Algoritmos, Ordenação

Prof. Ricardo Reis
Universidade Federal do Ceará
Sistemas de Informação
Estruturas de Dados

23 de Abril de 2013

1 Introdução

Ordenar objetos que seguem algum tipo de classificação é uma tarefa comum que as pessoas executam diariamente. Um conjunto de objetos, seja de números, livros, nomes próprios e etc, é mais facilmente manipulado quando está ordenado. A ordenação requer um *critério*, por exemplo, ao rearranjar uma prateleira de livros pode-se fazê-lo pela altura do livro, pelo ano de publicação, pelo título, pela espessura do volume e etc. Quando ocorre repetição de objetos então um processo de ordenação poderá aplicar sucessivos critérios, por exemplo, no caso dos livros, se estiverem sendo ordenados por altura então livros de mesma altura poderão ser ordenados por ano de publicação e os de mesmo ano de publicação poderão ser classificados por título.

No que diz respeito a computadores, existem muitos algoritmos para efetuar ordenação de dados e suas construções são baseadas basicamente no tipo de memória onde a informação é armazenada e na estrutura de dados utilizada. Neste artigo trataremos a ordenação em memória principal utilizando vetores como estruturas de armazenamento.

O princípio mais usual de ordenação de um conjunto genérico de objetos é a *troca*, ou seja, dado um vetor de entrada L e um critério válido pode-se então escrever um algoritmo que ordena L através de um número finito de trocas. Uma troca é o processo pelo qual duas variáveis (ou duas células de um vetor) têm seus respectivos conteúdos alternados pelo uso de uma variável intermediária. No esquema,

$t \leftarrow a$
$a \leftarrow b$
$b \leftarrow t$

as variáveis a e b são trocadas utilizando t como variável intermediária. O número de trocas em um algoritmo de ordenação é em geral função do comprimento n do vetor L a ser ordenado e muitas vezes é utilizado como equivalente a complexidade do algoritmo.

A função SWAP, Algoritmo-1, implementa para um vetor de entrada L, a troca dos dois elementos pertencentes a L de posições p e q. Esta função será usada nas sessões seguintes.

Algoritmo 1 Troca de elementos entre duas posições de um vetor.

```
1: Função SWAP(ref L, p, q)
2:     x ← L[p]
3:     L[p] ← L[q]
4:     L[q] ← x
```

2 Ordenação Elementar

Os algoritmos mais simples para ordenação dos elementos de um vetor de comprimento n possuem complexidade de ordem $O(n^2)$ e são denominados *métodos de ordenação elementares*. Os mais populares são ordenação por borbulhamento (*BubbleSort*), ordenação por seleção (*SelectionSort*) e ordenação por inserção (*InsertionSort*). Muitos outros métodos de mesma ordem de complexidade são variações destes três.

2.1 Ordenação por Borbulhamento

Consideremos um grupo de n pessoas que estão de pé uma do lado da outra em linha reta e posicionadas ombro-com-ombro. Cada uma tem uma carta numerada na mão (todas distintas) e não é permitido ficar com mais de uma carta ou sem nenhuma carta. Inicialmente as cartas não estão ordenadas na sequência das pessoas. Para fazer a ordenação devem ser efetuadas trocas de cartas apenas entre pessoas vizinhas. Uma ideia para ordenar as cartas sobre tais restrições é descrita a seguir. A primeira pessoa compara sua carta com a da segunda pessoa e caso a sua tenha um número maior então elas trocam suas cartas. O mesmo processo ocorre entre a segunda e terceira pessoa, depois entre a terceira e a quarta e assim sucessivamente até atingir a comparação entre a penúltima e última pessoa (posições $n-1$ e n). No final deste processo a carta de maior número atinge a mão da última pessoa (n) que deve dar um passo para trás indicando aos $n-1$ colegas restantes que repitam o processo sem sua participação. Na segunda rodada a carta que atinge

a última pessoa é a maior daquele grupo e segunda maior do grupo completo. Então esta pessoa dá um passo para trás para se unir ao colega que fez o mesmo na primeira rodada e outras rodadas se procedem até que todos tenham dado um passo para trás. O efeito final é naturalmente as cartas classificadas em ordem crescente de seus valores.

Este processo é conhecido como *ordenação por borbulhamento* ou *BubbleSort*. O termo *Bolha* surge da abstração que uma bolha encobre o par de posições vizinhas onde ocorre a troca corrente. A proporção que as trocas são feitas tem-se o efeito de um *andar de bolha*. Além do mais em cada rodada uma nova bolha parte em direção a um *teto mais baixo* que o da rodada anterior (conceber a lista de objetos verticalmente facilita a analogia pois permite visualizar as bolhas subindo).

A Figura-1 ilustra as rodadas em uma ordenação por borbulhamento, e suas respectivas trocas, no vetor [7, 8, 5, 1, 3, 9, 2, 6, 4] . Os pares de células vizinhas marcadas com molduras referem-se a comparação corrente do processo e podem ou não ocasionar uma troca. As células em tom escuro representam os elementos que já atingiram suas posições definitivas.

O Algoritmo-2 implementa a ordenação por borbulhamento para um vetor de comprimento n passado como argumento. A variável t representa o índice limite (*teto*) até onde uma dada bolha-de-troca pode ir numa rodada em específico. Como os maiores elementos encontram suas posições finais gradativamente então este limite diminui (*abaixamento de teto*). O controle de subida de bolhas e eventuais trocas é feito pela variável j e por isso em cada rodada inicia sempre em 1 (linha-4). Trocas entre posições vizinhas (j e $j+1$) ocorrem na linha-7.

