**Resolução Exercícios do Livro Algoritmos 3 edição Thomas Cormen.**

**Capítulo 1**

Exercício

1.1-1 – Dê um exemplo do mundo real que exige a classificação ou um exemplo do mundo real que requer a computação de um casco convexo.

R.

Um exemplo real que exija ordenação pode ser qualquer tipo de catálogo. Um catálogo de livros, por exemplo, precisa ter vários tipos de livros. Mas para que as pessoas possam encontrar o livro desejado é necessário ordenar este catálogo de alguma maneira. Por exemplo, área da computação teriam todos os livros de computação. E dentro desta área poderia ser ordenados por nome.

Localização da envoltória convexa de um conjunto finito de pontos é o menor polígono/poliedro convexo que contém todos os pontos desse conjunto. Pode ser definida também como inserção de todos os polígonos/poliedros que cont6em os pontos. Sabendo disso, podemos dar como exemplo seu uso em computação gráfica e animação, para detecção de colisão de objetos.

ordenação: Listagem de alunos ou produtos por nome, ou código ou sobrenome, ou fabricante

determinação da melhor ordem para multiplicação de matrizes: Para solução sistema de equações lineares, utilizando método de Gauss

localização da envoltória convexa: topografia, determinar o terreno que envolve os pontos levantados, Curvas de nível

1.1-2 Além da velocidade, que outras medidas de eficiência poderiam ser usadas em uma configuração real?

R.

Tomemos como exemplo um programa que exija muito espaço em disco, outro que exija pouco. Programas que exijam mais ou menos memória. Executem em mais ou menos tempo. Todas essas medidas de eficiência podem ser usadas em uma configuração real.

1.1-3 Selecione uma estrutura de dados que você já tenha visto antes e discuta seus pontos fontes e suas limitações.

R.

Um exemplo de estrutura de dados é um vetor. Um vetor é uma estrutura que armazena objetos do mesmo tipo sequencialmente, em posições consecutivas de memória RAM.

Pontos fortes são: rapidez de acesso devido ao tempo constante para acesso de qualquer posição.

Pontos fracos: alocação estática, tamanho predefinido.

1.1-4 Em que aspectos os problemas de caminho mais curso e do caixeiro-viajante anteriores são semelhantes? Em que aspectos eles são diferentes?

R.

O problema do caminho mais curto consiste na minimização do custo do caminho entre dois vértices(ou nós) de um grafo, custo calculado pela soma dos pesos de cada aresta percorrida.

Aspectos semelhantes: a dificuldade para encontrar a solução mais eficiente esta no tamanho da quantidade de pontos, cidades que se queira analisar, quanto maior o número de cidades, pontos do problema mais difícil fica a solução do problema.

o problema do caminho mais curto pertence a classe P

o problema do caixeiro viajante pertence a classe NP Completo

1.1-5 Mostre um problema real no qual apenas a melhor solução servirá. Em seguida, apresente um problema em que baste uma solução que seja “apropriadamente” a melhor.

R. Podemos utilizar como exemplo de problema real onde apenas a melhor solução serve a ordenação de um catálogo. Se o catálogo não ordenado corretamente, o item procurado não será encontrado na posição desejada.

Solução Ótima: Quantidade de dinheiro necessária para quitar as dívidas do dia (contas a pagar)

Solução Aproximada: Encontrar o zero de uma função utilizando o método das secantes.

1.2.1 Forneça um exemplo de aplicação que exige conteúdo algorítmico no nível de aplicação e discuta a função dos algoritmos envolvidos.

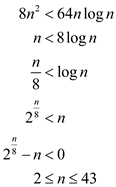
R. Consideremos um caixa eletrônico. Caixas eletrônicos processam transações financeiras de vários clientes de um banco. Assim, essas transações devem ser processadas usando algoritmos eficientes para que o sistema não caia quando um ou mais consumidores realizem transações. Além disso, algoritmos são necessários para o processamento das transações de modo que os dados de um cliente possam ser acessados somente por ele e não afetem os dados de nenhum outro cliente.

Caixas eletrônicos também precisam armazenar dados temporariamente de modo a permitir que as transações feitas pelos clientes seja processadas corretamente, e deletar esses dados ao fim das mesmas. Estes processos são feitos utilizando algoritmos eficientes.

Desse modo, os caixas eletrônicos precisam de algoritmos de ordenação e de processamento para funcionarem corretamente.

