INF 1010
Estruturas de Dados Avançadas

Complexidade

Complexidade Computacional

Termos criado por Juris Hartmanis e Richard Stearns (1965).

Relação entre o tamanho do problema e seu consumo de tempo e espaço durante execução.

Objetivo

Por que analisar a complexidade dos algoritmos?

- A preocupação com a complexidade de algoritmos é fundamental para projetar algoritmos eficientes.
- Podemos desenvolver um algoritmo e depois analisar a sua complexidade para verificar a sua eficiência.
- Mas o melhor ainda é ter a preocupação de projetar algoritmos eficientes desde a sua concepção.

Eficiência ou complexidade de algoritmos

Seja n um parâmetro que caracteriza o tamanho da entrada de um algoritmo.

Considere, por exemplo, o problema de ordenar n números ou multiplicar duas matrizes quadradas $n \times n$ (cada uma com n^2 elementos).

Considere dois algoritmos que resolvem o problema.

Como podemos comparar os dois algoritmos para escolher o melhor?

Complexidade de tempo ou de espaço

Precisamos definir alguma medida que expresse a eficiência. Costuma-se medir um algoritmo em termos de tempo de execução ou de espaço (ou memória) usado.

Complexidade Espacial: Quantidade de recursos utilizados para resolver o problema;

Complexidade Temporal: Quantidade de tempo utilizado. Pode ser visto também como o número de instruções necessárias para resolver determinado problema;

Em ambos os casos, a complexidade é medida de acordo com o tamanho dos dados de entrada (n).

Complexidade
medida com relação ao tamanho *n* da entrada

espacial

- recursos (memória) necessários

temporal

- tempo utilizado
- número de instruções necessárias
- perspectivas:
  - pior caso
  - caso médio
  - melhor caso

Tempo de execução

Fornece um indicativo de como o algoritmo se comporta;

Conforme a arquitetura utilizada, este tempo varia;

Necessário testar com diversos casos a fim de compreender o comportamento de um algoritmo.

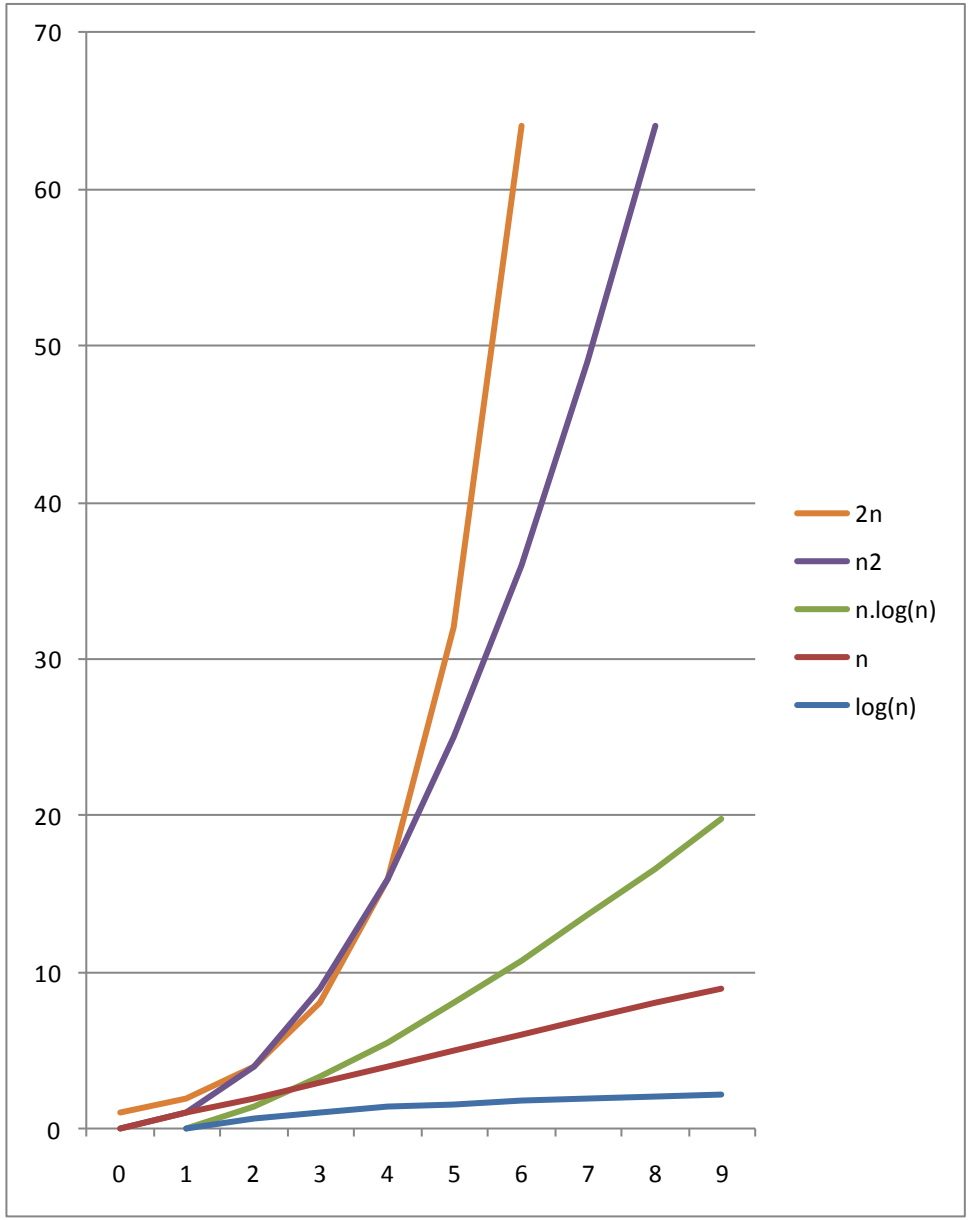
Tempo de execução – como calcular

```
float soma (float valores[], int n)
{
  int i;
  float somatemp = 0;
  for (i=0; i < n; i++)
    somatemp += valores[i];
  return somatemp;
}
```

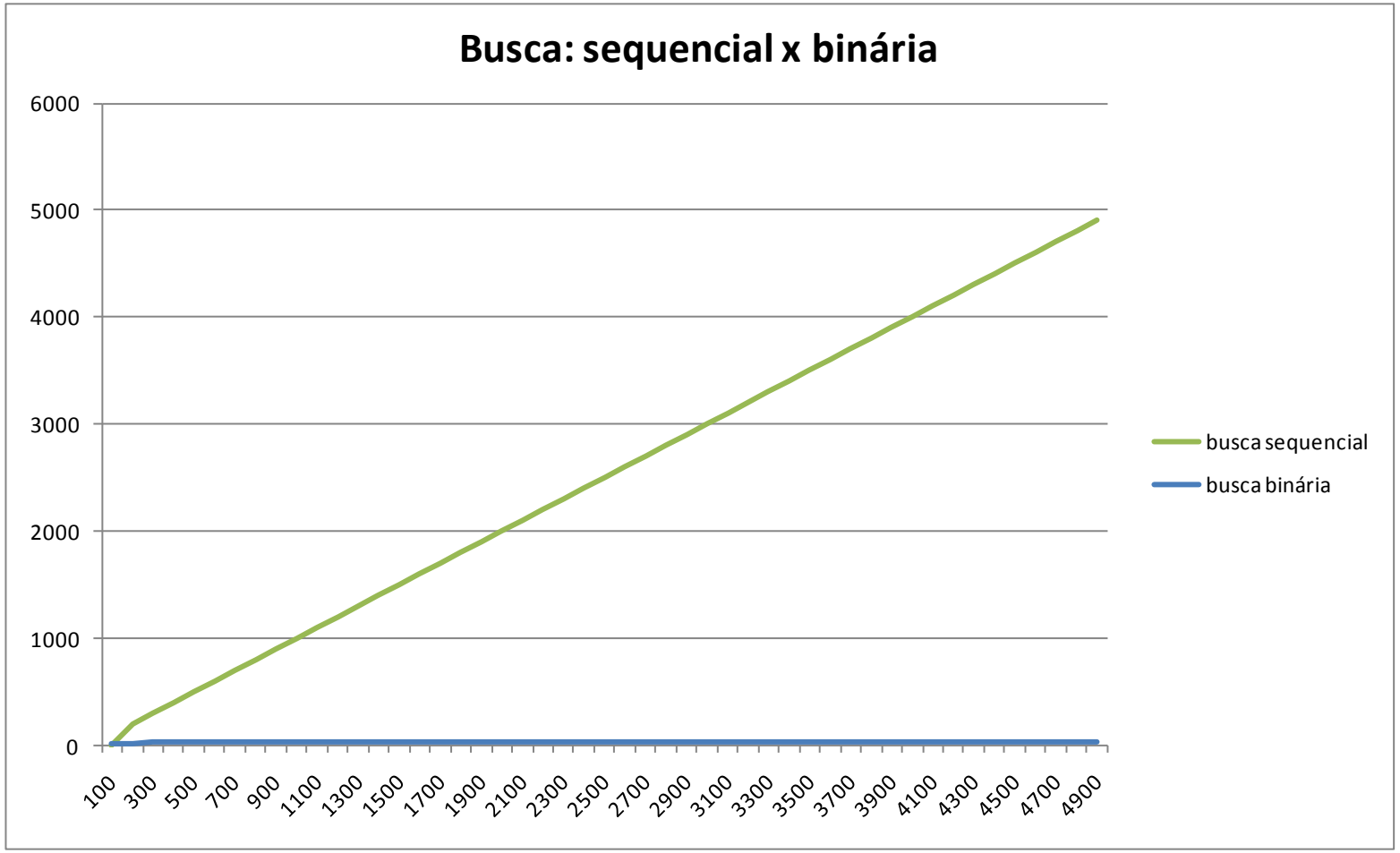
```
/* contando tempo */
#include <time.h>
double tempo;
float soma (float valores[], int n)
{
  int i;
  float somatemp = 0;
  clock_t tinicio = clock();
  for (i=0; i < n; i++)
  {
    somatemp += valores[i];
  }
  tempo=((double)clock()-tinicio)/CLOCK_PER_SEC;
  return somatemp;
}
```