Algoritmo 2 Ordenação por Borbulhamento, *BubbleSort*

```
1: Função BubbleSort(ref L, n)
2:     t ← n                                  ▷ Partida do teto
3:     Enquanto t > 1 faça                    ▷ Teto desce?
4:         j ← 1                              ▷ Partida da bolha
5:         Enquanto j < t faça                ▷ Bolha sobe?
6:             Se L[j] > L[j+1] então         ▷ Troca?
7:                 Swap(L, j, j+1)            ▷ Troca
8:             j ← j + 1                      ▷ Bolha sobe
9:         t ← t − 1                          ▷ Teto desce
```

Atentemos a presença do par de laços aninhados no Algoritmo-2. O laço mais externo (controle de teto), de contador t, possui $n-1$ iterações onde n é o comprimento do vetor de entrada, L. Esta medida é feita através de t que inicia em n e encerra em 2 (teto mínimo). Matematicamente o total de iterações é calculada por $pos_{max} - pos_{min} + 1 = (n) - (2) + 1 = n - 1$. A quantidade de iterações do laço mais interno, de contador j, depende da iteração do laço externo onde ela procede, e logo é função de t. O valor de j inicia em 1 e termina em $t-1$ (o teto não é incluído neste laço porque as comparações ocorrem aos pares entre posições vizinhas e o segundo elemento da última comparação precisa ser t). Assim o total de iterações do laço interno é $pos_{max} - pos_{min} + 1 = (t-1) - (1) + 1 = t - 1$.

Meçamos agora a complexidade da ordenação por borbulhamento $T(n)$ de um vetor L de comprimento n. É prática comum fazer isso contabilizando o número de trocas. No pior caso todas as trocas possíveis ocorrem (vetor inicialmente ordenado em ordem decrescente) e logo $T(n)$ corresponde a medida das associações de iterações dos laços externo e interno. Matematicamente,

$$\begin{aligned} T(n) &= \sum_{t=2}^{n} (t-1) \\ &= 1 + 2 + 3 + \cdots + (n-1) \\ &= \frac{n(n-1)}{2} \\ &= \frac{n^2 - n}{2} \\ &= O(n^2) \end{aligned}$$

Este resultado demonstra que de fato o *BubbleSort* é um método de ordenação elementar.

2.2 Ordenação por Seleção

Um professor está em sala de aula e resolve classificar os alunos em cada fileira em ordem crescente de idade. Então ele chega em uma das fileiras, pergunta a idade do primeiro aluno e depois lança aos demais a pergunta, *Alguém adiante deste colega é mais jovem que ele?*. Caso haja então os dois alunos se levantam e trocam de posição. Depois o professor passa a segunda posição da fileira, pergunta ao aluno ali sua idade e novamente lança a mesma pegunta. O processo continua até que ele atinja a penúltima posição onde lança pela última vez a pergunta direcionada neste caso exclusivamente ao último aluno da fileira. Após as perguntas e eventuais trocas a fileira de alunos se torna ordenada por idade.

Este processo de ordenação é conhecido como *ordenação por seleção*, ou *SelectionSort*. O nome se deve ao fato de em cada rodada do processo ocorrer a *seleção* do aluno mais jovem entre a posição avaliada (onde o professor para) e a posição final. Em outras palavras, cada parada do professor numa posição k, em uma fileira de n alunos, identifica o aluno mais jovem entre as posições k e n e o coloca na posição k (através de uma troca, se necessária). Os alunos atrás do professor (entre as posições 1 e $k-1$) estarão ordenados em qualquer etapa e toda fileira estará ordenada ao final.

A Figura-2 ilustra todas as rodadas de uma ordenação por seleção aplicada ao vetor [7, 8, 5, 1, 3, 9, 2, 6, 4] . Os pares de células marcadas com

