Comportamento Assintótico de Funções

É interessante comparar algoritmos para valores grandes de n.

Para valores pequenos de n, mesmo um algoritmo ineficiente não custa muito para ser executado

Fazendo estimativas e simplificações...

- O número de passos que um algoritmo dá para uma dada entrada pode depender de vários parâmetros:
  - Se for um grafo pode depender do número de nós, arestas, grau máximo do grafo, ou uma combinação destes.

- Mas, por simplicidade, computamos o tempo de execução em função da entrada
  - (que em grafos geralmente é o número de nós) e desconsideramos os outros parâmetros.

- Além disso, usamos uma estimativa – a análise assintótica – que somente considera o termo de ordem mais alta de uma expressão para o tempo de execução
  - Por exemplo, para $f(n) = 6\ n^3 + 2\ n^2 + 20\ n + 45$ consideramos 6n3 e descartamos o coeficiente 6 e dizemos que $f(n) = O(n^3)$.

Introdução

- Objetivos:

  Apresentar as notações assintóticas usadas para descrever a taxa de crescimento de funções de custo de algoritmos

Tópicos:

- Notação O – big-oh
- Notação Ω - ômega
- Notação Θ - theta
- Classes de comportamento assintótico
- Algoritmo Exponencial versus Polinomial
- Como calcular a complexidade de limite superior (big – Oh) de algoritmos

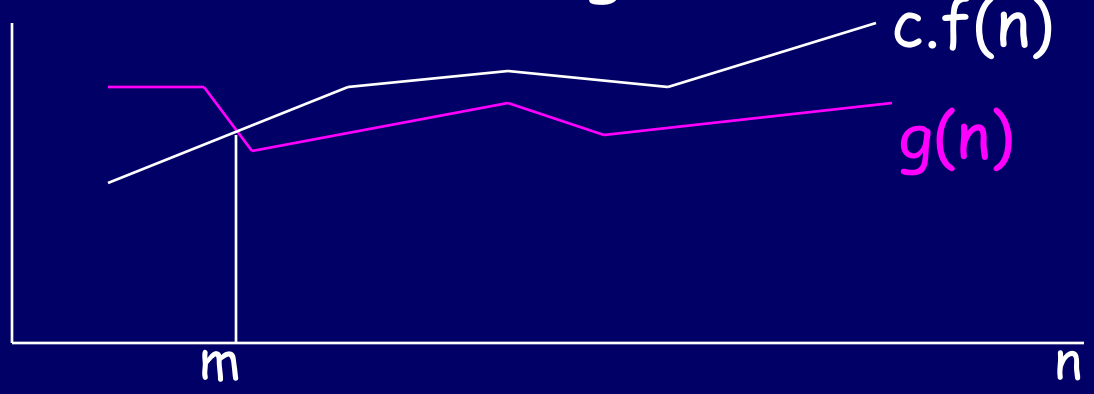
Custo Assintótico de Funções

- O custo assintótico de uma função f(n) representa o limite do comportamento de custo quando n cresce.

- Geralmente, um algoritmo que é assintoticamente mais eficiente será a melhor escolha para valores grandes de n.

Dominação Assintótica

- Definição: f(n) domina assintoticamente g(n) se existem duas constantes positivas c e m tais que, para n >= m, temos |g(n)| <= c.|f(n)|



- Exemplo: Seja g(n) = n e f(n) = $-n^2$
- Temos que |n| <= $|-n^2|$ para todo n pertencente aos N.
- Fazendo c = 1 e m = 0 a definição é satisfeita. Logo, f(n) domina assintoticamente g(n)

Notação O

- Notação trazida da matemática por Knuth (1968):

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

- Lê-se:

g(n) é de ordem no máximo f(n)

f(n) domina assintoticamente g(n)

(f(n) é um limite assintótico superior para g(n))

- O(f(n)) representa o conjunto de todas as funções que são assintoticamente dominadas por uma dada função f(n).
- Se $g(n) \in O(f(n))$ então dizemos $g(n) = O(f(n))$

Formalmente: $g(n) = O(f(n))$, $\exists$ (existe) $c > 0$ e $n_0 \mid 0 <= g(n) <= c.f(n)$, $\forall$ (qualquer) $n >= n_0$