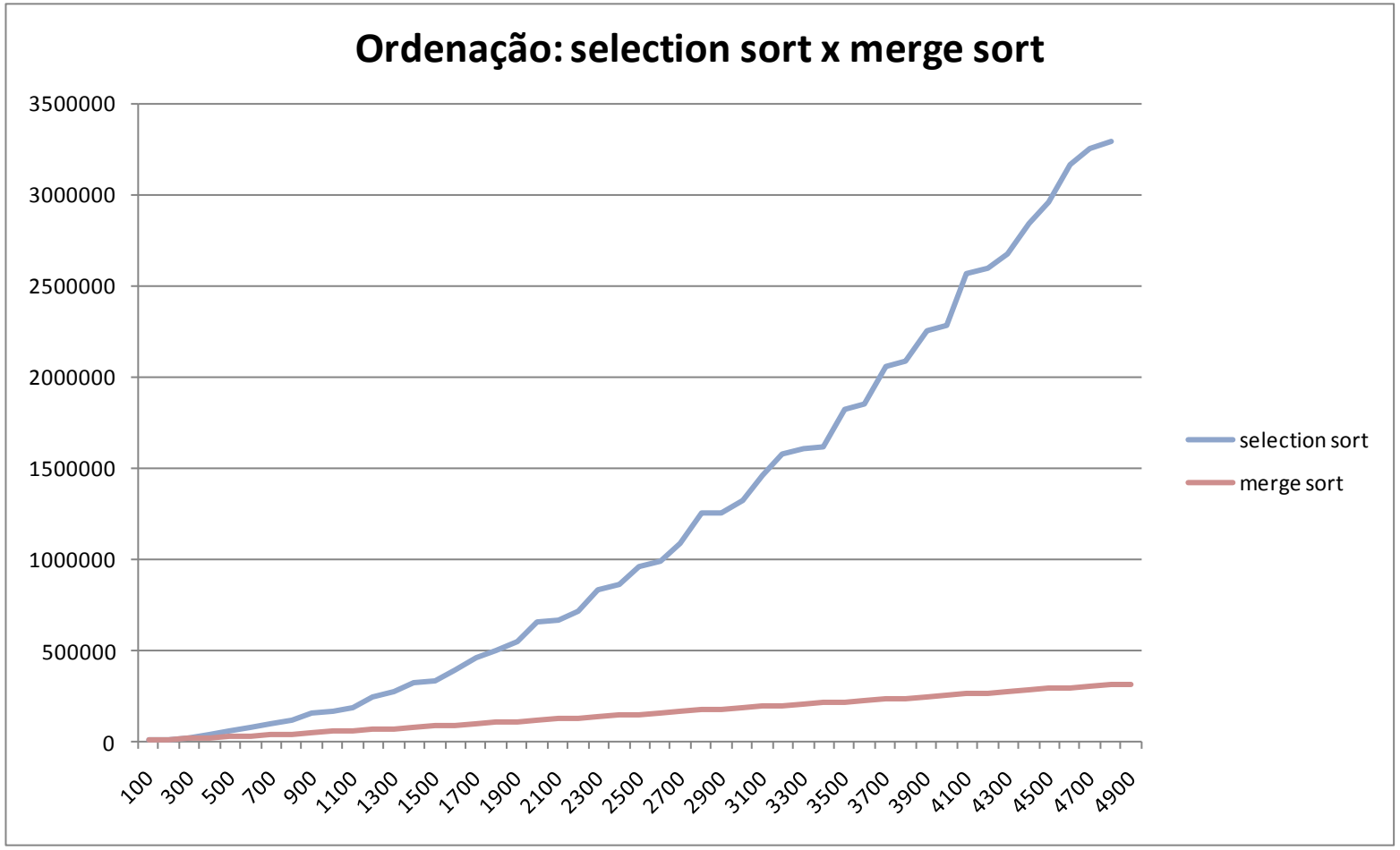
```
/* contando número de passos,
   considerando apenas atribuição
   e retorno de valores */

int count = 0;

float soma (float valores[], int n)
{
  int i;
  float somatemp = 0;
  count++; /* atribuição somatemp */
  for (i=0; i < n; i++)
  {
    count++; /* incremento for */
    count++; /* atrib somatemp */
    somatemp += valores[i];
  }
  count++; /* último incr. for */
  count++; /* return */
  return somatemp;
}
```

Por que isso importa?



log(n)	n	n.log(n)	n^2	2^n
0	1	0	1	2
0,69	2	1,39	4	4
1,10	3	3,30	9	8
1,39	4	5,55	16	16
1,61	5	8,05	25	32
1,79	6	10,75	36	64
1,95	7	13,62	49	128
2,08	8	16,64	64	256
2,20	9	19,78	81	512
2,30	10	23,03	100	1024
2,40	11	26,38	121	2048
2,48	12	29,82	144	4096
2,56	13	33,34	169	8192
2,64	14	36,95	196	16384
2,71	15	40,62	225	32768
2,77	16	44,36	256	65536
2,83	17	48,16	289	131072
2,89	18	52,03	324	262144
2,94	19	55,94	361	524288
3,00	20	59,91	400	1048576
3,04	21	63,93	441	2097152
3,09	22	68,00	484	4194304
3,14	23	72,12	529	8388608
3,18	24	76,27	576	16777216
3,22	25	80,47	625	33554432
3,26	26	84,71	676	67108864
3,30	27	88,99	729	1,34E+08
3,33	28	93,30	784	2,68E+08
3,37	29	97,65	841	5,37E+08
3,40	30	102,04	900	1,07E+09
3,43	31	106,45	961	2,15E+09
3,47	32	110,90	1024	4,29E+09
3,50	33	115,38	1089	8,59E+09
3,53	34	119,90	1156	1,72E+10
3,56	35	124,44	1225	3,44E+10
3,58	36	129,01	1296	6,87E+10
3,61	37	133,60	1369	1,37E+11
3,64	38	138,23	1444	2,75E+11
3,66	39	142,88	1521	5,5E+11
3,69	40	147,56	1600	1,1E+12
3,71	41	152,26	1681	2,2E+12
3,74	42	156,98	1764	4,4E+12
3,76	43	161,73	1849	8,8E+12
3,78	44	166,50	1936	1,76E+13
3,81	45	171,30	2025	3,52E+13
3,83	46	176,12	2116	7,04E+13
3,85	47	180,96	2209	1,41E+14
3,87	48	185,82	2304	2,81E+14
3,89	49	190,70	2401	5,63E+14
3,91	50	195,60	2500	1,13E+15


Busca: sequencial x binária
6000
5000
4000
3000
2000
1000
0
100
300
500
700
900
1100
1300
1500
1700
1900
2100
2300
2500
2700
2900
3100
3300
3500
3700
3900
4100
4300
4500
4700
4900
busca sequencial
busca binária


Ordenação: selection sort x merge sort
3500000
3000000
2500000
2000000
1500000
1000000
500000
0
100
300
500
700
900
1100
1300
1500
1700
1900
2100
2300
2500
2700
2900
3100
3300
3500
3700
3900
4100
4300
4500
4700
4900
selection sort
merge sort

Complexidade de Algoritmos

Melhor Caso (Ω - ômega)

É o menor tempo de execução em uma entrada de tamanho *n*.

É pouco usado, por ter aplicação em poucos casos.

Exemplo:

Se tivermos uma lista de *n* números e quisermos encontrar algum deles, assume-se que a complexidade no melhor caso é **f(n) = Ω(1)**, pois assume-se que o número estaria logo na topo da lista.

Complexidade de Algoritmos

Pior Caso (O - ômicron)

É o mais fácil de se obter. Baseia-se no maior tempo de execução sobre todas as entradas de tamanho n.

Exemplo:

Se tivermos uma lista de n números e quisermos encontrar algum deles, assume-se que a complexidade no pior caso é **O(n)**, pois assume-se que o número estaria, no pior caso, no final da lista.

Complexidade de Algoritmos

Caso Médio (θ - theta)

Dos três, é o mais difícil de se determinar

Deve-se obter a média dos tempos de execução de todas as entradas de tamanho N, ou baseado em probabilidade de determinada condição ocorrer.

No exemplo anterior:

A complexidade média é P(1) + P(2) + ... + P(n)

Dado que Pi = i/n; 1 ≤ i ≤ n

Segue que P(1) + P(2) + ... + P(n) = 1/n + 2/n + ... + 1 =

$$\frac{1}{n}(1 + 2 + \ldots + n) = \frac{1}{n}\left(\frac{n(n+1)}{2}\right)$$

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

Exemplo

Considere o número de operações de cada um dos dois algoritmos que resolvem o mesmo problema, como função de n.