1.2.2 Vamos supor que estamos comparando implementações de ordenação por inserção e ordenação por intercalação na mesma máquina. Para entradas de tamanho n, a ordenação por inserção é executada em 8n^2 etapas, enquanto a ordenação por intercalação é executada em 64n log n etapas. Para que valores de n a ordenação por inserção supera a ordenação por intercalação.

R. Para calcularmos essa questão, devemos escreve-la em forma de inequações e calcular seu resultado. Como queremos saber os valores de n para os quais a ordenação de inserção supera a ordenação por intercalação temos:

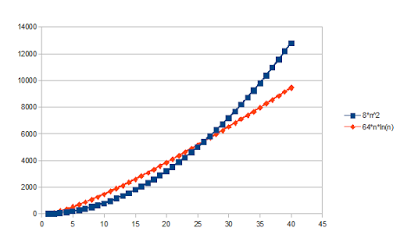


Ou

Elaborando uma tabela com três colunas, n, tempo para o algoritmo por inserção e o tempo para o algoritmo por intercalação tem-se a tabela abaixo. O algoritmo por inserção tem menos etapas para n variando de 1 a 43, depois disso, ou seja, para n>43 o algoritmo por intercalação tem menos etapas. Foi utilizado lg = log na base 2.

Tabela contendo os possíveis valores de *n* até que se encontre o ponto onde a ordenação por inserção supera a ordenação por intercalação:

|  |  |  |
| --- | --- | --- |
| **Valor de n** | **Inserção (8n2)** | **Intercalação (64n Ig n)** |
| 1 | 8,00 | - |
| **2** | **32,00** | **128,00** |
| 3 | 72,00 | 304,31 |
| 4 | 128,00 | 512,00 |
| 5 | 200,00 | 743,02 |
| 6 | 288,00 | 992,63 |
| 7 | 392,00 | 1.257,70 |
| 8 | 512,00 | 1.536,00 |
| 9 | 648,00 | 1.825,88 |
| 10 | 800,00 | 2.126,03 |
| 11 | 968,00 | 2.435,44 |
| 12 | 1.152,00 | 2.753,25 |
| 13 | 1.352,00 | 3.078,77 |
| 14 | 1.568,00 | 3.411,39 |
| 15 | 1.800,00 | 3.750,61 |
| 16 | 2.048,00 | 4.096,00 |
| 17 | 2.312,00 | 4.447,16 |
| 18 | 2.592,00 | 4.803,75 |
| 19 | 2.888,00 | 5.165,48 |
| 20 | 3.200,00 | 5.532,07 |
| 21 | 3.528,00 | 5.903,27 |
| 22 | 3.872,00 | 6.278,88 |
| 23 | 4.232,00 | 6.658,68 |
| 24 | 4.608,00 | 7.042,50 |
| 25 | 5.000,00 | 7.430,17 |
| 26 | 5.408,00 | 7.821,53 |
| 27 | 5.832,00 | 8.216,45 |
| 28 | 6.272,00 | 8.614,78 |
| 29 | 6.728,00 | 9.016,41 |
| 30 | 7.200,00 | 9.421,23 |
| 31 | 7.688,00 | 9.829,13 |
| 32 | 8.192,00 | 10.240,00 |
| 33 | 8.712,00 | 10.653,76 |
| 34 | 9.248,00 | 11.070,32 |
| 35 | 9.800,00 | 11.489,59 |
| 36 | 10.368,00 | 11.911,51 |
| 37 | 10.952,00 | 12.335,99 |
| 38 | 11.552,00 | 12.762,96 |
| 39 | 12.168,00 | 13.192,36 |
| 40 | 12.800,00 | 13.624,14 |
| 41 | 13.448,00 | 14.058,22 |
| 42 | 14.112,00 | 14.494,55 |
| **43** | **14.792,00** | **14.933,08** |



1.2-3 Qual é o menor valor de n tal que um algoritmo cujo tempo de execução é 100n^2 funciona mais rápido que um algoritmo cujo tempo de execução é 2^n na mesma máquina?

R.

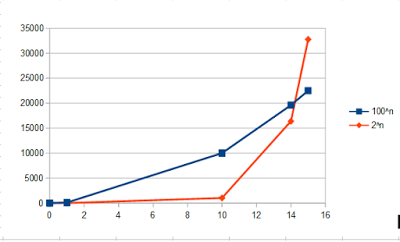
Devemos encontrar o menor valor de n para qual a seguinte inequação é verdadeira.

