Unidade 3 - Métodos de ordenação

3.1 - Primeiras palavras

A utilização dos dados segundo um critério de ordenação auxilia bastante na execução de várias tarefas. A ordenação alfabética de um índice remissivo é essencial para que a localização de um termo seja eficiente. A ordenação está presente em listas telefônicas, ordenadas alfabeticamente por sobrenome e, para um mesmo sobrenome, ordenadas alfabeticamente por nome; ou ainda os livros na biblioteca, ordenados por código de classificação.

Para que qualquer ordenação possa ser feita, é preciso que se defina um critério para ela. Este critério exige a definição do que será usado para a ordenação e qual a relação de desigualdade existente. Para valores numéricos o critério é a relação usual, dada pela magnitude dos números, de forma que valores menores vêm antes dos maiores. Este critério determina, por exemplo, que pessoas mais novas antecedem as mais velhas, definindo um modo natural de ordenação. Para o caso de nomes, a ordem alfabética é a mais comum, de maneira que a ordem definida pelo alfabeto é usada como critério para a ordem dos nomes. Para o caso dos ideogramas chineses, a ordenação se dá conforme a complexidade de cada ideograma em termos do número de traços que o compõe.

A tarefa de colocar em ordem é, portanto, importante e a eficiência com que isso é feito, relevante em muitos contextos.

Para a utilização de algoritmos para abordarem o problema da ordenação, é essencial que sejam visualizadas duas situações: os métodos para memória principal e os para memória secundária. Dadas as características diferentes entre os dois tipos de memória, as abordagens para efetuar a ordenação em si usam estratégias diferentes.

Um último ponto é que, embora importantes, os métodos de ordenação estão amplamente disponíveis na literatura e também *on-line*. Este texto, portanto, não tem o objetivo de descrevê-los a fundo, mas de caracterizá-los segundo seus comportamentos e aplicações.

3.2 - Ordenação na memória principal

Para os métodos em memória principal, uma característica interessante que todos possuem é que os dados são permutados entre si, não exigindo espaço extra para a transferência de dados, exceto o preciso para efetuar trocas individuais. Esta característica se contrapõe aos métodos usados para ordenação em memória secundária, na qual é mais interessante fazer cópias de todo o conjunto de dados, mas isso será abordado na próxima sessão.

Nos diversos exemplos, a ordenação será discutida sobre arranjos com valores numéricos, apenas por serem mais simples de visualizar e por serem mais compactos que nomes. É claro que, dado um critério qualquer de ordenação, basta adaptar para que as comparações obedeçam a este critério. Ordens inversas, isto é, do maior para o menor, podem ser definidas pela alteração destes critérios.

Para ver os métodos de ordenação mais conhecidos, eles estão divididos em dois grupos: os básicos e os avançados. Como básicos estão os métodos mais elementares e, em geral, menos eficientes. Os avançados incluem os que manipulam os dados com estratégia mais sofisticada, sendo estes mais eficientes para grandes volumes de dados.

3.2.1 - Métodos básicos de ordenação

Neste item são abordados três métodos para ordenação que usam estratégias diferentes. Eles são denominados **inserção direta**, **seleção direta** e **bubblesort**.

O método de **inserção direta**, também chamado de ordenação por inserção, utiliza uma estratégia bastante simples. Inicialmente é pego o primeiro elemento do conjunto; em seguida, um novo elemento é pego e já colocado em ordem em relação ao primeiro. A cada novo elemento, este é colocado já em ordem em relação ao grupo já ordenado, repetindo-se o processo até o último elemento.

Para a aplicação desta estratégia em arranjos, segue os seguintes passos:

1. Seleciona-se o primeiro elemento do arranjo, o qual define o subconjunto ordenado inicial.
2. São pegos os demais elementos um a um, a partir do segundo, e posicionados em relação à parte ordenada, fazendo os deslocamentos de dados necessários

Algoritmo 3-1

```
1  inserção_direta(dados[]: inteiro; tamanho: inteiro)
2       para i ← 1 até tamanho – 1 faça
3             { seleciona um dos elementos }
4             auxiliar ← dados[i]
5
```

```
6             { abre espaço deslocando os elementos maiores que dados[i] }
7             j ← i - 1
8             enquanto dados[j] > auxiliar e j ≥ 0 faça
9                   dados[j + 1] ← dados[j]  { desloca uma posição }
10                  j ← j - 1
11            fim-enquanto
12
13            { coloca na posição correta }
14            dados[j + 1] ← auxiliar
15      fim-para
```

A inserção direta é descrita em sua forma algorítmica no Algoritmo 3-1. Deve-se notar que a varredura pelos elementos se inicia no segundo elemento (posição 1 do vetor*), já que o elemento da posição zero já é considerado ordenado (quando há um único elemento no conjunto, o conjunto pode ser considerado ordenado). A partir do primeiro, cada um dos elementos é posicionado em relação aos demais, fazendo os deslocamentos de todos os valores maiores que o elemento que está sendo ajustado. O espaço "aberto" no arranjo permite a inserção do elemento na sua devida posição.

