Algoritmos Estrutura de Dados

Marcelo Lobosco

DCC/UFJF

Algoritmos de Ordenação

Parte 2 – Aula 04

Agenda

- Algoritmos de Ordenação
  - Insert Sort
    - Loop Invariante
    - Crescimento de funções

Algoritmos de Ordenação

- Primeiro algoritmo: ordenação por inserção
  - Entrada: Sequência de n números (a_1, a_2, ..., a_n)
  - Saída: Uma permutação (reordenação) da sequência de entrada (a'_1, a'_2, ..., a'_n), tal que $a'_1 <= a'_2 <= ... <= a'_n$
  - Números que desejamos ordenar conhecidos como chaves

Algoritmos de Ordenação

- Primeiro algoritmo: ordenação por inserção
  - Método parecido com a ordenação de cartas de um baralho
  - Inicialmente mão vazia; cartas viradas com face para baixo
  - Uma carta da pilha tirada e seu lugar na mão encontrado

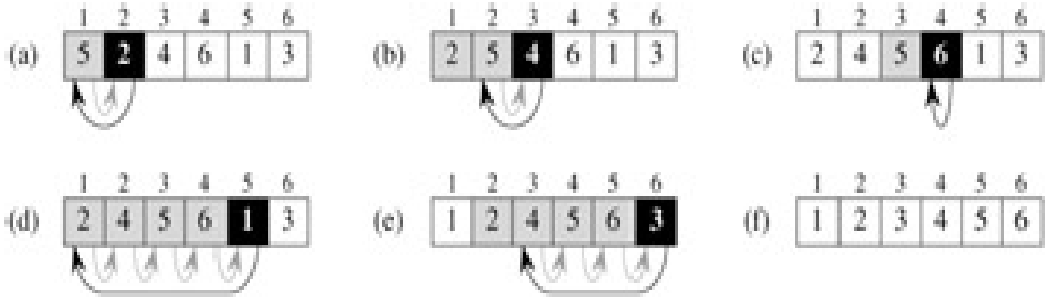
Algoritmos de Ordenação

- Algoritmo INSERTION-SORT
  - Arranjo A[1..n] passado como parâmetro
  - Números ordenados no local: reorganizados dentro do arranjo A
  - Ao terminar, arranjo A conterá a sequência de saída ordenada

```
INSERTION-SORT(A)
1  for j ← 2 to length[A]
2      do key ← A[j]
3         ▷ Insert A[j] into the sorted sequence A[1 .. j - 1].
4         i ← j - 1
5         while i > 0 and A[i] > key
6            do A[i + 1] ← A[i]
7               i ← i - 1
8         A[i + 1] ← key
```

Algoritmos de Ordenação

- Algoritmo INSERTION-SORT




```
INSERTION-SORT(A)
1  for j ← 2 to length[A]
2      do key ← A[j]
3         ▷ Insert A[j] into the sorted sequence A[1 .. j - 1].
4         i ← j - 1
5         while i > 0 and A[i] > key
6            do A[i + 1] ← A[i]
7               i ← i - 1
8         A[i + 1] ← key
```

Algoritmos de Ordenação

- Algoritmo INSERTION-SORT
  - Índice j indica a “carta atual”
  - No início de cada iteração do loop for, o subarranjo que consiste nos elementos A[1 .. j-1] está ordenado
    - Equivale a mão atualmente ordenada
  - Já o arranjo A[j+1 .. n] corresponde à pilha de cartas ainda na mesa
  - De fato, arranjo A [1 .. j-1] corresponde aos elementos que estavam originalmente nas posições de 1 a j-1, mas agora em sequência ordenada
    - Loop invariante

Algoritmos de Ordenação

- Loop invariante nos ajuda a entender porque um algoritmo é correto
  - Devemos demonstrar três coisas sobre um loop invariante:
    - Inicialização: ele é verdadeiro antes da primeira iteração do loop
    - Manutenção: se for verdadeiro antes de uma iteração do loop, ele permanecerá verdadeiro antes da próxima iteração
    - Término: quando o loop termina, o invariante nos fornece uma propriedade útil que ajuda a mostrar que o algoritmo é correto

Algoritmos de Ordenação

- Quando as duas primeiras propriedades são válidas, o loop invariante é verdadeiro antes de toda iteração do loop
- Semelhante a indução matemática: passo básico equivale a inicialização e etapa indutiva a manutenção
- Témino talvez seja a etapa mais importante, pois estamos usando o loop invariante para mostrar a correção do algoritmo
  - Difere da indução, visto que etapa indutiva usada indefinidamente

Algoritmos de Ordenação

- Vejamos como propriedades válidas para a ordenação por inserção:
  - Inicialização: devemos mostrar que ele é verdadeiro antes da primeira iteração do loop, ou seja, j = 2. Subarranjo A[1 .. j-1] consiste apenas de A[1], o elemento original de A quando passado como parâmetro. Além disso, esse subarranjo é obviamente ordenado. Assim, o loop invariante é válido antes da primeira iteração do loop.

Algoritmos de Ordenação

- Vejamos como propriedades válidas para a ordenação por inserção (cont.):
  - Manutenção: devemos demonstrar que cada iteração mantém o loop invariante. Informalmente, corpo do loop for exterior funciona deslocando-se A[j-1], A[j-2], e daí por diante, uma posição à direita, até ser encontrada a posição adequada para A[j] (linhas 4 a 7), e nesse ponto A[j] é inserido (linha 8). Um tratamento mais formal nos obrigaria a analisar o loop while interno, mas por ora não vamos nos prender a esse formalismo.

Algoritmos de Ordenação

- Vejamos como propriedades válidas para a ordenação por inserção (cont.):
  - Término: finalmente examinamos o que ocorre quando o loop termina. Loop for termina quando j excede n, ou seja, quando j = n + 1. Substituindo j por n+1 no enunciado do loop invariante, temos que o subarranjo A[1 .. n] consiste nos elementos originalmente contidos em A[1..n], mas em sequência ordenada. Contudo, o subarranjo A[1..n] é o arranjo inteiro! Desse modo, o arranjo inteiro é ordenado, o que significa que o algoritmo é correto.

Análise de algoritmos

- Análise da ordenação por inserção
  - Tempo de INSERTION-SORT depende da entrada
  - Mesmo para distintas entradas do mesmo tamanho, tempo pode variar
    - Depende de quanto entradas já ordenadas
  - Assim, tempo de execução é uma função do tamanho da entrada
    - Tamanho da entrada depende do problema que está sendo estudado
  - Tempo de execução: número de operações ou etapas executadas

Análise de algoritmos

- Análise da ordenação por inserção (cont.)
  - Por ora, custo de execução da i-ésima linha igual a c_i, onde c_i é uma constante
  - Loop executado n-1 vezes
    - Contador j iniciado em 2
    - Mas veja que instrução de teste para fim do loop é executado uma vez a mais
  - Veja ainda que número de vezes que loop de deslocamento é executado é uma função da entrada
    - Depende de quão ordenado vetor já esteja
    - t_j representa número de vezes que loop é executado na iteração j

Análise de algoritmos

- Análise da ordenação por inserção (cont.)

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j \leftarrow 2$ **to** *length*$[A]$	c_1	n
2 **do** $key \leftarrow A[j]$	c_2	$n-1$
3 ▷ Insert $A[j]$ into the sorted sequence $A[1 \,..\, j-1]$.	0	$n-1$
4 $i \leftarrow j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 **do** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7 $i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8 $A[i+1] \leftarrow key$	c_8	$n-1$

Análise de algoritmos

- Análise da ordenação por inserção (cont.)
  - Tempo de execução do algoritmo é a soma dos tempos de execução para cada instrução executada

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1)$$
$$+ c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1).$$

  - No melhor caso, quando arranjo já ordenado, número de vezes que teste do loop (t_j) executado igual a 1, para todo valor de j

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8). \end{aligned}$$

Análise de algoritmos

- Análise da ordenação por inserção (cont.)
  - Pior caso ocorre quando arranjo ordenado em ordem decrescente. Neste caso, t_j igual a j (lembre-se da comparação final para saída do loop)
  - Assim:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1)$$

$$+ c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1).$$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n-1)}{2} - 1\right)$$

$$+ c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1)$$

$$= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n$$

$$- (c_2 + c_4 + c_5 + c_8).$$

Análise de algoritmos

- Análise da ordenação por inserção (cont.)
  - Podemos observar que, no melhor caso, tempo de execução é uma função linear de n e, no pior caso, é uma função quadrática de n
  - A princípio vamos nos concentrar na descoberta do pior caso. Três motivos:
    - Por ser o limite superior de execução para qualquer entrada: algoritmo nunca demorará mais tempo
    - Pior caso ocorre com bastante frequência
    - Caso médio é quase tão ruim quanto pior caso e pode ser difícil calcular
      - Temos de levar em conta as probabilidades de ocorrência das instâncias

