = o(g(n)) & \text{and} & g(n) = o(h(n)) & \text{imply} & f(n) = o(h(n)) \\
f(n) = \omega(g(n)) & \text{and} & g(n) = \omega(h(n)) & \text{imply} & f(n) = \omega(h(n))
\end{array}
$$

- Reflexividade

$$
\begin{aligned}
f(n) &= \Theta(f(n)) \\
f(n) &= O(f(n)) \\
f(n) &= \Omega(f(n))
\end{aligned}
$$

Propriedades

- Simetria

$$f(n) = \Theta(g(n)) \text{ if and only if } g(n) = \Theta(f(n))$$

- Simetria transposta

$$f(n) = O(g(n)) \text{ if and only if } g(n) = \Omega(f(n))$$
$$f(n) = o(g(n)) \text{ if and only if } g(n) = \omega(f(n))$$

Propriedades

- Transitividade:
  - se f = O(g) e g = O(h), então f = O(h)
  - se f = Ω(g) e g = Ω(h), então f = Ω(h)
  - Se f = Θ (g) e g = Θ(h), então f =Θ(h)
- Seja f, a função de complexidade de um algoritmo, um polinômio de grau d. Então f = O(n^d)
- Para todo b > 1 e x > 0: $log_b n = O(n^x)$
- Para todo a> 1 e b > 1: $log_a n$ = Θ($log_b n$)
- Para todo r > 1 e todo d > 0: $n^d = O(r^n)$
- Para r > s > 1, $r^n \neq \Theta(s^n)$

Questões

1. Como comparar duas funções assintoticamente?

$$\lim_{n \to \infty} \frac{f(n)}{g(n)}$$

- Se lim → ∞: f(n) é assintoticamente maior que g(n)
- Se lim → 0: g(n) é assintoticamente maior que f(n)
- Se lim = c: f(n) e g(n) são equivalentes assintoticamente

2. Uma função f(n) = O(n^4) é assintoticamente $\Theta(n^4)$?

- Não, a notação O define apenas um limite superior, enquanto a notação Θ define limites superior e inferior. Por exemplo: f(n) = n^2 = O(n^4) mas f(n) $\neq$ $\Theta(n^4)$

3. Se uma função f < g, é possível que g = O(f)?

- Sim, em análise assintótica, a comparação é entre classes de funções.
- Sejam f(n) = n-1 e g(n) = 2n. Temos que f(n) < g(n)
- Mas f(n) = O(n) e g(n) = O(n); então g(n) = O(f(n))

Questões

4. Qual função cresce mais lentamente: log(n) ou n^x, onde x é um número muito pequeno?
   - Qualquer polinomial tem uma taxa de crescimento mais rapida que qualquer logaritmo.

5. Qual função domina assintoticamente: f(n) = $log_2 n$ ou g(n) = $log_4 n$?
   - São assintoticamente equivalentes, isto é $f(n) = \Theta\big(g(n)\big)$

$$\log_a n = \frac{\log_b n}{\log_b a} = Constant * \log_b n$$

6. $2^{2n} = O(2^n)$?
   - 2^{2n}=$(2^2)^n = 4^n$
   - Se SIM, então devem existir constantes c e m tal que $2^{2n} \leq c2^n$ para n $\geq m$. Não existem tais constantes
   - Logo, a afirmativa é FALSA. 2^{2n} = $O(4^n)$
   - Mensagem: diferentemente do logaritmo, a base da exponencial importa!

Classes de Comportamento Assintótico

- Em geral, é interessante agrupar os algoritmos / problemas em **Classes de Comportamento Assintótico**, que vão determinar a complexidade inerente do algoritmo

- Como explicado, o comportamento assintótico é medido quando o tamanho da entrada (n) tende a infinito, com isso, as constantes são ignoradas e apenas o componente mais significativo da função de complexidade é considerado

- **Quando dois algoritmos fazem parte da mesma classe de comportamento assintótico, eles são ditos equivalentes**. Nesse caso, para escolher um deles deve-se analisar mais cuidadosamente a função de complexidade ou o seu desempenho em sistemas reais

Principais Classes de Problemas

f(n) = O(1)

– Algoritmos de complexidade $O(1)$ são ditos de **complexidade constante**.