O método é chamado de inserção por pegar cada elemento e o *inserir* na parte ordenada do arranjo.

A análise deste algoritmo passa, inicialmente, por identificar que o tamanho do problema *n* é dado pelo número de elementos presentes no vetor. Quanto maior o número de elementos, mais repetições são necessárias tanto para o comando **para** (linha 2) quanto para o **enquanto** (linha 8). O segundo ponto é notar que existe também uma dependência dos dados, a qual ocorre no comando **enquanto**. Neste comando, deve-se observar que a quantidade de elementos que são deslocados depende do valor do elemento; se ele for maior que a maioria dos demais, poucos elementos devem ser movidos; se for o menor de todos, haverá movimentação de todos para que seja colocado na posição correta. Assim, para este algoritmo, existe **melhor** e **pior** casos e também caso **médio**.

Para seu **melhor caso**, inserção direta faz muito poucos movimentos, situação que ocorre quando o vetor original já se encontra praticamente ordenado. Para este caso, pode-se considerar que a repetição interna (comando **enquanto**) possui tempo constante; isto é, não depende do número total de elementos, já que repete poucas vezes. Portanto, somente a repetição do **para** é relevante, fazendo uma única passagem pelos dados. Para o melhor caso, o algoritmo se comporta como $O(n)$.

O pior caso ocorre quando os elementos estão originalmente em ordem decrescente, ou quase nesta situação. Neste cenário, o comando **enquanto** repete praticamente o número máximo de vezes, pois todos os dados têm que ser deslocados para cada novo elemento que é ordenado. Desta forma, a repetição do **enquanto** é proporcional ao número de elementos do arranjo. Considerando a repetição do **para** e a do enquanto, o algoritmo torna-se pertencente a $O(n^2)$.

Finalmente, existe a situação do caso médio, que considera que os dados estejam em uma ordem aleatória qualquer, ou seja, nem ordenados nem inversamente ordenados. Para este quadro, é possível considerar que, em média, a repetição do **enquanto** desloca metade dos itens do conjunto já ordenada ao inserir um novo elemento. É possível, assim, considerar que exige metade do esforço computacional para esta repetição comparada ao que exige o pior caso. Mesmo assim, esta repetição é proporcional a *n* e, portanto, o algoritmo também é $O(n^2)$ para o caso médio.

O importante, em geral, é o caso médio, pois se aplica a uma solução genérica. O pior caso é uma situação extrema, assim como o melhor caso. O conhecimento do bom comportamento, $O(n)$, para o melhor caso passa a ser importante em alguns casos, quando já se sabe que há certa ordenação no conjunto de dados. De fato, um dos algoritmos avançados, o *shellsort*, faz uma organização dos dados para fazer uma "pré-ordenação" geral e depois utiliza desta vantagem da inserção direta para finalizar a ordenação, obtendo com isso um excelente desempenho.

Usando uma estratégia diferente, o método conhecido por **seleção direta** (ordenação por seleção ou *selection*) é outro método básico. Seu princípio de operação é bastante simples, também: o menor elemento deve ocupar a primeira posição; o segundo menor elemento deve ocupar a segundo; e assim por diante.

Sua estrutura básica reflete:

1. Considere todo o arranjo como não ordenado.
2. Para a parte do arranjo ainda não ordenada, localize o menor elemento e o mova para a posição, reduzindo o tamanho da partição não ordenada.

Algoritmo 3-2

```
1  seleção_direta(dados[]: inteiro; tamanho: inteiro)
```

* Lembra-se, aqui, que os dados nos arranjos são representados com a primeira posição sendo considerada a de índice zero.

```
2       para i ← 0 até tamanho – 2 faça
3             { localiza o menor elemento da partição }
4             menor ← dados[i]
5             posição ← i
6             para j ← i + 1 até tamanho – 1 faça
7                   se dados[j] < menor então
8                         menor ← dados[j]
9                         posição ← j   { armazena a posição }
10                  fim-se
11            fim-para
12
13            { coloca o menor elemento no início da partição }
14            dados[posição] ← dados [i]
15            dados[i] ← menor
16      fim-para
```

A estrutura básica da abordagem por seleção é apresentada no Algoritmo 3-2. Note-se que, a cada passo, é feita a seleção do elemento que deve ocupar cada posição.

Em termos de análise, pode-se notar que não existe dependência de nenhuma repetição em relação ao valor dos dados. Isso quer dizer que não há casos pior, médio ou pior, pois o comportamento é igual qualquer que seja a distribuição inicial dos dados no arranjo. A repetição do **para** mais externo (linha 2) executa sempre *n* vezes, enquanto a do interno (linha 6) se inicia com *n* – 1, depois passa por *n* – 2, *n* – 3 e assim por diante, até 2. Ambos os laços de repetição, como já foi visto, são proporcionais a *n* e o algoritmo tem comportamento $O(n^2)$. Desta maneira, pode-se considerar que seu desempenho é conhecido, pois não depende dos dados.