Algoritmo 1: $f_1(n) = 2n^2 + 5n$ operações

Algoritmo 2: $f_2(n) = 50n + 4000$ operações

Dependendo do valor de n, o Algoritmo 1 pode requerer mais ou menos operações que o Algoritmo 2.

n = 10
$f_1(10) = 2(10)^2 + 5*10 = 250$
$f_2(10) = 50*10 + 4000 = 4500$

n = 100
$f_1(100) = 2(100)^2 + 5*100 = 20500$
$f_2(100) = 50*100 + 4000 = 9000$

Comportamento assintótico

Algoritmo 1: $f_1(n) = 2n^2 + 5n$ operações

Algoritmo 2: $f_2(n) = 500n + 4000$ operações

Um caso de particular interesse é quando n tem valor muito grande ($n \rightarrow \infty$), denominado comportamento assintótico.

Os termos inferiores e as constantes multiplicativas contribuem pouco na comparação e podem ser descartados.

O importante é observar que $f_1(n)$ cresce com n^2, ao passo que $f_2(n)$ cresce com n. Um crescimento quadrático é considerado pior que um crescimento linear. Assim, vamos preferir o Algoritmo 2 ao Algoritmo 1.

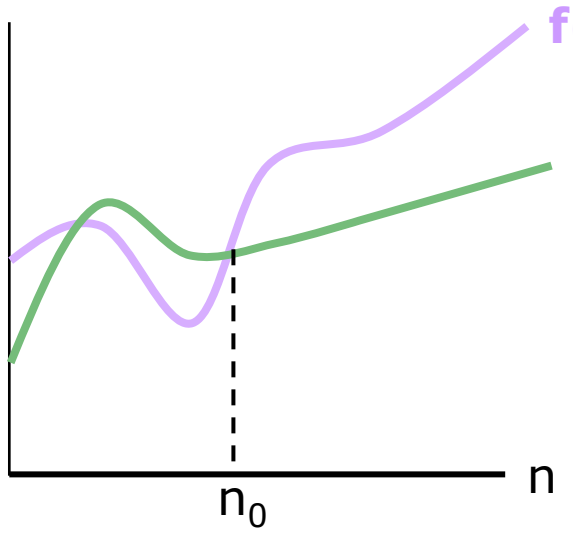
A notação Ω

Sejam f e g duas funções de domínio X.

Dizemos que a função f é **Ω(g(n))** se,

$$\exists c \in \Re^{+}, \exists n_0 \in X : c|g(n)| \leq |f(n)|, \forall n \geq n_0$$

Assim, a notação **Ω** dá-nos um **limite inferior** assintótico.



f(n) = Ω(g(n))

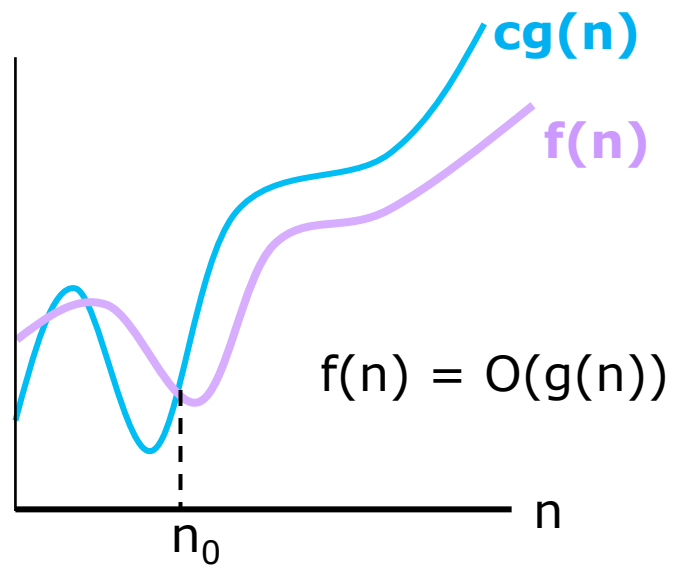
A notação O

Sejam f e g duas funções de domínio X.

Dizemos que a função f é O(g(n)) se,

$$\exists c \in \Re^+, \exists n_0 \in X : |f(n)| \leq c|g(n)|, \forall n \geq n_0$$

Assim, a notação **O** nos dá um **limite superior** assintótico.



Exemplos:

$3n + 2 = O(n)$, pois
$3n + 2 \leq 4n$ para todo $n \geq 2$

$1000n^2 + 100n - 6 = O(n^2)$, pois
$1000n^2 + 100n - 6 \leq 1001n^2$ para $n \geq 100$

$f(n) = a_m n^m + \ldots + a_1 n + a_0 \Rightarrow f(n) = O(n^m)$

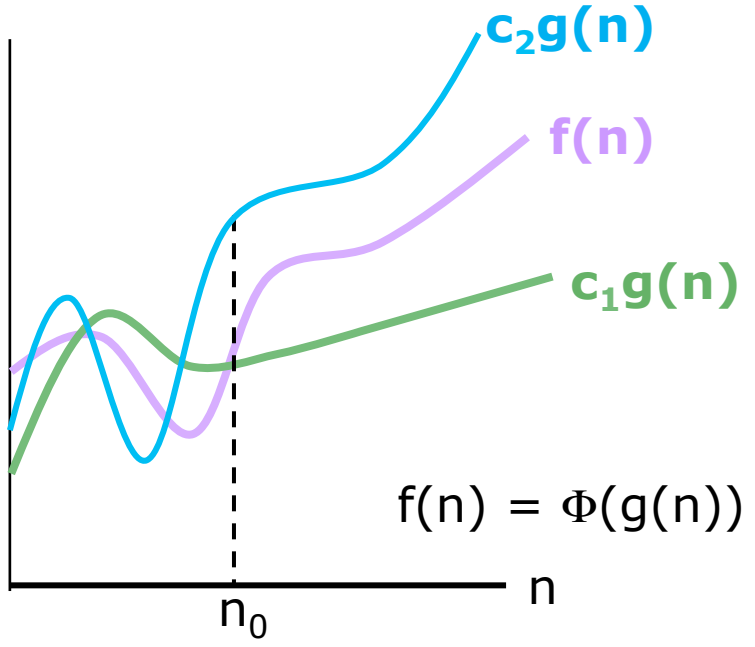
A notação Φ

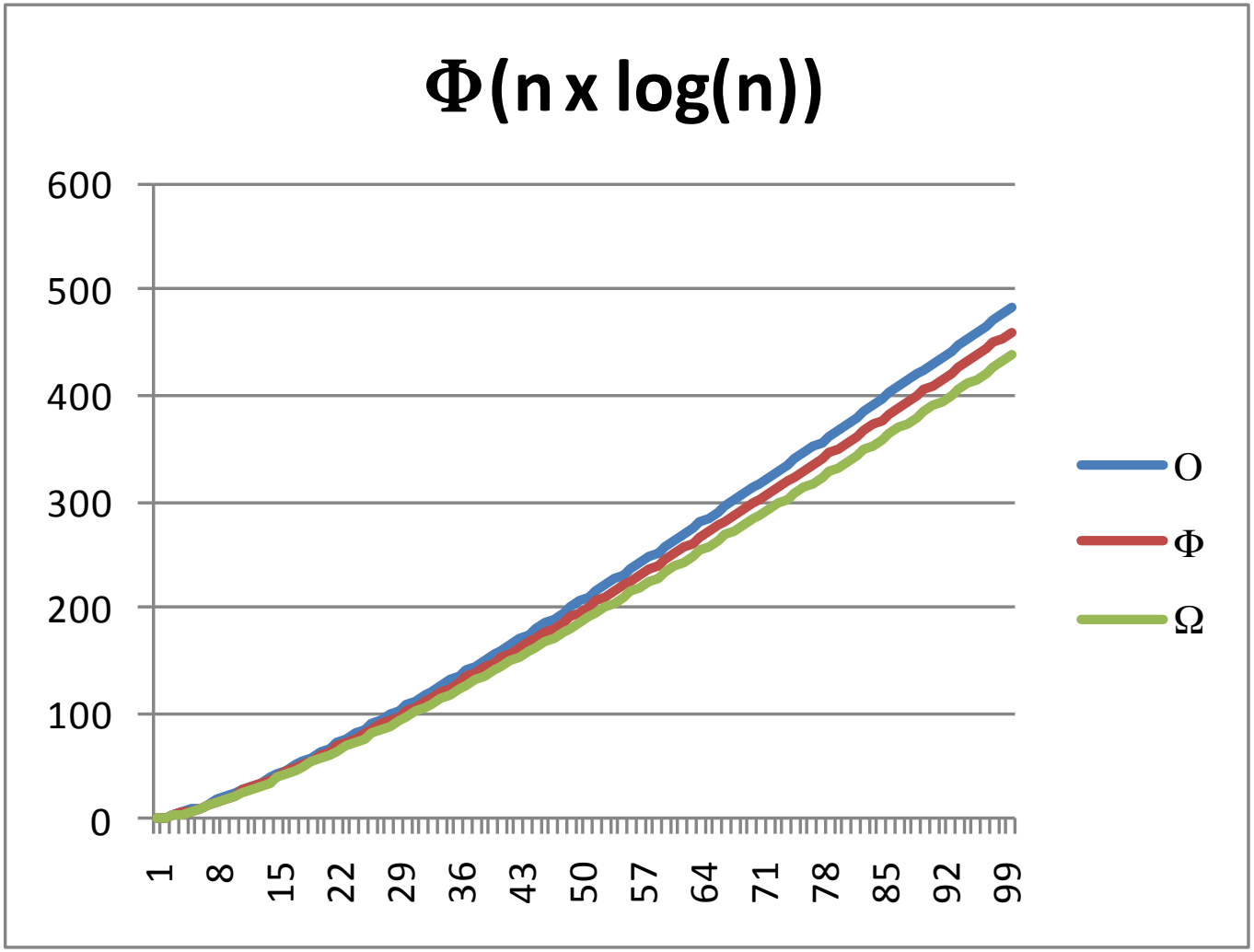
Sejam f e g duas funções de domínio X.