– Uso do algoritmo independe de *n*.

– As instruções do algoritmo são executadas um número fixo de vezes.

Principais Classes de Problemas

f(n) = O(log n).

– Um algoritmo de complexidade $O(\log n)$ é dito ter **complexidade logarítmica**.

– Típico em algoritmos que transformam um problema em outros menores.

– Pode-se considerar o tempo de execução como menor que uma constante grande.

– Quando n é mil, $\log_2 n \approx 10$, quando n é 1 milhão, $\log_2 n \approx 20$.

– Para dobrar o valor de $\log n$ temos de considerar o quadrado de n.

– A base do logaritmo muda pouco estes valores: quando n é 1 milhão, o $\log_2 n$ é 20 e o $\log_{10} n$ é 6.

Principais Classes de Problemas

f(n) = O(n)

– Um algoritmo de complexidade $O(n)$ é dito ter **complexidade linear**.

– Em geral, um pequeno trabalho é realizado sobre cada elemento de entrada.

– É a melhor situação possível para um algoritmo que tem de processar/produzir n elementos de entrada/saída.

– Cada vez que n dobra de tamanho, o tempo de execução dobra.

Principais Classes de Problemas

f(n) = O(n log n)

– Típico em algoritmos que quebram um problema em outros menores, resolvem cada um deles independentemente e ajuntando as soluções depois.

– Quando n é 1 milhão, $n\log_2 n$ é cerca de 20 milhões.

– Quando n é 2 milhões, $n\log_2 n$ é cerca de 42 milhões, pouco mais do que o dobro.

Principais Classes de Problemas

f(n) = O(n²)

– Um algoritmo de complexidade $O(n^2)$ é dito ter **complexidade quadrática**.

– Ocorrem quando os itens de dados são processados aos pares, muitas vezes em um anel dentro de outro.

– Quando n é mil, o número de operações é da ordem de 1 milhão.

– Sempre que n dobra, o tempo de execução é multiplicado por 4.

– Úteis para resolver problemas de tamanhos relativamente pequenos.

Principais Classes de Problemas

$f(n) = O(n^3)$

– Um algoritmo de complexidade $O(n^3)$ é dito ter **complexidade cúbica**.

– Úteis apenas para resolver pequenos problemas.

– Quando n é 100, o número de operações é da ordem de 1 milhão.

– Sempre que n dobra, o tempo de execução fica multiplicado por 8.

Principais Classes de Problemas

f(n) = O(2^n)

– Um algoritmo de complexidade $O(2^n)$ é dito ter **complexidade exponencial**.

– Geralmente não são úteis sob o ponto de vista prático.

– Ocorrem na solução de problemas quando se usa **força bruta** para resolvê-los.

– Quando n é 20, o tempo de execução é cerca de 1 milhão. Quando n dobra, o tempo fica elevado ao quadrado.

Principais Classes de Problemas

f(n) = O(n!)

– Um algoritmo de complexidade $O(n!)$ é dito ter complexidade exponencial, apesar de $O(n!)$ ter comportamento muito pior do que $O(2^n)$.

– Geralmente ocorrem quando se usa **força bruta** para na solução do problema.

– n = 20 => 20! = 2432902008176640000, um número com 19 dígitos.

