Gerardo Valdisio Rodrigues Viana
Glauber Ferreira Cintra

Pesquisa e Ordenação de Dados

2011

Capítulo 2

Ordenação Interna

Objetivos:

- Definir o problema da ordenação
- Apresentar os métodos de classificação interna
- Analisar a complexidade dos algoritmos de ordenação

Apresentação

Neste capítulo iniciamos definindo formalmente o problema da ordenação e os principais critérios de classificação de dados. Em seguida, apresentamos em detalhes os métodos de ordenação interna, discutindo sua eficiência em termos de consumo de tempo e de memória e outros aspectos relevantes. Na ordem, abordaremos os métodos Bolha, Inserção e Seleção que são bastante simples e servem para introduzir ideias a serem utilizadas em outros métodos mais eficientes. Seguimos com os métodos Shellsort, Mergesort, Quicksort e Heapsort, considerados superiores aos anteriores, também, como eles, baseados na operação de comparação. Por fim, discutiremos os métodos Countingsort, Bucketsort e Radixsort que têm em comum o fato de não utilizarem a operação de comparação e, sob algumas hipóteses, eles ordenam em tempo linearmente proporcional ao tamanho da lista.

1. Ordenação Interna

Como dito no capítulo anterior, a ordenação de dados é um dos problemas mais importantes e mais estudados dentro da ciência da computação. Em diversas situações cotidianas é preciso colocar uma lista em ordem para facilitar a busca de informações nela contidas.

Neste capítulo discorremos sobre os principais critérios de ordenação, definindo antes, na seção 2, formalmente o problema da ordenação. Nas seções seguintes são apresentados em detalhes os diversos métodos de ordenação interna. Para cada um deles é mostrada sua eficiência em termos de consumo de tempo e de memória e outros aspectos considerados relevantes.

Inicialmente, nas seções 3, 4 e 5, abordaremos os métodos *Bolha*, *Inserção* e *Seleção*. Tais métodos, considerados inferiores, são bastante simples, mas introduzem ideias que servem de base para outros métodos mais eficientes. Esses métodos utilizam como uma de suas operações básicas a comparação de elementos da lista.

Em seguida, nas seções 6, 7, 8 e 10, apresentaremos os métodos *Shellsort*, *Mergesort*, *Quicksort* e *Heapsort*. Esses métodos, considerados superiores, utilizam estratégias mais sofisticadas como, por exemplo, *divisão e conquista* e o uso de estruturas de dados conhecidas como *heaps binários*, descritos na Seção 9. Esses métodos também são baseados na operação de comparação.

Por fim, nas seções 11, 12 e 13, discutiremos os métodos *Countingsort*, *Bucketsort* e *Radixsort*. Esses últimos métodos têm em comum o fato de não utilizarem a operação de comparação. Além disso, sob certas hipóteses, eles ordenam em tempo linearmente proporcional ao tamanho da lista.

2. O Problema da Ordenação

Para que uma lista possa ser ordenada os seus elementos devem ser *dois a dois comparáveis*. Isso significa que dados dois elementos da lista deve ser sempre possível determinar se eles são equivalentes ou se um deles é "menor" que o outro. A idéia associada ao termo "menor" depende do critério de comparação utilizado.

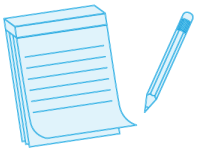
Diz-se que uma lista está em ordem crescente se cada um de seus elementos é menor ou igual ao seu sucessor segundo algum critério de comparação. A definição de ordem decrescente é análoga. Como dito, uma lista somente pode ser colocada em ordem se os seus elementos forem dois a dois comparáveis. O problema da ordenação consiste em, dada uma lista, colocá-la em ordem crescente ou decrescente. Os principais critérios são comparação numérica, comparação alfabética e comparação cronológica.

- Comparação numérica: um número x é menor do que um número y se a expressão $x - y$ resulta em um número negativo. Esse é o tipo mais comum de comparação e, de certa forma, todos os demais critérios de comparação derivam dele.
- Comparação alfabética: um caractere é menor do que outro se precede esse outro na ordem do alfabeto. Por exemplo, o caractere "a" precede o "b", o "b" precede o "c" e assim por diante. No computador, os caracteres são representados através de números (segundo alguma codificação, por exemplo, ASCII, UNICODE etc). Dessa forma, a comparação entre caracteres acaba sendo uma comparação entre os seus códigos numéricos e, portanto, é uma comparação numérica.
- Comparação cronológica: Uma data é menor do que outra se ela antecede essa outra. Uma data pode ser representada através de três números, um para representar o ano, outro para o mês e outro para o dia. Ao comparar uma data com outra, comparamos o ano. Em caso de empate, comparamos o mês. Em caso de novo empate, comparamos o dia. Sendo assim, a comparação cronológica também é feita comparando-se números.

É possível definir outros critérios de comparação. Por exemplo, podemos comparar caixas, considerando menor aquela que apresentar menor volume. Como o volume é expresso por um número, esse tipo de comparação também é uma comparação numérica.

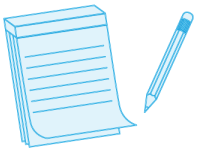
Poderíamos, no entanto, considerar uma caixa menor do que outra se ela *cabe* nessa outra. Nesse caso, podemos ter caixas incomparáveis. Utilizando esse critério de comparação, podemos ter uma lista de caixas que não pode ser ordenada. Além disso, quando temos listas heterogêneas, pode não ser possível ordená-la. Por exemplo, a lista (4, "bola", 2, 15/02/2011) não pode ser ordenada, pois seus elementos não são todos dois a dois comparáveis.

Dizemos que uma lista está em *ordem crescente* se cada elemento da lista é menor ou igual (segundo o critério de comparação adotado) ao seu sucessor. Se cada elemento da lista é estritamente menor do que seu sucessor, dizemos que a lista está em *ordem estritamente crescente*. Analogamente, dizemos que uma lista está em *ordem decrescente* se cada elemento da lista é maior ou igual ao seu sucessor. Se cada elemento da lista é estritamente maior do que seu sucessor, dizemos que a lista está em *ordem estritamente decrescente*. Obviamente, ordenação estrita somente é possível se todos os elementos da lista forem distintos.



ANOTE

ASCII - American Standart Code for Information Interchange - Código padrão americano para o intercâmbio de informação



ANOTE

Aquilo que definimos como *ordem crescente* é chamada por alguns autores de *ordem não decrescente* enquanto que o que chamamos de *ordem estritamente crescente* é chamada por esses mesmos autores de *ordem crescente*. O que chamamos de *ordem decrescente* é chamada por alguns de *ordem não crescente* enquanto que a *ordem estritamente decrescente* é chamada de *ordem decrescente*.

O problema da ordenação consiste em, dada uma lista cujos elementos sejam todos dois a dois comparáveis, permutar a ordem de seus elementos de modo a deixar a lista em ordem crescente ou decrescente.

Se a lista não for muito grande ela pode ser armazenada na memória principal (interna) do computador e ser ordenada usando apenas a memória interna. Os algoritmos que ordenam utilizando apenas memória interna são chamados de *métodos de ordenação de interna*.

Em contrapartida, os algoritmos que fazem uso de memória externa são chamados de *métodos de ordenação de externa*. Alguns desses métodos serão discutidos no Capítulo 3.

No restante deste capítulo descreveremos diversos métodos de ordenação interna. Embora uma lista possa ser armazenada usando-se uma lista encadeada, descreveremos os algoritmos considerando que a lista está armazenada num vetor indexado a partir da posição zero. Além disso, consideraremos que o vetor armazena números e, portanto, o critério de comparação será numérico. Na descrição dos algoritmos de ordenação, vamos sempre considerar que o vetor é passado por referência.

Convém salientar que a lista pode ser constituída de registro e podemos ordená-la por um de seus campos, que chamamos de chave de ordenação. A comparação entre os elementos da lista é feita comparando-se os valores das chaves de ordenação dos dois elementos.

Obviamente, os métodos aqui descritos podem ser facilmente adaptados por ordenar listas encadeadas cujos elementos são registros. Em alguns exercícios tais adaptações são requeridas.

3. Bolha

O Método Bolha é bastante simples e intuitivo. Nesse método os elementos da lista são movidos para as posições adequadas de forma contínua, assim como uma bolha move-se num líquido. Se um elemento está inicialmente numa posição i e, para que a lista fique ordenada, ele deve ocupar a posição j, então ele terá que passar por todas as posições entre i e j.

Em cada iteração do método, percorremos a lista a partir de seu início comparando cada elemento com seu sucessor, trocando-os de posição se houver necessidade. É possível mostrar que, se a lista tiver *n* elementos, após no máximo (n–1) iterações a lista estará em ordem. A seguir fornecemos uma descrição em pseudocódigo do Bolha.

```
ALGORITMO BOLHA
  ENTRADA: UM VETOR L COM N POSIÇÕES
  SAÍDA: O VETOR L EM ORDEM CRESCENTE

  PARA i = 1 até n - 1
    PARA j = 0 até n - 1 - i
      SE  L[j] >  L[j+1]
        AUX = L[j]          // SWAP
        L[j] = L[j+1]
        L[j+1] = AUX
FIM {BOLHA}
```

Algoritmo 2.1: Bolha

Na descrição do Algoritmo Bolha deixamos claro como realizar a troca (*swap*) de elementos da lista. No restante deste capítulo, faremos o *swap*

usando o procedimento *troca*. Assim, a chamada *troca(L[i], L[j])* serve para trocar os conteúdo das posições i e j do vetor L.