100m^2< 2^n

Elaborando uma tabela com três colunas, n, tempo para o algoritmo 1 e o tempo para o algoritmo 2 tem-se a tabela abaixo. O algoritmo 2 é melhor até n=14, a partir de n=15 o algoritmo 1 tem melhor desempenho

.

|  |  |  |
| --- | --- | --- |
| n | 100^n | 2^n |
| 0 | 0 | 1 |
| 1 | 100 | 2 |
| 10 | 10000 | 1024 |
| 14 | 19600 | 16384 |
| **15** | **22500** | **32768** |

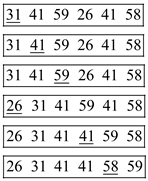


**Capítulo 2**

2.1-1 Ilustre a operação de ordenação por intercalação sobre o arranjo A = {3, 41, 52, 26, 38, 57, 9, 49}.

R. Sabemos que o **insertion-sort** funciona da seguinte maneira: o menor elemento é colocado na primeira posição a esquerda de um vetor, “empurrando” os outros elementos para a direita, e assim sucessivamente até que o vetor esteja ordenado.

A partir do arranjo A = {3, 41, 52, 26, 38, 57, 9, 49}.



Portanto, teremos ao fim da operação de **insertion-sort** o arranjo ordenado:

https://chegg-html-solutions.s3.amazonaws.com/9788535236996/719463-2.1-1E-i3.png

2.1-2 Reescreva o procedimento **Insertion-Sort** de modo que ele não utilize sentinelas e, em vez disso, se interrompa depois que o arranjo L ou R tiver todos os seus elementos copiados de volta para A, e então copie o restante do outro arranjo de volta em A.

R. Para alterarmos o procedimento de ordenar em ordem não crescente ao invés de ordem não decrescentes, basta que invertamos a comparação dos elementos do vetor como o menor elemento que é armazenado na quinta linha do pseudocódigo.

R.

INSERTION-SORT(A)

1 for j = 2 to A.length

2 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 A[i+1] = A[i]

7 i = i - 1

8 A[i + 1] = key

2.1-3 Considere o problema de pesquisa:

**Entrada**: Uma sequência de n números A = (a1, a2, ..., an) e um valor v.

**Saída**: Um índice i tal que v = A[i] ou o valor especial NIL, se v não aparecer em A.

Escreva um pseudocódigo para pesquisa linear, que faça a varredura da sequencia, procurando por v. Usando um loop invariante, prove que seu algoritmo é correto. Certifique-se de que seu loop invariante satisfaz às três propriedades necessárias.

R.

A seguir, encontra-se o pseudocódigo para a pesquisa linear:

bucaLinear(A, v)

1 for i = 1 to A.length do

2 if A [i] = v then

3 return i

4 return NIL

Seu loop invariante é:

Loop Invariante:

( ( v = A ) e ( i < A.length ) ) ou ( v ∉ A [1 .. i - 1] )

O invariante de laço pode ser definido como uma relação entre as variáveis de um algoritmo (programa) que é verdadeira nas seguintes condições:

antes do início do laço;

durante a manutenção do laço;

na saída do laço.

**Inicialização**

Inicialmente, o subarray é um vetor vazio, provando que é trivial.

**Manutenção**

Em cada passo, sabemos que A[1..i-1] não contém v. Comparamos v com A [i]. Se forem iguais, retornamos i, o que é um resultado correto. Caso contrário, continuamos para o próximo passo. Asseguramos também que A[A..i-1] não contém v e que A [i] é diferente de v, então este passo preserva o invariante.

**Terminação**

O loop termina quando i> A.length. Uma vez que i aumenta em 1 e i> A.length, sabemos que todos os elementos em A foram verificados e verificou-se que v não está entre eles. Assim, devolvemos NIL.

2.1-4 Considere o problema de somar dois inteiros binários de n bits, armazenados em dois arranjos de n elementos A e B. A soma dos dois inteiros deve ser armazenada em forma binária em um arranjo de (n + 1) elementos C. Enuncie o problema de modo formal e escreva o pseudocódigo para somar os dois inteiros.

R.

O problema dado pode ser enunciado formalmente da seguinte maneira:

**Entrada** : dois vetores, A e B, como mostrados abaixo, onde cada vetor representa um inteiro em formato binário(0 ou 1) e cada inteiro tem o comprimento de n.

https://word-to-html-images.s3.amazonaws.com/9788535236996/719463-2.1-4e-i3.png

**Saída** : Um vetor C representando um inteiro em formato binário (0 ou 1), onde C=A + B.

https://word-to-html-images.s3.amazonaws.com/9788535236996/719463-2.1-4e-i4.png

Escreva o algoritmo para a soma de dois inteiros binários de n bits, os quais estão armazenados em dois arranjos (A e B) em forma binária, onde o resultado deve ser armazenado em forma binária no arranjo C de n + 1 elementos.