Por final, existe um algoritmo elementar amplamente apresentado nos cursos de computação: o chamado ***bubblesort*** ou método das bolhas (ou borbulhamento). A estratégia de ordenação consiste em varrer os dados do arranjo, do fim para o início e, sempre que encontrado um par de elementos adjacentes fora de ordem, fazer a troca. Após um número de passadas pelo arranjo, a ordenação será obtida. Na primeira varredura, o menor elemento migrará para a primeira posição; a cada passagem subsequente, o próximo menor elemento migra para sua posição.

Sua estratégia é descrita por:

1. Faça a varredura, do fim ao início do arranjo, trocando os itens adjacentes que estejam fora de ordem.
2. Esta mesma estratégia é aplicada sucessivamente, até que a ordenação seja obtida.

Algoritmo 3-3

```
1   bubblesort(dados[]: inteiro; tamanho: inteiro)
2       para i ← 0 até tamanho – 2 faça
3             { varredura }
4             j ← tamanho – 1
5             enquanto j > i faça
6                   se dados[j] < dados[j – 1] então
7                         { troca se forem adjacentes }
8                         auxiliar ← dados[j]
9                         dados[j] ← dados[j – 1]
10                        dados[j – 1] ← auxiliar ]
11                  fim-se
12                  j ← j – 1
13            fim-enquanto
14      fim-para
```

Como característica de funcionamento, pesa para o *bubblesort* o fato de o número de trocas poder ser elevado. Isso acontece, por exemplo, quando o menor elemento do arranjo está na última posição. Na primeira varredura, é feita sua troca com o elemento da posição anterior (exigindo três movimentações entre variáveis – linhas 8 a 10); decrementando **j**, nova troca é feita (mais três movimentos); e isso ocorre para cada valor de **j** na primeira varredura.

A análise do algoritmo do *bubblesort* permite verificar que o número total de repetições é fixo. O laço mais externo do **para** repete *n* – 1 vezes; a repetição interna, comando **enquanto**, repete uma vez a menos cada vez, mas é também proporcional a *n*. O algoritmo como um todo é pertencente a $O(n^2)$. O importante a notar é que, embora tenha mesma complexidade dos anteriores, isso significa apenas que seu comportamento é limitado por uma quadrática quando *n* cresce. Dado o custo das trocas necessárias, seu desempenho costuma ser bastante inferior aos demais métodos.

3.2.2 - Métodos avançados de ordenação

Para as estratégias mais sofisticadas de ordenação para memória principal serão considerados três métodos: ***shellsort***, ***heapsort*** e ***quicksort***. Cada um deles possui uma estratégia distinta e, também, características distintas.

O ***shellsort*** é um método inspirado na estratégia do inserção direta e se aproveita de seu melhor caso, que é quando o arranjo já está praticamente ordenado. Ao invés de comparar elementos adjacentes, as comparações do *shellsort* são feitas entre elementos relativamente distantes entre si. Esta distância entre os elementos é chamada **incremento**. Assim, para um incremento valendo 5, as comparações são feitas entre os elementos das posições 0, 5, 10 etc.; das posições 1, 6, 11 etc.; das posições 2, 7, 12 etc. e assim por diante. Cada um destes sub-conjuntos é chamado **subsequência**. Na prática, para um dado valor de incremento *p*, são criadas *p* subsequências, cada uma com (aproximadamente) *n*/*p* elementos em cada uma.

A estratégia geral do *shellsort* pode ser dada por:

1. Defina alguns incrementos diferentes p_k ($p_k > p_{k+1}$) e, para cada incremento ordene cada subsequência usando a estratégia do método inserção direta.
2. Aplique, como última etapa, o inserção direta para garantir a ordenação completa.

Para cada incremento diferente, as várias subsequência são ordenadas de forma individual. Uma vantagem desta abordagem é a movimentação de dados a "distâncias" razoáveis dentro do arranjo. Por exemplo, para um incremento igual a 43, as trocas ocorrerão diretamente entre as posições *i* e *i* – 43. Para um dado valor de incremento, o que ocorre apenas é uma organização grosseira do arranjo, com os valores menores tendendo a ir (dentro de sua própria subsequência) para o início do arranjo e os maiores ficando mais para o final.

Como este processo é aplicado para vários incrementos diferentes, cada um menor que seu anterior, a cada etapa as subsequências ficam com mais elementos e as "distâncias" entre as comparações se reduz. Isso significa que o arranjo tende a ficar com uma ordenação melhor a cada passo. Adicionalmente, a cada novo incremento utilizado, cada subsequência ordenada já se aproveita das ordenações grosseiras dos passos anteriores, de forma que a estratégia do inserção direta se aproveita das reduções das comparações e deslocamentos de dados da repetição interna.