É fácil perceber que ao final da primeira iteração do laço mais externo do Algoritmo Bolha, o maior elemento da lista estará na última posição do vetor. Sendo assim, na iteração seguinte não precisamos mais nos preocupar com a última posição do vetor. Ao final da segunda iteração, os dois maiores valores da lista estarão nas duas últimas posições do vetor L, em ordem crescente. Não precisamos então nos preocupar com as duas últimas posições do vetor.

De fato, no Algoritmo Bolha é válido o seguinte invariante: ao final da iteração k do laço mais externo, os k maiores valores da lista estarão nas k últimas posições do vetor, em ordem crescente. A corretude do algoritmo decorre desse invariante. Além disso, o invariante explica porque no laço mais interno o contador j só precisa variar de 0 (primeira posição do vetor) até n – 1 – i.

Claramente, a instrução crítica do Algoritmo Bolha é a comparação (se). Em cada iteração do laço mais interno é feita uma comparação. Como a quantidade de iterações do laço interno diminui à medida que i aumenta, o número de comparações realizadas pelo algoritmo é:

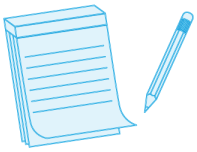
i	Comparações
0	n – 1
1	n – 2
...	...
n – 1	1
Total	$(n^2 - n)/2$

Assim, a complexidade temporal do Algoritmo Bolha pertence a $\Theta(n^2)$. Além dos parâmetros de entrada (L e n), o algoritmo utiliza apenas três variáveis escalares (i, j e aux). Dessa forma, a quantidade extra de memória utilizada pelo algoritmo é constante. Sendo assim, sua complexidade espacial pertence a $O(1)$. Os métodos de ordenação cuja complexidade espacial pertence a $O(1)$ são chamados métodos de ordenação *in situ*. Esse é o caso do Método Bolha.

Outro aspecto importante dos métodos de ordenação é a *estabilidade*. Dizemos que um método de ordenação é *estável* se ele preserva a ordem relativa existente entre os elementos repetidos da lista. Por exemplo, se a ordenação da lista (4, 7, 6, 7, 2) resulta em (2, 4, 6, 7, 7), a ordenação foi estável. Observe que o sete sublinhado estava à direita do outro sete e ao final da ordenação ele continuou à direita. Se todas as ordenações produzidas por um método de ordenação são estáveis, dizemos que o método é estável.

Observe que no Algoritmo Bolha, quando comparamos dois elementos da lista, somente fazemos o *swap* se o antecessor for menor do que o sucessor, pois tivemos o cuidado de utilizar uma desigualdade estrita na comparação. Como consequência, o Algoritmo Bolha é estável.

A seguir exibimos o conteúdo de um vetor após cada iteração do laço mais externo do Algoritmo Bolha.



ANOTE

A letra grega "Θ" é usada para denotar a ordem de complexidade media e "O" para o pior caso.

GLOSSÁRIO

In situ: Expressão latina, usada em vários contextos, que significa "no lugar".

	0	1	2	3	4	5
Lista original	9	4	5	10	5	8
1ª iteração	4	5	9	5	8	10
2ª iteração	4	5	5	8	9	10
3ª iteração	4	5	5	8	9	10
4ª iteração	4	5	5	8	9	10
5ª iteração	4	5	5	8	9	10

Figura 2.1: Ordenação com o Algoritmo Bolha

Neste exemplo percebemos que no final da segunda iteração a lista já estava ordenada. Na terceira iteração não houve nenhuma alteração na lista. Essa fato nos permitiria concluir que a lista já estava em ordem, de modo que as duas últimas iterações foram inúteis.

Podemos introduzir uma pequena modificação no Método Bolha de modo a torná-lo um pouco mais "esperto" e evitar iterações inúteis. Tal modificação consiste em usar uma variável (*flag*) para sinalizar se foi realizada alguma troca de elementos numa iteração do método. Caso nenhuma troca tenha sido realizada, a lista já estará em ordem e, portanto, o método deve parar. No exemplo anterior, após a terceira iteração a ordenação seria finalizada.

Essa variante do Bolha é conhecida como *Bolha com Flag* e seu pseudocódigo é fornecido a seguir.

```
ALGORITMO BOLHA_COM_FLAG
  ENTRADA: UM VETOR L COM N POSIÇÕES
  SAÍDA: O VETOR L EM ORDEM CRESCENTE

  i = 0
  FAÇA
    FLAG = FALSO
    PARA j = 0 até n - 1 - i
      SE  L[j] >  L[j+1]
        TROCA(L[J], L[J+1])
        FLAG = VERDADEIRO
    i = i + 1
  ENQUANTO FLAG = VERDADEIRO
FIM {BOLHA_COM_FLAG}
```

Algoritmo 2.2: Bolha com Flag

Enquanto que o tempo requerido pelo Bolha é sempre quadrático em relação ao tamanho da lista, o comportamento do Bolha com Flag é sensível ao tipo de lista fornecida como entrada. No melhor caso, que ocorre quando a lista fornecida já esta ordenada, o Algoritmo Bolha_com_Flag requer tempo $\Theta(n)$ pois a comparação será sempre falsa e a variável flag jamais receberá o valor verdadeiro. Dessa forma, após a primeira iteração do laço externo o algoritmo será finalizado.

Suponha agora que o menor elemento está inicialmente na última posição da lista. Observe que a cada iteração do laço externo o menor elemento será movido apenas uma posição para a esquerda. Serão necessárias n – 1 iterações para que o menor elemento alcance a primeira posição da lista. Nesse caso, o comportamento do Algoritmo Bolha_com_Flag será idêntico ao do Algoritmo Bolha. Concluímos que no pior caso o Bolha com Flag requer tempo $\Theta(n^2)$. Finalmente, é possível mostrar que o desempenho do Bolha com Flag no caso médio também pertence a $\Theta(n^2)$.

4. Inserção

O Método Inserção, também conhecido como Inserção Direta, é bastante simples e apresenta um desempenho significativamente melhor que o Método Bolha, em termos absolutos. Além disso, como veremos no final dessa seção, ele é extremamente eficiente para listas que já estejam substancialmente ordenadas.

Nesse método consideramos que a lista está dividida em parte esquerda, já ordenada, e parte direita, em possível desordem. Inicialmente, a parte esquerda contém apenas o primeiro elemento da lista. Cada iteração consiste em inserir o primeiro elemento da parte direita (pivô) na posição adequada da parte esquerda, de modo que a parte esquerda continue ordenada. É fácil perceber que se a lista possui n elementos, após n – 1 inserções ela estará ordenada.

Para inserir o pivô percorremos a parte esquerda, da direita para a esquerda, deslocando os elementos estritamente maiores que o pivô uma posição para direita. O pivô deve ser colocado imediatamente à esquerda do último elemento movido.

```
ALGORITMO INSERÇÃO
  ENTRADA: UM VETOR L COM N POSIÇÕES
  SAÍDA: O VETOR L EM ORDEM CRESCENTE

  PARA i = 1 até n - 1
    PIVO = L[i]
    j = i - 1
    ENQUANTO j ≥ 0 e L[i] > PIVO
      L[j+1] = L[j]
      j = j - 1
    L[j+1] = PIVO
FIM {INSERÇÃO}
```

Algoritmo 2.3: Inserção

No melhor caso, que ocorre quando a lista fornecida como entrada já está ordenada, cada inserção é feita em tempo constante, pois o pivô é maior ou igual a todos os elementos da parte esquerda e a condição do laço interno nunca é verdadeira. Nesse caso, a complexidade temporal do Algoritmo Inserção pertence a $\Theta(n)$.

No pior caso, que ocorre quando a lista está inversamente ordenada, cada inserção requer que todos os elementos da parte esquerda sejam movidos para a direita, pois todos eles serão maiores do que o pivô. Nesse caso, a quantidade total de iterações do laço interno será $(n^2 - n)/2$ e a complexidade temporal do Algoritmo Inserção será $\Theta(n^2)$.

A complexidade temporal do Algoritmo Inserção no caso médio também pertence a $\Theta(n^2)$. Isso decorre do fato de que, em média, cada iteração requer que metade dos elementos da parte esquerda sejam movidos para a direita.

É possível mostrar ainda que se todos os elementos da lista estão inicialmente à distância no máximo c (onde c é uma constante) da posição em que deve ficar quando a lista estiver ordenada, o Método Inserção requer tempo linear.

O Algoritmo Inserção requer o uso de apenas três variáveis escalares. Sendo assim, sua complexidade espacial é $O(1)$. Observe que um elemento da parte esquerda somente é movido para a direita se ele for estritamente maior do que o pivô. Por conta disso, o Algoritmo Inserção é estável.

Na figura a seguir exibimos o conteúdo de um vetor após cada iteração do laço mais externo do Algoritmo Inserção. A parte esquerda da lista aparece sombreada.

	0	1	2	3	4	5
Lista original	9	4	5	10	5	8
1ª iteração	4	9	5	10	5	8
2ª iteração	4	5	9	10	5	8
3ª iteração	4	5	9	10	5	8
4ª iteração	4	5	5	9	10	8
5ª iteração	4	5	5	8	9	10

Figura 2.2: Ordenação com o Algoritmo Inserção

5. Seleção

O Método Seleção tem como ponto forte o fato de que ele realiza poucas operações de *swap*. Seu desempenho, em termos absolutos, costuma ser superior ao do Método Bolha, mas inferior ao do Método Inserção.