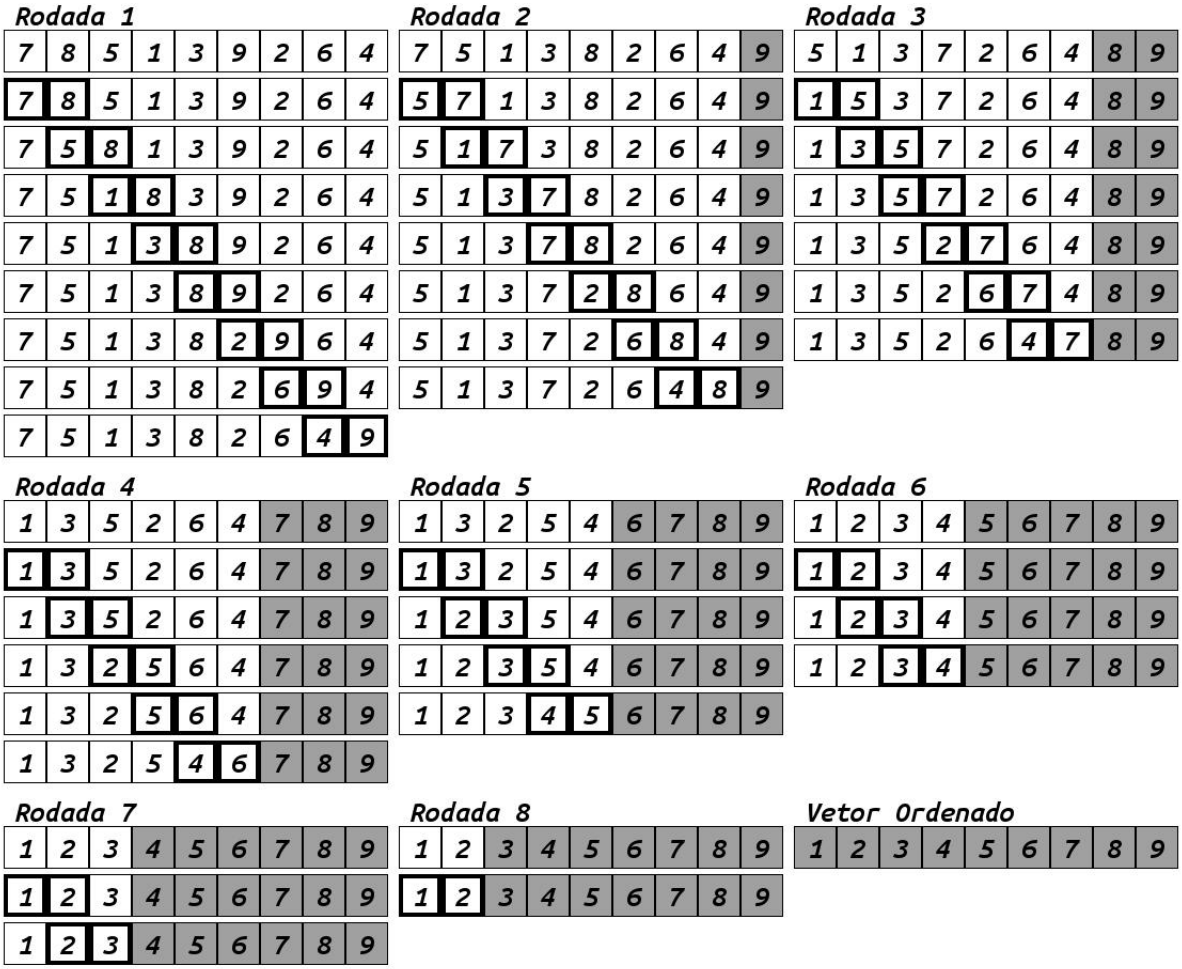

Figura 1: Ordenação por borbulhamento (BubbleSort).

bordas escuras em cada rodada indicam os elementos trocados e as células em tom escuro representam os elementos que já atingiram a posição final.

O Algoritmo-3 implementa a ordenação por seleção para um vetor de comprimento n passado como argumento. Como na ordenação por borbulhamento, na ordenação por seleção existem também dois laços aninhados. O laço mais externo (linha-2) controla o que equivale à posição onde o professor para e efetua o teste de idade. Seu contador é i que vai de 1 (primeiro aluno da fileira) até $n-1$ (penúltimo aluno da fileira). Para se identificar o menor elemento de cada rodada (aluno mais jovem) utiliza-se o índice auxiliar min que parte da hipótese que a menor chave, entre as posições i e n, está na posição i (linha-3). O laço interno (linha-4) testa as demais posições (entre $i+1$ e n) atualizando min sempre que um elemento menor que $L[min]$ for encontrado (teste na linha-5). Encerrado o laço interno, o elemento mínimo encontrado é trocado com o elemento da posição i (linha-7).

Note que as chamadas a SWAP(()) efetuadas na ordenação por seleção ocorrem dentro do laço mais externo contabilizando um total de trocas de ordem $O(n)$. Assim, devido a desconsideração do laço interno, o total de trocas não representa corretamente

Algoritmo 3 Ordenação por Seleção *SelectionSort*

```
1: Função SELECTIONSORT(ref L, n)
2:     Para i ← 1 até n − 1 faça
3:         min ← i                                  ▷ Mínimo?
4:         Para j ← i + 1 até n faça
5:             Se L[j] < L[min] então               ▷ Menor?
6:                 min ← j
7:         SWAP(L, i, min)                          ▷ Troca
```

a complexidade do algorítimo. Ao invés das trocas é mais conveniente medir a complexidade $T(n)$ como o número de testes efetuados na linha-5.

O número de iterações do laço interno é $pos_{max} - pos_{min} + 1 = (n-1) - (1) + 1 = n-1$. O laço interno por sua vez tem um total de iterações $pos_{max} - pos_{min} + 1 = (n) - (i+1) + 1 = n - i$. Como há um teste por

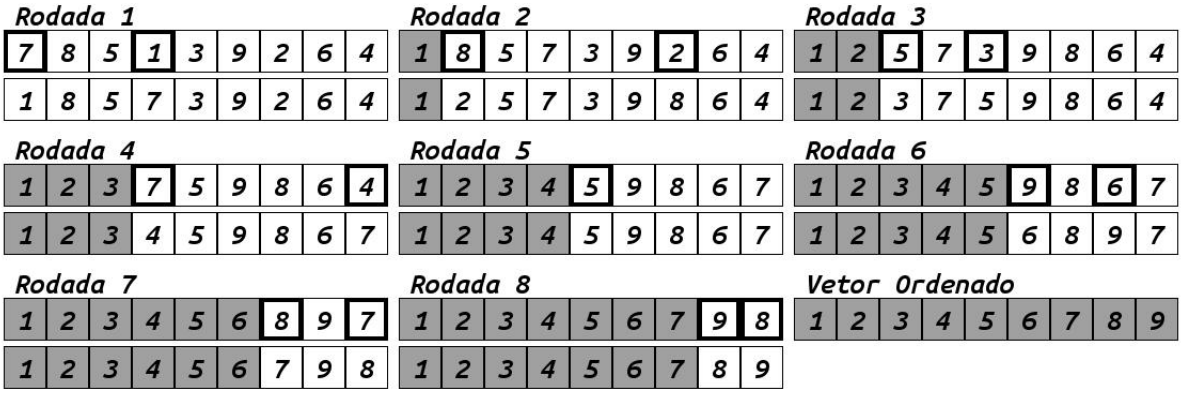

Figura 2: Ordenação por seleção (SelectionSort).