O último passo é a aplicação do inserção direta, o que equivale efetivamente a usar incremento igual a 1. Quando isso acontece, o arranjo já tem sua ordenação geral bastante melhorada e o melhor caso da ordenação é válido para praticamente todo o arranjo.

Pelo uso desta estratégia, a ordenação passa a ter desempenho substancialmente superior a qualquer um dos métodos básicos apresentados na seção anterior.

Uma questão em aberto no algoritmo é a determinação de quais são os incrementos que devem ser usados. De forma geral, testes empíricos sugerem que bons desempenhos podem ser obtidos seguindo-se algumas sugestões* simples:

1. O último passo usa incremento 1 (o que equivale à aplicação do inserção direta).
2. Cada incremento anterior pode ser calculado como o incremento do próximo passo multiplicado por 3 e acrescido de 1. Assim, se um incremento for *p*, então o incremento anterior deve ser 3*p* + 1.
3. O número total de incrementos diferentes a ser usados deve ser equivalente a $\log_3 n$.

Como exemplo, para um vetor com 100.000 elementos devem ser usados 10 incrementos diferentes ($\log_3$ 100.000 vale aproximadamente 10,48). Cada incremento fica definido como: 29524, 9841, 3280, 1093, 364, 121, 40, 13, 4 e 1 (calculados do último para o primeiro).

A análise da complexidade do *shellsort* não é trivial e muito do seu desempenho foi medido por análises experimentais. De forma geral, funções na forma $T(n) = n\log^2 n$ ou mesmo $T(n) = n\log n$ representam satisfatoriamente seu comportamento. Em geral, o *shellsort* é considerado um algoritmo $O(n\log^2 n)$.

Algoritmo 3-4

```
1  shellsort(dados[]: inteiro, tamanho:inteiro)
2        { define os incrementos }
3        númeroIncrementos ← log(tamanho)/log(3)
4        p[0] = 1 { último incremento }
5        para i ← 1 até númeroIncrementos - 1 faça
6              p[i] ← p[i – 1] * 3 + 1
7        fim-para
```

* Há algumas variações destas sugestões, dependendo dos diversos autores. Portanto, elas não podem ser consideradas regras.

```
8
9       { ordenação das subsequências para cada incremento }
10      i ← númeroIncrementos – 1  { começa pelo maior }
11      enquanto i > 0 faça
12            { ordena cada subsequência }
13            incremento ← p[i]
14            para j ← incremento até tamanho – 1 faça
15                  auxiliar ← dados[j]
16
17                  { deslocamentos dentro da mesma subsequência }
18                  k ← j – incremento
19                  enquanto dados[k] > auxiliar e k ≥ 0 faça
20                        dados[k + incremento] ← dados[k]
21                        k ← k - incremento
22                  fim-enquanto
23                  dados[k + incremento] ← auxiliar
24            fim-para
25
26            { passa para o próximo incremento }
27            i ← i - 1
28      fim-enquanto
```

O Algoritmo 3-4 mostra uma implementação para o *shellsort*, que é um "algoritmo de inserção" (que insere um valor em relação aos demais já ordenados). Para entender a parte de ordenação das subsequências é interessante notar que elas não são tratadas separadamente. Na realidade, a repetição da linha 14 passa por todos os elementos do arranjo, mas o uso da variável **incremento** (nas linhas 18 a 21) garantem que cada comparação seja restrita a sua própria subsequência.

Outro exemplo de "algoritmo de seleção", ou seja, daquele que seleciona qual item deve ocupar cada posição, é o *heapsort*. A cada passo, um elemento é selecionado para ocupar cada posição do arranjo.

Para tornar a escolha do item selecionada a cada passo mais eficiente, os dados são inicialmente arrumados para permitirem uma busca binária. Esta organização recebe o nome de **monte**, que segue a estrutura de uma árvore binária.

Esta solução é uma alternativa bastante prática, já que a árvore é organizada no próprio arranjo de dados, sem a necessidade de qualquer estrutura adicional. O arranjo pode ser visto como uma árvore binária se, considerando que a posição *i* do arranjo representa um nó, as posições 2*i* + 1 e 2*i* + 2 forem usadas para armazenar os filhos esquerdo e direito. A raiz é escolhida como a posição zero, ou seja, o primeiro elemento do vetor. A determinação de qual nó é pai ou filho de outro se torna simples e dispensa os usuais ponteiros.

Como exemplo, tome-se o arranjo de inteiros da Figura 3-1. A visão usual dos dados do arranjo é como uma lista linear de dados. Porém, considerando-se a posição zero a raiz, as posições 1 e 2 são seus filhos esquerdo e direito, quando a visão é alterada para uma árvore. Desta forma, a posição 3 do arranjo tem seu filho esquerdo na posição 7 e o filho direito na posição 8. A árvore em si não existe, mas o arranjo é usado como se fosse uma árvore, apenas pela manipulação adequada dos índices das posições.