Notação O

Se dizemos que $g(n) = O(n^2)$

Significa que existem constantes c e m | $g(n) <= c.\ n^2$ para $n >= m$

Exemplo:

$g(0) = 1$

$g(1) = 4$

$g(n) = (n+1)^2$

n	$(n+1)^2$	$2.n^2$
0	1	0
1	4	2
2	9	8
3	16	18

$(n+1)^2 <= c.n^2$

$n^2 + 2n + 1 <= c.n^2$

$2n + 1 <= (c-1).n^2$

Fazendo c = 2 e m = 3 a definição é satisfeita

Notação Ω

- Ω define um limite inferior para a função, por um fator constante.
- Escreve-se $g(n) = \Omega(f(n))$, se existirem constantes positivas c e n_0 | para $n >= n_0$, o valor de g(n) é maior ou igual a c.f(n)
- Neste caso, diz-se que f(n) é um limite assintótico inferior para g(n).

Formalmente: $g(n) = \Omega(f(n))$, ∃(existe) $c > 0$ e n_0 | $0 <= c.f(n) <= g(n)$, ∀ (qualquer) $n >= n_0$

- A notação Ω é usada para expressar o limite inferior do tempo de execução de qualquer algoritmo para resolver um problema específico

- Exemplo: O limite inferior para qualquer algoritmo de ordenação que utiliza comparação de chaves é $\Omega(n \log n)$

Comportamento Assintótico de Funções

- Assim, existem duas notações principais na análise assintótica de funções: O e Ω

$O(f(n))$ depende do algoritmo (limite superior)

$\Omega(f(n))$ depende do problema (limite inferior)

Notação Θ

- A notação Θ limita a função por fatores constantes.
- g(n) = Θ(f(n)) se existirem constantes positivas c1 e c2 e n_0 tais que para n >= n_0, o valor de g(n) está sempre entre c1.f(n) e c2.f(n) inclusive.
- Neste caso, dizemos que f(n) é um limite assintótico firme para g(n).

Formalmente:
g(n) = Θ(f(n)), ∃ (existe) $c_1 > 0$ e $c_2 > 0$ e n_0 | 0 <= c_1.f(n) <= g(n) <= c_2.f(n), ∀ (qualquer) n >= n_0

Exemplo de uso da Notação Θ

- Provem que $1/2n^2 - 3n = \Theta(n^2)$

Achem constantes $c_1 > 0$, $c_2 > 0$, $n_0 > 0$, tais que:

$$c_1.n^2 <= 1/2n^2 - 3n <= c_2.n^2$$

para todo $n >= n_0$

OBS: as constantes c1 e c2 podem ser reais.

Classes de Comportamento Assintótico

- f(n) = O(1) (complexidade constante)

  O uso do algoritmo independe do tamanho de n. Neste caso, as instruções do programa são executadas um número fixo de vezes.

- f(n) = O(log n) (complexidade logaritmica)

  Ocorre tipicamente em algoritmos que resolvem um problema transformando-o em problemas menores.

  Nestes casos, o tempo de execução pode ser considerado como sendo menor que uma constante grande .

  Exemplos: n = 1000 -- $\log_2 n \sim 10$; n = 10^6 – $\log_2 n \sim 20$.

- f(n) = O(n) (complexidade 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.

Classes de Comportamento Assintótico

- $f(n) = O(n \log n)$

  Este tempo de execução ocorre tipicamente em algoritmos resolvidos com o método dividir&conquistar. Exemplos: n = 1 milhão – n log n ~20 milhões;

  n = 2 milhões, n log n ~ 42 milhões, pouco mais que o dobro.

- $f(n) = O(n^2)$ (complexidade quadrática)

  Algoritmos desta ordem de complexidade ocorrem quando os itens são processados aos pares, em um laço dentro de outro. São úteis para resolver problemas de tamanho pequeno – métodos diretos de ordenação.