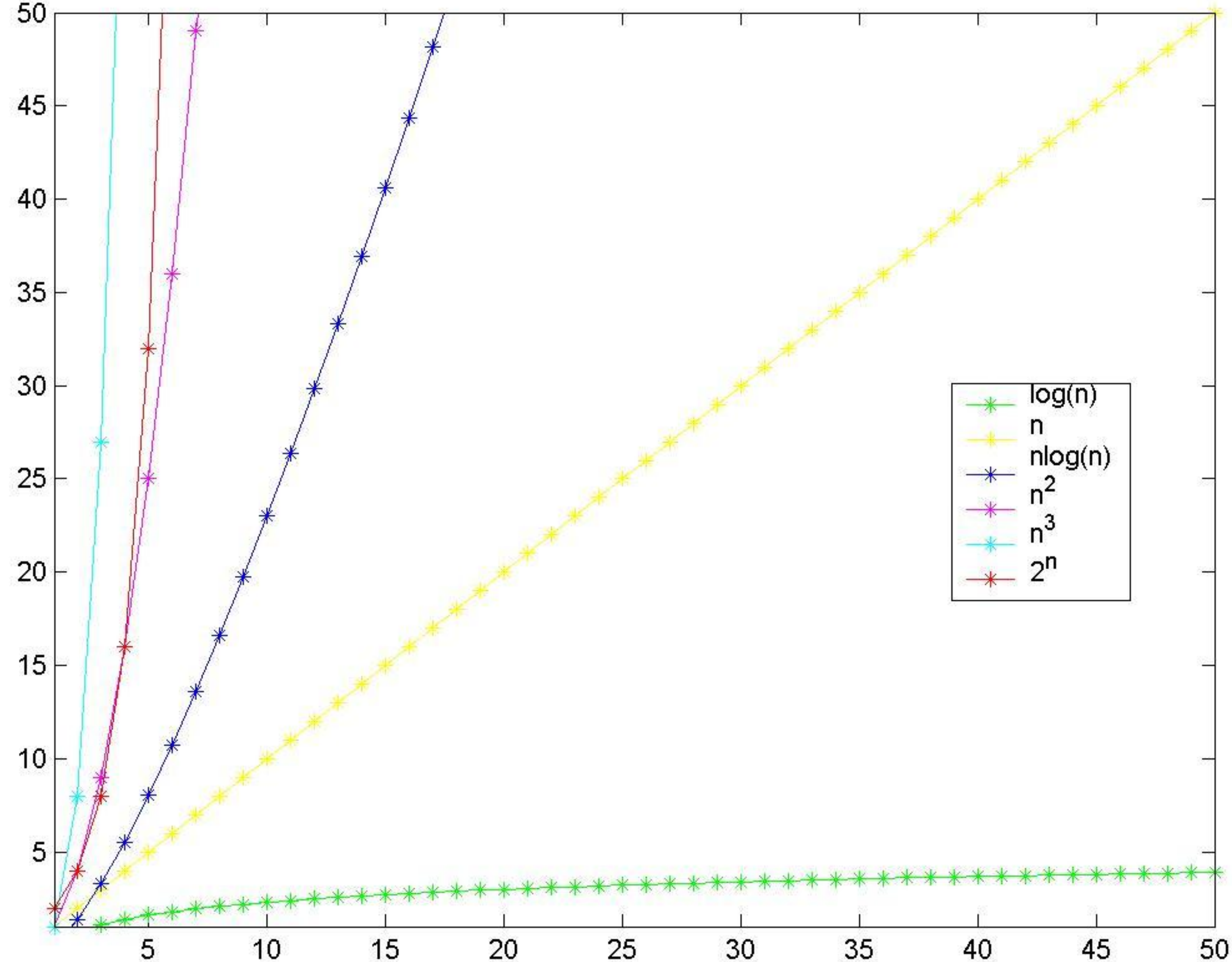
– n = 40 => um número com 48 dígitos.