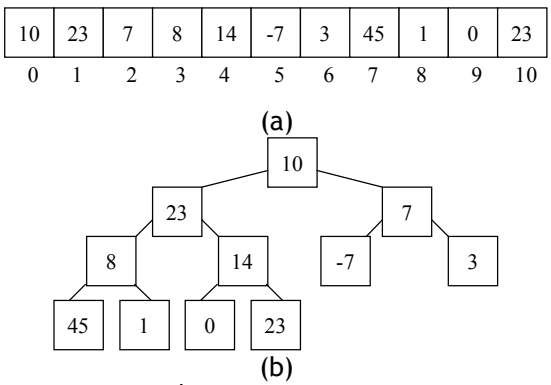

Figura 3-1. Ilustração de um arranjo de números inteiros com 11 posições. (a) Visão usual dos dados no arranjo; (b) Visão do arranjo como uma árvore binária.

O monte[*] é uma árvore binária com a seguinte propriedade: o valor de um dado nó é sempre maior ou igual ao de seus dois filhos. Desta propriedade decorre que a raiz armazena o maior elemento da árvore.

[*] No original, monte é chamado *heap*.

Para a ordenação pelo *heapsort*, então, é preciso inicialmente criar o monte com os dados do arranjo. Assim os elementos são trocados de posição até que os elementos do arranjo, sob a visão de árvore binária, formem um monte.

O algoritmo para realizar esta tarefa é apresentado no Algoritmo 3-5. Para cada elemento do arranjo, do segundo até o último, é feita sua "subida" na árvore em direção à raiz, desde que seu valor seja maior que o de seu pai. Assim, pela estrutura da árvore, os valores maiores migram, um a um, até sua posição correta na árvore. O resultado é que o maior valor fica armazenado na raiz. A complexidade desta fase é O(*n*log *n*): o comando **para** é linear, enquanto a repetição interna do **enquanto** é O(log *n*), pois usa a estrutura binária da árvore.

Algoritmo 3-5

```
{ criação do monte }
para i ← 1 até tamanho – 1 faça
      auxiliar ← dados[i]
      filho ← i
      pai ← (filho – 1)/2  { posição do pai }
      enquanto filho > 0 e dados[pai] < auxiliar faça
            dados[filho] ← dados[pai]
            filho ← pai
            pai ← (filho – 1)/2
      fim-enquanto
      dados[pai] ← auxiliar
fim-para
```

Criado o monte, pode ser feita então a ordenação por seleção. Para esta segunda fase, é feita a seleção do maior elemento do arranjo (que está na raiz) e este é trocado com o que está na última posição do arranjo. Com a troca, o monte é deixado incorreto, pois na raiz há um elemento qualquer. O ajuste do monte é simplesmente conseguido fazendo com que o item que acabou de ser colocado na raiz "desça" na estrutura da árvore até sua posição. O elemento ordenado é considerado fora do monte.

O processo é repetido para o segundo maior elemento, que agora está na raiz da árvore, fazendo-se a troca para a posição correta e novo ajuste do monte. O processo é repetido até a raiz.

O monte é usado para que a seleção dos elementos seja eficiente, permitindo que, um a um, cada valor seja colocado na posição correta, do maior até o menor.

Algoritmo 3-6

```
{ ordenação com uso do monte }
i ← tamanho – 1
enquanto i < 2 faça
      { coloca o valor da raiz na posição definitiva }
      auxiliar ← dados[0] { raiz }
      dados[i] ← dados[0]
      dados[0] ← auxiliar

      { corrige o monte }
      auxiliar ← dados[0]  { valor incorreto, colocado na raiz }
      pai ← 0
      se dados[2*pai + 1] > dados[2*pai + 2] e 2*pai + 2 ≥ i então
            filho ← 2*pai + 1
      senão
            filho ← 2*pai + 2
      fim-se
      enquanto pai < i e dados[filho] < dados[pai] faça
            dados[pai] ← dados[filho]
            pai ← filho
            se dados[2*pai + 1] > dados[2*pai + 2] e 2*pai + 2 ≥ i então
                  filho ← 2*pai + 1
            senão
                  filho ← 2*pai + 2
            fim-se
      fim-enquanto
      dados[filho] ← auxiliar
fim-enquanto
```

No Algoritmo 3-6 é apresentada a parte da ordenação do *heapsort*. A troca da raiz (maior elemento atualmente no monte) coloca o valor na posição correta, pois o maior vai para a última posição, o segundo maior para a penúltima posição e assim por diante. O reajuste do monte é feito fazendo com que o valor colocado na raiz desça pela árvore até sua posição correta. Para isso se escolhe qual de seus dois filhos possui valor maior, o que determina qual ramo da árvore será usado para a descida. A descida pela árvore é proporcional a log *n* e o processo é repetido para cada elemento do arranjo. A complexidade da fase de ordenação é, portanto, O(*n*log *n*).