Dizemos que a função f é Φ(g(n)) se,

$$\exists c_1, c_2 \in \Re^+, \exists n_0 \in X : c_1|g(n)| \leq |f(n)| \leq c_2|g(n)|, \forall n \geq n_0$$

Assim, a função f(n) é Φ(g(n)) se existirem duas constantes positivas c_1, c_2 de tal modo que se possa **limitar a função** |f(n)| por $c_1|g(n)|$ e $c_2|g(n)|$, para n suficientemente grande.




Φ(n x log(n))
600
500
400
300
200
100
0
1
8
15
22
29
36
43
50
57
64
71
78
85
92
99
O
Φ
Ω

Busca sequencial

```
int pesquisaSequencial(char vet[], int tam, char dado)
{
    int i;
    for (i=0; i<tam; i++){
        if ( vet[i] == dado )
            return(i);
    }
    return(0);
}
```

Busca sequencial
Análise de melhor caso (Ω)

Quando acontece o melhor caso?

Quando o dado procurado está na **primeira posição** do vetor.

Logo, f(n) = 1, o algoritmo realizará apenas uma comparação.

Assim, a complexidade no melhor caso é $\Omega(1)$.

Busca sequencial
Análise de pior caso (O)

Quando acontece o pior caso?

Quando o dado procurado está na **última** posição do vetor ou **o dado não está no vetor**.

Dado um vetor de tamanho n

tem-se que f(n) = n.

Assim, a complexidade no pior caso é O(n).

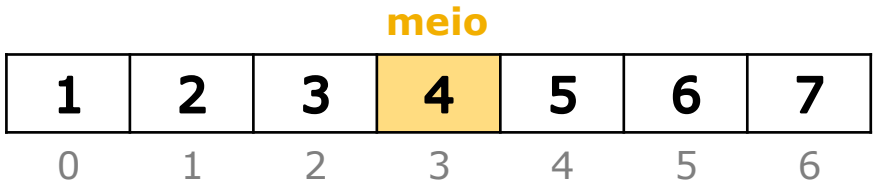
Busca binária (vetor ordenado)

```
int pesquisaBinaria( char vet[], char dado, int inic, int fim)
{
    int meio = (inic + fim)/2;
    if ( vet[meio] == dado )
        return (meio);
    if ( inic >= fim )
        return (-1);
    if ( dado < vet[meio] )
        return pesquisaBinaria (vet, dado, inic, meio-1);
    else
        return pesquisaBinaria (vet, dado, meio+1, fim);
}
```

Busca binária

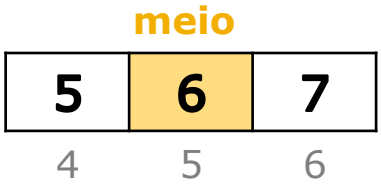
O dado a ser procurado é o '7'.

inic = 0
fim = 6
meio = 0 + 6 / 2 = 3



pesquisaBinaria (vet, dado, meio+1, fim);

inic = 4
fim = 6
meio = 4 + 6 / 2 = 5



pesquisaBinaria (vet, dado, meio+1, fim);

inic = 6
fim = 6
meio = 6 + 6 / 2 = 6

Busca binária
Análise de melhor caso (Ω)

Quando acontece o melhor caso?

Quando o elemento procurado está **no meio do vetor** (já na primeira chamada). Nesse caso, será executada apenas uma comparação, e a posição já será retornada.

Busca binária
Análise de melhor caso (Ω)

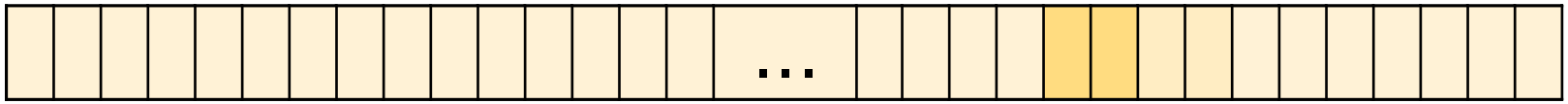
```
int pesquisaBinaria( char vet[], char dado, int inic, int fim){
int meio = (inic + fim)/2;
    if( vet[meio] == dado )
        return(meio);
    if (inic >= fim)
        return(-1);
    if (dado < vet[meio])
        pesquisaBinaria (vet, dado, inic, meio-1);
    else
        pesquisaBinaria (vet, dado, meio+1, fim);
}
```

Neste caso, o algoritmo em particular tem um comportamento constante: f(n) = 1.

Logo, podemos dizer que é Ω(1).

Busca binária
Análise de pior caso (O)

O pior caso acontece quando o elemento procurado não está no vetor.



n elementos

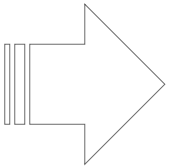
1° iteração: n elementos

2° iteração: n/2 elementos

3° iteração: n/4 elementos

4° iteração: n/8 elementos

5° iteração: n/16 elementos



K-ésima iteração: $n/(2^{k-1})$ elementos

Busca binária
Análise de pior caso (O)

As chamadas param quando:

- a posição do elemento é encontrada ou
- quando não há mais elementos a serem procurados, isto é, quando n < 1.

Para qual valor de k tem-se n < 1?

$$\frac{n}{2^{k-1}} = 1 \Rightarrow n = 2^{k-1} \Rightarrow \log_2 n = \log_2 2^{k-1} \Rightarrow \log_2 n = (k-1)\log_2 2 \Rightarrow$$

$$\Rightarrow \log_2 n = k - 1 \Rightarrow k = 1 + \log_2 n$$

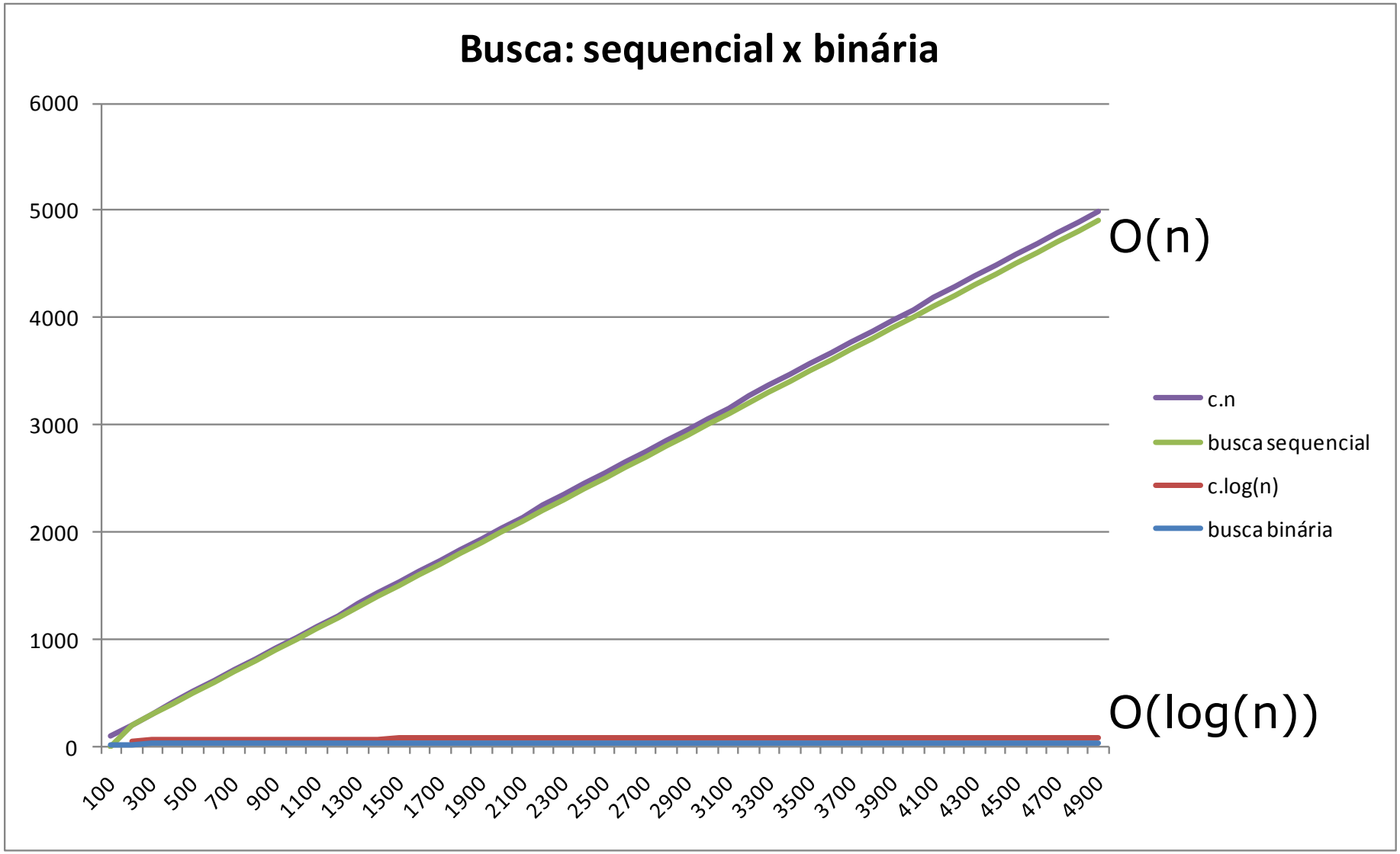
Portanto, quando $k = \log_2 n$ temos um elemento no vetor ainda e para $k > 1 + \log_2 n$ o algoritmo pára.