SOMABINARIOS(A, B):

C : vetor[A.length + 1]

resultado = 0

for i = 1 to A.length

C[i] = (A[i] + B[i] + resultado) % 2 // resto da divisão

resultado = (A[i] + B[i] + resultado) / 2 // quociente

C[i] = resultado

return C

A complexidade é linear, pois passa uma vez por cada elemento do vetor.

O(n).

2.2-1 Expresse a função n^3/100 – 100m^2 -100n+3 em termos da notação Θ(theta)

R.

Para escrevermos uma função em termos da notação Θ(theta) costumamos considerar a taxa de crescimento do tempo de execução, que é dada pelo maior termo. Na função data, o maior termo é n^3.

Portanto a função data reescrita em termos da notação Θ é Θ (n^3)

2.2-2 Considere a ordenação de n números armazenados no arranjo A, localizando primeiro o menor elemento de A e permutando esse elemento com o elemento contido em A[1]. Em seguida, encontre o segundo menor elemento de A e o troque pelo elemento A[2]. Continue dessa maneira para os primeiros n − 1 elementos de A. Escreva o pseudocódigo(a) para esse algoritmo, conhecido como ordenação por seleção. Que loop invariante esse algoritmo mantém? (b) Por que ele só precisa ser executado para os primeiros n−1 elementos e não para todos os n elementos?(c) Forneça os tempos de execução do melhor caso e do pior caso da ordenação por seleção em notação Θ(d).

R.

a) Segue abaixo o pseudocódigo para o algoritmo de ordenação por seleção:

1 SELECTION-SORT(A):

2 for i = 1 to A.length - 1

3 min = i

4 for j = i + 1 to A.length

5 if A[j] < A[min]

6 min = j

7 temp = A[i]

8 A[i] = A[min]

9 A[min] = temp

b) No início de cada iteração do loop externo para, o subvetor A[1..i - 1] contém os menores elementos i - 1 do vetor, classificados em ordem não decrescente.

E:

No início de cada iteração do loop interno para, A [min] é o menor número no subvertor A

[i..j - 1] .

c) Porque trata-se de um algoritmo baseado na comparação de um elemento chave com seus vizinhos de modo que se possa selecionar o menor elemento do subconjunto em questão e se realizar uma troca. Quando resta apenas um último elemento a ser ordenado, já não há mais outros elementos com os quais comparar, além disto, este último elemento já ocupa a sua posição correta na ordenação de maior elemento.

d)

No melhor caso (o vetor está ordenado), o corpo do if nunca é invocado. O número de operações (contando a comparação como uma operação) é:

(n - 1)/ {n + 2} {2} + 4)

No pior caso (a matriz é invertida), o corpo do if é invocado em todas as ocasiões, o que duplica o número de etapas no circuito interno, ou seja:

(n - 1) (n + 6)

Ambos são claramente Θ (n^2) .

2.3-3 Considere mais uma vez a pesquisa linear(ver exercício 2.1-3). Quantos elementos da sequencia de entrada precisam ser verificados em média, supondo-se que o elemento que está sendo procurado tenha a mesma probabilidade de ser qualquer elemento do arranjo (matriz)? E no pior caso? Quais são os tempos de execução do caso médio e pior caso de pesquisa linear em notação Θ? Justificar suas respostas

R. Em média a busca linear precisa chegar n/2 elementos. Isso ocorre porque cada elemento na sequência tem a mesma chance de ser o elemento procurado. Assim, para o primeiro elemento o número de passos seria 1, para o segundo 2, e assim por diante até n.

Se o elemento estiver presente na sequencia, é provável que a metade dos elementos seja verificada antes de ser encontrada no caso médio. No pior dos casos, todos são verificados. Ou seja n/2 para o caso médio e n para o pior dos casos. Ambos tem a notação Θ(n).

2.2-4 Como podemos modificar praticamente qualquer algoritmo para ter um bom tempo de execução de melhor dos casos?

R.