Em termos gerais, o *heapsort* possui duas fases O(*n*log *n*) consecutivas e é, portanto, O(*n*log *n*) como um todo.

O algoritmo do *heapsort* não faz qualquer consideração sobre como os dados estão no arranjo. Isso significa que, mesmo que os dados já estejam ordenados, haverá a troca de suas posições para que o monte seja criado e depois a ordenação seja feita. O custo total do tempo de execução é distribuído praticamente metade para cada fase, o que o torna custoso para arranjos pequenos, mas interessante para arranjos com grade quantidade de itens para ser ordenados.

É importante salientar que, usualmente, a implementação do *heapsort* se apresenta em algumas formas diferentes. Este é o caso da implementação indicada por Drozdek [6]. Os princípios discutidos aqui, entretanto, estão presentes.

Existe, ainda, um último algoritmo para ordenação, o qual é amplamente utilizado. Ele é chamado de *quicksort*, o que quer dizer "método rápido de ordenação". Essencialmente recursivo em sua implementação, utiliza algoritmo de particionamento para efetuar a ordenação.

Algoritmo 3-7

```
1  partição(dados[]: inteiro, início, fim: inteiro, var esquerda, direita: inteiro)
2  { particionamento dos dados entre as posições início e fim do arranjo }
3        esquerda ← início
4        direita ← fim
5        pivô ← dados[(início + fim)/2] { elemento do meio }
6        enquanto esquerda ≤ direita faça
7              { procura um valor maior ou igual ao pivô }
8              enquanto dados[esquerda] < pivô faça
9                    esquerda ← esquerda + 1
10             fim-enquanto
11
12             { procura um valor menor ou igual ao pivô }
13             enquanto dados[direita] > pivô faça
14                   direita ← direita – 1
15             fim-enquanto
16
17             { faz a troca }
18             se esquerda ≤ direita então
19                   auxiliar ← dados[esquerda]
20                   dados[esquerda] ← dados[direita]
21                   dados[direita] ← auxiliar
22                   direita ← direita – 1
23                   esquerda ← esquerda + 1
24             fim-se
25       fim-enquanto
```

O Algoritmo 3-7 mostra o particionamento. Para um dado intervalo dentro do arranjo, delimitado pelos índices **início** e **fim**, são feitas as trocas necessárias. Um valor de referência, chamado *pivô*, é escolhido e as trocas seguem um princípio simples: todos os valores menores que **pivô** devem ficar à esquerda (isto é, no início do intervalo) e os maiores à direita. Os índices **esquerda** e **direita** são usados para buscar os valores e as trocas são feitas conforme necessário. Deve-se notar que este procedimento não se preocupa com qualquer aspecto da ordenação, pois apenas separa os dados em dois conjuntos: menores ou iguais ao pivô de um lado, maiores ou iguais ao pivô do outro[*]. A propriedade mais interessante deste particionamento é que, depois de concluído, cada partição se torna independente. Em outras palavras, os elementos que ficaram em uma partição devem apenas ser ordenados naquela partição e nunca ocorrem trocas entre elementos de partições diferentes.

O *quicksort* aplica o particionamento descrito para o arranjo como um todo e depois para cada partição criada. As partições ficam, então cada vez menores e, quando não atingem tamanho menor que 2, significa que estão ordenadas.

Algoritmo 3-8

```
1  quicksort(dados[]: inteiro, tamanho: inteiro)
2        quickRecursivo(dados, 0, tamanho – 1)
3
4
5  quickRecursivo(dados[]: inteiro, início, fim: inteiro)
6        partição(dados, início, fim, esquerda, direita)
7
8        { ordena primeira partição }
9        se início < direita então
10             quickRecursivo(dados, início, direita)
```

[*] Os valores que forem iguais ao do pivô podem ficar tanto na partição da esquerda quanto da direita, sendo indiferente para o funcionamento do algoritmo.

```
11      fim-se
12
13      { ordena segunda partição }
14      se esquerda < fim então
15            quickRecursivo(dados, esquerda, fim)
16      fim-se
```

No Algoritmo 3-8, **quicksort** é usado para chamar a parte recursiva do algoritmo, definindo o arranjo inteiro como a partição inicial. Para cada chamada de **quickRecursivo**, a partição é sub-dividida em duas novas partições e cada uma delas, se tiver mais que um elemento, é repassada para nova sub-divisão. A primeira partição é definida de **início** até **direita** e a segunda de **esquerda** até **fim**.