Análise de algoritmos

- Encontrar o tempo médio da busca de um elemento x numa lista de n elementos

Algoritmo: Busca_Seqüencial(L, x)

```
    L[n+1] = x;
     i = 1;
    enquanto L[i]  ≠ x faça
        i = i + 1;
    imprima i; // posição de x
```

- Tempo médio $T_M = \sum_{1 \le i \le m} p(E_i) T(E_i)$
  - m : número total de instâncias de tamanho n

Análise de algoritmos

- As instâncias podem ser classificadas em n+1 classes distintas:
  - E_1 = L tal que L[1] = x,
  - E_2 = L tal que L[2] = x , ...
  - E_n = L tal que L[n] = x,
  - E_{n+1} = L tal que x $\notin$ L
- Probabilidade de cada instância ocorrer
  - $p(E_i) = 1/(n+1), \ \forall \ i = 1,..., n+1$
- Tempo para cada instância (número de comparações)
  - $t(E_i) = i, \ \forall \ i = 1,...,n+1$

$$T_M(n) = \sum_{i=1}^{n+1} p(E_i).t(E_i) = \sum_{i=1}^{n+1} \frac{1}{(n+1)}.i = \frac{1}{(n+1)}.\frac{(n+1)(n+2)}{2} = \frac{n+2}{2} \text{ comparações}$$

Análise de algoritmos

- Análise da ordenação por inserção (cont.)
  - Podemos fazer mais simplificações na busca do tempo de execução de INSERTION_SORT
  - Já havíamos ignorado o custo real de cada instrução ao usarmos constantes c_i para representar esses custos
  - Tempo de execução no pior caso pode ser reescrito como $an^2 + bn + c$, onde a, b e c dependem dos custos de instrução c_i
  - Assim, podemos ignorar os próprios custos abstratos c_i
  - Entretanto, podemos considerar apenas o termo de mais alta ordem, an^2
    - Termos de baixa ordem insignificantes para grandes valores de n

Análise de algoritmos

- Análise da ordenação por inserção (cont.)
  - Da mesma forma, podemos ignorar a constante **a**
    - Visto que fatores constantes são menos significativos do que a taxa de crescimento na determinação da eficiência computacional para grandes entradas
    - Portanto, ordenação por inserção tem tempo de execução do pior caso igual a $\Theta(n^2)$
    - Algoritmos mais eficiente que outro se sua ordem de crescimento mais baixa

Crescimento de Funções

- Ordem de crescimento do tempo de execução de um algoritmo
  - Permite caracterização simples da eficiência do algoritmo
  - Permite comparar desempenho relativo de algoritmos alternativos
  - Podemos calcular tempo de execução exato, mas não vale o esforço: constantes multiplicativas e termos de mais baixa ordem dominados pelos efeitos do próprio tamanho da entrada

Crescimento de Funções

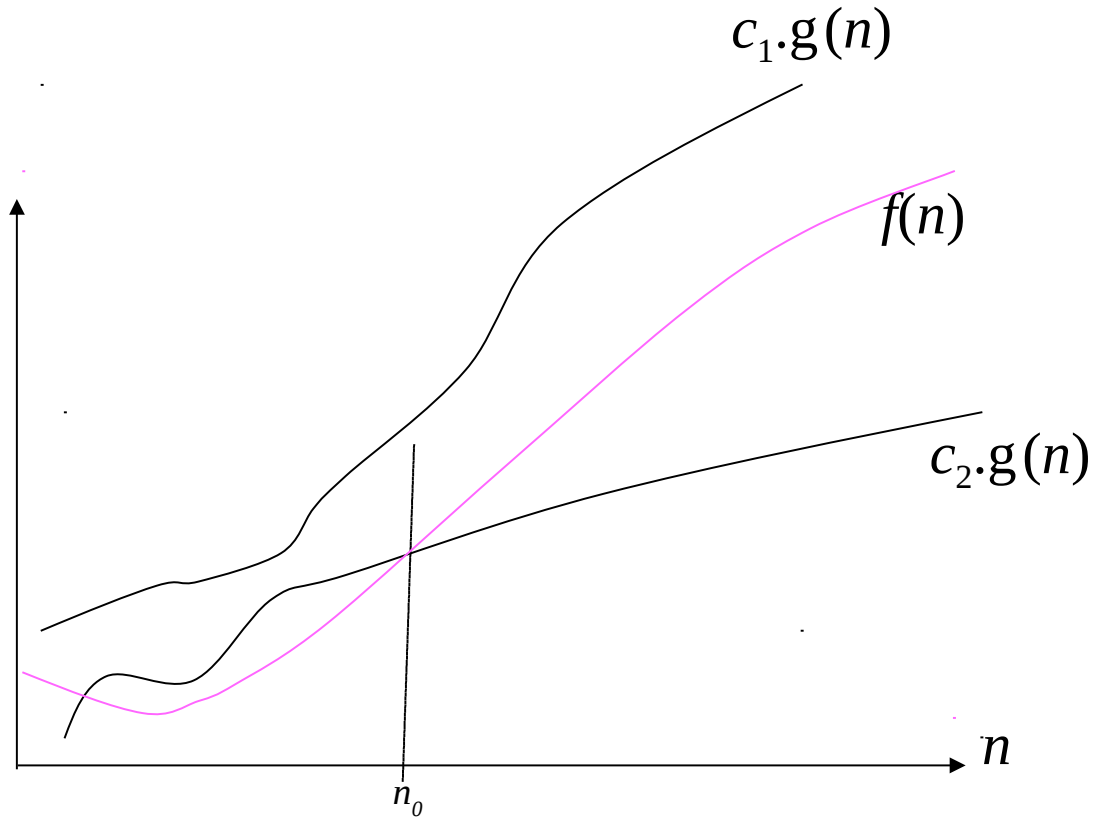
- Eficiência Assintótica: quando observamos apenas entradas grandes o suficiente
  - Estamos preocupados com modo como tempo de execução aumenta com tamanho de entrada no limite
  - Algoritmo assintoticamente mais eficiênte será a melhor escolha para todas as entradas, exceto talvez as muito pequenas
  - Utilizamos notação assintótica

Notação Assintótica

- Notação Θ
  - Denotamos por Θ(g(n)) o conjunto de funções

    Θ(g(n)) = { f(n): existem constantes positivas c_1, c_2, e n_0 tais que $0 <= c_1 g(n) <= f(n) <= c_2 g(n)$ para todo $n >= n_0$}
  - Ou seja, uma função f(n) pertence ao conjunto Θ(g(n)) se existem constantes c_1 e c_2 que a mantenham limitada entre $c_1.g(n)$ e $c_2.g(n)$ para um valor suficientemente grande de n
  - Em geral, usamos f(n) = Θ(g(n)) ao invés de f(n) ϵ Θ(g(n))