Dividir & Conquistar

- Muitos algoritmos possuem uma natureza recursiva:
  - para resolver um dado problema eles se chamam recursivamente uma ou mais vezes para tratar de subproblemas do mesmo tipo mas de tamanho menor e,
- Então combinam estas soluções para criar a solução original, seguindo a abordagem dividir e conquistar.
- Método Top-down

Os 3 Passos da Abordagem

- Dividir o problema em vários subproblemas
- Conquistar os subproblemas resolvendo eles recursivamente. Quando os tamanhos são pequenos resolver diretamente
- Combinar as soluções dos subproblemas na solução para o problema original

- Lembrem que o uso da recursão facilita a descrição do algoritmo e garante um mecanismo "limpo" para dividir o problema em 2 menores com a metade do tamanho do original.

MergeSort

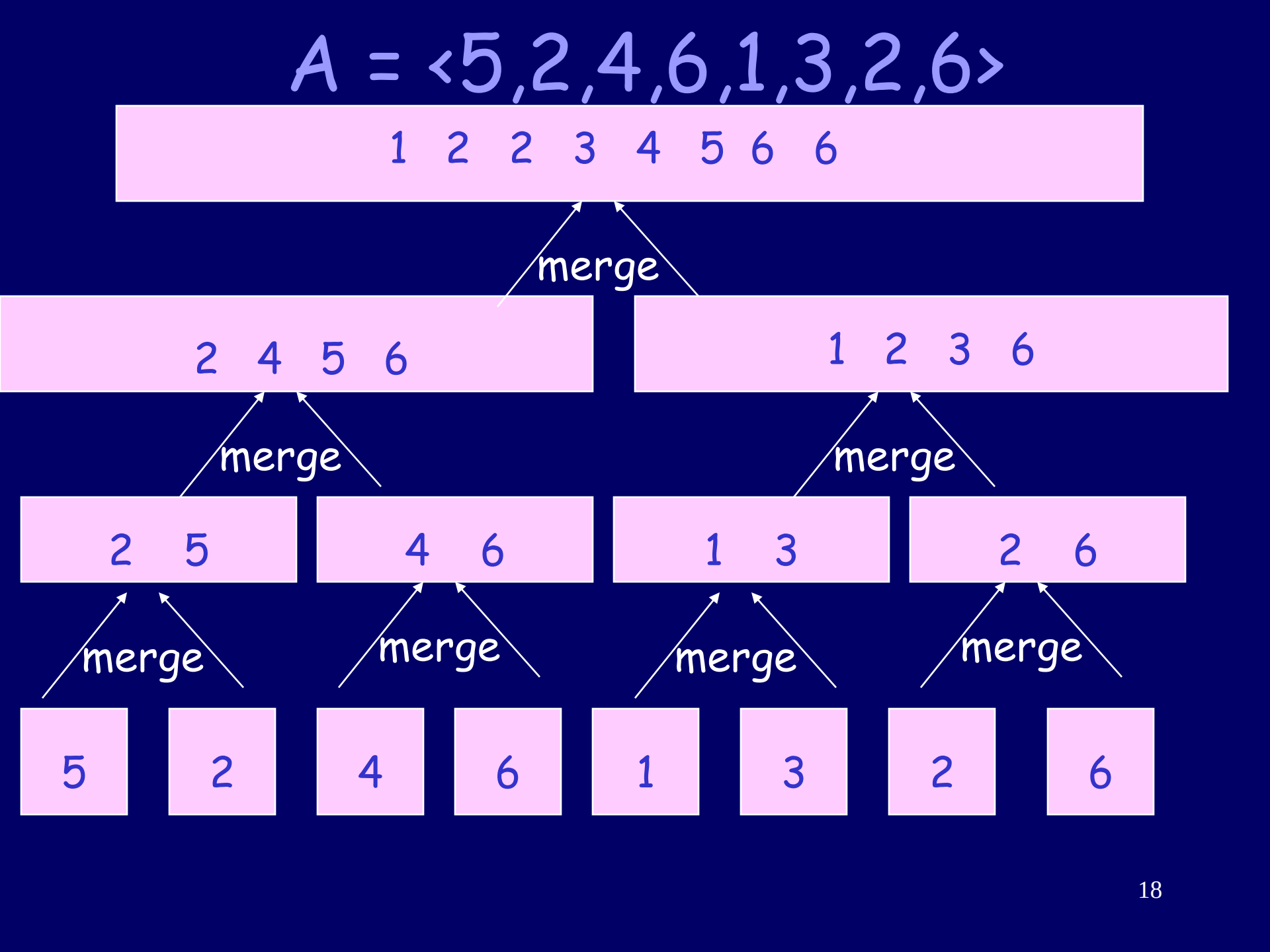
(Dividir): divida a lista L de tamanho n em duas listas de tamanho n/2