Comparação de Funções de Complexidade

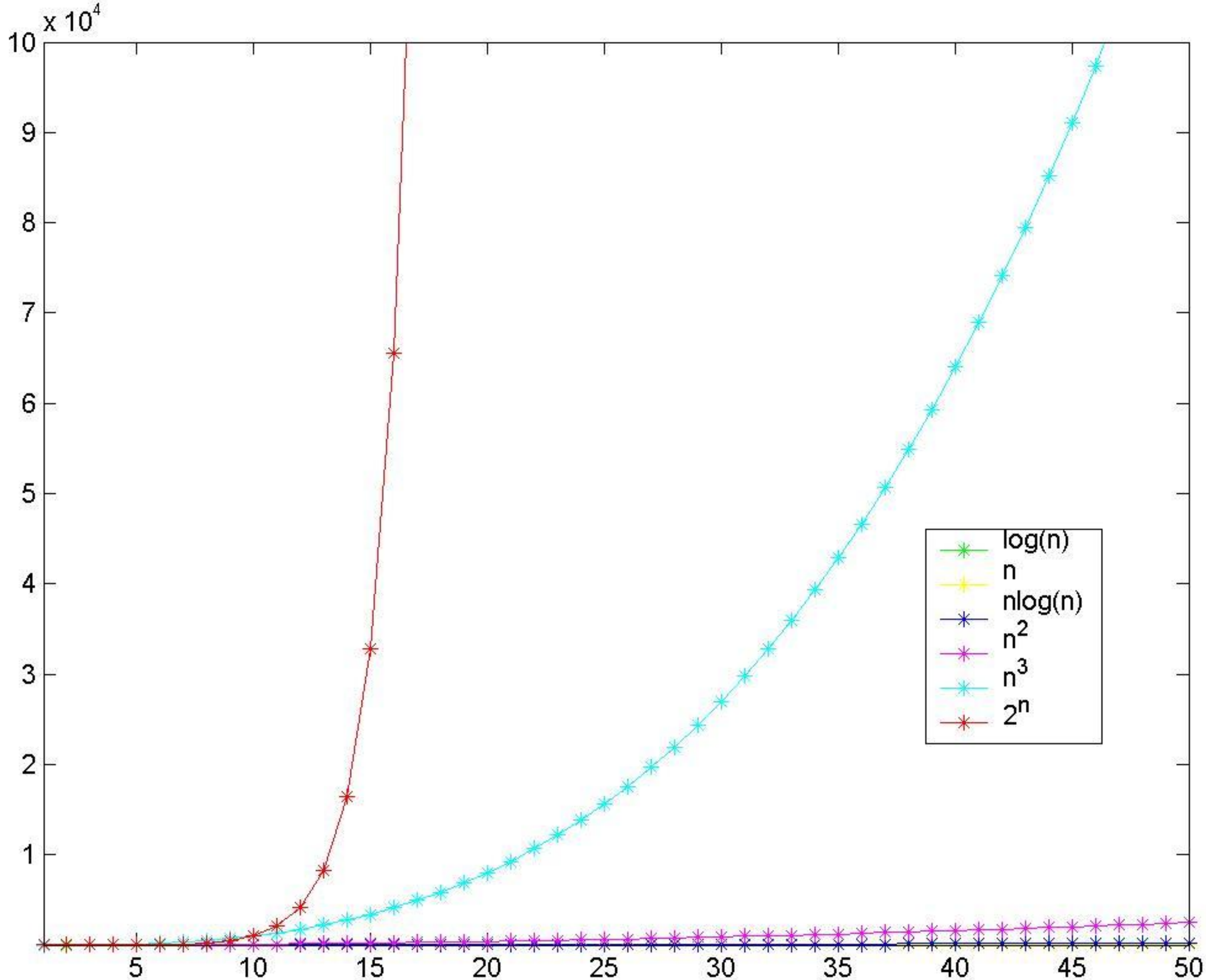
Função de custo	Tamanho n					
	10	20	30	40	50	60
n	0,00001 s	0,00002 s	0,00003 s	0,00004 s	0,00005 s	0,00006 s
n^2	0,0001 s	0,0004 s	0,0009 s	0,0016 s	0,0.35 s	0,0036 s
n^3	0,001 s	0,008 s	0,027 s	0,64 s	0,125 s	0.316 s
n^5	0,1 s	3,2 s	24,3 s	1,7 min	5,2 min	13 min
2^n	0,001 s	1 s	17,9 min	12,7 dias	35,7 anos	366 séc.
3^n	0,059 s	58 min	6,5 anos	3855 séc.	10^8 séc.	10^{13} séc.