iteração do laço interno então o total de testes é,

$$\begin{aligned} T(n) &= \sum_{i=1}^{n-1}(n-i) \\ &= (n-1)+(n-2)+(n-3)+\cdots+1 \\ &= \frac{n(n-1)}{2} \\ &= O(n^2) \end{aligned}$$

2.3 Ordenação por Inserção

Suponha que Pedro possui uma prateleira cheia de DVDs e deseja ordená-los em ordem de aquisição, ou seja, começando pelos mais antigos até os mais recentes. Para realizar esta tarefa Pedro toca com o indicador da mão direita o segundo DVD da fileira e então gradativamente desloca esta mão para a direita verificando individualmente cada DVD que possui. Para cada uma destas verificações ele toca com o indicador da mão esquerda o DVD imediatamente anterior e a desloca para a esquerda à procura do primeiro DVD de aquisição mais antiga que aquele apontado pela mão direita. Quando isso acontece (e pode não acontecer) Pedro retira o DVD apontado pela mão direita e o coloca na posição imediatamente a frente da mão esquerda (note que é necessário empurrar um pouco os DVDs à direita para fazer cabê-lo). Em seguida Pedro recoloca a mão direita na posição que estava antes da retirada e prossegue efetuando o mesmo procedimento até atingir o final da prateleira.

Este procedimento é conhecido como *ordenação por inserção* (ou *InsertionSort*). O nome se deve ao fato de extrair-se elementos da sequência e em seguida *inseri-los* novamente.

Em termos computacionais a ordenação por inserção ocorre como descrito a seguir. Seja L um vetor de comprimento n que deva ser ordenado, i o contador que simula a mão direita de Pedro e j o contador que simula a mão esquerda. O contador i varia desde 2 até n (verificações de Pedro) ao passo que j, em cada iteração de i, regride desde $i-1$ até 1 ou até encontrar um elemento que seja menor que aquele da posição i (DVD precisa ser movido). Antes da regressão de j o elemento em i deve ser copiado para uma variável auxiliar x (possível DVD a ser movido) e durante a regressão os elementos de posição j devem ser copiados para a posição imediatamente a frente, $j+1$ (equivale a deslocar os DVDs à direita para abrir espaço). Quando a regressão encerra o valor de x é colocado em $j+1$ (O DVD movido é colocado de volta). O Algoritmo-4 implementa este procedimento. Note que de forma análoga aos outros métodos de ordenação elementares, o InsertionSort possui dois laços aninhados, um externo e outro interno.

Algoritmo 4 Ordenação por Inserção, *InsertionSort*

```
1:  Função INSERTIONSORT(ref L, n)
2:      i ← 2
3:      Enquanto i ≤ n faça
4:          x ← L[i]
5:          j ← i − 1
6:          Enquanto j ≥ 1 e L[j] > x faça:
7:              L[j + 1] ← L[j]
8:              j ← j − 1
9:          L[j + 1] ← x
10:         i ← i + 1
```

As rodadas de uma ordenação por inserção sobre o vetor [7, 8, 5, 1, 3, 9, 2, 6, 4] estão ilustradas na Figura-3. Os pares de células em tom escuro em cada rodada representam o retirar (cópia para x) e colocar (inserir) desta rodada. Os pares de células marcadas em bordas representam os gradativos movimentos à direita que os elementos precisam fazer para abrir espaço à inserção. Note que há rodadas onde não ocorrem movimentos e a inserção ocorre no próprio lugar (em i).

E as trocas? Onde ocorrem na ordenação por inserção? Ao contrário dos dois métodos de ordenação elementares descritos anteriormente, no InsertionSort as trocas ocorrem no escopo do laço externo (linha-3, Algoritmo-4) e ainda, o laço interno existe para determinar o índice j cujo elemento será trocado com o elemento da posição i. Em outras pa-