Assim como no Método Inserção, nesse método consideramos que a lista está dividida em parte esquerda, já ordenada, e parte direita, em possível desordem. Além disso, os elementos da parte esquerda são todos menores ou iguais aos elementos da parte direita.

Cada iteração consiste em selecionar o menor elemento da parte direita (pivô) e trocá-lo com o primeiro elemento da parte direita. Com isso, a parte esquerda aumenta, pois passa a incluir o pivô, e a parte direita diminui. Note que o pivô é maior que todos os demais elementos da parte esquerda, portanto a parte esquerda continua ordenada. Além disso, o pivô era o menor elemento da parte direita e, portanto, continua sendo verdade que os elementos da parte esquerda são todos menores ou iguais aos elementos da parte direita. Inicialmente, a parte esquerda é vazia. Se a lista possui n elementos, após n – 1 iterações ela estará ordenada.

```
ALGORITMO SELEÇÃO
  ENTRADA: UM VETOR L COM N POSIÇÕES
  SAÍDA: O VETOR L EM ORDEM CRESCENTE

  PARA i = 0 ate n - 2
    MIN = i
    PARA j = i + 1 até n - 1
      SE L[j] < L[MIN]
        MIN= j
    TROCA(L[i], L[MIN])
FIM {SELEÇÃO}
```

Algoritmo 2.4: Seleção

A análise do Algoritmo Seleção é semelhante à análise do Algoritmo Bolha. Os dois algoritmos são constituídos de dois laços contados encaixados que realizam as mesmas quantidades de iterações. Dessa forma, se torna evidente que o Algoritmo Seleção realiza $(n^2 - n)/2$ comparações e sua complexidade temporal pertence a $\Theta(n^2)$. Assim como no Bolha, não faz sentido falar em melhor nem pior caso. Ambos os algoritmos requerem tempo quadrático, qualquer que seja a lista.

O Algoritmo Seleção necessita de apenas quatro variáveis escalares para fazer o seu trabalho (uma variável auxiliar é usada no procedimento troca). Sendo assim, a complexidade espacial do Seleção é $O(1)$.

Exibimos na Figura 2.3 o conteúdo de um vetor após cada iteração do laço mais externo do Algoritmo Seleção. A parte esquerda da lista aparece sombreada. Por esse exemplo percebemos que o Método Seleção é *não-estável*.

	0	1	2	3	4	5
Lista original	10	4	5	3	10	2
1ª iteração	2	4	5	3	10	10
2ª iteração	2	3	5	4	10	10
3ª iteração	2	3	4	5	10	10
4ª iteração	2	3	4	5	10	10
5ª iteração	2	3	4	5	10	10

Figura 2.3: Ordenação com o Algoritmo Seleção

6. Shellsort

O Método Shellsort foi proposto por Donald Shell em 1959 e é um exemplo de um algoritmo fácil de descrever e implementar, mas difícil de analisar. Uma explicação precisa para seu surpreendente desempenho é um dos enigmas que desafiam os pesquisadores há décadas.

Para descrever o Shellsort com mais facilidade é conveniente definir o termo *h-lista*. Dada uma lista L, uma h-Lista de L é uma sublista maximal de L na qual cada elemento está a distância h de seu sucessor. Por exemplo, para h = 3, a lista a seguir contém três h-listas. A primeira h-lista é constituída dos elementos nas posições 0, 3 e 6 do vetor. A segunda h-lista contém os elementos nas posições 1, 4 e 7. A terceira h-lista é constituída dos elementos nas posições 2 e 5.

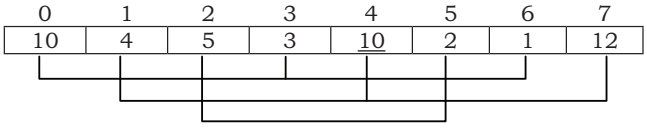


Figura 2.4: Uma lista dividida em h-listas (h = 3)

No Shellsort, a lista é subdividida em h-listas, as quais são ordenadas com um método de ordenação qualquer. Esse procedimento é repetido para valores decrescentes de h, sendo que o último valor de h tem que ser 1. A lista estará então ordenada.

Observe que o Shellsort permite diversas implementações diferentes. Cabe ao programador decidir qual método de ordenação ele usará para ordenar as h-listas. Uma boa opção é utilizar o método Inserção.

Além disso, o programador também terá decidir quais valores de h ele utilizará. Donald Shell sugeriu usar potências de 2 como valores de h. Por exemplo, para ordenar uma lista de 100 elementos, seriam utilizados os números 64, 32, 16, 8, 4, 2 e 1 como valores de h.

Posteriormente, outro pesquisador, Donald Knuth, mostrou que o desempenho do Shellsort não é bom se os valores que h assume ao longo da execução do método são múltiplos uns dos outros. Ele então sugeriu calcular os valores de h através da seguinte fórmula de recorrência: h = 3 * h + 1. Como o último valor de h tem que ser 1, usando essa fórmula, o valor anterior de h seria 4. Antes de 4, o valor de h seria 13, e assim por diante. Por exemplo, para ordenar uma lista de 100 elementos, seriam utilizados os números 40, 13, 4 e 1 como valores de h. Donald Knuth mostrou empiricamente que o desempenho do Shellsort rivaliza com o desempenho dos

melhores métodos de ordenação ao ordenar listas de pequeno e médio porte quando é usada a fórmula sugerida por ele.

```
ALGORITMO SHELLSORT
  ENTRADA: UM VETOR L COM N POSIÇÕES
  SAÍDA: O VETOR L EM ORDEM CRESCENTE

  H = 1
  ENQUANTO H < n FAÇA H = 3 * H + 1
  FAÇA
    H = H / 3            // divisão inteira
    PARA i = H até n - 1 // Inserção adaptado para h-listas
      PIVO = L[i]
      j = i - H
      ENQUANTO j ≥ 0 e L[j] > PIVO
        L[j + H] = L[j]
        j= j - H
      L[j + H] = PIVO
  ENQUANTO H > 1
     FIM {SHELLSORT}
```

Algoritmo 2.5: Shellsort

Note que os valores de h decrescem praticamente numa progressão geométrica de razão 1/3. Isso implica que a quantidade de iterações do laço mais externo pertence a Θ(logn). No melhor caso, que ocorre quando a lista fornecida já está ordenada, a condição do laço interno nunca é verdadeira. Nesse caso, cada iteração do laço mais externo é feita em tempo linear e, portanto, a complexidade temporal do Algoritmo Shellsort pertence a Θ(nlogn). Não se conhece a exata complexidade do Shellsort no pior caso. Sabe-se, no entanto, que tal complexidade está entre Θ(nlogn) e $\Theta(n^{1,5})$.

O Shellsort é mais um método de ordenação *in situ*, pois sua complexidade espacial é constante. Sobre a estabilidade, o exemplo da Figura 2.5 deixa claro que ao ordenar as h-listas o Shellsort pode inverter a ordem existente entre os elementos repetidos, mesmo que seja usado um método de ordenação estável (como o Inserção) para ordenar as h-listas. Consequentemente, o Shellsort é *não-estável*.

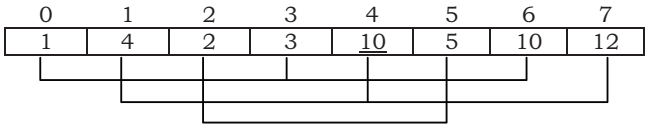
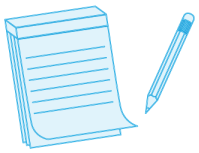


Figura 2.5: h-listas da Figura 2.4 em ordem crescente



ANOTE

Em 2001, Marcin Ciura mostrou que utilizar a sequência 1, 4, 10, 23, 57, 132, 301, 701 e 1750 como valores de h produz resultados ainda melhores que utilizar a fórmula proposta por Knuth. A sequência de Ciura é a melhor conhecida atualmente.

7. Mergesort

Esse método, proposto por John Von Neumann em 1945, é baseado numa estratégia de resolução de problemas conhecida como *divisão e conquista*. Essa técnica consiste, basicamente, em decompor a instância a ser resolvida em instâncias menores do mesmo tipo de problema, resolver tais instâncias (em geral, recursivamente) e por fim utilizar as soluções parciais para obter uma solução da instância original.

Naturalmente, nem todo problema pode ser resolvido através de divisão e conquista. Para que seja viável aplicar essa técnica a um problema ele deve possuir duas propriedades estruturais. O problema deve ser *decompo-*

nível, ou seja, deve ser possível decompor qualquer instância não trivial do problema em instâncias menores do mesmo tipo de problema.

Vale salientar que a técnica de divisão e conquista costuma apresentar desempenho melhor quando as instâncias obtidas com a decomposição têm aproximadamente o mesmo tamanho.

Uma instância *trivial* é uma instância que não pode ser decomposta, mas cuja solução é muito simples de ser calculada. Por exemplo, ordenar uma lista constituída de apenas um elemento é uma instância trivial do problema da ordenação.

Além disso, deve ser sempre possível utilizar as soluções obtidas com a resolução das instâncias menores para chegar a uma solução da instância original. Como veremos mais adiante, o problema da ordenação tem essas duas propriedades estruturais e, portanto, pode ser resolvido por divisão e conquista.