Em termos de desempenho, o *quicksort* tem seu melhor caso quando o pivô escolhido para a partição leva à criação de duas novas partições de tamanhos equivalentes (idealmente dividindo a partição anterior em duas de mesmo comprimento). Neste caso, o algoritmo tem desempenho O($n$log n). Entretanto as partições podem ter tamanhos muito diferentes, o que é prejudicial. No pior caso, a escolha do pivô faz com que uma partição de k elementos seja subdividida em uma partição de tamanho 1 e outra de k – 1. Nesta situação crítica e isso ocorrendo para todos os particionamentos, o *quicksort* cai para desempenho O(n^2). O fato é que, para arranjos usuais, o pior caso é uma hipótese remota e o desempenho O($n$log n) é a situação comum como caso médio.

Comparativamente aos outros métodos de ordenação mais avançados (*shellsort* e *heapsort*), o *quicksort* é bastante leve e apresenta, para arranjos com muitos elementos, um desempenho geralmente bastante superior aos demais.

3.3 - Ordenação na memória secundária

Diferentemente dos métodos usados em memória principal, a ordenação em memória secundária tem, necessariamente, que se preocupar com os acessos a disco. Como o acesso a dados em disco é feito por blocos, leituras e gravações desnecessárias se tornam uma penalização excessiva para o tempo de execução.

O princípio das técnicas usadas para ordenação em disco é tentar fazer acesso sequencial aos dados, o que minimiza acesso a registros isolados. Assim, todos os registros lidos em um bloco são processados.

Porém, inicialmente, será apresentada a solução padrão de um método de ordenação por fusão, conhecido como *mergesort*. O método se baseia na "fusão" de listas ordenadas, ou seja, na criação de uma lista ordenada a partir de duas outras listas também já ordenadas.

A criação de uma lista ordenada a partir de outras duas também ordenadas é simples. É sempre necessário ver o início de cada lista, sendo que o menor é movido para lista de saída. Este processo é repetido até que uma das duas listas se esgote e, então, o restante dos itens da lista remanescente é movido para a lista de saída. Este processo é exemplificado, para memória principal, para o caso da geração de um arranjo, chamado **listaSaída**, a partir de outros dois, **lista1** e **lista2** (Algoritmo 3-9).

Algoritmo 3-9

```
1  { fusão de duas listas ordenadas
2  i ← 0
3  i1 ← 0
4  i2 ← 0
5  enquanto i1 < tamanhoLista1 e i2 < tamanhoLista2 faça  { até o fim de uma das listas }
6        se lista1[i1] < lista2[i2] então
7              listaSaída[i] ← lista1[i1]
8              i1 ← i1 + 1
9        senão
10             listaSaída[i] ← lista2[i2]
11             i2 ← i2 + 1
12       fim-se
13       i ← i + 1
14 fim-enquanto
15
16 enquanto i1 < tamanhoLista1 faça      { restante da lista 1, se houver }
17       listaSaída[i] ← lista1[i1]
18       i ← i + 1
19       i1 ← i1 + 1
20 fim-enquanto
21
22 enquanto i2 < tamanhoLista2 faça      { restante da lista 2, se houver }
23       listaSaída[i] ← lista2[i2]
24       i ← i + 1
25       i2 ← i2 + 1
26 fim-enquanto
```

Para usar este processo na ordenação, parte-se do princípio que uma lista com um único elemento se encontra sempre ordenada. Então, duas listas de um elemento cada são ordenadas em outra lista com dois elementos; depois, listas com 2 elementos são ordenadas em listas com 4 itens, e assim sucessivamente até que se obtenha, por fusões sucessivas, a ordenação de todo o conjunto.

Um detalhe importante é que a fusão de listas requer que haja espaço disponível para armazenar a lista de saída. Uma observação atenta do código do Algoritmo 3-9 permite verificar que as variáveis **lista1** e **lista2** não podem ser reaproveitadas para guardar a saída, de forma que outra variável, **listaSaída**, é necessária para armazenar o resultado da intercalação.

O *mergesort* faz uma abordagem recursiva para a ordenação, dividindo o arranjo a ser ordenado pela metade passo a passo, até que cada parte tenha comprimento igual a 1 (ou seja, um único elemento). A partir daí, volta fazendo as fusões sucessivas, até obter a ordenação completa.

Algoritmo 3-10

```
1  mergesort(dados[]: inteiro, tamanho: inteiro)
2       mergeRecursivo(dados, 0, tamanho – 1)  { chama recursivamente para todos os dados }
3
4
5  mergeRecursivo(dados[]: inteiro, início, fim: inteiro)
6       { ordena recursivamente duas partições }
7       se início < fim então
8             meio ← (início + fim)/2
9             mergeRecursivo(dados, início, meio)    { primeira metade }
10            mergeRecursivo(dados, meio + 1, fim)   { segunda metade }
11
12            { fusão das metades ordenadas }
13            fusão(dados, início, fim)
14      fim-se
```

O *mergesort* é apresentado no Algoritmo 3-10, no qual a primeira parte faz apenas a chamada para o procedimento recursivo. O procedimento **mergeRecursivo** faz o trabalho, dividindo todos as partições do arranjo (se forem maior que 1) em duas e acionando o procedimento para cada uma delas.