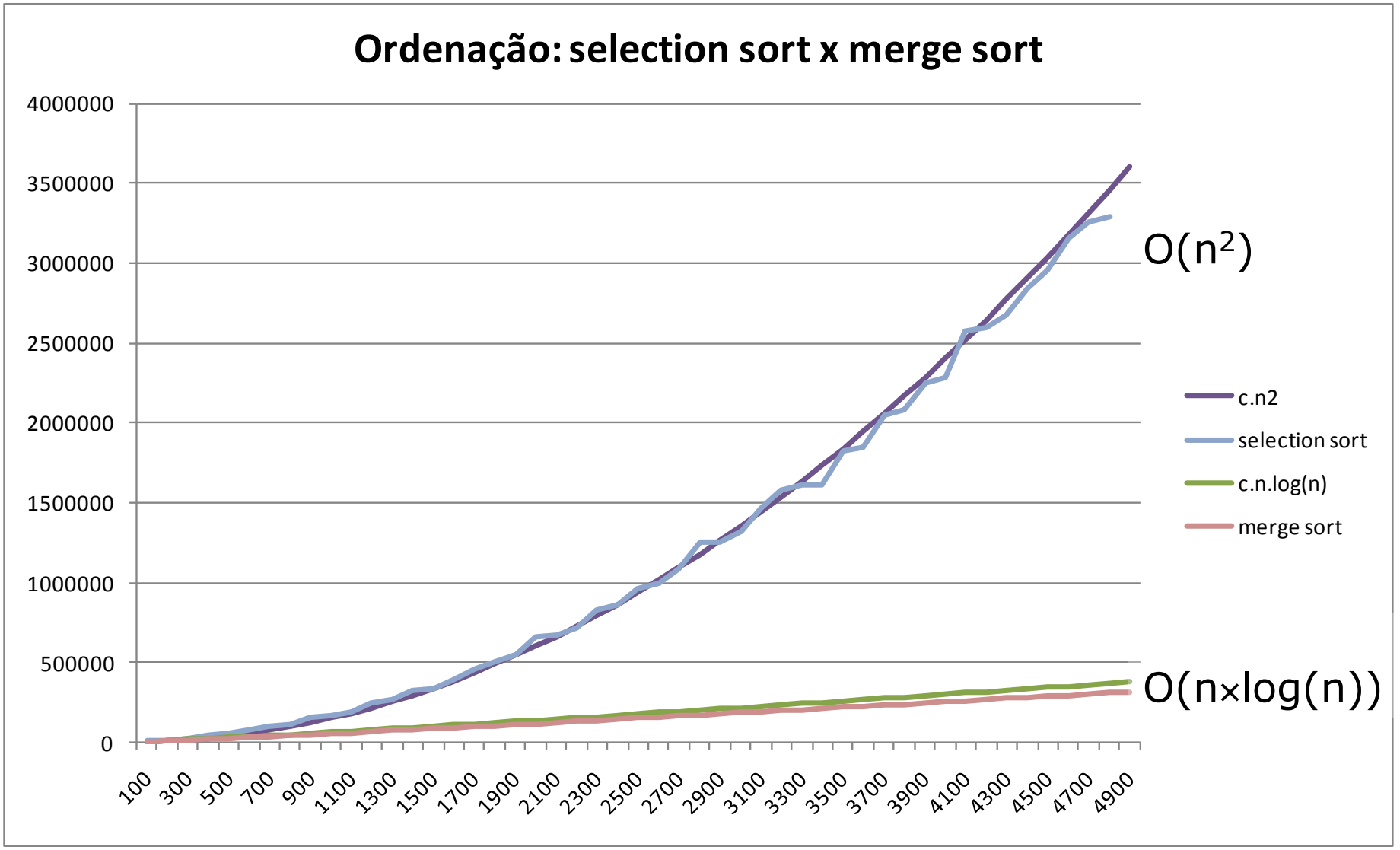
Busca binária
Análise de pior caso (O)

Assim, temos $1 + \log_2 n$ passos no pior caso.

Mas, $1 + \log_2 n < c(\log_2 n)$ para algum $c > 0$.

Portanto, a complexidade no algoritmo no pior caso é **$O(\log_2 n)$**.


Busca: sequencial x binária
6000
5000
4000
3000
2000
1000
0
100
300
500
700
900
1100
1300
1500
1700
1900
2100
2300
2500
2700
2900
3100
3300
3500
3700
3900
4100
4300
4500
4700
4900
O(n)
O(log(n))
c.n
busca sequencial
c.log(n)
busca binária



Complexidades comumente encontradas

Notação	Nome	Característica	Exemplo
O(1)	constante	independe do tamanho n da entrada	determinar se um número é par ou ímpar; usar uma tabela de dispersão (hash) de tamanho fixo
O(log n)	logarítmica	o problema é dividido em problemas menores	**busca binária**
O(n)	linear	realiza uma operação para cada elemento de entrada	**busca sequencial**; soma de elementos de um vetor
O(n log n)	log-linear	o problema é dividido em problemas menores e depois junta as soluções	heapsort, quicksort, **merge sort**
$O(n^2)$	quadrática	itens processados aos pares (geralmente loop aninhado)	bubble sort (pior caso); quick sort (pior caso); **selection sort**; insertion sort
$O(n^3)$	cúbica		multiplicação de matrizes n x n; todas as triplas de n elementos
$O(n^c)$, c>1	polinomial		caixeiro viajante por programação dinâmica
$O(c^n)$	exponencial	força bruta	todos subconjuntos de n elementos
O(n!)	fatorial	força bruta: testa todas as permutações possíveis	caixeiro viajante por força bruta

Em geral: $O(1) < O(\log n) < O(n) < O(n \log n) < O(n2) < O(n3) < O(c^n) < O(n!)$

Soma de vetor – passos de execução

comando	passo	frequência	subtotal
`float soma(float v[], int n)`	0	0	0
`{`	0	0	0
`  int i;`	0	0	0
`  float somatemp = 0;`	1	0	1
`  for (i=0; i < n; i++)`	1	n+1	n+1
`    somatemp += vet[i];`	1	n	n
`  return somatemp;`	1	1	1
`}`	0	0	0
Total			**2n+3**

O(n)

Soma de matriz – passos de execução

comando	passo	frequência	subtotal
`float soma(int a[][N], ..., int rows, int cols)`	0	0	0
`{`	0	0	0
`  int i, j;`	0	0	0
`  for (i=0; i < rows; i++)`	1	rows+1	rows+1
`    for (j=0; j < cols; j++)`	1	rows × (cols+1)	rows × (cols+1)
`      c[i][j] = a[i][j]+b[i][j];`	1	rows × cols	rows × cols
`}`	0	0	0
Total			**2rows × cols + 2rows + 1**

$O(n^2)$

Soma de matriz – complexidade

comando	complexidade assintótica
`float soma(int a[][N], ..., int rows, int cols)`	0
`{`	0
`  int i, j;`	0
`  for (i=0; i < rows; i++)`	$\Phi(rows)$
`    for (j=0; j < cols; j++)`	$\Phi(rows \times cols)$
`      c[i][j] = a[i][j]+b[i][j];`	$\Phi(rows \times cols)$
`}`	0
Total	**$\Phi(rows \times cols)$**

$O(n^2)$

Multiplicação de matriz – complexidade

comando	complexidade assintótica
`float multi(double *a, double *b, double *c, int n)`	0
`{`	0
`  int i, j, k;`	0
`  for (i=0; i < n; i++)`	n
`    for (j=0; j < n; j++)`	n x n
`    {`	0
`      c[i][j] = 0`	n x n
`      for (k=0; k < n; k++)`	0
`        c[i][j] += a[i][l] * b[l][j];`	n x n x n
`    }`	0
`}`	0
Total	**Φ(rows × cols)**

$$O(n^3)$$

Cota Superior (Upper Bound)

A cota superior de um problema é definida pelo algoritmo mais eficiente para resolver este problema.

Ou seja, para se resolver um problema a complexidade dele não pode ser maior que a do algoritmo conhecido.

Conforme novos (e mais eficientes) algoritmos vão surgindo esta cota vai diminuindo.

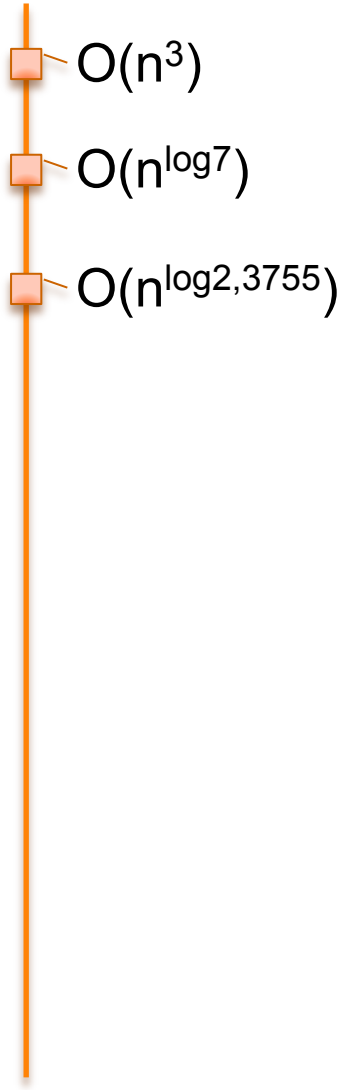
Cota Superior: Multiplicação de Matrizes

Um problema clássico é a operação de multiplicação em matrizes quadradas:

O algoritmo tradicional resolve em $O(n^3)$. Assim imediatamente podemos afirmar que a cota superior é no máximo $O(n^3)$.