Notação Assintótica



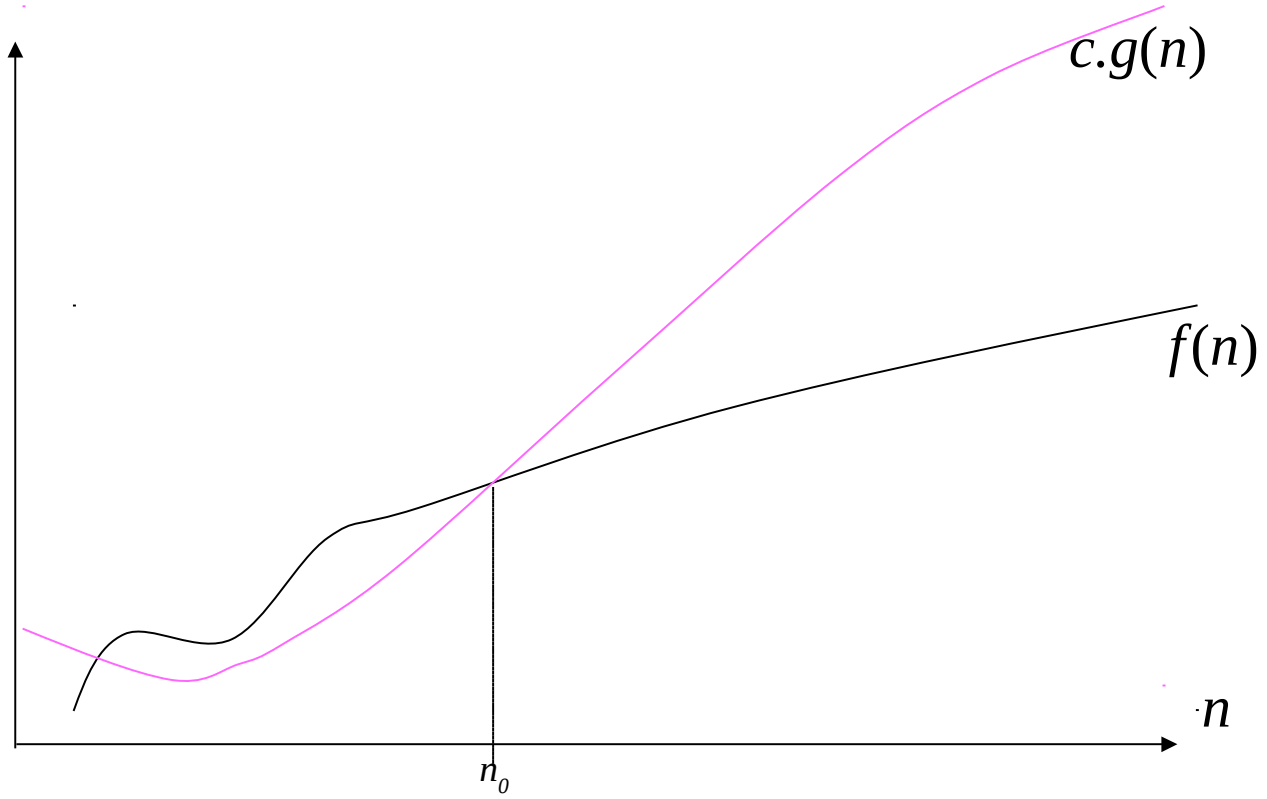
Notação Assintótica

- Notação Θ
  - Exemplo: $1/2.n^2 – 3n = \Theta(n^2)$
  - Para isso devemos definir constantes c_1, c_2 e n_0 tais que; $c_1.n^2 <= 1/2.n^2 – 3n <= c_2.n^2$ para todo $n >= n_0$
  - P. ex., podemos escolher os valores $c_1= 1/14$, $c_2 = 1/2$ e $n_0=7$
  - Existem outras opções para as constantes, mas o importante é que existe alguma opção

Notação Assintótica

- Notação O
  - Usada quando temos apenas um limite assintótico superior
  - Denotamos por O(g(n)) o conjunto de funções O(g(n)) = { f(n): existem constantes positivas c e n_0 tais que 0<= f(n) <= cg(n) para todo n>=n_0}
  - Ou seja, uma função f(n) pertence ao conjunto O(g(n)) se existe constante c que a mantenha limitada a c.g(n) para um valor suficientemente grande de n

Notação Assintótica



Notação Assintótica

- Notação O
  - Veja que f(n) = Θ (g(n)) implica f(n) = O(g(n))
  - Com a notação O, podemos descrever o tempo de execução de um algoritmo apenas inspecionando sua estrutura global

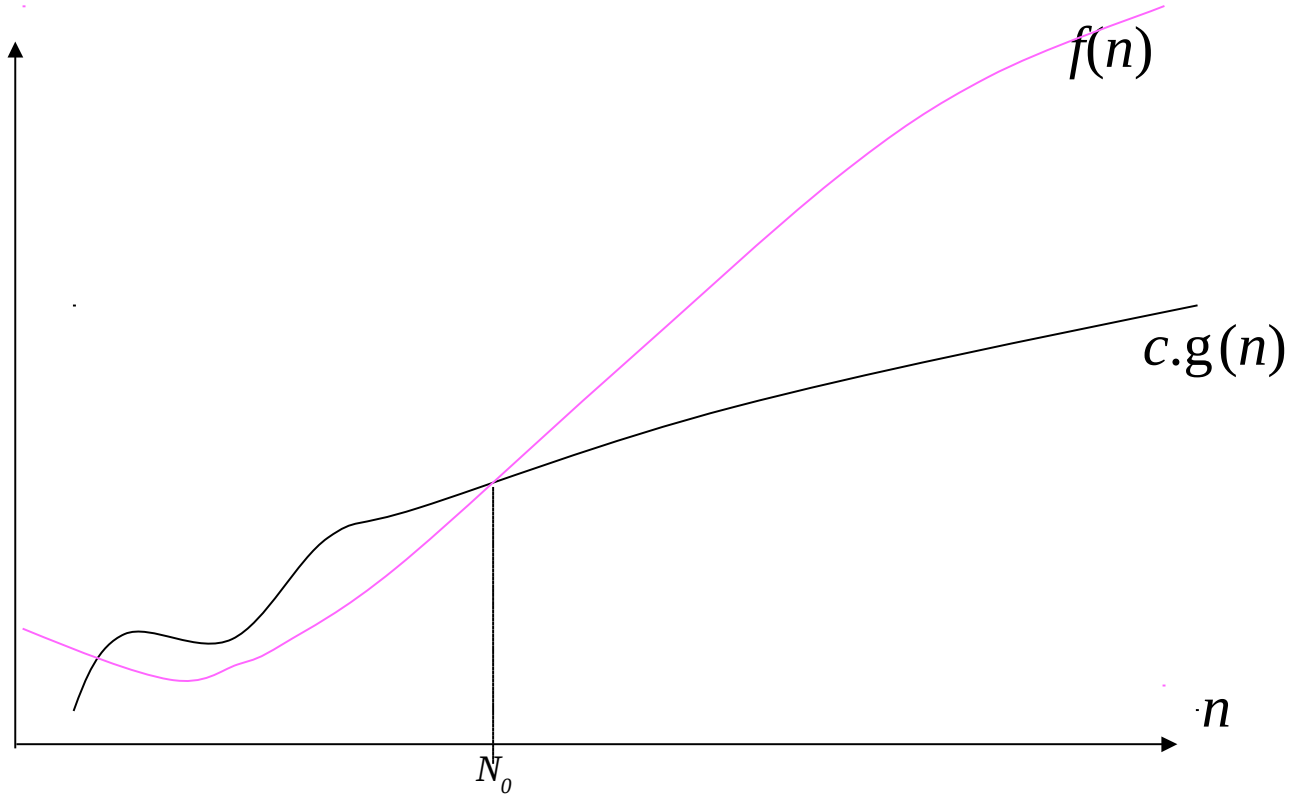
Notação Assintótica

- Exemplo
  - $T(n) = 3n^2 + 4n + 50$ é $O(n^2)$
  - Basta encontrar duas constantes c e N, tal que $T(n) \leq c.n^2, \ \forall n \geq N$
  - Com c = 57, temos $3n^2 + 4n + 50 \leq 57n^2, \ \forall n \geq 1 = N$
  - Pode-se mostrar também que $T(n) = 3n^2 + 4n + 50$ é $O(n^3)$ ou $O(n^4)$, entretanto estamos interessados no menor limite superior possível

Notação Assintótica

- Notação Ω
  - Fornece limite assintótico inferior
  - Denotamos por Ω(g(n)) o conjunto de funções Ω(g(n)) = { f(n): existem constantes positivas c e n_0 tais que 0<= cg(n) <= f(n) para todo n>=n_0}
  - Ou seja, uma função f(n) pertence ao conjunto Ω(g(n)) se existe constante c que limite c.g(n) a f(n) para um valor suficientemente grande de n

Notação Assintótica



Notação Assintótica

- Notação Ω
  - $T(n) = n^4 - n$ é $\Omega(n^4)$
  - Basta encontrar duas constantes c e n tal que $n^4 - 8n >= c.n^4$, para todo $n >= N$
  - Para $c = 1/2$, temos $n^4 - n >= 1/2.n^4$, para todo $n >= 3 = N$
  - Pode-se mostrar também que $T(n) = n^4 - 8n$ é $O(n^3)$ ou $O(n^2)$, entretanto estamos interessados no maior limite inferior