No Mergesort dividimos a lista em duas metades. Essas metades são ordenadas recursivamente (usando o próprio Mergesort) e depois são *intercaladas*. Embora seja possível descrever o Mergesort sem fazer uso de recursividade, a versão recursiva desse é algoritmo é elegante e sucinta, como vemos a seguir.

```
ALGORITMO MERGESORT
  ENTRADA: UM VETOR L E AS POSIÇÕES INICIO E FIM
  SAÍDA: O VETOR L EM ORDEM CRESCENTE DA POSIÇÃO INICIO ATÉ
         A POSIÇÃO FIM

  SE inicio < fim
    meio = (inicio + fim)/2          // divisão inteira
    SE inicio < meio
      MERGESORT(L, inicio, meio)
    SE meio + 1 < fim
      MERGESORT(L, meio + 1, fim)
    MERGE(L, inicio, meio, fim)
FIM {MERGESORT}
```

Algoritmo 2.6: Mergesort

Observe que a intercalação do intervalo a ser ordenado é feita com o Procedimento Merge descrito mais adiante. Tal procedimento é a parte principal do Mergesort e é nele onde os elementos são movimentados para as posições adequadas.

Nesse procedimento fazemos uso das variáveis i e j para percorrer a metade esquerda e a metade direita, respectivamente. Em cada iteração comparamos o elemento na posição i com o elemento na posição j. O menor deles é copiado para um vetor auxiliar. Esse procedimento é repetido até que uma das duas metades tenha sido totalmente copiada para o vetor auxiliar. Se alguns elementos da metade esquerda não tiverem sido copiados para o vetor auxiliar, eles devem ser copiados para o final do intervalo. Finalmente, copiamos os elementos do vetor auxiliar de volta para o intervalo.

```
PROCEDIMENTO MERGE
  ENTRADA: UM VETOR L E AS POSIÇÕES INICIO, MEIO E FIM
           (L TEM QUE ESTAR EM ORDEM CRESCENTE DA POSIÇÃO
           INICIO ATÉ MEIO E DA POSIÇÃO MEIO + 1 ATÉ FIM)
  SAÍDA: O VETOR L EM ORDEM CRESCENTE DA POSIÇÃO INICIO ATÉ
         A POSIÇÃO FIM

i = inicio, j = meio + 1, k = 0
ENQUANTO i ≤ meio e j ≤ fim
  SE L[i] ≤ L[j]
    AUX[k] = L[i], i = i + 1
  SENAO
    AUX[k] = L[j], j = j + 1
  k = k + 1
SE i ≤ meio
  PARA j = meio até i(passo -1)
    L[fim-meio+j] = L[j]
PARA i=0 ate k-1
    L[i+inicio] = AUX[i]
FIM {MERGE}
```

Algoritmo 2.7: Procedimento Merge

Uma análise cuidadosa desse procedimento nos leva a concluir que ele requer tempo linearmente proporcional ao tamanho do intervalo. Vamos chamar de n o tamanho do intervalo, ou seja, n = fim – inicio + 1. Observe que a condição do primeiro laço somente é verdadeira se i ≤ meio e j ≤ fim. Logo, para que o laço continue e ser executado é preciso termos i + j ≤ meio + fim.

Observe que inicialmente i + j = inicio + meio + 1. Como a cada iteração do primeiro laço ou i ou j é incrementado, após n – 1 iterações teremos:

$$i + j = inicio + meio + 1 + n - 1$$
$$i + j = inicio + meio + n$$
$$i + j = inicio + meio + fim - inicio + 1$$
$$i + j = meio + fim + 1$$

Dessa forma, após no máximo n – 1 iterações a condição do primeiro laço será violada. Claramente, o segundo laço, se executado, terá no máximo n/2 iterações. Note que a cada iteração do primeiro laço a variável k é incrementada. Dessa forma, a quantidade de iterações do primeiro laço será k. Como k é no máximo n – 1 e o último laço terá k iterações, concluímos que o Procedimento Merge requer tempo linear. Observe ainda que o Procedimento Merge faz uso do vetor AUX e, portanto, requer espaço linear.

Além disso, note que na comparação contida no primeiro laço, se L[i] e L[j] forem iguais, é copiado L[i] para o vetor auxiliar e isso a preserva a ordem relativa entre os elementos repetidos. Esse cuidado na implementação faz com que o Procedimento Merge faça sempre intercalações estáveis. Como consequência, o Mergesort também é estável.

A figura a seguir ilustra o funcionamento do Procedimento Merge. Observe que o intervalo de L que vai da posição INICIO até MEIO e o intervalo que vai de MEIO + 1 até FIM já estão em ordem crescente.

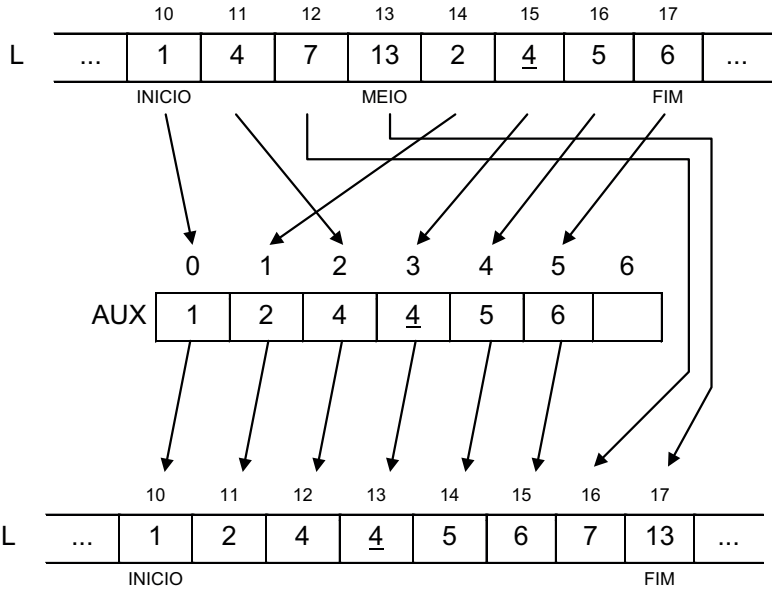
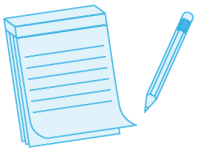


Figura 2.6: Exemplo de intercalação



ANOTE

Esse método é descrito e exemplificado no excelente livro Algoritmos – Teoria e Prática, de Cormen et al.

Como o Algoritmo **Mergesort** foi descrito de maneira recursiva, podemos facilmente fazer sua análise utilizando fórmulas de recorrência. Para isso, vamos denotar por T(n) o tempo requerido para ordenar um intervalo de tamanho n. Sendo assim o tempo necessário para ordenar um intervalo de tamanho n/2 será T(n/2). Como o Procedimento Merge requer tempo linear, podemos supor que ele requer tempo cn (onde c é uma constante positiva). Temos então a seguinte fórmula de recorrência para T:

$$T(n) = 2T(n/2) + cn$$
$$T(1) = c$$

Podemos resolver essa fórmula facilmente utilizando o **método da iteração** supondo $n = 2^k$. A partir da recorrência original, podemos obter as seguintes recorrências:

$$T(n) = 2T(n/2) + cn$$
$$2T(n/2) = 4T(n/4) + cn$$
$$4T(n/4) = 8T(n/8) + cn$$
$$\ldots$$
$$2^{k-1}T(n/2^{k-1}) = 2^kT(n/2^k) + cn$$
$$2^kT(n) = 2^kc$$

Somando tais fórmulas chegamos a $T(n) = cnk + 2^kc$. Lembre-se de que supomos $n = 2^k$, logo, $k = \log_2 n$. Sendo assim, $T(n) = cn\log_2 n + cn$. Concluímos que a complexidade temporal do Algoritmo Mergesort pertence a $\Theta(n\log n)$.

Vamos agora denotar por E(n) o espaço extra de memória requerido para ordenar um intervalo de tamanho n. Como o Procedimento Merge requer espaço linear e a memória utilizada numa chamada recursiva pode

ser reaproveitada na chamada recursiva seguinte, temos então a seguinte fórmula de recorrência para E:

$$E(n) = \max(E(n/2), cn)$$
$$E(1) = c$$

Novamente podemos resolver essa fórmula utilizando o método da iteração, obtendo $E(n) = cn$. Concluímos que a complexidade espacial do Algoritmo Mergesort pertence a $\Theta(n)$. Diferente de todos os métodos que estudamos anteriormente, o Mergesort não é *in situ*.

8. Quicksort

Esse método, desenvolvido em 1960 por C. A. R. Hoare e publicado em 1962, é talvez o mais utilizado de todos os métodos de ordenação. Isso ocorre porque quase sempre o Quicksort é significativamente mais rápido do que todos os demais métodos de ordenação baseados em comparação. Além disso, suas características fazem com que ele, assim como o Mergesort, possa ser facilmente paralelizado. Ele também pode ser adaptado para realizar ordenação externa (Quicksort Externo).

Um aspecto que torna o Quicksort muito interessante sob o ponto de visto teórico é o fato de que, embora em geral ele seja extremamente rápido, existem casos (muito raros, é verdade) em que seu desempenho é bem ruim, chegando a ser pior do que o de métodos de ordenação considerados inferiores como os métodos Bolha, Inserção e Seleção.