Para modificarmos qualquer algoritmo de forma que tenha bom tempo de execução no melhor caso devemos considerar alguns pontos:

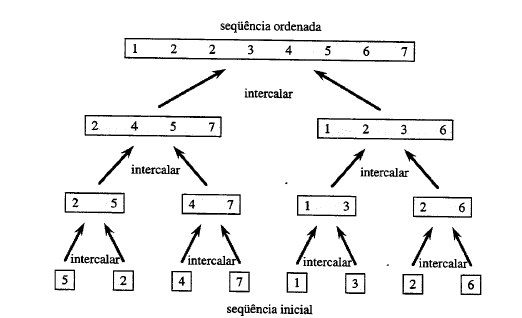
- Não executar trabalho desnecessário. Seu algoritmo deve fazer o mínimo necessário de modo a alcançar seu objetivo, gerando eficiência máxima.

- Processa dados de modo eficiente, isto é, não processar dados que seja grande demais para seu algoritmo.

- Preparar o algoritmo para chegar o melhor caso. Por exemplo, podemos modificar uma algoritmo de ordenação por intercalação para verificar se o vetor já esta ordenando. Neste caso, o melhor caso teria tempo de execução Θ(n).

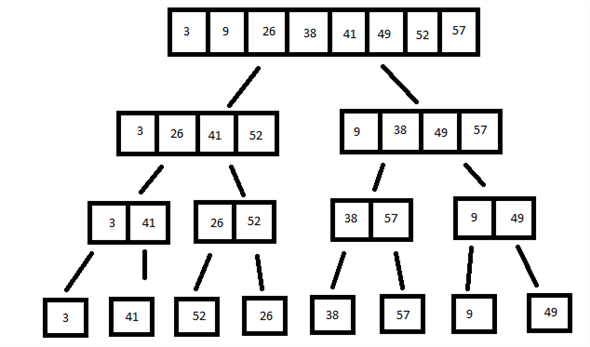
Podemos modificá-lo para lidar com o melhor caso de forma eficiente. Por exemplo, se modificarmos o tipo de mesclagem para verificar se a matriz está ordenada e apenas devolvê-la, o tempo de execução do melhor caso será Θ (n).

2.3-1 Usando a figura 2.4 como modelo, ilustre a operação de ordenação por intercalação sobre o arranjo A = (3; 41; 52; 26; 38; 57; 9; 49)



R. De forma resumida, podemos dizer que na ordenação por intercalação os elementos (e subvetores) são agrupados dois a dois e ordenadores, sendo posteriormente intercalados e o processo repetido, de modo a termos no fim um vetor ordenado.

Portanto, temos abaixo a ordenação por intercalação do arranjo dado, partindo de baixo para cima:



2.3-2 Reescreva o procedimento MERGE de modo que ele não utilize sentinelas e, em vez disso , interrompa depois que o arranjo L ou R tiver todos os seus elementos copiados de volta para A, e então copie o restante do outro arranjo de volta em A.

R.

MERGE(A, p, q, r )

1 **para** i de p **ate** q **faca**

2 B[i] <- A[i]

3 **para** j de q + 1 **ate** r **faca**

4 B[r + q + 1 − j] <- A[j]

5 i <- p

6 j <- r

7 **para** k de p **at e** r **faca**

8 **se** B[i] <= B[j]

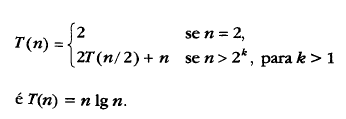
9 **entao** A[k] <- B[i]

10 i <- i + 1

11 **senao** A[k] <- B[j]

12 j <- j – 1

2.3-3 Use indução matemática para mostrar que, quando n é uma potência exata de 2, a solução da recorrência



R.:

Potencias de 2, { 1, 2, 4, 7, 16, 23, 64, ....}

T(2) = 2 se n = 2.

Primeiro, devemos escrever uma função na qual usaremos indução.

F(k) = T(2^k)

Queremos provar o seguinte:

F(k) = 2^k log 2^k

F(2) = 2^1 log 2^1 = 2

O caso base é quando n = 2, e nós temos n log n = 2 log 2 = 2 \* 1 = 2

Para o passo indutivo, nossa hipótese é que T(n/2) = (n/2) log (n/2)

T(n) = 2T (n/2) + n

= 2 (n/2) log (n/2) +n)

= 2 n/2(log n -1) + n

= n(log n -1) + n (distributiva de n)

= n log n – n + n (elimina -n com + n)

= n log n

Que completa a prova indutiva de potências exatas de 2.

Vamos assumir que log = log2.