Comparação de Funções de Complexidade

Função de custo de tempo	Computador atual	Computador 100 vezes mais rápido	Computador 1.000 vezes mais rápido
n	t_1	$100\ t_1$	$1000\ t_1$
n^2	t_2	$10\ t_2$	$31,6\ t_2$
n^3	t_3	$4,6\ t_3$	$10\ t_3$
2^n	t_4	$t_4 + 6,6$	$t_4 + 10$



log(n)
n
nlog(n)
n^2
n^3
2^n
50
45
40
35
30
25
20
15
10
5
5
10
15
20
25
30
35
40
45
50


x 10^4
10
9
8
7
6
5
4
3
2
1
5
10
15
20
25
30
35
40
45
50
log(n)
n
nlog(n)
n^2
n^3
2^n

Exercício

Ordene as seguintes funções pela complexidade assintótica:

- 10^n; $n^{1/3}$; n^n; $log_2 n$; $log_{1000} n$; $n^{10^{-1000}}$; $2^{(log_2 n)^2}$

Técnicas de Análise de Algoritmos

- Determinar o tempo de execução de um programa pode ser um problema matemático complexo;
- Determinar a ordem do tempo de execução, sem preocupação com o valor da constante envolvida, pode ser uma tarefa mais simples.
- A análise utiliza técnicas de matemática discreta, envolvendo contagem ou enumeração dos elementos de um conjunto:
  - manipulação de somas,
  - produtos,
  - permutações,
  - fatoriais,
  - coeficientes binomiais,
  - solução de **equações de recorrência**.

Análise do Tempo de Execução

- Comando de atribuição, de leitura ou de escrita: $O(1)$.
- Sequência de comandos: determinado pelo maior tempo de execução de qualquer comando da sequência.
- Comando de decisão: tempo dos comandos dentro do comando condicional, mais tempo para avaliar a condição, que é $O(1)$.
- Anel: soma do tempo do corpo do anel mais o tempo de avaliar a condição para terminação (geralmente $O(1)$), multiplicado pelo número de iterações.
- **Procedimentos não recursivos**: cada um deve ser computado separadamente um a um, iniciando com os que não chamam outros procedimentos. Avalia-se então os que chamam os já avaliados (utilizando os tempos desses). O processo é repetido até chegar no programa principal.
- **Procedimentos recursivos**: associada uma função de complexidade $f(n)$ desconhecida, onde n mede o tamanho dos argumentos.

Procedimento não Recursivo

Algoritmo para ordenar os n elementos de um conjunto A em ordem ascendente.

```
void Ordena(TipoVetor A)
{ /*ordena o vetor A em ordem ascendente*/
  int i, j, min,x;
  for (i = 1; i < n; i++)
    { min = i;
      for (j = i + 1; j <= n; j++)
      if ( A[j − 1] < A[min − 1] ) min = j;
      /*troca A[min] e A[i]*/
      x = A[min − 1];
      A[min − 1] = A[i − 1];
      A[i − 1] = x;
    }
}
```

- Seleciona o menor elemento do conjunto.
- Troca este com o primeiro elemento $A[1]$.
- Repita as duas operações acima com os $n-1$ elementos restantes, depois com os $n-2$, até que reste apenas um.

Análise do Procedimento não Recursivo

Anel Interno

- Contém um comando de decisão, com um comando apenas de atribuição. Ambos levam tempo constante para serem executados.
- Quanto ao corpo do comando de decisão, devemos considerar o pior caso, assumindo que serSS sempre executado.
- O tempo para incrementar o índice do anel e avaliar sua condição de terminação é $O(1)$.
- O tempo combinado para executar uma vez o anel é $O(max(1, 1, 1)) = O(1)$, conforme regra da soma para a notação O.
- Como o número de iterações é $n - i$, o tempo gasto no anel é $O((n - i) \times 1) = O(n - i)$, conforme regra do produto para a notação O.

Análise do Procedimento não Recursivo

Anel Externo

- Contém, além do anel interno, quatro comandos de atribuição. $O(max(1, (n-i), 1, 1, 1)) = O(n-i)$.
- A linha (1) é executada $n-1$ vezes, e o tempo total para executar o programa está limitado ao produto de uma constante pelo **somatório** de $(n-i)$: $\sum_1^{n-1}(n-i) = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} = O(n^2)$
- Considerarmos o número de comparações como a medida de custo relevante, o programa faz $(n^2)/2 - n/2$ comparações para ordenar n elementos.
- Considerarmos o número de trocas, o programa realiza exatamente $n-1$ trocas.