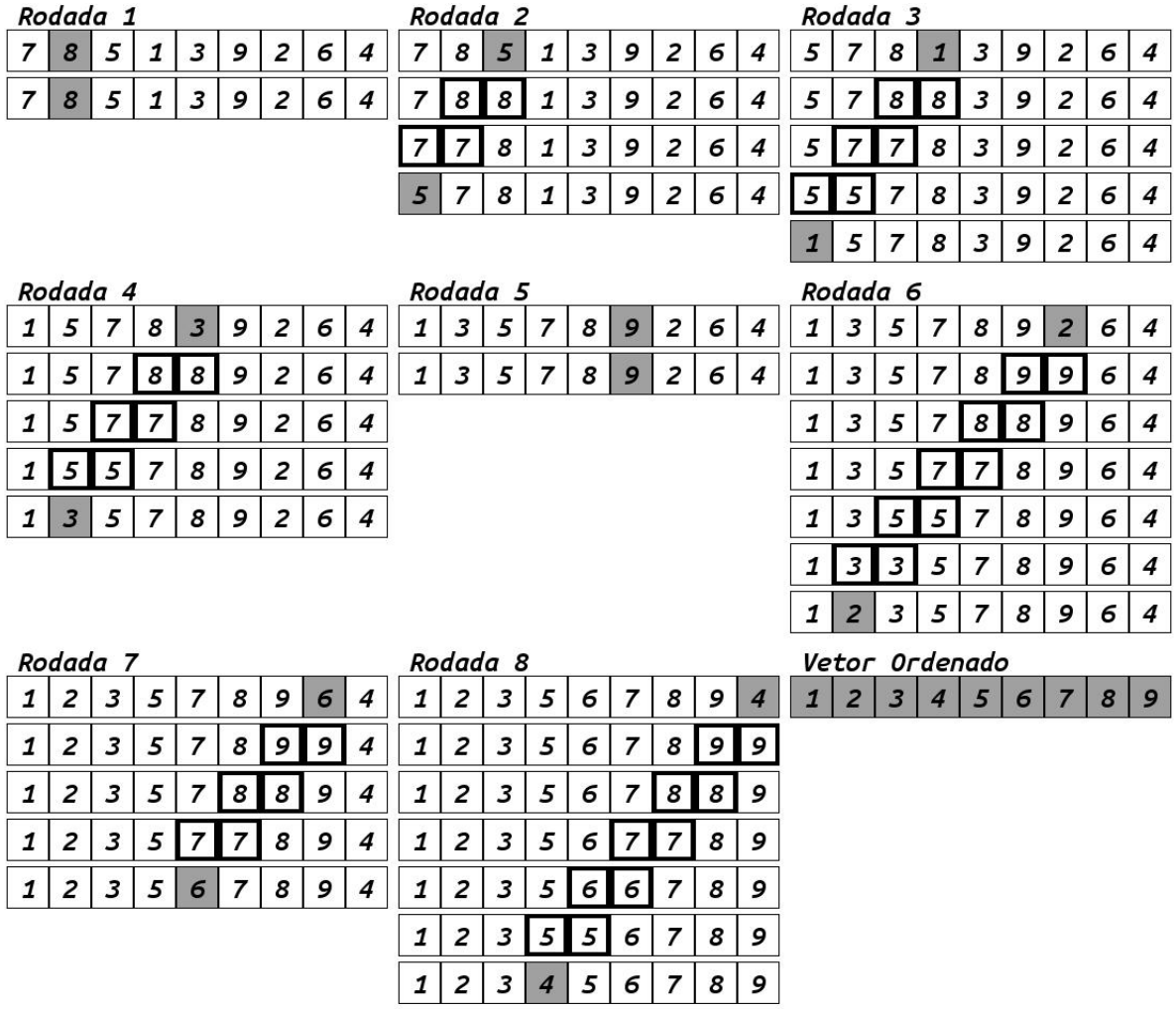

Figura 3: Ordenação por inserção (InsertionSort).

lavras, cada troca começa na linha-4 (x é variável auxiliar), depois ocorre a determinação do segundo elemento da troca entre as linhas 6 e 8 e por fim encerra na linha-9.

Como as trocas são segmentadas entre escopos então elas não indicam a complexidade real da ordenação por inserção (o total de trocas deve ser $O(n)$. Mostre!). A melhor opção neste caso é fazer a contagem do número de deslocamentos para esquerda que ocorrem na linha-7. No pior caso, para cada iteração do laço externo (onde i varia entre 2 e n), *todos* os deslocamentos ocorrem e logo o valor de j decresce de $i-1$ até 1 ($pos_{max} - pos_{min} + 1 = (i-1) - (1) + 1 = i - 1$ deslocamentos). Assim a complexidade da ordenação por inserção, $T(n)$, pode ser determinada por,

$$\begin{aligned} T(n) &= \sum_{i=2}^{n} (i-1) \\ &= 1 + 2 + 3 + 4 + \cdots + n - 1 \\ &= \frac{n(n-1)}{2} \\ &= O(n^2) \end{aligned}$$

3 Ordenação Eficiente

Nesta sessão abordaremos dois algoritmos de ordenação substancialmente mais rápidos que aqueles apresentados na sessão anterior. Implementaremos versões recursivas destes algoritmos (que são as mais usuais) e ainda mostraremos que a ordem de complexidade deles é $O(n\log(n))$.

3.1 Ordenação Rápida - *QuickSort*

Como primeiro passo para a apresentação da *ordenação rápida* (ou *QuickSort*), apresentamos o processo de partição de um vetor. *Particionar* um vetor L de comprimento n em relação a um elemento $k \in L$ (pivô) significa redistribuir em ordem $O(n)$ os elementos de L de forma que os elementos anteriores a k sejam menores que k e os posteriores sejam maiores. Note que isso não significa necessariamente ordenar L. Por exemplo, o particionamento do vetor,

$$\{4, 1, 7, 3, 9, 5\}$$

utilizando 5 como pivô pode produzir o rearranjo,

$$\{4, 1, 3, \mathbf{5}, 9, 7\}$$

Veja que as sequências $\{4,1,3\}$ e $\{9,7\}$ possuem elementos respectivamente menores e maiores que 5 e que L *não* se torna ordenado. Note também que existem outras opções de particionamento.

O Algoritmo-5 implementa o particionamento entre as posições p e q de um vetor L de comprimento n utilizando o elemento na posição q como pivô. Ele executa em $O(n)$ e funciona como descrito a seguir. O índice i é utilizado para visitar cada elemento de L, exceto o pivô (que está na posição q). O índice j controla a posição do maior elemento menor que o pivô e é gradualmente deslocado para a direita (note que ele inicia em $p-1$, linha-3, pois a primeira hipótese é a de que todos os elementos são menores que o pivô). Um deslocamento ocorre sempre que i encontra um elemento menor que o pivô (linha-5) e corresponde ao incremento de j da unidade (linha-6) mais uma troca (linha-7) entre a nova posição de j e a posição i. Quando o laço principal termina todos os elementos entre as posições p e j são menores que o pivô na posição q. Neste ponto uma última troca entre as posições $j+1$ e q () posiciona o pivô de forma que todos os elementos anteriores ficam menores e os posteriores maiores (exatamente como desejado). A função PART ainda retorna a posição final onde ficou o pivô, ou seja, $j+1$. A presença de um laço único denota a ordem $O(n)$ do algoritmo (Mostre!).