Hipótese: T(n)= n log n

Caso base: T(2) = 2 log 2 = 2, correto

Passo indutivo: assumindo T(n), vamos encontrar T(2n)

T(2n) = 2n log 2n

= 2n (log 2 + log n)

= 2n + 2n log n

= 2 (n log n) \_ 2n

= 2T(n) + 2n

2.3-4 A ordenação por inserção pode ser expressa sob a forma de um procedimento recursiva como a seguir. Para ordenar A[1..n], ordenamos recursivamente A[1..n-1] e depois inserimos A[n] no arranjo ordenado A[1..n-1]. Escreva uma recorrência para o tempo de execução dessa versão recursiva da ordenação por inserção.

R.:

O problema é divido em um problema menor com tamanho n -1. Resolvemos então esse problema por recursão. Temos agora um subvetor de tamanho n-1. Inserimos então o elemento faltante no vetor.

A recorrência é dada pelo custo de ordenar os n-1 primeiros elementos mais o custo de colocar o elemento na posição correta (tendo que deslocar, no máximo, (n-1) elementos).

T(n)=1, se n = 1

T(n)=T(n−1)+(n−1), se n>1

2.3-5 Voltando ao problema de pesquisa (ver Exercício 2.1-3), observe que, se a sequência A estiver ordenada, podemos verificar o ponto médio da sequência com v e eliminar metade da sequência de consideração posterior. A pesquisa binária é um algoritmo que repete esse procedimento, dividindo ao meio o tamanho da porção restante da sequência a cada vez. Escreva pseudocódigo, sendo ele iterativo ou recursivo, para pesquisa binária. Demostre que o tempo de execução do pior caso da pesquisa binária é Θ(log n).

Vamos mostrar, por indução, que a complexidade é Θ (logn). Recorrência:

T(n) = c1, se n = 1

T(n) = T(n/2)+c2, se n>1

Caso base: T(1)=log1=c1, correto.

Passo indutivo:

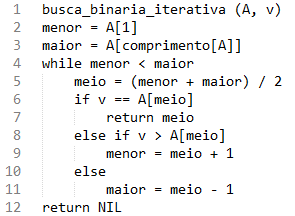
T(2n) = log2n

= log2+logn

= 1+logn

= logn+c2

= T(n)+c2



O procedimento BINARY-SEARCH pega um vetor ordenado A, um valor v e um intervalo [low. . high] do vetor, no qual procuramos o valor v. O procedimento compara v para a entrada do vetor no ponto médio do intervalo e decide eliminar a metade do alcance de uma análise adicional. Damos versões iterativas e recursivas, cada uma das quais retorna um índice i tal que A [i] = v, ou NIL se nenhuma entrada de A [low .. high] contém o valor v. A chamada inicial para qualquer das versões deve ter os parâmetros A, v, 1, n.

**Interativo:**

**ITERATIVE-BINARY-SEARCH**(A, v, low, high)

while low ≤ high

do mid ← (low+high)/2

if v = A[mid]

then return mid

if v > A[mid]

then low ← mid +1

else high ← mid −1

return NIL

**Recursivo:**

**RECURSIVE-BINARY-SEARCH**(A, v, low, high)

if low > high

then return NIL

mid ← (low+high)/2

if v = A[mid]

then return mid

if v > A[mid]

then

return RECURSIVE-BINARY-SEARCH(A, v, mid +1, high)

else

return RECURSIVE-BINARY-SEARCH(A, v, low, mid −1)

Ambos os procedimentos terminam a pesquisa sem sucesso quando o intervalo está vazio (ou seja, low> high) e termina com sucesso se o valor v for encontrado. Com base na comparação de v para o elemento do meio no intervalo pesquisado, a pesquisa continua com o intervalo dividido pela metade. A recorrência para esses procedimentos é, portanto, T (n) = T (n / 2) + Θ (1), cuja solução é T (n) = Θ (Lg n).

2.3-6 Observe que o laço while das linhas 5 a 7 do procedimento Insertion-Sort na seção 2.1 utiliza uma pesquisa linear para varrer(no sentido inverso) o subarranjo ordenado A[1..j-1]. Podemos usar, em vez disso, uma busca binária (veja Exercício 2.3-5) para melhorar o tempo de execução global do pior caso da ordenação por inserção para Θ (n log n)?

R. Se o tempo logarítmico for usado para encontrar os elementos, ainda assim esses elementos precisarão ser movidos e eventualmente ordenados. Mover elemento é linear no pior caso. O algoritmo executará o mesmo número de trocas, porém reduziria o número de comparações.