O algoritmo de Strassen (1969) resolve multiplicações de matrizes em $O(n^{log7})$, levando a cota superior para $O(n^{log7})$.

O algoritmo de Coppersmith-Winograd (1990), melhorou ainda mais, levando a multiplicação de matrizes para $O(n^{2,3755})$.

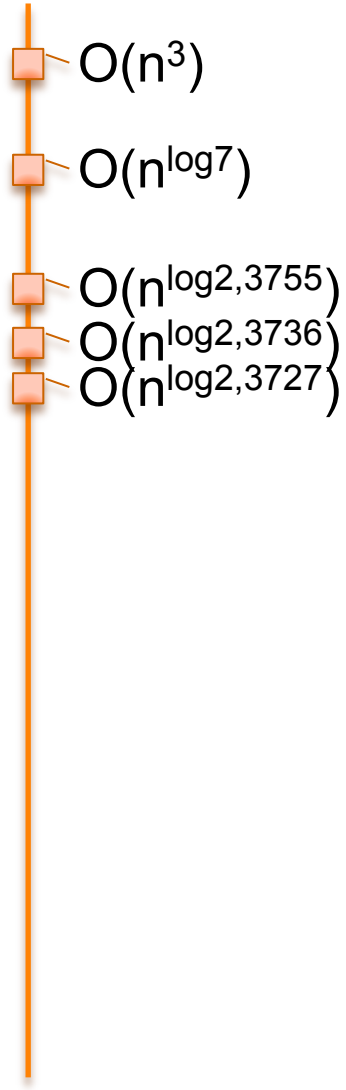


Cota Superior: Multiplicação de Matrizes

Andrew Stothers (2010)melhorou o algoritmo de Coppersmith-Winograd, chegando a $O(n^{2,3736})$.

Virginia Williams (2011) melhorou ainda mais o algoritmo chegando a $O(n^{2,3727})$, definindo a cota superior conhecida atualmente.

Vale notar que esses algoritmos só se aplicam a matrizes muito grandes, que dependendo do caso não podem nem ser processadas pelos computadores atuais.

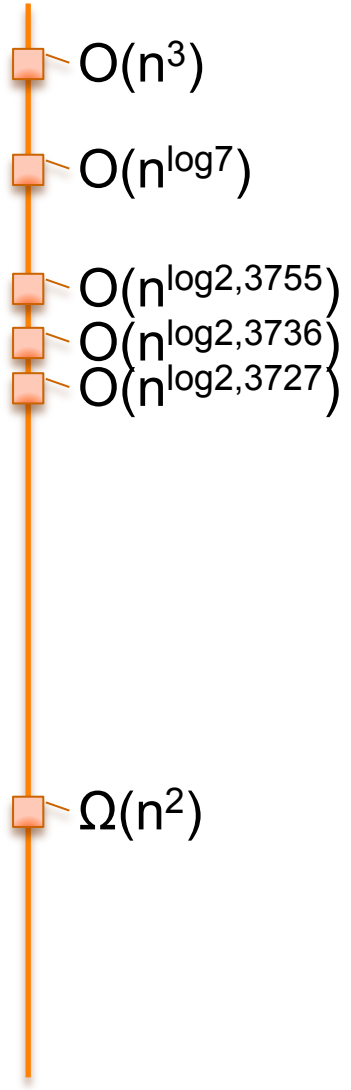


Cota Inferior (lower bound)

Número mínimo de operações para resolver um problema, independente de qualquer algoritmo a usar.

Ou seja, independente de como o algoritmo seja feito ele vai precisar de no mínimo um certo número de operações.

No problema de multiplicação de matrizes, apenas para ler e escrever as matrizes seria necessário n^2 operações, assim a cota inferior seria $\Omega(n^2)$.



Assintoticamente Ótimos

Algoritmos que tem a complexidade igual a cota inferior são considerados assintoticamente ótimos.

No caso da multiplicação de matrizes não é conhecido nenhum algorítmo assintoticamente ótimo atualmente.

Fatorial de um número - complexidade

Seja T(n) o tempo de processamento para um entrada de tamanho n.

T(n) = 1 se n = 0 ou 1

T(n) = T(n-1) + 1 se n > 1

Mas quanto vale T(n-1)?

(T(n-1)) + 1 =

(T(n-2) + 1) + 1 =

T(n-2) + 2 =

(T(n-3) + 1) + 2 = T(n-3) + 3

...

T(n) = T(n-k) + k; $1 < k \leq n$

```
int fatorial (int n)
{
   if (n<= 1)
      return(1);
   else
      return( n * fatorial(n-1) );
}
```


Fazendo k = n temos

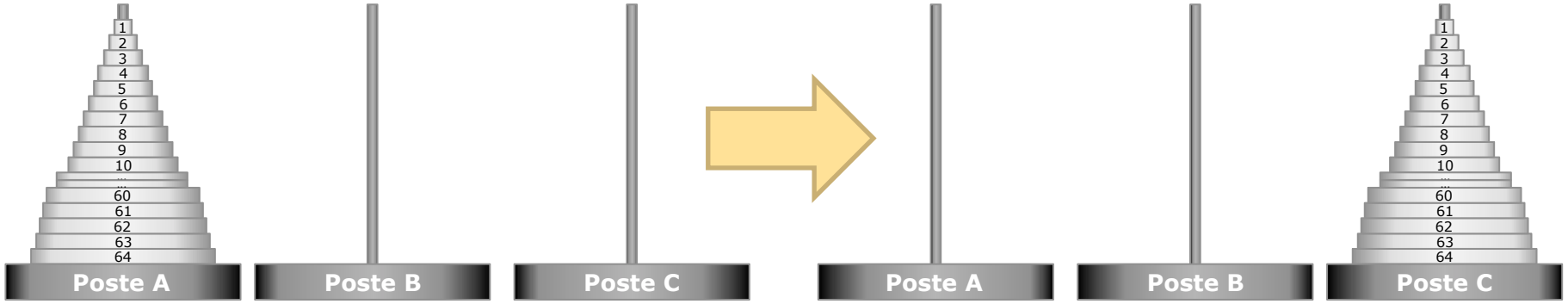
T(n) = T(n-n) + n = T(0) + n = n

T(n) = n

Exemplo: Torres de Hanói

Diz a lenda que um monge muito preocupado com o fim do Universo perguntou ao seu mestre quando isto iria ocorrer.

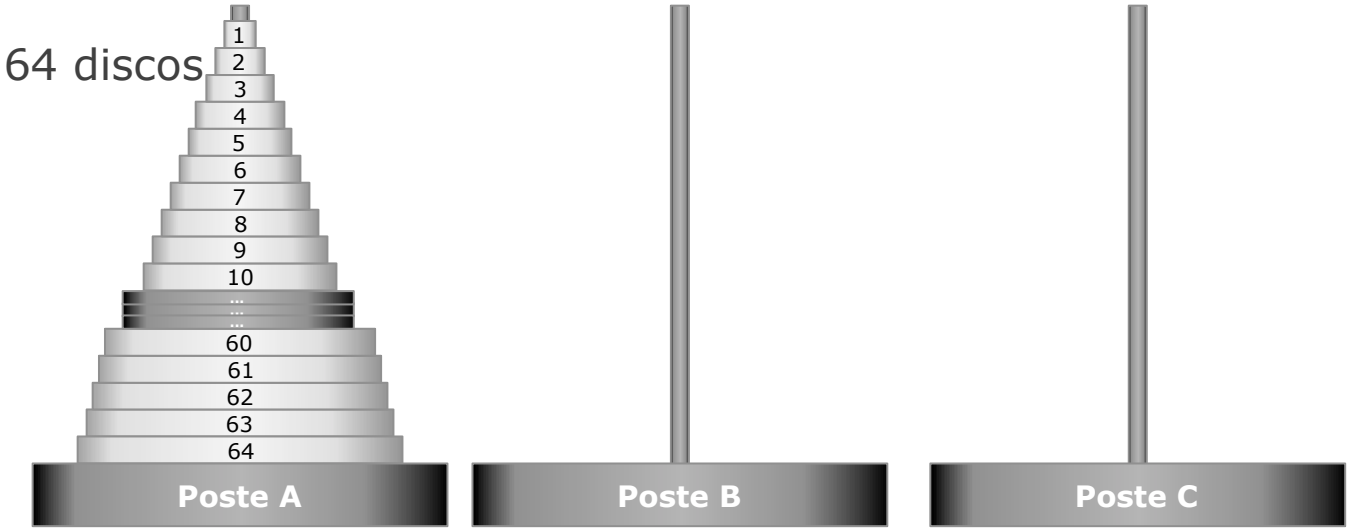
O mestre, vendo a aflição do discípulo, pediu a ele que olhasse para os três postes do monastério e observasse os 64 discos de tamanhos diferentes empilhados no primeiro deles. Disse que se o discípulo quisesse saber o tempo que levaria para o Universo acabar bastava que ele calculasse o tempo que levaria para ele mover todos os discos do Poste A para o Poste C seguindo uma regra simples: ele nunca poderia colocar um disco maior sobre um menor e os discos teriam que repousar sempre num dos postes.