Algoritmo 5 Particionamento de um vetor L na faixa $p..q$ com pivô em $L[q]$.

```
1:  Função PART(ref L, p, q)
2:      i ← p
3:      j ← p − 1
4:      Enquanto i < q faça
5:          Se L[i] < L[q] então
6:              j ← j + 1
7:              SWAP(L, i, j)                ▷ Trocas
8:          i ← i + 1                        ▷ Próxima
9:      SWAP(L, j + 1, q)                    ▷ Última troca
10:     Retorne j + 1                        ▷ Destino do pivô
```

A figura-4 mostra os passos de um particionamento aplicado ao vetor,

$$\{7,6,9,3,1,8,4,2,5\}$$

cuja última chave (5) é tomada como pivô e as chaves de valores menores que o pivô são marcadas em tom escuro. O contador i visita o vetor entre as posições das chaves 7 e 2 e inicialmente j aponta para a posição antes da chave 7. Quando i atinge a chave de valor 3 (primeira chave menor que a pivô), então j avança uma posição ($j \leftarrow j+1$) e a chave desta nova posição (7) é trocada com a chave da posição atual de i (3). A próxima menor chave em relação a pivô é 1 que quando encontrada faz novamente avançar j (que passa a apontar 6) e provoca novamente uma troca entre as posições i e j. O mesmo acontece adiante com as chaves 4 e 2 finalizando o deslocamento das chaves de valores inferiores ao da pivô. O passo final troca chave 5 (pivô) com a da posição $j+1$ (6) encerrando o processo.

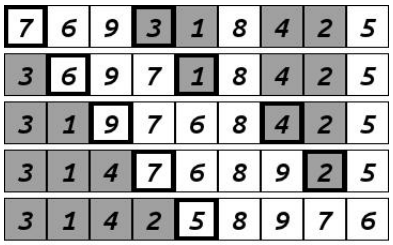

Figura 4: Particionamento de um vetor com pivô na última posição.

A ordenação rápida é um processo de ordenação eficiente que utiliza particionamentos apropriados para promover a ordenação dos elementos de um vetor dado. A forma recursiva é a mais usual e processa-se como descrito a seguir. Dado um vetor de entrada L executa-se um particionamento global (entre a primeira e última posição) utilizando-se como pivô a última chave, que denotaremos por k. Tal particionamento desloca k para sua posição definitiva e cria dois sub-vetores: o de chaves menores que k e o de chaves maiores que k. Então recursivamente aplica-se o particionamento a estes dois sub-vetores que por sua vez causam outras chamadas recursivas e assim sucessivamente. Quando os sub-vetores atingem comprimento menor que 2 as pendências recursivas começam a resolver-se e o resultado final é a ordenação de L. Este procedimento é implementada no Algoritmo-6. Os argumentos p e q em QUICKSORT permitem manipular faixas distintas do mesmo vetor L em iterações recursivas distintas. Os particionamentos ocorrem na linha-3 e as chamadas recursivas aos sub-vetores nas linhas 4 e 5. Atente ao fato de que a posição retorno do particionamento, manipulada pela variável r, não entrar nas chamadas recursivas, ou seja, na chamada da linha-4 o intervalo para particionamento é $\{p, r-1\}$ e da segunda chamada recursiva é $\{r+1, q\}$. Isso naturalmente ocorre porque a chave na posição já está em seu lugar definitivo. De fato cada particionamento põe seu respectivo pivô em sua posição definitiva. A condição da linha-2 restringe a ordenação a sub-vetores cujo comprimento é maior que a unidade. A figura-5 ilustra as etapas de uma ordenação rápida. Em cada etapa o sub-vetor de particionado é marcado em tom escuro.

Algoritmo 6 Ordenação rápida, *QuickSort*, entre as posições p e q de um vetor L.

```
1:  Função QUICKSORT(ref L, p, q)
2:      Se p < q então
3:          r ← PART(L, p, q)
4:          QUICKSORT(L, p, r − 1)
5:          QUICKSORT(L, r + 1, q)
```

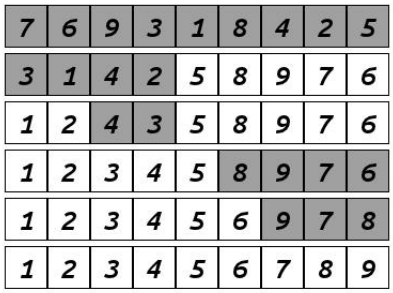

Figura 5: Ordenação Rápida (QuickSort).

O trecho de pseudocódigo a seguir ilustra o uso da função QUICKSORT,

```
1: L ← {7, 6, 9, 3, 1, 8, 4, 2, 5}          ▷ Vetor de 9 chaves
2: QUICKSORT(L, 1, 9)                        ▷ Classifica vetor inteiro
3: Para k ← 1 até 9 faça
4:     Escreva L[k]                          ▷ Impressão de chaves ordenadas
```

No melhor caso da ordenação QuickSort *todos* os particionamentos contam com pivôs que são *medianas* do sub-vetor avaliado, ou seja, quando deslocados ocupam a posição central do arranjo. Essa situação representa o melhor caso porque equilibra o número de chamadas recursivas que os dois sub-vetores provenientes do particionamento sofrem. Para medir a complexidade neste caso meçamos inicialmente o total $T(n)$ (onde n denota o comprimento do vetor por ordenar) de particionamentos que se procedem por ação da linha-3 na função QUICKSORT, Algoritmo-6. A lei de recorrência neste caso é,

$$T(n) = \begin{cases} 2T\left(\frac{n-1}{2}\right) + 1 & n > 1 \\ 0 & n = 1 \end{cases} \qquad (1)$$

onde o 2 que multiplica a chamada recorrente denota as duas chamadas recursivas a QUICKSORT, $\frac{n-1}{2}$ denota o tamanho dos sub-vetores oriundos do particionamento no melhor caso descrito anteriormente e o, $+1$, denota a contagem de chamadas a PART. A solução desta lei de recorrência conduz a $O(\log n)$ (Veja questão-9). Como cada particionamento ocorre em $O(n)$ (Veja questão-8) então, no melhor caso a complexidade do quicksort é de $O(n \log n)$.