Ao terminar a execução das duas chamadas **mergeRecursivo** das linhas 9 e 10, as duas partições estarão ordenadas. Então, o procedimento de fusão (linha 13) é chamado para fazer a intercalação dos dois segmentos do arranjo. Na prática, o procedimento **fusão** é uma adaptação da fusão do Algoritmo 3-9. Em **fusão**, os equivalentes a **lista1** e **lista2** são partes do arranjo, definidas de **início** até **início + meio** e de **meio + 1** até **fim**, sendo **meio** igual a **(início + fim)/2**. Um arranjo temporário é necessário para guardar o resultado da fusão, a qual, quando terminada, é copiada de volta para o arranjo.

Em termos de desempenho, o tempo de execução do *mergesort* é O(*n*log *n*), o que o equipara aos algoritmos avançados já descritos. O detalhe é que, necessariamente, um arranjo auxiliar tem que ser alocado para guardar os valores das intercalações, exigindo o dobro do espaço originalmente usado para guardar o vetor de dados.

A utilização do *mergesort* para arquivos utiliza algumas modificações importantes. Em primeiro lugar, as listas a serem mescladas estão em disco, assim como em disco deve ser armazenado o resultado da fusão. Realizar a fusão de apenas duas listas também é contraproducente, dado que o número de vezes que cada item deve lido para a memória aumenta. Finalmente, não é de se esperar que o início seja a fusão de uma grande quantidade de arquivos com apenas um registro.

Uma solução para a ordenação de dados em arquivo, portanto, pode ser estabelecida pelos seguintes passos:

1. Carregar tantos registros do arquivo quanto caibam na memória principal, ordenando-os com um método apropriado para memória principal (*quicksort*, por exemplo) e gravando-os como uma sequência ordenada em um arquivo separado.
2. Realizar a fusão de *k* arquivos com listas ordenadas, gerando uma nova lista ordenada, a qual é também armazenada em disco. Repetir o processo de fusão com as listas existentes até que uma única lista seja criada.

Supondo que o arquivo seja bastante grande e que não caibam todos os dados em memória, o passo 1 acima viabiliza a criação de listas ordenadas com um conjunto razoável de registros. Como o custo para recuperar os dados do disco e regravá-los é mais significativo que a ordenação em memória principal, o custo da ordenação inicial de cada lista não é tão relevante no processo como um todo. Esta primeira fase é completada fazendo-se apenas uma leitura e uma gravação de cada registro em disco.

A fusão de *k* arquivos é feita adaptando-se a fusão de listas para *k* listas, e não apenas para 2, como apresentado no *mergesort* original. O valor de *k* depende de quantos blocos (um de cada arquivo a ser intercalado) podem ser mantidos simultaneamente em memória principal. Para cada fusão, os dados são lidos apenas uma vez do disco e gravados também apenas uma vez. Se o valor de *k* permitir que todos

os arquivos do primeiro passo sejam fundidos de uma vez (ou seja, se o número de sequências geradas for menor ou igual a *k*), então cada registro será lido e escrito apenas mais uma vez.

3.4 - Considerações finais

A necessidade de ordenação é de incontestável importância e a escolha de um método adequado para cada situação é essencial. Para tanto, foram apresentados os principais métodos elementares de ordenação, cujos tempos são $O(n^2)$, englobando o *inserção direta*, o *seleção direta* e o *bubblesort*. Para estes métodos, o desempenho para quantidades pequenas de dados é razoável, o que permite fazer a escolha pelos dois primeiros em algumas situações. O *bubblesort*, pelo grande número de trocas, não deve ser usado.

Para os métodos avançados em memória principal, sua aplicação é para grandes volumes de dados que possam ser simultaneamente guardados em um vetor. A opção pelo *shellsort*, *heapsort* e *quicksort* é, em geral, indiferente. Porém, o *quicksort* é, de longe, o mais popular, com um desempenho excepcional. O *shellsort* necessita de um arranjo auxiliar para guardar os incrementos e o *heapsort* não considera pré-ordenações existentes.

Finalmente, quando a situação recai para a ordenação de dados em arquivos, mudam todos os critérios que devem ser considerados. Como a leitura e escrita de dados passam a ser o cerne do consumo de tempo, quanto menos os dados forem lidos e gravados, melhor o desempenho final. O uso de alternativas mistas como a descrita, fazendo ordenações em memória principal do que for possível e usando o *mergesort* adaptado para realizar as fusões necessárias, caracteriza uma estratégia importante para esta situação.

Não existem métodos absolutamente piores ou melhores. A situação da aplicação é que determina a escolha por um ou outro método de ordenação.

3.5 - Sugestões de leitura

Como leitura suplementar, sugere-se que outros métodos de ordenação sejam vistos. Em particular, o *radixsort* é bastante interessante.