Assim como o Mergesort, o Quicksort também é baseado na estratégia de divisão e conquista. Ocorre que, enquanto no Mergesort a fase de divisão é trivial e a fase de conquista é trabalhosa, no Quicksort, com veremos a seguir, acontece o contrário, a fase de divisão é trabalhosa e a fase de conquista é trivial.

Nesse método, a lista é dividida em parte esquerda e parte direita, sendo que os elementos da parte esquerda são todos menores que os elementos da parte direita. Essa fase do método é chamada de *partição*.

Em seguida, as duas partes são ordenadas recursivamente (usando o próprio Quicksort). A lista estará então ordenada.

Uma estratégia para fazer a partição é escolher um valor como *pivô* e então colocar na parte esquerda os elementos menores ou iguais ao pivô e na parte direita os elementos maiores que o pivô.

Embora existam diversas maneiras de escolher o pivô, algumas das quais serão apresentadas nessa seção, adotaremos a princípio uma estratégia extremamente simples, mas que costuma apresentar resultados práticos muito bons. Além disso, tal estratégia facilita perceber os casos em que o procedimento de partição não divide a lista de maneira equilibrada. Dessa forma, torna-se mais fácil caracterizar e analisar o pior caso do Método Quicksort.

Utilizaremos como pivô o primeiro elemento da lista. Nesse caso, o pivô deve ser colocado entre as duas partes. O Procedimento Partição, que implementa tal estratégia, é descrito a seguir.

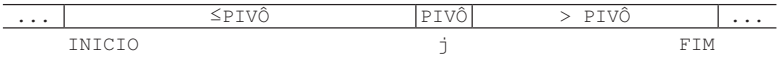


Figura 2.7: Partição

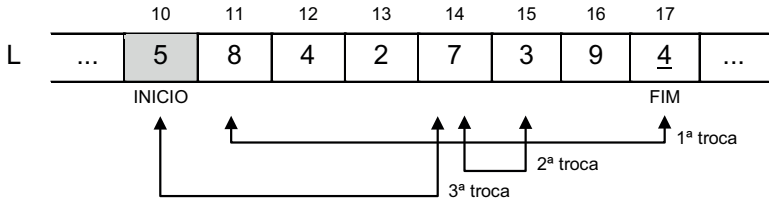
```
PROCEDIMENTO PARTIÇÃO
  ENTRADA: UM VETOR L E AS POSIÇÕES INICIO E FIM
  SAÍDA:j, tal que L[INICIO]..L[j-1] ≤ L[j] e
        L[j+1]..L[FIM] > L[j]
     // j é a posição onde será colocado o pivô, como
     // ilustrado na figura abaixo
       PIVO = L[INICIO], i = INICIO + 1, j = FIM
        ENQUANTO i ≤ j
          ENQUANTO i ≤ j e L[i] ≤ PIVO
            i = i + 1
          ENQUANTO L[j] > PIVO
            j = j - 1
          SE i ≤ j
            TROCA(L[i],L[j])
            i = i + 1, j = j - 1
        TROCA(L[INICIO], L[j])
        DEVOLVA j
      FIM {PARTIÇÃO}
```

Algoritmo 2.8: Procedimento Partição

No Procedimento Partição a variável i avança até encontrar um elemento maior do que o pivô e a variável j recua até encontrar um elemento menor ou igual ao pivô. É feita então a troca do elemento que está na posição i com o elemento que está na posição j. Esse processo é repetido até termos i > j. Para finalizar, o pivô é colocado entre as duas partes fazendo troca(L[INICIO], L[j]). A posição j é devolvida ao chamador do procedimento.

A região crítica do Procedimento Partição é constituída dos dois laços mais internos. Seja n o tamanho do intervalo. Observe que inicialmente j – i = n – 2. Note ainda que a cada iteração de um dos laços internos ou i é incrementado ou j é decrementado. Isso significa que a cada iteração de um desses laços a expressão j – i é decrementada de uma unidade. Consequentemente, após no máximo n – 1 iterações desses laços teremos j – i = -1 o que implica que j será menor do que i. O laço será então finalizado. Concluímos que o Procedimento Partição requer tempo linear. Claramente, esse procedimento tem complexidade espacial constante.

Na figura 2.8 é ilustrada a partição de uma lista. Observe que nesse exemplo que a partição ficou equilibrada, pois quatro elementos ficaram à esquerda do pivô e três elementos ficaram à direita do pivô.



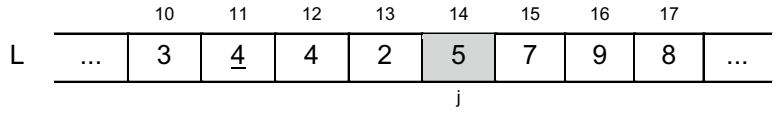


Figura 2.8: Exemplo de partição equilibrada

Nesse exemplo, percebemos que o Procedimento Partição pode inverter a ordem existente entre os elementos repetidos. Consequentemente, o Quicksort é *não-estável*.

Se o elemento escolhido como pivô for o maior elemento do intervalo, teremos uma partição desequilibrada, como ilustrado na figura a seguir. Note que apenas uma troca será executada.

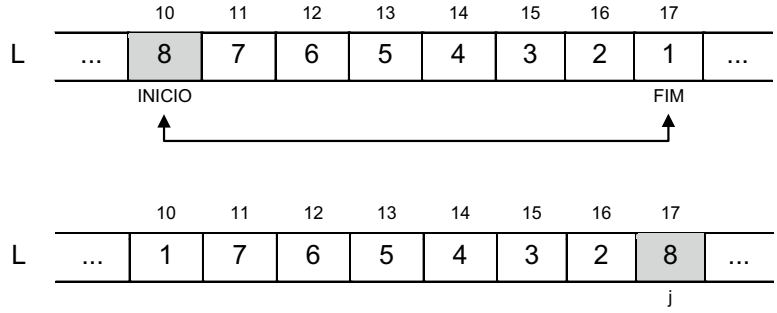


Figura 2.9: Exemplo de partição desequilibrada

É fácil perceber que se o menor elemento do intervalo for escolhido como pivô nenhuma troca será efetuada. Teremos outra partição desequilibrada. Em ambos os casos uma das metades produzidas com a partição ficará vazia e a outra terá todos os elementos do intervalo exceto o pivô. Esse fato terá um impacto importante sobre o desempenho do Quicksort como veremos mais adiante.

Descrevemos a seguir uma versão recursiva do Quicksort.

```
ALGORITMO QUICKSORT
  ENTRADA: UM VETOR L E AS POSIÇÕES INICIO E FIM
  SAÍDA: O VETOR L EM ORDEM CRESCENTE DA POSIÇÃO INICIO ATÉ
         A POSIÇÃO FIM

  SE INICIO < FIM
    j = PARTICAO(L, INICIO, FIM)
    SE INICIO < j - 1
      QUICKSORT(L, INICIO, j - 1)
    SE j + 1 < FIM
      QUICKSORT(L, j + 1, FIM)
FIM {QUICKSORT}
```

Algoritmo 2.9: Quicksort

Para analisar o Algoritmo Quicksort vamos denotar por T(n) o tempo requerido para ordenar um intervalo de tamanho n e por E(n) o espaço extra requerido para ordenar um intervalo de tamanho n. No pior caso, que ocorre quando todas as escolhas do pivô recaem sempre sobre um elemento extremo (o maior ou o menor) do intervalo, a complexidade temporal do Quicksort é dada por:

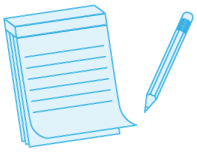
$$T(n) = T(n-1) + cn$$
$$T(1) = c$$

Essa fórmula pode ser facilmente resolvida resultando em $T(n) = (n^2 - n)/2$. A complexidade espacial do Quicksort no pior caso é dada por:

$$E(n) = E(n-1) + c$$
$$E(1) = c$$

Resolvendo essa fórmula obtemos $E(n) = cn$. Concluímos que no pior caso o Algoritmo Quicksort requer tempo $\Theta(n^2)$ e espaço $\Theta(n)$. Para nossa surpresa, essa complexidade temporal é tão ruim quanto as dos métodos Bolha, Inserção e Seleção e essa complexidade espacial é tão ruim quanto a do Mergesort. Some-se a isso o fato de que o Quicksort é não-estável e então concluímos que no pior caso o Quicksort é um dos piores métodos de ordenação.

É fácil perceber que com o critério de escolha do pivô que adotamos, se a lista estiver inversamente ordenada, todos os pivôs escolhidos serão elementos extremos de seus intervalos e, portanto, recairemos no pior caso do Quicksort. Para nossa surpresa, se a lista já estiver ordenada, todos os pivôs escolhidos também serão elementos extremos de seus intervalos e novamente recairemos no pior caso do Quicksort. Felizmente, a ocorrência do pior caso é extremamente rara.



ANOTE

No livro Algoritmos – Teoria e Prática é descrito um método para calcular a mediana em tempo linear.

No melhor caso, que ocorre quando as escolhas do pivô recaem sempre sobre a **mediana** do intervalo, a complexidade temporal do Quicksort é dada por:

$$T(n) = 2T(n/2) + cn$$
$$T(1) = c$$

Resolvemos essa fórmula ao discutir o Mergesort e sabemos que $T(n) \in \Theta(n\log n)$. A complexidade espacial do Quicksort é dada por:

$$E(n) = E(n/2) + c$$
$$E(1) = c$$

Resolvendo essa fórmula concluímos que $E(n) \in \Theta(\log n)$. Sendo assim, no melhor caso a complexidade temporal do Quicksort pertence a $\Theta(n\log n)$ e a complexidade espacial pertence a $\Theta(\log n)$. Essas também são as complexidades do Quicksort no caso médio.