3.2 Ordenação por Fusão - *MergeSort*

A *ordenação por fusão*, ou *MergeSort*, consiste em um método eficiente de ordenação em vetores que efetua um conjunto de *fusões* internas apropriadas no vetor de entrada provocando ordenação de chaves. Uma fusão (*merge*) é um procedimento que toma duas faixas pré-ordenadas adjacentes de um mesmo vetor de entrada e as rearranja de forma a gerar um arranjo único ordenado. Por exemplo, no vetor,

$$\{2, 4, 7, 1, 5, 9\}$$

podemos *fundir* as faixas $\{2, 4, 7\}$ e $\{1, 5, 9\}$, que já são ordenadas, e gerar o novo arranjo,

$$\{1, 2, 4, 5, 7, 9\}$$

Um algoritmo de complexidade $O(n)$ para fusão em um vetor L das duas faixas adjacentes de índices respectivos entre p e r e entre $r+1$ e q é apresentado no Algoritmo-7. Seu funcionamento é descrito a seguir. Inicialmente as duas faixas são copiadas em vetores intermediários locais A (linha-4) e B (linha-6) cujos comprimentos são mantidos pelas variáveis m e n respectivamente. Uma vez copiadas todas as chaves para A e B o passo seguinte consiste em trazê-las de volta para L sobrescrevendo as originais pelas cópias ordenadas. Para reinserir as chaves já ordenadas utilizam-se três contadores, i, j e k. O contador k aponta para a posição de L onde ocorrerá a próxima inserção (note que ele inicia em p, linha-8) ao passo que i e j apontam para as últimas chaves não transferidas de A e B respectivamente. Em cada etapa deste processo de transferência (linha-9) é comparada a chave apontada por i em A com aquela apontada por j em B (linha-10) e a menor é escolhida para ser transferida para L (linha-11 ou linha-14) e seu respectivo contador (i ou j) incrementado em 1 (próxima chave não transferida). Ocasionalmente um dos vetores intermediários poderá ter todas as suas chaves transferidas ao passo que o outro não. Neste caso as chaves excedentes, sejam de A ou B, deverão ser transferidas diretamente para L concluindo o processo. O laço na linha-17 transfere chaves excedentes de A para L ao passo que o laço da linha-21 transfere chaves excedentes de B. Note que apenas uma deles ou nenhum ocorrerá.

De forma similar ao que ocorre na ordenação rápida, a ordenação por fusão efetua fusões apropriadas que ocasionam a ordenação global do vetor. O Algoritmo-8 implementa a versão recursiva do MergeSort mais usual a qual recebe um vetor L e a faixa de índices $p..q$ de L onde se dará a ordenação. Neste algoritmo o vetor é recursivamente dividido em duas faixas de mesmo tamanho aproximadamente, ambas as faixas são ordenadas e por fim fundidas. Muitas divisões e fusões ocorrem durante o processo que aplica como critério de continuidade de recursão o comprimento da faixa ser maior que a unidade (ou seja, $p < q$, linha-2). Note que a divisão em faixas de tamanho aproximadamente iguais é construída com o índice r que é média entre p e q (linha-3) sendo logo tais faixas $p..r$ e $r+1..q$ (distintas). Note também que as chamadas recursivas a MERGESORT sobre as faixas descritas, e que são responsáveis pela ordenação de cada uma delas, ocorrem antes da fusão (de fato só deve- se fundir faixas já ordenadas). Esta característica em

particular distingue o MergeSort do QuickSort no que diz respeito a ordem de colocação definitiva de chaves. No QuickSort a colocação definitiva de chaves se faz na primeira metade do processo sendo a segunda apenas o desembaraço das recursões. No MergeSort a primeira metade gera um embarace recursivo de faixas enquanto a segunda metade desfaz este embarace fundindo faixas e naturalmente realizando as colocações definitivas das chaves.

Algoritmo 8 Ordenação por Fusão.

```
1: Função MERGESORT(ref L, p, q)
2:     Se p < q então
3:         r ← ⌊(p + q)/2⌋                          ▷ índice central
4:         MERGESORT(L, p, r)                       ▷ Ordena faixa p, r
5:         MERGESORT(L, r + 1, q)                   ▷ Ordena faixa r + 1, q
6:         MERGE(L, p, r, q)                        ▷ Fusão de faixas
```

A figura-6 ilustra as etapas de um processo de ordenação por fusão. Note que inicialmente o vetor se fragmenta em faixas continuamente até atingirem tamanho um e depois se *fundem* gradativamente até recomporem o vetor ordenado.

Em MERGE a faixa é varrida uma vez para cópia e outra para a devolução de chaves ordenadas o que demonstra uma ordem de complexidade $O(n)$. Já o processo recursivo de MERGESORT é análogo ao de QUICKSORT e logo tem ordem de $O(\log n)$ iterações recursivas. Associando estes resultados presumimos que a complexidade global seja $O(n \log n)$.