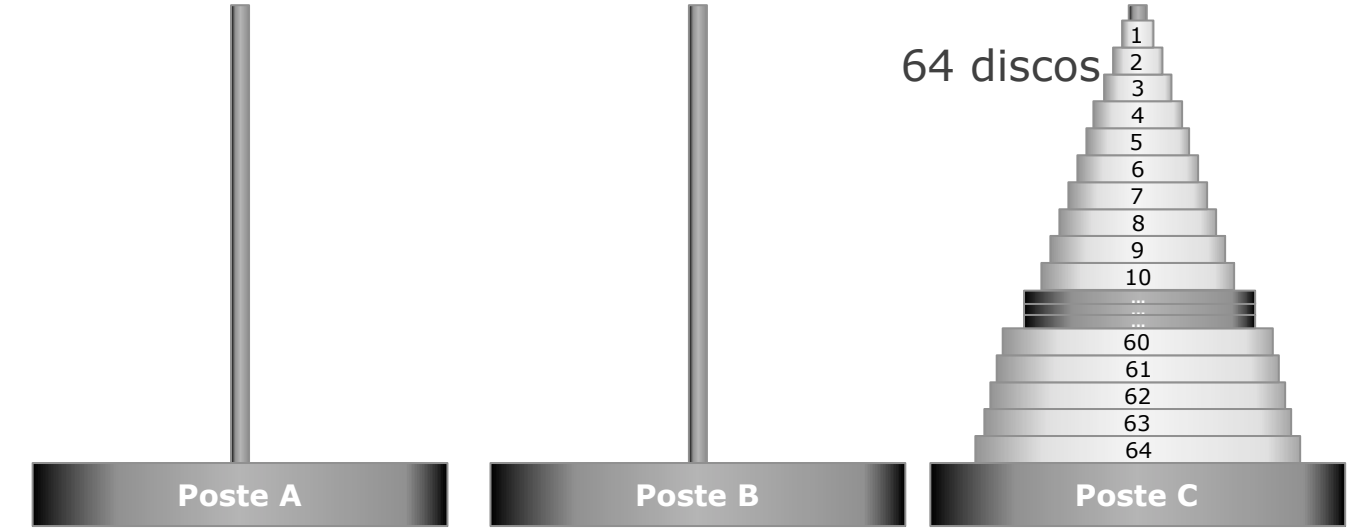
Em quanto tempo você estima que o mestre disse que o Universo vai acabar?

Torres de Hanói

inicial

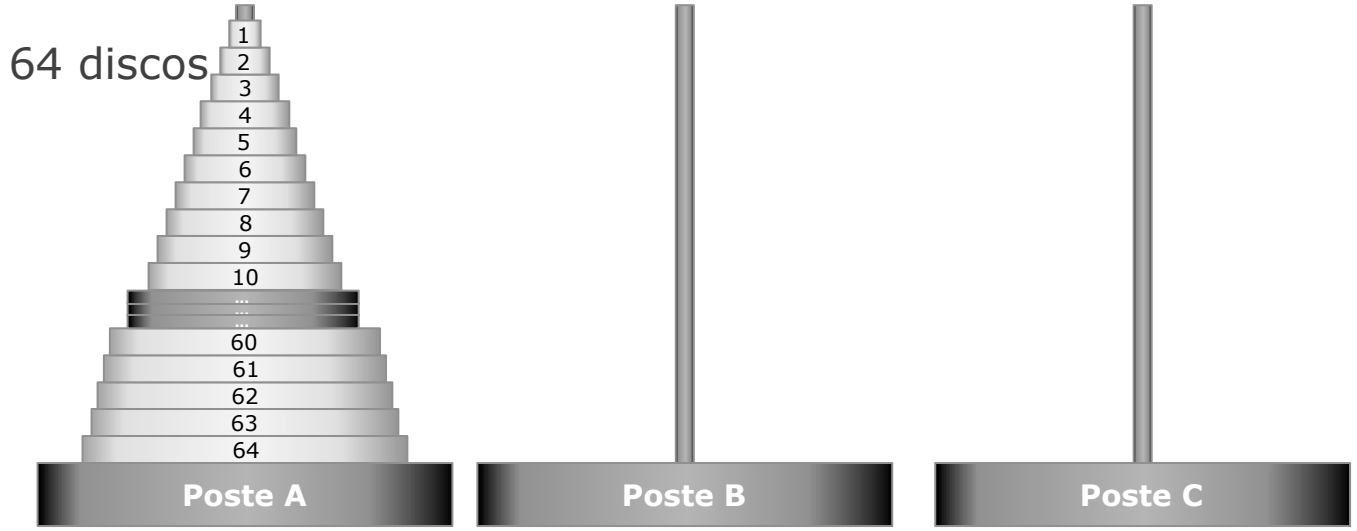


final



Torres de Hanói – algoritmo recursivo

Suponha que você tem uma solução para mover n-1 discos, e a partir dela crie uma solução para n discos.

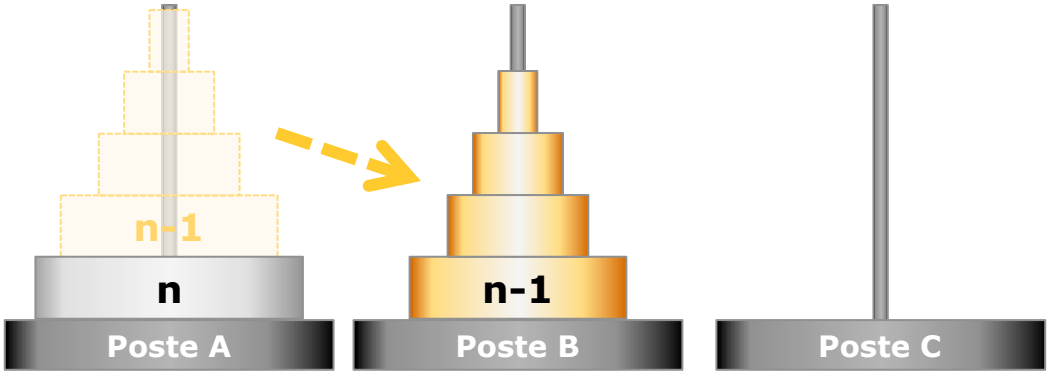


Torres de Hanói– algoritmo recursivo

Passo 1

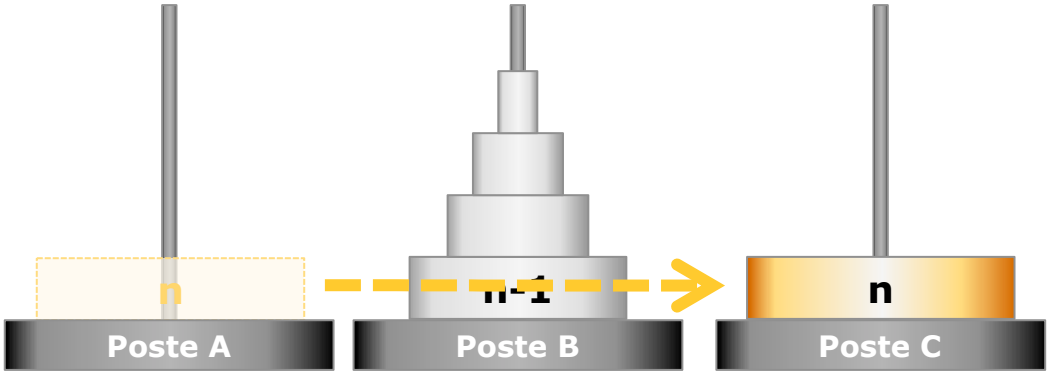
Mova n-1 discos do poste A para o poste B

(hipótese da recursão)



Passo 2

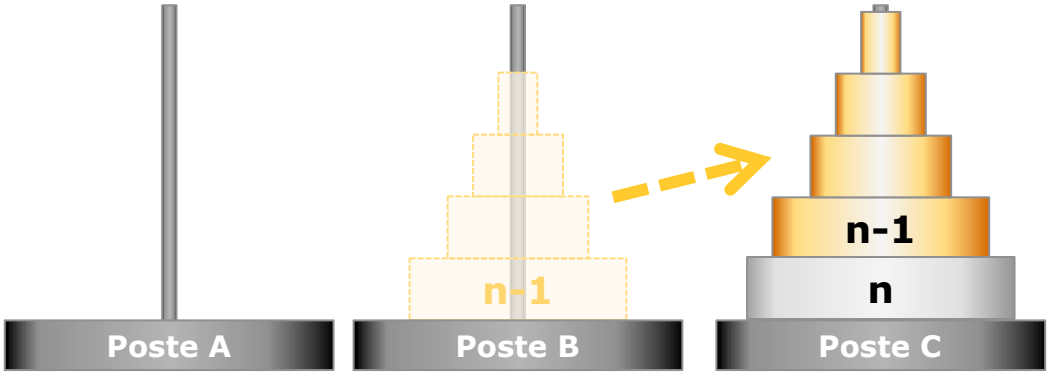
Mova o n-ésimo disco de A para C



Passo 3

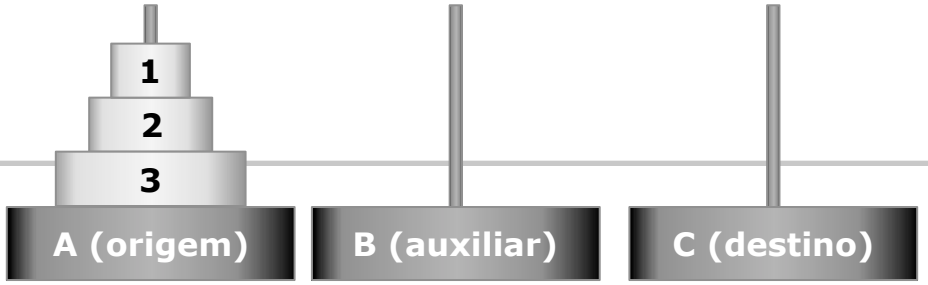
Mova n-1 discos de B para C

(hipótese da recursão)