Podemos utilizar estratégias mais elaboradas para escolha do pivô, com o objetivo de produzir partições mais equilibradas. Eis algumas ideias:

- Utilizar a média aritmética do intervalo.
- Utilizar a mediana do intervalo.
- Escolher aleatoriamente um elemento do intervalo.

Nos dois primeiros casos precisaremos de tempo linear para escolher o **pivô**, mas isso não afetará a complexidade temporal assintótica da

partição que é linear. No entanto, em termos absolutos, o tempo requerido pela partição aumentará. Além disso, o pivô poderá não ser um elemento do intervalo e isso exigirá algumas alterações adicionais do Procedimento Partição.

No último caso temos a versão conhecida como *Quicksort probabilístico*. A principal vantagem dessa versão é o fato de tornar impossível garantir que o Quicksort vai requerer tempo quadrático para uma determinada lista. Como vimos anteriormente, ao escolher deterministicamente o pivô, é possível que alguém disposto a desmoralizar o Quicksort construa listas para as quais seu desempenho seja quadrático. Isso não é mais possível na versão probabilística do algoritmo.

9. Heaps Binários

Um *heap* é uma coleção de dados parcialmente ordenados. Num heap *crescente* é válida a seguinte propriedade: "o pai é menor ou igual aos seus filhos". Num heap *decrescente*, temos a propriedade análoga: "o pai é maior ou igual aos seus filhos". Existem diversos tipos de heaps: binários, binomiais, de Fibonacci etc. Abordaremos apenas heaps binários.

Um heap binário se caracteriza pelo fato de que cada elemento possui no máximo dois filhos. Podemos implementar heaps binários usando uma árvore binária. Tal árvore terá todos os níveis completos, com exceção do último nível, que pode estar incompleto. Além disso, os níveis são preenchidos da esquerda para a direita.

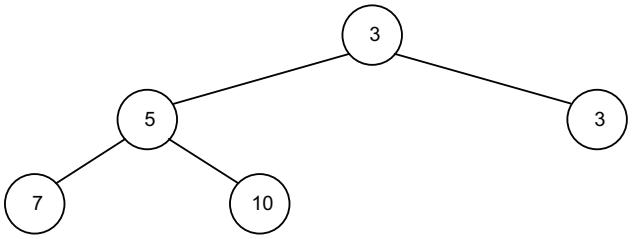


Figura 2.10: Exemplo de heap binário crescente armazenado numa árvore

Um heap binário também pode ser implementado através de um vetor. Nesse caso, os filhos da posição i são as posições 2 * i e 2 * i + 1 (A posição zero do vetor não é utilizada). Discutiremos em detalhes como fazer inserção e como remover o menor elemento de um heap binário crescente implementado num vetor.

1	2	3	4	5	6
3	5	3	7	10	

Figura 2.11: Exemplo de heap binário crescente armazenado num vetor

Observe que num heap binário crescente implementado numa árvore, o menor valor estará sempre na raiz da árvore. Se o heap binário crescente for implementado num vetor, o menor valor estará sempre na primeira posição do vetor. Dessa forma, encontrar o menor valor contido num heap binário crescente é uma operação que pode ser feita em tempo constante. Analogamente, o maior elemento num heap binário decrescente estará sempre no seu início.

Para implementar um heap binário usando um vetor, podemos usar uma estrutura com os seguintes campos:

- vetor: armazena os dados
- tamanho: indica o tamanho do vetor
- ultima: indica a última posição usada do vetor

Para inserir um valor x num heap binário devemos inseri-lo após o último elemento e depois comparar com seu pai, trocando-os de posição se houver necessidade. Tal processo é repetido até que x seja maior ou igual ao seu pai ou até x atingir a posição 1 do vetor. O algoritmo a seguir implementa a ideia descrita nesse parágrafo.

```
ALGORITMO INSERE_HBC
  ENTRADA: UM HEAP BINÁRIO CRESCENTE H E UM VALOR X
  SAÍDA: SE HOUVER ESPAÇO DISPONIVEL, INSERE X EM H;
         CASO CONTRÁRIO, DEVOLVE UM ERRO.

SE H.ultima = H.tamanho                  //HEAP OVERFLOW
  devolva ERRO
H.ultima = H.ultima + 1
i = H.ultima
ENQUANTO i > 1 e H.vetor[i/2] > x        // DIVISÃO INTEIRA
        H.vetor[i] = H.vetor[i/2]
        i = i/2
H.vetor[i] = x
FIM {INSERE_HBC}
```

Algoritmo 2.10: Insere_HBC

Claramente, a região crítica do Algoritmo Insere_HBC é o laço. Dessa forma, o tempo requerido pelo algoritmo é determinado pela quantidade de iterações do laço. Seja n = H.ultima. Note que inicialmente i = n e, portanto, i possui $\lfloor \log_2 n \rfloor + 1$ bits **significativos**. A cada iteração do laço, a variável i perde um bit significativo, pois o valor de i é atualizado fazendo-se i = i/2. Após $\lfloor \log_2 n \rfloor$ iterações, i terá apenas um bit significativo. Sendo assim, teremos $i \leq 1$ e o laço será finalizado. Assim, a quantidade máxima de iterações do laço é $\lfloor \log_2 n \rfloor$. Concluímos que a complexidade temporal do Algoritmo Insere_HBC pertence a $O(\log n)$.

A figura a seguir ilustra a inserção do valor 4 num heap binário crescente. Observe que o valor 7 e depois o valor 5 precisam ser movidos para que o valor 4 possa ser colocado na posição 2 do vetor. Note ainda que a inserção pode inverter a ordem entre os elementos repetidos.

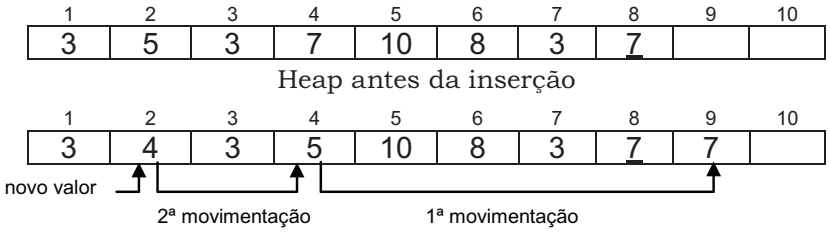


Figura 2.12: Exemplo de inserção num heap binário crescente

Embora seja possível remover um elemento qualquer do heap, vamos discutir apenas a remoção do menor elemento. Como vimos anteriormente, o menor elemento estará sempre na primeira posição do heap.

ANOTE

$\lfloor x \rfloor$ (lê-se "chão de x") é o maior inteiro que é menor ou igual a x.

Para remover o menor elemento de um heap binário crescente implementado num vetor, movemos o último valor contido no heap (pivô) para a posição 1 do vetor. Em seguida, comparamos o pivô com seus filhos, trocando-o com o menor de seus filhos se houver necessidade. Esse procedimento é repetido até que o pivô seja menor ou igual aos seus filhos ou até o pivô atingir uma posição que não tenha filhos.

```
ALGORITMO REMOVE_MENOR
  ENTRADA: UM HEAP BINÁRIO CRESCENTE H
  SAÍDA: SE H NÃO ESTIVER VAZIO, REMOVE E DEVOLVE O MENOR
         ELEMENTO DE H; CASO CONTRÁRIO; DEVOLVE UM ERRO

  SE H.ultima = 0            //HEAP UNDERFLOW
    devolva ERRO e PARE
  valor= H.vetor[1]
  H.vetor[1] = H.vetor[H.ultima]
  H.ultima = H.ultima - 1
  i = 1
  ENQUANTO (2*i ≤ H.ultima e H.vetor[i] > H.vetor[2*i]) ou
           (2*i < H.ultima e H.vetor[i] > H.vetor[2*i+1])
    menor = 2*i
    SE 2*i < H.ultima e H.vetor[2*i+1] ≤ H.vetor[2*i]
      menor= menor+1
    TROCA(H.vetor[i], H.vetor[menor])
    i = menor
DEVOLVA valor
FIM {REMOVE_MENOR}
```

Algoritmo 2.11: Remove_Menor

A figura a seguir mostra a remoção do menor elemento de um heap binário crescente. Observe que após mover o valor 10 para a primeira posição do vetor, precisamos trocá-lo com o valor 3 e depois com o valor 6 para restaurar a ordenação parcial do heap. Note ainda que a remoção do menor elemento pode inverter a ordem entre elementos repetidos.