(Conquistar): Ordene as duas listas recursivamente usando o mergesort. Se a lista L tem tamanho 1 ela está ordenada.

(Combinar): Intercale as duas produzindo uma lista ordenada

A complexidade de tempo de MergeSort é O(n log n)

```
procedure mergesort(A,p,r);
Se p < r então
   inicio
    q <- (p + q) div 2
    mergesort(A,p,q)
    mergesort(A,q+1,r)
    merge(A,p,q,r) {assume que os vetores A[p..q] e
    A[q+1..r] estão ordenados e os intercala para
    formar um único array A[p..r]}
  fim
```


A = <5,2,4,6,1,3,2,6>
1 2 2 3 4 5 6 6
merge
2 4 5 6
1 2 3 6
merge
merge
2 5
4 6
1 3
2 6
merge
merge
merge
merge
5
2
4
6
1
3
2
6

Classes de Comportamento Assintótico (cont.)

- $f(n) = O(n^3)$ (complexidade cúbica)

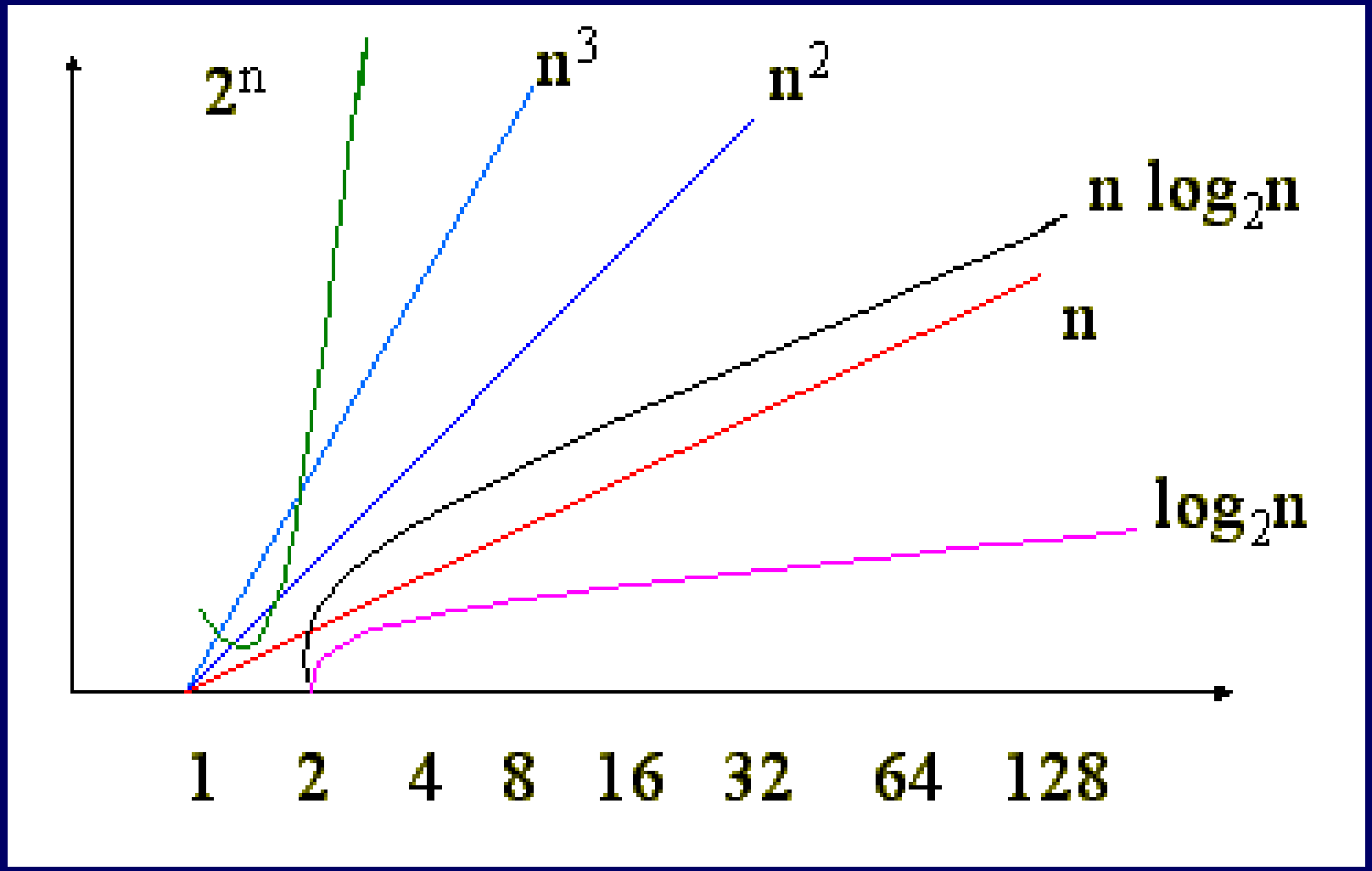
  Algoritmos desta ordem são úteis apenas para resolver pequenos problemas.