Notação Assintótica

- Para duas funções quaisquer f(n) e g(n), temos f(n) = Θ(g(n)) se e somente se f(n) = O(g(n)) e f(n) = Ω(g(n))

Notação Assintótica

- Operações com a notação O
  - $f(n) = O(f(n))$
  - $c.O(f(n)) = O(f(n))$, c = constante
  - $O(f(n)) + O(f(n)) = O(f(n))$
  - $O(O(f(n))) = O(f(n))$
  - $O(f(n)) + O(g(n)) = O(\max\{f(n), g(n)\})$
  - $O(f(n)).O(g(n)) = O(f(n).g(n))$
  - $f(n).O(g(n)) = O(f(n).g(n))$

Técnicas para Análise de Complexidade de Algoritmos

- Comandos simples, atribuições, incrementos, decrementos, ifs, elses
  - Tempo constante: O(1)
- Blocos for

for i = 1 to n do ..... n vezes

v[i] ← 0; ..... O(1)

- Complexidade (tempo): T(n) = n.O(1) = O(n)

for i = 1 to n do ..... n vezes

for j = 1 to n do ..... n vezes M[i][j] = 0; ..... O(1)

$O(n)$

- Complexidade T(n) = n.O(n) = $O(n^2)$

Técnicas para Análise de Complexidade de Algoritmos

- Teste

if < condição > then ..... O(1)

< Corpo do if > ..... O(T1(n))

else

< Corpo do else > ..... O(T2(n))

  - Complexidade T(n) = O(max{T1(n), T2(n))

- Exemplo

if M[i,i] == 0 then

```
for i = 1 to n do
    for  j = 1 to n M[i][ j] = 0;
```

$O(n^2)$

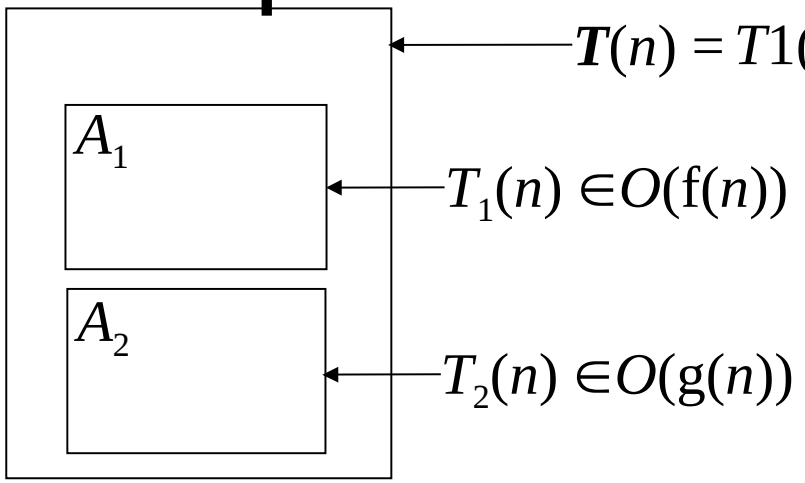
else

```
for i = 1 to n M[i][ j] = 1;
```

$O(n)$

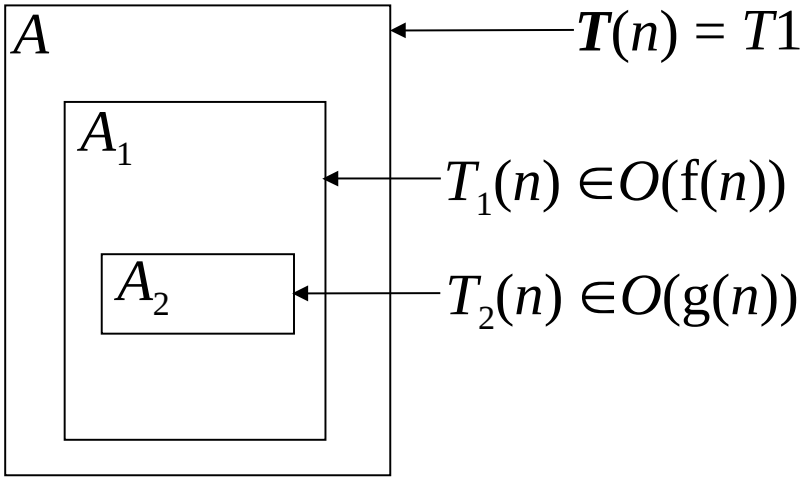
Complexidade T(n) = $O(n^2)$

Técnicas para Análise de Complexidade de Algoritmos



Regra das Somas

Se um algoritmo ***A*** se divide em duas partes independentes $\boldsymbol{A}_1$ e $\boldsymbol{A}_2$, onde $\boldsymbol{A}_1$ de complexidade $T_1(n) \in O(f(n))$ e $\boldsymbol{A}_2$ de complexidade $T_2(n) \in O(g(n)) \Rightarrow \boldsymbol{T}(n) = T1(n) + T2(n) = O(\max\{f(n), g(n)\})$ será a complexidade de ***A***.



Regra do Produto

Se um algoritmo ***A*** contém dois "aninhamentos" $\boldsymbol{A}_1$ e $\boldsymbol{A}_2$, onde $\boldsymbol{A}_1$ de complexidade $T_1(n) \in O(f(n))$ e $\boldsymbol{A}_2$ de complexidade $T_2(n) \in O(g(n))$

$\Rightarrow \boldsymbol{T}(n) = T1(n).T2(n) = O(f(n).g(n))$ será a complexidade de ***A***.

Técnicas para Análise de Complexidade de Algoritmos

- Exemplos

```
p = 0;                                   O(1)
for i = 1 to n do
    if mod(i, 2) = 0 then
        p = p + i*i;                     O(1)
```

n.O(1) = O(n)

Complexidade: T(n) = O(max{1, n}) = O(n)

```
para i=1 até n faça
    v[i]=i;
    para j = 2 até n faça
        p=v[i] * j;
```

O(max{1, n}) = O(n)

Complexidade: T(n) = n.O(n) = $O(n^2)$

Técnicas para Análise de Complexidade de Algoritmos

- Exemplos:

```
para i = 1 até n faça
    para j = 1 até n faça
        M[i, j] = 0;                                  O(1)
        para k = 1 até n faça                         O(n)
            M[i, j] = M[i, j] + A[i, k]* B[k, j];
```

$O(n^2)$

Complexidade: $T(n) = n.O(n^2) = O(n^3)$

Técnicas para Análise de Complexidade de Algoritmos

- Exemplos

```
if n = 0 then p = a[0];   O(1)
else
    p=a[0];               O(1)
    y=x;
    for i = 1 to n do     O(n)
       p = p + a[i] * y;
       y=y * x;
                                  O(n)
```

Complexidade: T(n) = O(n)

Técnicas para Análise de Complexidade de Algoritmos

- Exemplos

```
for i = 1 to n-1 do
  for j = n downto i+1 do
    if V[j-1] > V[ j] then
       temp = V[ j-1];
       V[j-1] = V[j];
       V[j] = temp;
```

O(1)

i	nº de vezes que o For interno é executado
1	n-1
2	n-2
3	n-3
⋮	⋮
n-1	1

Total: $\sum_{k=1}^{n-1} k = \frac{(n-1)n}{2} = O(n^2)$

Técnicas para Análise de Complexidade de Algoritmos

- Exemplos:

```
for i = 1 to n-1 do
    Soma = 0;
    for j = i to n do
        S =  S + V[j];
        if S > Smax  then
            Smax ← S;
            i_max ← i;
            j_max ← j;
```

O(1)

i	nº de vezes que o For interno é executado
1	n
2	n-1
3	n-2
⋮	⋮
n	1

Total: $\sum_{k=1}^{n} k = \frac{n(n+1)}{2} = O(n^2)$

Projeto de algoritmos

- Existem vários modos de projetar algoritmos
- Ordenação por inserção usou abordagem incremental
- Outra forma de projetar algoritmo de ordenação: usando abordagem “dividir e conquistar”
  - Desmembram problema em vários subproblemas semelhantes ao problema original, mas menores em tamanho
  - Problemas resolvidos recursivamente
    - Algoritmos chamam a si mesmos uma ou mais vezes para lidar com subproblemas intimamente relacionados
  - Soluções combinadas para criar uma solução para o problema original