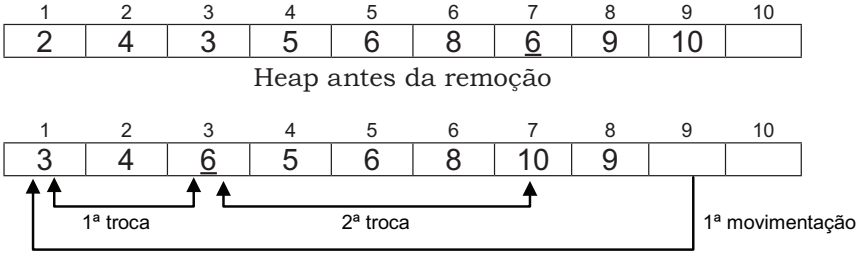


Figura 2.13: Remoção do menor elemento num heap binário crescente

Seja n = h.ultima. Observe que h.ultima / 2 possui $\lfloor \log_2 n \rfloor$ bits significativos. Inicialmente, a variável i possui apenas um bit significativo. A cada iteração do laço, i ganha um bit significativo. Após $\lfloor \log_2 n \rfloor$ iterações, i terá $\lfloor \log_2 n \rfloor + 1$ bits significativos e, portanto, teremos i > h.ultima/2. Consequentemente, o laço será finalizado. Concluímos que o Algoritmo Remove_Menor requer tempo *O*(logn).

10. Heapsort

Podemos usar heaps binários para ordenar com bastante eficiência e elegância. O Método Heapsort é baseado no uso de heap binário. Inicialmente, inserimos os elementos da lista num heap binário crescente. Em seguida, fazemos sucessivas remoções do menor elemento do heap, colocando os elementos removidos do heap de volta na lista. A lista estará então em ordem crescente.

```
ALGORITMO HEAP SORT
  ENTRADA: UM VETOR L COM N POSIÇÕES
  SAÍDA: O VETOR L EM ORDEM CRESCENTE

inicialize um HBC H com n posições
PARA i = 0 até n - 1
  INSERE_HBC(H, L[i])
PARA i = 0 até n - 1
  L[i] = REMOVE_MENOR(H)
FIM {HEAP SORT}
```

Algoritmo 2.12: Heapsort

Na seção anterior mostramos que a inserção e a remoção do menor elemento requer tempo logarítmico no tamanho do heap. Sendo assim, a complexidade temporal do Heapsort pertence a Θ(nlogn).

Como vimos anteriormente, a inserção e a remoção do menor elemento do heap não preservam a ordem relativa existente entre os elementos repetidos. Como consequência desse fato, o Heapsort é não-estável.

Finalmente, a complexidade espacial do Algoritmo Heapsort é Θ(n). No entanto, podemos facilmente adaptar o Heapsort para fazer ordenação *in situ*. Para isso, devemos usar o próprio espaço do vetor como heap. Nesse caso sua complexidade espacial do Heapsort será $O(1)$.

O sete métodos que descrevemos neste capítulo até agora utilizam como operação fundamental a comparação de elementos da lista. É possível mostrar que todo método de ordenação baseado na operação de comparação requer tempo **Ω(nlogn)**. No entanto existem métodos de ordenação que, sob certas condições, ordenam em tempo linear. Naturalmente, tais métodos não são baseados em comparação. Discutiremos a seguir três desses métodos.

ANOTE

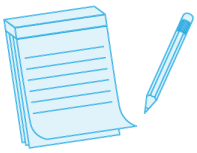
O leitor interessado poderá encontrar uma prova desse fato no livro Algoritmos – Teoria e Prática. (Comen et al., 2002).

11. Countingsort

Esse método foi proposto por Harold H. Seward em 1954 e faz a ordenação usando a ideia de contagem. Uma restrição desse método é que ele somente ordena listas de números naturais.

Essa não é uma restrição muito severa, pois podemos converter diversos tipos de listas em listas de números naturais. Por exemplo, uma lista de números inteiros onde o menor valor é -x pode ser convertida numa lista de números naturais somando x a cada elemento da lista. Uma lista de números racionais pode ser convertida numa lista de números inteiros multiplicando-se os elementos da lista por uma constante apropriada.

De fato, sempre que existir uma bijeção entre o tipo dos dados da lista e um subconjunto dos naturais é possível converter a lista em uma lista de números naturais. Infelizmente, algumas dessas conversões podem ter um impacto negativo sobre o desempenho do método.



ANOTE

Ω denota complexidade no melhor caso.

Seja n o tamanho da lista e k o maior valor contido na lista. Fazemos uso de um vetor C com k + 1 posições. No primeiro laço do Algoritmo Countingsort, descrito a seguir, o vetor C é inicializado com zeros.

No laço seguinte, contamos quantas vezes cada um dos elementos de 0 a k aparece na lista. Para fazer isso, percorremos a lista e para cada valor i contido na lista incrementamos de uma unidade a posição i do vetor C. Dessa forma, ao final desse laço cada posição i de C indica quantas vezes o valor de i ocorre na lista.

Em seguida, fazemos a *soma acumulada* do vetor C. Percorremos novamente o vetor C, dessa vez a partir da segunda posição, somando cada posição i do vetor c com a posição anterior. O resultado da soma é guardado na própria posição i. Após realizar a soma acumulada, cada posição i do vetor C indicará quantos elementos da lista são menores ou iguais a i. Obviamente, se i ocorre na lista e C[i] = j, quando a lista estiver ordenada o valor i deverá ocupar a j-ésima posição da lista.

Percorremos mais uma vez a lista, dessa vez, da direita para a esquerda (para garantir que a ordenação seja estável) e colocamos cada valor i contido na lista na posição c[i] – 1 de um vetor auxiliar. Para evitar que múltiplas cópias de um valor contido na lista sejam colocadas numa mesma posição do vetor auxiliar, após colocar um valor i no vetor auxiliar, decrementamos de uma unidade a posição i de C. No último laço, copiamos o vetor auxiliar para a lista.

```
ALGORITMO COUNTING SORT
  ENTRADA: UM VETOR L CONTENDO N NÚMEROS NATURAIS NO
           INTERVALO DE 0 A K
  SAÍDA: O VETOR L EM ORDEM CRESCENTE

  PARA i = 0 até k                  // INICIALIZACAO DE C
    c[i]=0
  PARA i = 0 até n - 1              // CONTAGEM
    c[L[i]]= c[L[i]] +1
  PARA i = 1 até k                  // SOMA ACUMULADA
    c[i]= c[i] + c[i-1]
  PARA i= n - 1 até 0 (passo -1)    // ORDENACAO
    c[L[i]] = c[L[i]] - 1
    aux[c[L[i]]]= L[i]
  PARA i = 0 até n - 1              // COPIANDO AUX PARA L
    L[i]= aux[i]
FIM {COUNTING SORT}
```

Algoritmo 2.13: Countingsort

O Algoritmo Countingsort é constituído de cinco laços independentes. Três desses laços gastam tempo $\Theta(n)$ e os outros dois gastam tempo $\Theta(k)$. Além disso, vetor C tem k + 1 posições e o vetor aux tem n posições. Concluímos que a complexidade temporal e a complexidade espacial do Countingsort pertencem a $\Theta(n + k)$. Para a classe de listas onde k não é muito maior do que n, mais precisamente, se $k \in \Theta(n)$, então tais complexidades pertencem a $\Theta(n)$. O cuidado que tivemos ao percorrer a lista do final para o início no quarto laço garante a estabilidade de Countingsort .

Na Figura 2.14 exibimos o conteúdo do vetor C ao final de cada laço do Algoritmo Countingsort. O conteúdo do vetor aux corresponde à lista original em ordem crescente.

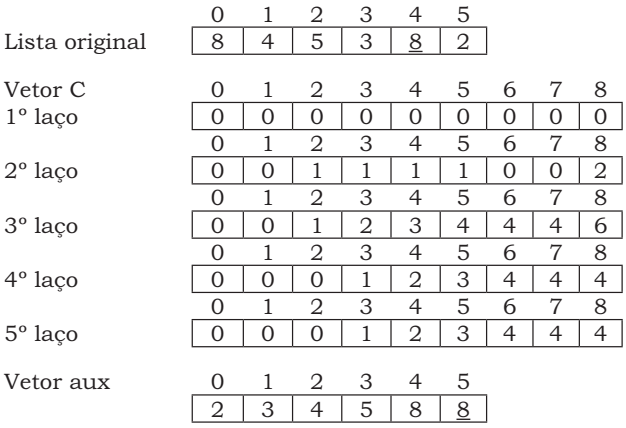


Figura 2.14: Ordenação com o Algoritmo Countingsort

12. Bucketsort

Embora esse método permita ordenar listas de números quaisquer, descreveremos uma versão desse método que ordena listas de números não negativos.

Se a lista a ser ordenada tem n elementos, será usado um vetor de ponteiros (bucket) com n posições. Se o maior elemento da lista é k, cada posição do bucket apontará para uma lista encadeada na qual serão inseridos os elementos da lista que pertencem ao intervalo [i * (k + 1) / n , (i + 1) * (k + 1) / n). Observe que o intervalo é fechado à esquerda e aberto à direita.

A ideia do método é dividir o intervalo que vai de 0 até k em n subintervalos de mesmo tamanho. Cada subintervalo estará associado a uma lista ligada que irá conter os elementos da lista que pertencem àquele subintervalo. Por exemplo, se a lista tem oito elementos é o maior deles é 71, teremos oito intervalos: [0, 9), [9, 18), ..., [63, 72). A posição zero do bucket apontará para uma lista encadeada que irá conter os elementos da lista que são maiores ou iguais a zero e menores que 9, e assim por diante.