- $f(n) = O(2^n)$ (complexidade exponencial)

  Algoritmos desta ordem geralmente não são úteis sob o ponto de vista prático.

  Eles ocorrem na solução de problemas quando se usa a força bruta para resolvê-los.

Relação entre as ordens



Onde estão n! e n^n?

Algoritmo Exponencial Versus Polinomial

- Um algoritmo cuja função de complexidade é $O(c^n)$, $c > 1$, é chamado de algoritmo exponencial de tempo de execução.
- Um algoritmo cuja função de complexidade é $O(p(n))$, onde $p(n)$ é um polinômio, é chamado de algoritmo polinomial de tempo de execução.
- A distinção entre os dois torna-se significativa quando n cresce.
- Um problema é considerado intratável se ele é tão difícil que não existe um algoritmo polinomial para resolvê-lo
  - (torres de hanoi, damas e xadrez generalizados).
- Um problema é considerado bem resolvido quando existe um algoritmo polinomial para resolvê-lo.
  - São obtidos através de um entendimento mais profundo da estrutura do problema.

Princípios para a Análise de Algoritmos -- Prática

Tempo de execução para:

1. Atribuição, leitura e escrita: $O(1)$
   Exceção linguagens que permitem atribuição direta em vetores de tamanho grande

2. Seqüência: comando de maior tempo

3. Decisão: avaliação da condição: $O(1)$; comandos dentro: regra 2.

4. Laço: avaliação do término: $O(1)$; comandos dentro: regra 2.

5. Programa com procedimentos não recursivos: o tempo de execução de cada procedimento deve ser calculado separadamente um a um.

   Inicia-se com aqueles que não chamam outros. A seguir são avaliados aqueles que chamam os que não chamam outros, utilizando os tempos já calculados.

6. Programa com procedimentos recursivos: para cada procedimento é associada uma função de complexidade f(n) desconhecida, onde n mede o tamanho dos argumentos para o procedimento.

Operações com a Notação O

- f(n) = O(f(n))
- c. O(f(n)) = O(f(n)) se c = constante
- O(f(n) + O(f(n)) = O(f(n))
- O(O(f(n))) = O(f(n))
- O(f(n)) + O(g(n)) = O(max(f(n),g(n)))
- O(f(n))O(g(n)) = O(f(n)g(n))
- f(n)O(g(n)) = O(f(n)g(n))

Regra da Soma

Suponha 3 trechos de programas cujos tempos de execução são O(n), $O(n^2)$ e O(n log n). O tempo dos 2 primeiros é $O(\max(n,n^2))$ que é $O(n^2)$ e dos 3 é $O(\max(n^2, n \log n)$ que é $O(n^2)$.