Projeto de algoritmos

- Abordagem “dividir e conquistar” envolve três passos em cada nível da recursão
  - Dividir: desmembra problema em determinado número de subproblemas
  - Conquistar: resolve os problemas recursivamente
    - Se tamanhos dos subproblemas forem pequenos o bastante, problema resolvido de modo direto
  - Combinar: soluções obtidas combinadas para criar uma solução para o problema original

Projeto de algoritmos

- Algoritmo de ordenação por intercalação (merge sort) obedece o paradigma dividir e conquistar
  - Dividir: sequência de n elementos dividida em duas subsequências de n/2 elementos cada uma
  - Conquistar: classifica as duas subsequências recursivamente, utilizando a própria ordenação por intercalação
    - Quando sequência de comprimento um, não há trabalho a ser feito
  - Combinar: faz a intercalação das duas sequências ordenadas, de modo a produzir a resposta ordenada

Projeto de algoritmos

- Para executar a intercalação, usamos procedimento auxiliar MERGE(A, p, q, r)
  - A é o arranjo, e p, q e r são índices de enumeração dos elementos do arranjo tais que p <= q < r
  - Procedimento pressupõe que subarranjos A[p .. q] e A[q+1 .. r] estejam ordenados
  - Seu papel é o de intercala-los (mescla-los) para formar único subarranjo ordenado A[p .. r]
  - Procedimento leva tempo $\Theta(n)$, onde n = r – p + 1

Projeto de algoritmos

- MERGE(A, p, q, r) funciona como a seguir
  - Imagine duas pilhas de cartas ordenadas, com a face voltada para cima
  - Carta de menor valor em cima
  - Escolhe-se menor das duas cartas com a face voltada para baixo na pilha de saída
  - Repete-se essa operação até que uma das pilhas vazias
  - Demais cartas da outra pilha simplesmente colocadas sobre a pilha de saída
  - A cada passo computacional, estamos verificando apenas duas cartas superiores
  - Como executamos no máximo n passos básicos, intercalação demorará um tempo $\Theta(n)$

Projeto de algoritmos

```
MERGE(A, p, q, r)
 1  n1 ← q − p + 1
 2  n2 ← r − q
 3  create arrays L[1 .. n1 + 1] and R[1 .. n2 + 1]
 4  for i ← 1 to n1
 5        do L[i] ← A[p + i − 1]
 6  for j ← 1 to n2
 7        do R[j] ← A[q + j]
 8  L[n1 + 1] ← ∞
 9  R[n2 + 1] ← ∞
10  i ← 1
11  j ← 1
12  for k ← p to r
13        do if L[i] ≤ R[j]
14              then A[k] ← L[i]
15                   i ← i + 1
16              else A[k] ← R[j]
17                   j ← j + 1
```

Projeto de algoritmos

- MERGE(A, p, q, r) usa sentinela
  - Contém valor especial que empregamos para simplificar o código
  - No nosso caso, evita a necessidade de verificar se pilha está vazia
  - Valor infinito utilizado, de modo que ele nunca poderá ser o menor valor
    - A menos que ambas as pilhas tenham suas sentinelas expostas
    - Mas quando isso ocorre, todas as cartas já colocadas na pilha de saída
    - Contudo, como já sabemos que r-p+1 cartas serão colocadas sobre a pilha, podemos parar após esse número de passos

Projeto de algoritmos

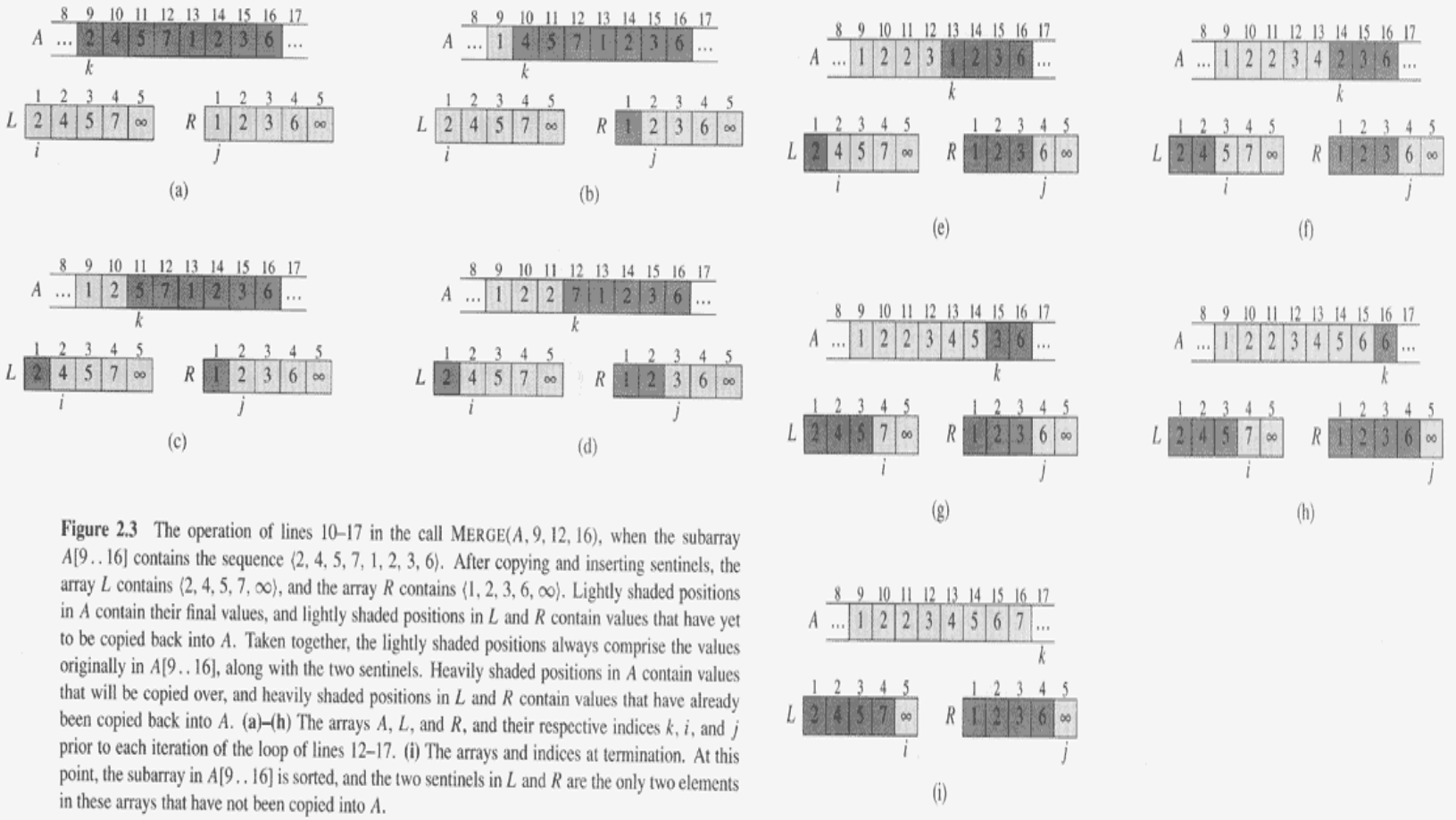


Figure 2.3 The operation of lines 10–17 in the call MERGE($A, 9, 12, 16$), when the subarray $A[9..16]$ contains the sequence $\langle 2, 4, 5, 7, 1, 2, 3, 6\rangle$. After copying and inserting sentinels, the array L contains $\langle 2, 4, 5, 7, \infty\rangle$, and the array R contains $\langle 1, 2, 3, 6, \infty\rangle$. Lightly shaded positions in A contain their final values, and lightly shaded positions in L and R contain values that have yet to be copied back into A. Taken together, the lightly shaded positions always comprise the values originally in $A[9..16]$, along with the two sentinels. Heavily shaded positions in A contain values that will be copied over, and heavily shaded positions in L and R contain values that have already been copied back into A. **(a)–(h)** The arrays A, L, and R, and their respective indices k, i, and j prior to each iteration of the loop of lines 12–17. **(i)** The arrays and indices at termination. At this point, the subarray in $A[9..16]$ is sorted, and the two sentinels in L and R are the only two elements in these arrays that have not been copied into A.