Torres de Hanoi: Implementação

```
#include <stdio.h>

void torres(int n, char origem, char destino, char auxiliar)
{
  if (n == 1) {
    printf("Mova o Disco 1 do Poste %c para o Poste %c\n", origem, destino);
    return;
  }
  else {
    torres(n-1, origem, auxiliar, destino);
    printf("Mova o Disco %d do Poste %c para o Poste %c\n", n, origem, destino);
    torres(n-1, auxiliar, destino, origem);
  }
}

int main( void )
{
  torres(3, 'A', 'C', 'B');
  return 0;
}
```



Execução para 3 discos:

Mova o disco 1 do Poste A para o Poste C

Mova o disco 2 do Poste A para o Poste B

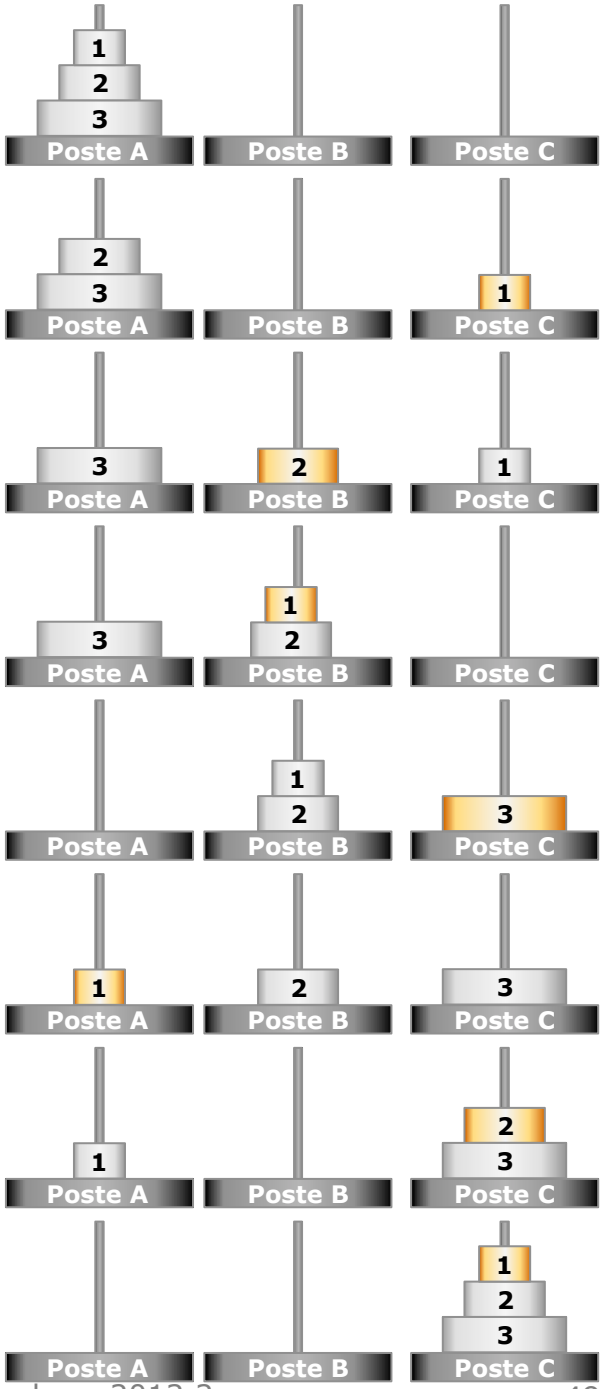
Mova o disco 1 do Poste C para o Poste B

Mova o disco 3 do Poste A para o Poste C

Mova o disco 1 do Poste B para o Poste A

Mova o disco 2 do Poste B para o Poste C

Mova o disco 1 do Poste A para o Poste C



Torres de Hanoi: Análise da complexidade

Seja t_n o tempo necessário para mover n discos

$t_n = 1 + 2t_{n-1}$ (a constante 1 pode ser ignorada)

$t_n \approx 2t_{n-1} = \underbrace{2(2(2(2(2\ldots(2t_1)))))}_{n-1}$

$t_n \approx \mathbf{2^{n-1}}t_1$ (exponencial)

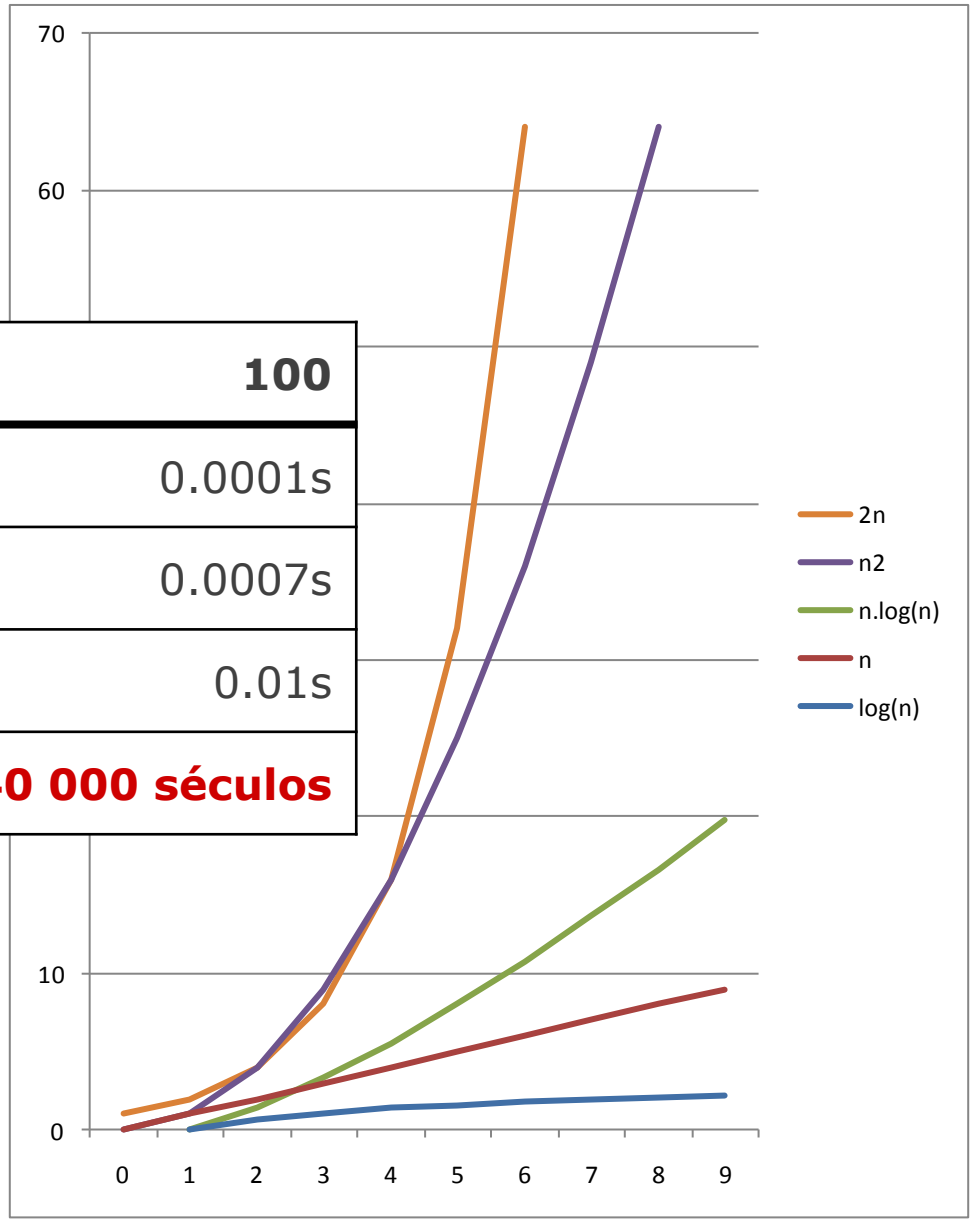
Para 64 discos: $t_{64} \approx 2^{63}t_1 = 9.2 \times 10^{18}t_1$

Supondo que o tempo para mover um disco seja t_1 = 1 s, o monge levaria **292.277.265 milênios** para terminar a tarefa!

n	2^{n-1}
1	1
2	2
3	4
4	8
5	16
6	32
7	64
8	128
9	256
10	512
11	1024
12	2048
13	4096

Importância da complexidade de um algoritmo

	10	20	100
n	0.00001s	0.00002s	0.0001s
n log(n)	0.00003s	0.00009s	0.0007s
n^2	0.0001s	0.0004s	0.01s
2^n	**intratáveis!**		**40 000 séculos**



dúvidas?