Regra do Produto

Produto de [log n + k + O(1/n)] por [n + O(raiz(n))] =

n log n

Fórmulas Úteis

$\sum_{1<=i<=n} 1 = n = O(n)$ (linearidade)

$\sum_{1<=i<=n} i = n(n+1)/2 = O(n^2)$ (série aritmética)

$\sum_{1<=i<=n} i^2 = [n(n+1)(2n+1)]/6 = O(n^3)$

Exemplo

```
Procedure Ordena (var A: vetor);
{ ordena o vetor em ordem ascendente}
Var i, j, min, x: integer;
Begin
(1) For i:= 1 to n-1 do
    Begin
(2)       min := i;
(3)       for j:= i + 1 to n do
(4)         if A[j] < A[min]
(5)            then min := j;
          {troca A[min] e A[i]}
(6)       X := A[min];
(7)       A[mim] := A[i];
(8)       A[i] := X;
    End;
End;
```

Análise do Programa

- O programa contém 2 laços, um dentro do outro.
  - Laço mais externo: (2) a (8)
  - Laço mais interno: (4) e (5)

- Começamos pelo mais interno: comando de atribuição e avaliação da condição levam tempo constante.
- Como não sabemos se o corpo da decisão vai ou não ser executado, considera-se o pior caso: assumir que a linha (5) sempre será executada.
- O tempo para incrementar o índice do laço e avaliar sua condição de termo é também O(1).

- O tempo para executar uma vez o laço composto por (3), (4) e (5) é O(max(1,1,1)) = O(1).

→ Seguindo regra da soma para a notação O.

- Como o número de iterações do laço é n-i então o tempo gasto é $O(n-i \,.\, 1) = O(n - i)$.

→Seguindo a regra do produto para notação O.

- O corpo do laço mais externo contém, além do laço interno, os comandos de atribuição nas linhas (2), (6), (7), (8).

  O tempo das linhas (2) a (8) é: $O(\max(1,(n-i),1,1,1)) = O(n-i)$

A linha (1) é executada n-1 vezes, o tempo total do programa é:

$$\sum_{i=1}^{N-1} (n - i) = n.\sum_{i=1}^{N-1} 1 - \sum_{i=1}^{N-1} i = n.(n-1) - [(n-1).(n)]/2 = n^2 - n - n^2/2 - n/2$$

$$n^2/2 - n/2 = O(n^2)$$

Nro de comparações X nro de trocas

- Considerando o número de comparações como a mais relevante (4): $n^2/2 - n/2$ comparações

- Considerando o número de trocas (6), (7), (8) a mais relevante: 3(n-1) trocas.

Procedimento Recursivo para calcular n!

```
function fat(n: integer): integer;
Begin
(1)   If n <= 1
(2)   then fat := 1 { escape da recursão }
(3)   Else fat := n * fat (n-1)
End;
```

Seja T(n) o tempo de execução para fat(n)

(1)-> O(1)

(2)-> O(1)

(3)-> O(1) + T(n-1)

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

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

...

T(2) = 1 + T(1)

T(n) = 1 + 1 + 1+...+1 + T(1)

(n - 1 vezes)

Método da Iteração: expande a recorrência e expressa ela como uma soma.

Relação de Recorrência:

Assumindo custos de multiplicações

T(1) = 0 pois não multiplica

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

Procedimento não recursivo para fatorial

```
function fat (n : integer): integer;
var f, i: integer;
begin
   f:= 1;
   for i:= 2 to n do f:= f*i;
   fat := f
End;
```

F(n) -> custo de multiplicações

F(1) = 0

F(n) = n - 1

Resolvendo Recorrências

- Quando um algoritmo contém uma chamada recursiva a ele mesmo , seu tempo de execução pode ser expresso como uma recorrência.

- Existem 3 métodos (Cormen et all, 1999):

- Método Mestre:
  - Fornece 3 casos para resolver T(n) =aT(n/b) + f(n), a>=1; b>1
- Método da Substituição:
  - O objetivo é encontrar uma função (chute) tal que T(n) = O(f(n)) e provar por indução em n.
- Método da Iteração:
  - Expande a recorrência e expressa ela como soma. Método mais utilizado.