Projeto de algoritmos

- MERGE(A, p, q, r) mantém o loop invariante
  - No início de cada iteração do último loop for, subarranjo A[p..k-1] contém os k-p menores elementos de L[1..n_1+1] e R[1..n_2+1] em sequência ordenada
  - Inicialização: antes da primeira iteração do loop, k = p. Neste caso, subarranjo A[p .. k-1] está vazio, portanto contém os k-p=0 menores elementos de L e R e, como i = j = 1, tanto L[i] quanto R[j] são os menores elementos do arranjo, ainda não copiados de volta para A
  - Manutenção: vamos supor primeiro que L[i] <= R[j]. Então L[i] é o menor elemento ainda não copiado de volta em A. Como A[p..k-1] contém os k-p menores elementos, depois da linha 14 copiar L[i] em A[k], o subarranjo A[p..k] conterá os k-p+1 menores elementos. Incremento de k e i mantém loop invariante. Semelhante se L[i] > R[j].

Projeto de algoritmos

- MERGE(A, p, q, r) mantém o loop invariante (cont.)
  - Término: no término, k = r + 1. Pelo loop invariante, subarranjo A[p..k-1], ou seja, A[p..r], contém os k-p=r-p+1 menores elementos de L[1..n_1+1] e R[1..n_2+1] em sequência ordenada. Os arranjos L e R contêm juntos n_1+n_2+2 = q-p+1+r-q+2=r-p+3. Assim, todos os elementos, exceto os dois maiores, foram copiados de volta em A. Os dois maiores elementos são as sentinelas

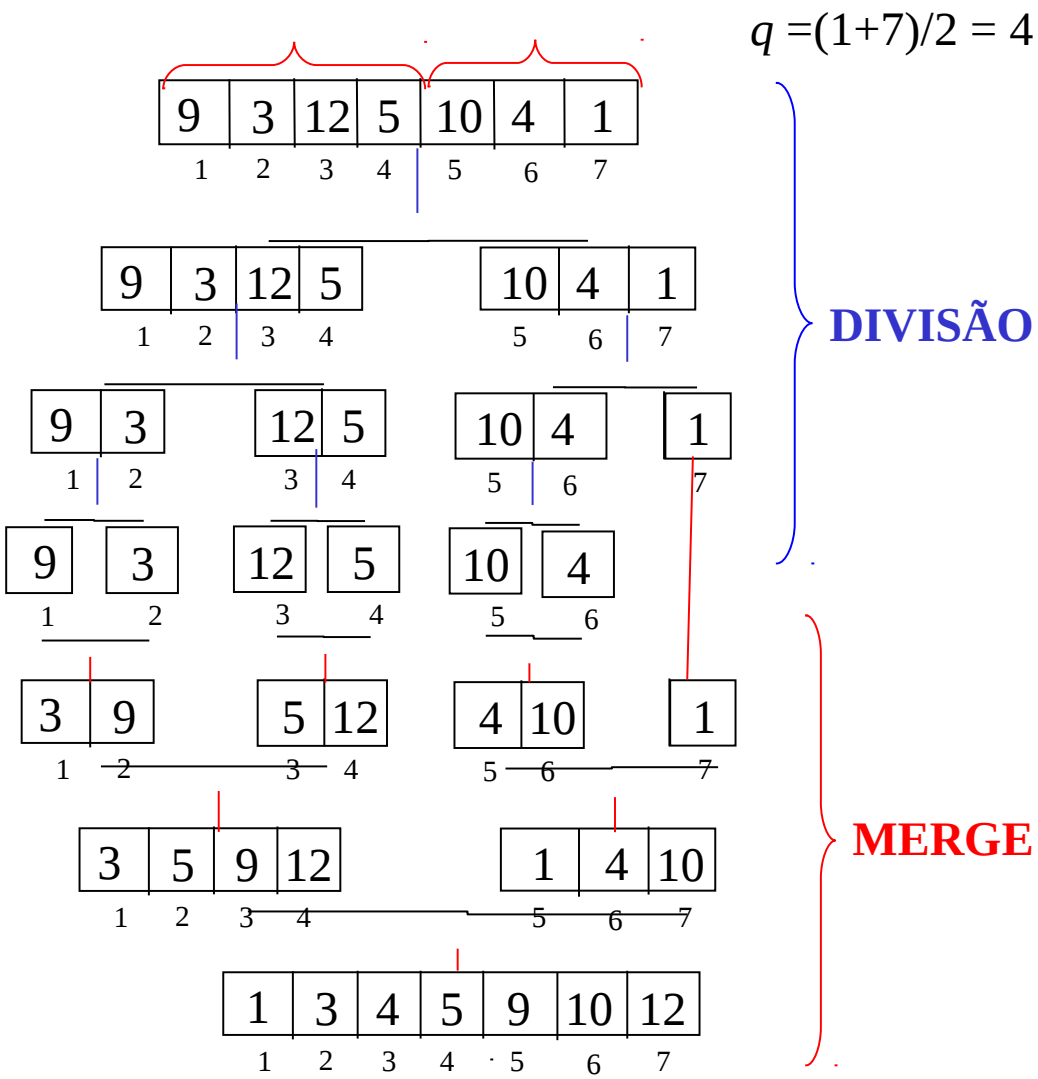
Projeto de algoritmos

- Podemos agora usar MERGE como sub-rotina no algoritmo de ordenação por intercalação
- MERGE-SORT(A,p,r) ordena os elementos de A[p..r]
- Se p >= r, subarranjo tem no máximo um elemento e consequentemente já está ordenado
- Caso contrário, etapa de divisão calcula índice q que divide A[p..r] em dois arranjos: A[p..q] e A[q+1..r]

Projeto de algoritmos

```
MERGE-SORT(A, p, r)
1   if p < r
2       then q ← ⌊(p + r)/2⌋
3            MERGE-SORT(A, p, q)
4            MERGE-SORT(A, q + 1, r)
5            MERGE(A, p, q, r)
```

Projeto de algoritmos



Projeto de algoritmos

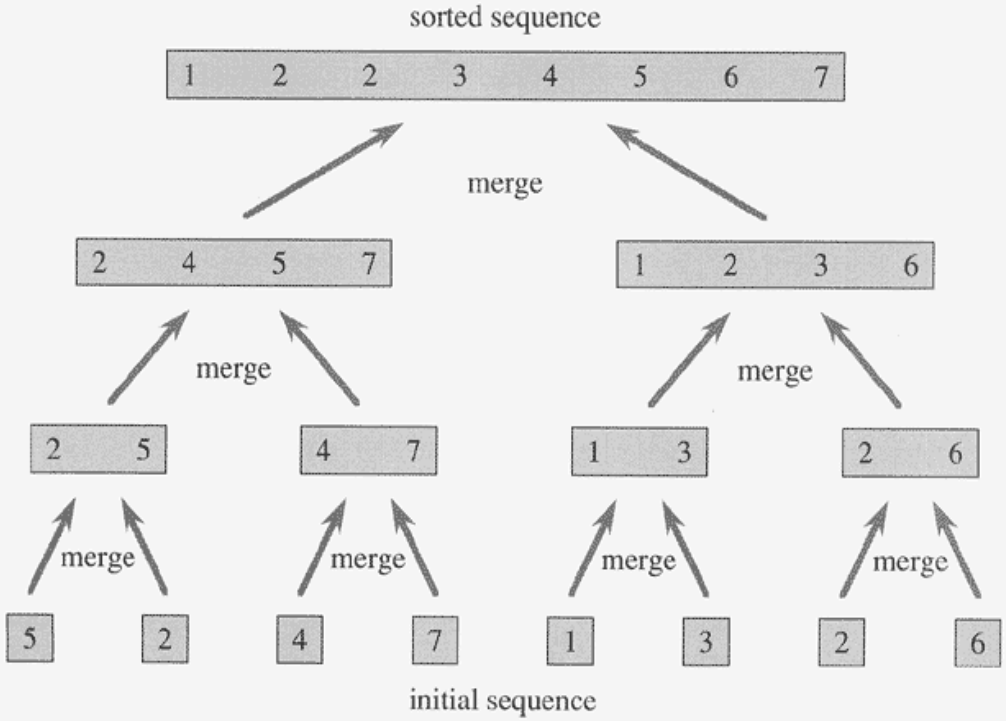


Figure 2.4 The operation of merge sort on the array $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$. The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.