Chamaremos o bucket de vetor B. Para construir as listas encadeadas devemos inserir cada valor j contido na lista a ser ordenada na lista encadeada apontada por B[j * n / (k + 1)]. Em seguida, ordenamos as listas encadeadas com um método de ordenação qualquer (de preferência estável). Após isso, a concatenação das listas encadeadas produz a lista original ordenada.

```
ALGORITMO BUCKET SORT
  ENTRADA: UM VETOR L CONTENDO N NÚMEROS NO INTERVALO DE
           0 ATÉ K
  SAÍDA: O VETOR L EM ORDEM CRESCENTE

PARA i = 0 até n - 1                    //INICIALIZACAO
        B[i] = NULO
//CONSTRUINDO AS LISTAS ENCADEADAS
PARA i = n - 1 até 0 (PASSO -1)
  insira L[i] no inicio da lista ligada apontada por
  B[L[i] * n / (K + 1)]                 // DIVISÃO INTEIRA

//ORDENANDO E CONCATENANDO AS LISTAS ENCADEADAS
j = 0
PARA i = 0 até n - 1
```

```
    coloque a lista apontada por B[i] em ordem crescente
    p = B[i]
    ENQUANTO p ≠ NULO
      L[j] = p.info
      p = p.proximo
      j = j + 1
    remova a lista apontada por B[i]
FIM {BUCKET SORT}
```

Algoritmo 2.14: Bucketsort

Claramente os primeiros dois laços podem ser executados em tempo Θ(n). Note que o tamanho *médio* de cada lista encadeada será 1. De fato, se os elementos de L estiverem uniformemente distribuídos no intervalo [0, k), é possível mostrar que o tamanho *esperado* de cada lista encadeada será constante. Nesse caso, independente do método utilizado, a ordenação de cada lista encadeada será feita em tempo esperado constante.

A cópia dos elementos de uma lista encadeada requer tempo amortizado constante. A remoção de uma lista encadeada também é feita em tempo amortizado constante. Consequentemente, o terceiro laço irá requerer tempo esperado constante, e a complexidade temporal *esperada* do Algoritmo Bucketsort será Θ(n).

Note que no segundo laço percorremos a lista da direita para a esquerda. Com isso, ao construir as listas encadeadas preservamos a ordem relativa existente entre os elementos repetidos. Se o método utilizado para ordenar as listas encadeadas for estável, o Algoritmo Bucketsort também será estável.

O bucket terá n posições. A quantidade de nós nas listas encadeadas será n. Concluímos que a complexidade espacial do Bucketsort pertence a Θ(n). Finalmente, observe que Bucketsort não irá requerer comparações se o método usado como subrotina não for baseado em comparações.

A figura a seguir exemplifica as listas encadeadas construídas pelo Bucketsort após sua ordenação. Claramente, a concatenação das listas encadeadas produz a lista original em ordem.

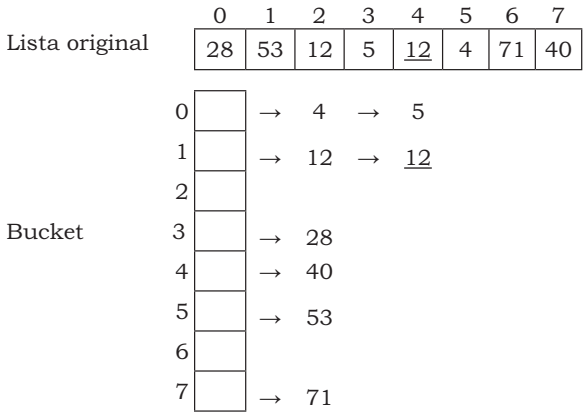


Figura 2.15: Ordenação com o Algoritmo Bucketsort

13. Radixsort

Esse método permite ordenar listas cujos elementos sejam dois a dois comparáveis. Uma versão desse método foi utilizada em 1887 por Herman Hollerith, fundador de uma empresa que mais tarde deu origem à IBM.

Em cada iteração do Radixsort, ordenamos a lista por uma de suas posições, começando pela posição menos significante, até chegar à posição mais significante (os elementos da lista devem que ser representados com a mesma quantidade de posições). Cada ordenação tem que ser feita com um método estável.

```
ALGORITMO RADIX SORT
  ENTRADA: UM VETOR L COM N POSIÇÕES CUJOS ELEMENTOS
           POSSUEM D SÍMBOLOS
  SAÍDA: O VETOR L EM ORDEM CRESCENTE

  PARA i = 1 até d
    Coloque L em ordem crescente pelo i-ésimo símbolo menos
    significante usando um método de ordenação estável
FIM {RADIX SORT}
```

Algoritmo 2.15: Radixsort

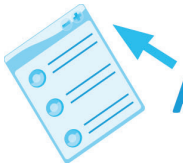
Podemos adaptar o Countingsort e utilizá-lo como subrotina no Radixsort. Observe que os elementos da lista são constituídos de símbolos de algum alfabeto. Por exemplo, se a lista contém números decimais, o alfabeto é formado pelos dígitos do sistema numérico decimal. Note que o maior símbolo encontrado numa posição de um elemento não excede o maior símbolo do alfabeto. Como o tamanho do alfabeto é constante, podemos garantir que a complexidade temporal e a complexidade espacial dessa versão do Countingsort pertencem a $\Theta(n)$.

Outra possibilidade seria adaptar o Bucketsort e utilizá-lo como subrotina. Para isso, o bucket deverá ter uma posição para cada elemento do alfabeto. Não haverá necessidade de ordenar as listas encadeadas visto que cada uma delas contém apenas elementos com o mesmo símbolo na posição que está sendo ordenada. Podemos então garantir que a complexidade temporal dessa versão do Bucketsort será $\Theta(n)$.

Note que tanto o Countingsort quanto o Bucketsort são estáveis. Utilizando qualquer um deles como subrotina, o Radixsort terá complexidade temporal $\Theta(dn)$ e complexidade espacial $\Theta(n)$. Se d for constante a complexidade temporal será $\Theta(n)$. Note que o Radixsort é necessariamente estável, qualquer que seja o método usado como subrotina.

32		500		500		9
500		431		9		32
431		32		431		32
248	→	32	→	32	→	48
9	Ordenando pela unidade	63	Ordenando pela dezena	32	ordenando pela centena	63
32		248		248		248
63		48		48		431
48		9		63		500

Figura 2.16: Ordenação com o Algoritmo Radixsort



ATIVIDADES DE AVALIAÇÃO

1. Complete a tabela abaixo.

Método	Complexidade temporal			Complexidade espacial	É estável?
	Melhor caso	Pior caso	Caso médio		
Bolha	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$O(1)$	Sim
Bolha com flag					
Inserção					
Seleção					
Shellsort					
Mergesort					
Quicksort					
Heapsort					

2. Caracterize o pior e o melhor caso, em termos assintóticos, de cada um dos métodos que aparecem na tabela acima.

3. Mostre como ficaria a lista (18, 23, 37, 22, 25, 16, 3, 31, 5, 28, 6, 34, 9, 45) após cada iteração do laço mais externo dos algoritmos bolha, inserção, seleção, shellsort e radixsort, descritos neste capítulo.

4. Mostre como ficaria a lista (18, 23, 37, 22, 25, 16, 3, 31, 5, 28, 6, 34, 9, 45) após ser particionada pelo Procedimento Partição apresentado neste capítulo.

5. Dentre os métodos bolha, bolha com flag, inserção, seleção, shellsort, mergesort e quicksort, quais seriam os mais eficientes, em termos de tempo, para colocar listas da forma (n, 1, 2, 3, ..., $n - 1$) em ordem crescente? Justifique sua resposta.

6. Escreva um algoritmo que implemente o método bolha para ordenar listas encadeadas contendo números. Considere que cada nó da lista possui os campos *info, anterior* e *proximo*.

7. Indique quais tipos de listas podem ser ordenadas pelos métodos Countingsort, Bucketsort e Radixsort e que condições precisam ser satisfeitas para que eles tenham complexidade de tempo linear no tamanho da lista.

8. Você deseja ordenar em tempo linear listas de n elementos contendo números inteiros entre 1 e $10n$. Qual método você utilizaria ? Justifique sua resposta.

9. Mostre como ficaria o vetor C no final de cada um dos cinco laços do Algoritmo Countingsort ao ordenar a lista (18, 23, 17, 22, 25, 16, 3, 11, 5, 8, 6, 4, 9, 15).

10. Mostre como ficariam as listas encadeadas geradas no método bucket sort ao ordenar a lista (18, 23, 17, 22, 25, 16, 3, 41, 5, 8, 6, 34, 9, 15).

11. Após cada iteração do laço mais externo de um algoritmo de ordenação, a lista (382, 142, 474, 267) ficou como mostrado abaixo. Qual foi o algoritmo utilizado (bolha, inserção, seleção ou radixsort)? Justifique por que nenhum dos outros algoritmos estudados neste capítulo poderia ter produzido as listas abaixo.

(142, 382, 474, 267)

(142, 267, 474, 382)

(142, 267, 382, 474)

12. Escreva um procedimento que receba um heap binário crescente (passado por referência) e uma posição i e então remova o elemento que estiver na posição i do heap.

13. Escreva um procedimento que transforme um vetor num heap binário crescente em tempo linear.

14. Mostre como ficaria um heap binário crescente no qual foram inseridos os valores 18, 23, 17, 22, 25, 16, 3, 41, 5, 8, 6, 34, 9, 15, nesta ordem (seu heap deve ser implementado num vetor de 15 posições). Em seguida, remova o elemento que estiver na posição 3, depois o elemento que estiver na posição 2 e por fim o elemento que estiver na posição 1 e então mostre como ficaria o heap.