Projeto de algoritmos

- Análise do algoritmo dividir e conquistar
  - Quando algoritmo contém chamada recursiva, seu tempo de execução frequentemente pode ser descrito por equação de recorrência
    - Descreve tempo de execução global sobre um problema de tamanho n em termos do tempo de execução sobre entradas menores
  - Recorrência no caso de algoritmo dividir e conquistar se baseia nos três passos do paradigma básico

Projeto de algoritmos

- Análise do algoritmo dividir e conquistar (cont.)
  - Se tamanho do problema for pequeno o bastante, p.ex. n <=c, a solução direta demorará um tempo constante
    - Consideraremos Θ=1
  - Vamos supor que problema seja dividido em a subproblemas, cada um dos quais com 1/b do tamanho original
    - No nosso caso, a = 2 e b = 2
  - Se tempo D(n) levado para dividir problema e C(n) para combinar soluções, obteremos a recorrência

Projeto de algoritmos

- Análise do algoritmo dividir e conquistar (cont.)

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c\,, \\ aT(n/b) + D(n) + C(n) & \text{otherwise}\,. \end{cases}$$

- Análise da ordenação por intercalação
  - MERGE-SORT funciona para número de elementos impares
  - Contudo, análise facilitada se fizermos suposição de que tamanho do problema é potência de dois
    - Cada passo de dividir produzirá duas subsequências de tamanho n/2

Projeto de algoritmos

- Análise da ordenação por intercalação (cont.)
  - Dividir: Etapa de dividir calcula ponto médio do subarranjo, o que demora um tempo constante. Logo, $D(n) = \Theta(1)$
  - Conquistar: Resolvemos recursivamente dois subproblemas, cada um com tamanho n/2. Eles contribuem com $2.T(n/2)$ para o tempo de execução. Logo $a = 2$ e $b = 2$.
  - Combinar: Já vimos que MERGE leva tempo $\Theta(n)$. Logo $C(n) = \Theta(n)$
  - Quando somamos C(n) e D(n), obtemos uma função linear de n, ou seja, $\Theta(n)$

Projeto de algoritmos

- Análise da ordenação por intercalação (cont.)

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

- Vamos por ora entender intuitivamente a solução da recorrência acima
- Vamos reescrever a recorrência como

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

- Onde c representa o tempo exigido para resolver problemas de tamanho 1, bem como para a etapa de combinar

Projeto de algoritmos

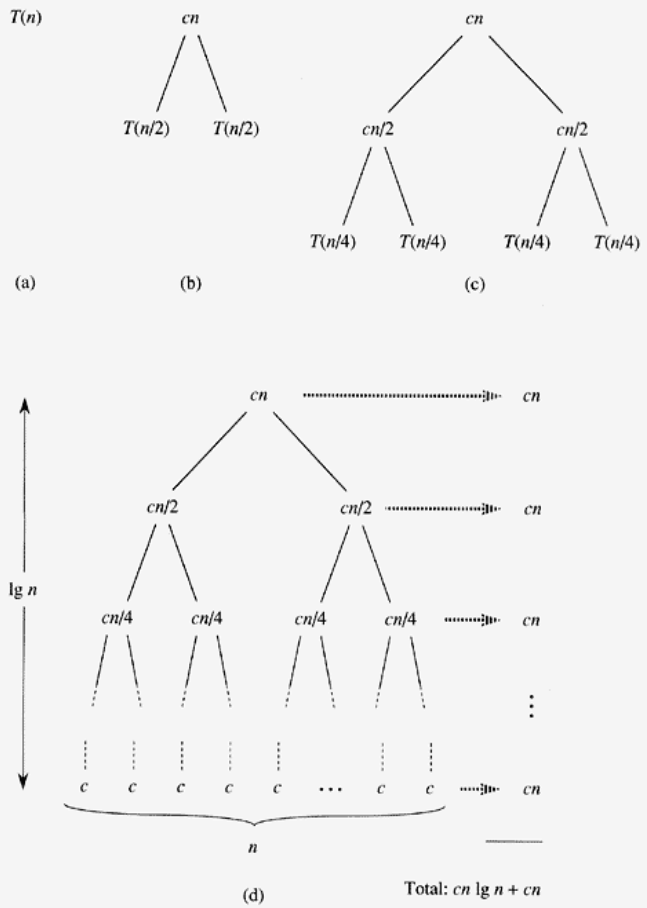


Figure 2.5 The construction of a recursion tree for the recurrence $T(n) = 2T(n/2) + cn$. Part **(a)** shows $T(n)$, which is progressively expanded in **(b)–(d)** to form the recursion tree. The fully expanded tree in part (d) has $\lg n + 1$ levels (i.e., it has height $\lg n$, as indicated), and each level contributes a total cost of cn. The total cost, therefore, is $cn \lg n + cn$, which is $\Theta(n \lg n)$.

Projeto de algoritmos

- Árvore têm lg n de altura, e lg n + 1 níveis
- Complexidade do algoritmo: cn * (lg n + 1) = cn.lg n + cn = Θ(n lg n)
- Complexidade menor do que do algoritmo de inserção, quando entrada n grande o suficiente

Técnicas para Análise de Complexidade de Algoritmos

- Algoritmo Recursivo
  - Contém, em sua descrição, uma ou mais chamadas a si mesmo
- Exemplo: calcular o fatorial de um número *n*

$$\text{Temos que : } n! = \begin{cases} 1, & \text{se } n = 0 \text{ (base da recursão)} \\ n(n-1)!, & \text{se } n > 0 \text{ (fórmula de recorrência)} \end{cases}$$

Análise de Algoritmos Recursivos

```
FAT(n : inteiro) : inteiro;
início
    se n = 0 então
        FAT = 1
    senão
        FAT =  n * FAT(n – 1);
fim;
```

Análise de Algoritmos Recursivos

```
FAT(n : inteiro) : inteiro;
início
    Se n = 0 então          O(1)
        FAT = 1             O(1)
    Senão
        FAT =  n * FAT(n – 1);
Fim;
```

$T(n) = ?$

$T(n\text{-}1)$

Se $n = 0$, $T(n) = 1$

Caso contrário, $T(n) = 1 + T(n\text{-}1)$

Análise de Algoritmos Recursivos

- Desenvolvendo a fórmula de recorrência

$$T(n) = \begin{cases} 1, & se \quad n = 0 \\ 1 + T(n-1), & se \quad n > 0 \end{cases}$$

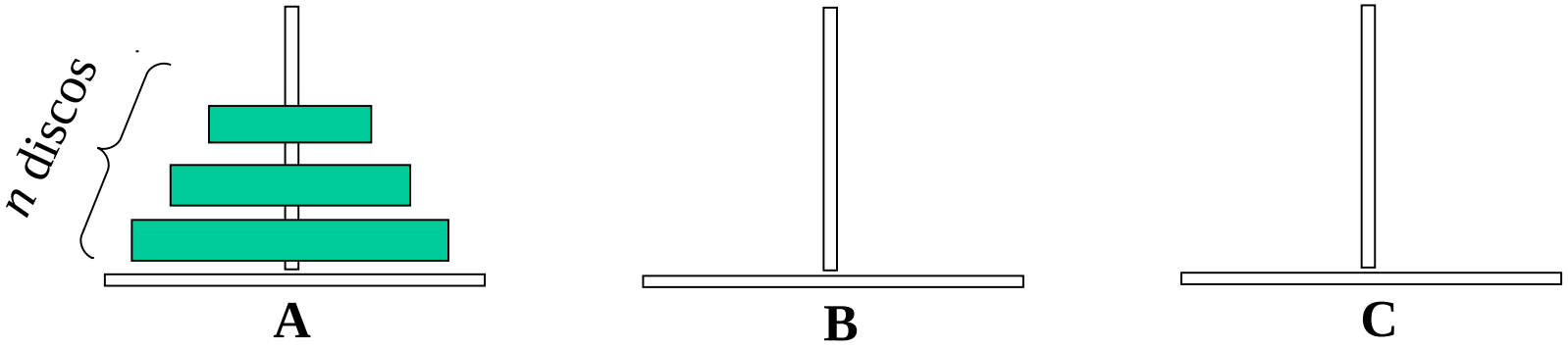
(condição de parada)

(fórmula de recorrência)

  - T(n) = 1 + T(n – 1) =
    1 + (1 +T(n – 2)) =
    1 + (1 +(1 +T(n – 3)))
  - Generalizando:
    - T(n) = k + T(n – k) ...(*)
    - Condição de parada: T(0) = 1,
    - Fazemos n – k = 0 → k = n
    - Substituindo em (*): T(n) = n + 1 = O(n)

Análise de Algoritmos Recursivos

- Exemplo: Torres de Hanói
  - Deslocar os n discos do pino A para o pino C usando o pino B
  - Só um disco pode ser movimentado de cada vez
  - Um disco maior não pode ser colocado sobre um disco menor
  - Realizar o menor número de movimentos



Análise de Algoritmos Recursivos

```
HANOI(n, A, C, B);
Início
    se n = 1 então move disco n de A para C      ----- O(1)
    senão
    HANOI(n-1, A, B, C);                         T(n-1)
     move disco n de A para C;
    HANOI(n-1, B, C, A);                         T(n-1)
Fim;
```

$T(n)$

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

Análise de Algoritmos Recursivos

- Desenvolvendo a fórmula de recorrência:
  - $T(n) = 2.T(n-1) + 1 =$

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

    $2^2.T(n-2) + 2 + 1 =$

    $2^3.T(n-3) + 2^2 + 2 + 1 =$

    .....

    $T(n) = 2^k.T(n-k) + 2^{k-1} + 2^{k-2} + ... + 2 + 1 \quad (*)$
  - Condição de parada: $n - k = 1 \rightarrow k = n - 1$
  - Substituindo em (*):

    $T(n) = 2^{n-1} + 2^{n-2} + ... + 2^1 + 2^0$. Por soma dos termos da PG: $T(n) = 2^n - 1 = O(2^n)$

Análise de Algoritmos Recursivos

```
MERGE-SORT(A, p, r)
1   if p < r
2       then q ← ⌊(p + r)/2⌋
3            MERGE-SORT(A, p, q)
4            MERGE-SORT(A, q + 1, r)
5            MERGE(A, p, q, r)
```

Algoritmo de Ordenação por Intercalação

- Assim:
  - $T(1) = 0$, se $n = 1$ (base da recursão)
  - $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n$, se $n > 1$ (fórmula de recorrência)
- Para n potência de dois temos: $T(\lceil n/2 \rceil) = T(\lfloor n/2 \rfloor) = T(n/2)$. Assim:

$$T(n) = 2.T(n/2) + n = 2.[\ 2.T(n/4) + n/2] + n =$$
$$= 2.[\ 2.\{2.T(n/8) + n/4\} + n/2] + n$$
$$= 8T(n/8) + 3n$$
$$= 2^3.T(n/2^3) + 3n = \ldots\ldots = 2^k.\ T(n/2^k) + kn$$

Condição de parada: $n/2^k = 1 \Rightarrow$ (aplicando $\log_2$) $k = \lg n$

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

Algoritmo de Ordenação por Intercalação

- Para qualquer n temos:
  - $2^{k-1} \leq n \leq 2^k$, $k > 0$
  - Se $n \leq 2^k$: $T(n) \leq T(2^k) = 2^k.\log 2^k$ (**)
  - Se $2^{k-1} \leq n$: $\log 2^{k-1} \leq \log n$
    - $k - 1 \leq \log n$
    - $k \leq \log n + 1$
  - Substituindo em (**): $T(n) \leq 2^{\log n + 1}.\log 2^{\log n + 1} = 2n\log n + 2n$

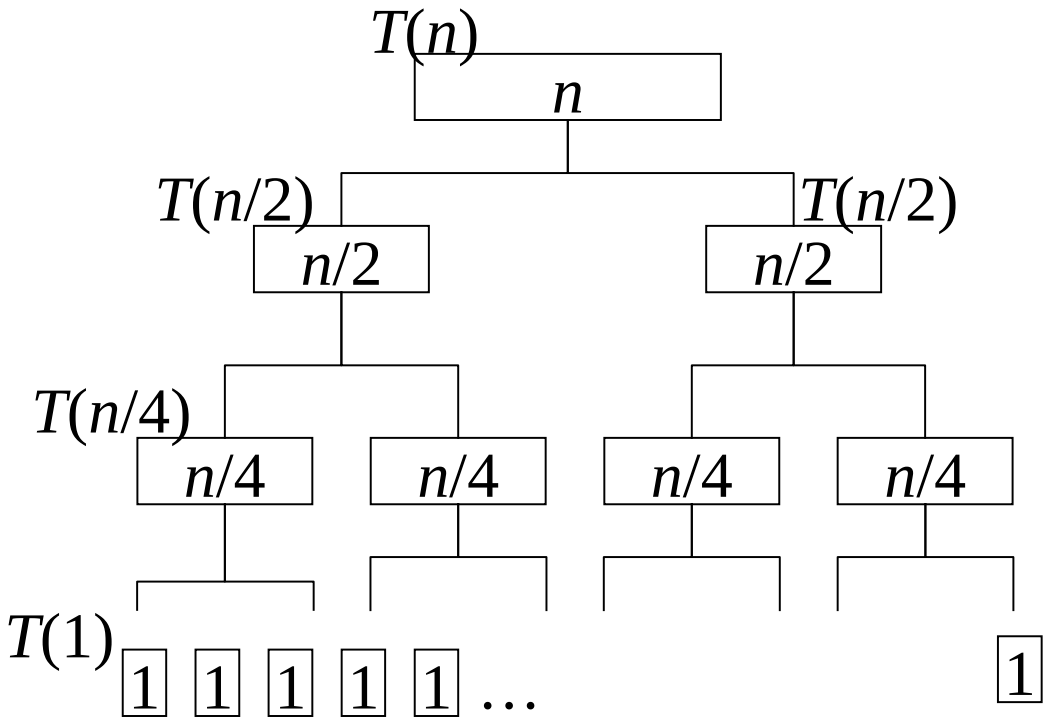
$\therefore \forall$ n inteiro, $T(n) = O(n\log n)$

Análise de Algoritmos Recursivos

- Árvore de Recursão
  - Maneira gráfica de visualizar a estrutura de chamadas recursivas do algoritmo
  - Cada nó da árvore é uma instância (chamada recursiva)
  - Se uma instância chama outras, estas são representadas como nós-filhos
  - Cada nó é rotulado com o tempo gasto apenas nas operações locais (sem contar as chamadas recursivas)

Análise de Algoritmos Recursivos

- Vamos revisitar o exemplo do Mergesort



$$
\begin{aligned}
&= n \\
&+ \\
2(n/2) &= n \\
&+ \\
4(n/4) &= n \\
&+ \\
n\,(n/n) &= n \\
\hline
n\,(\log_2 n + 1)
\end{aligned}
$$

Análise de Algoritmos Recursivos

- Observamos que cada nível de recursão efetua no total n passos
- Como há lg n + 1 níveis de recursão, o tempo total é dado por n lg n + n, o mesmo que encontramos na solução por iteração

Análise de Algoritmos Recursivos

- Teorema Mestre (simplificado)
  - Fórmulas de recorrência provenientes de algoritmos do tipo Dividir-para-Conquistar são muito semelhantes
  - Algoritmos tendem a dividir o problema em a partes iguais, cada uma de tamanho b vezes menor que o problema original
  - Quando trabalho executado em cada instância da recursão é uma potência de n, pode-se usar teorema que nos dá diretamente a complexidade assintótica do algoritmo
  - Teorema Mestre pode resolver recorrências cujo caso geral é da forma $T(n)=a\,T\,(n/b) + n^k$

Análise de Algoritmos Recursivos

- Teorema Mestre (cont).
  - Dadas as constantes a >= 1 e b >= 1 e uma recorrência da forma $T(n)=a\ T\ (n/b) + n^k$, então:
    - Caso 1: se $a > b^k$ então $T(n) = \Theta(n^{\log_b a})$
    - Caso 2: se $a = b^k$ então $T(n) = \Theta(n^k \log n)$
    - Caso 3: se $a < b^k$ então $T(n) = \Theta(n^k)$
  - Assumimos que n é uma potência de b e que o caso base T(1) tem complexidade constante

Análise de Algoritmos Recursivos

- Teorema Mestre: Exemplos
  - MergeSort: $T(n)=2T(n/2)+ n$
    - a=2, b=2, k=1.
    - caso 2 se aplica e $T(n) = \Theta(n \log n)$
  - $T(n)=3T(n/2)+ n^2$
    - a=3, b=2, k=2
    - caso 3 se aplica ($3<2^2$) e $T(n) = \Theta(n^2)$
  - $T(n)=2T(n/2)+ n \log n$
    - Teorema mestre (simplificado ou completo) não se aplica
    - Pode ser resolvida por iteração

Próxima aula...

Algoritmos de Ordenação

- Heapsort