Algoritmo 7 Fusão num vetor de faixas adjacentes pré-ordenadas.

```
1: Função MERGE(ref L, p, r, q)
2:     m ← r − p + 1
3:     n ← q − r
4:     Para k ← 1 até m faça                        ▷ A copia faixa (p, r)
5:         A[k] ← L[k + p − 1]
6:     Para k ← 1 até n faça                        ▷ B copia faixa (r + 1, q)
7:         B[k] ← L[k + r]
8:     i, j, k ← 1, 1, p
9:     Enquanto i ≤ m e j ≤ n faça
10:        Se A[i] < B[j] então                     ▷ Quem é menor?
11:            L[k] ← A[i]                          ▷ Cópia de A para L
12:            i ← i + 1
13:        senão
14:            L[k] ← B[j]                          ▷ Cópia de B para L
15:            j ← j + 1
16:        k ← k + 1
17:    Enquanto i ≤ m faça                          ▷ Excedente de A pra L
18:        L[k] ← A[i]
19:        i ← i + 1
20:        k ← k + 1
21:    Enquanto j ≤ n faça                          ▷ Excedente de B pra L
22:        L[k] ← B[j]
23:        i ← i + 1
24:        k ← k + 1
```

4 Estabilidade

Um algoritmo de ordenação é considerado *estável* quando mantém inalteradas, no vetor de entrada, as posições relativas das chaves que se repetem. Por exemplo, no vetor de entrada,

$$\{2, 1, 3_1, 7, 4, 3_2\}$$

a chave 3 se repete duas vezes e estão marcados com índices. Um algoritmo de ordenação estável mantém a posição relativa das chaves 3, ou seja, 3_1 deve vir antes de 3_2. Esquematicamente a ordenação tem aspecto,

$$\{1, 2, 3_1, 3_2, 4, 7\}$$

Na ordenação instável 3_2 *pode* vir antes de 3_1. Note que isso não compromete de fato a ordenação. Então qual o diferencial de um algoritmo de ordenação estável? O diferencial está nas ordenações que aplicam mais de um critério à mesma base de dados, por exemplo, numa tabela de registros contendo nome (string) e setor (inteiro), de empregados de uma empresa, se os dados forem previamente ordenados por nome e em seguida por setor é necessário que os nomes mantenham, ao final do segundo processo, em seus respectivos setores, a ordem obtida na primeira ordenação. Para ilustrar o exemplo considere os seguintes registros hipotéticos de empresa já classificados por nome,

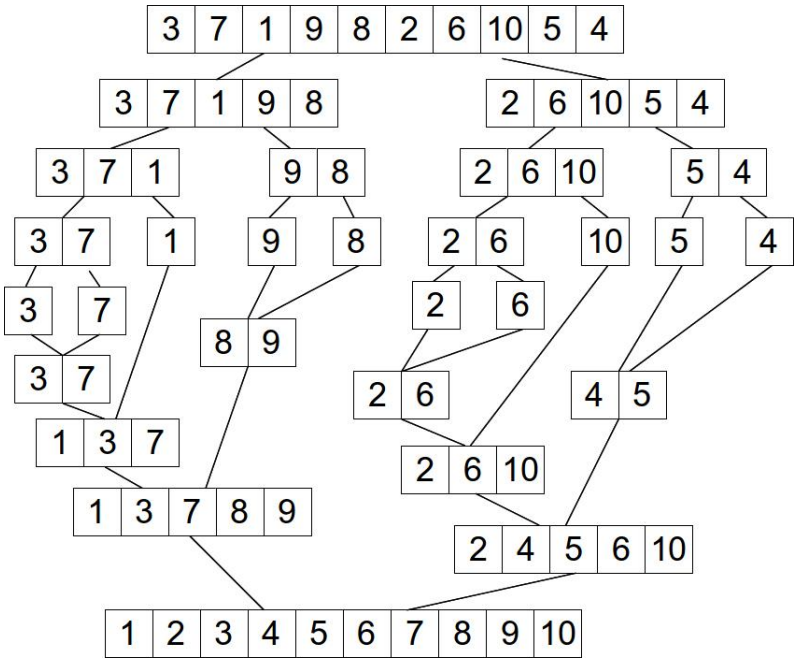

Figura 6: Etapas de uma ordenação por fusão (MergeSort)

Nome	**Setor**
Beatriz Lavor	2
Fabiano Augusto	3
Irã Marques	1
Karine Bráz	3
Pedro Marques	2

Quando reclassificados por setor, utilizando um algoritmo estável, é obtido,

Nome	**Setor**
Irã Marques	1
Beatriz Lavor	2
Pedro Marques	2
Fabiano Augusto	3
Karine Bráz	3

Note que a ordem por setor dos nomes se mantém a mesma da primeira classificação.

Os algoritmos de ordenação por borbulhamento (*BubbleSort*), por inserção (*InsertionSort*) e por fusão (*MergeSort*) são estáveis. As ordenações por seleção (*SelectionSort*) e rápida (*QuickSort*) são instáveis.

5 Problemas

1. Reimplemente a ordenação BubbleSort de forma que ao invés de *subir*, as bolhas *desçam* pela coluna. Note que a ordenação ainda deve ser ascendente.
2. Modifique a implementação da ordenação por borbulhamento de forma que, ao invés de duas células as bolhas abarquem três células do vetor de entrada por iteração. O que muda substancialmente?
3. O que muda nas ordenações por seleção e inserção quando devem rearranjar seus elementos em ordem descendente? Reconstrua os algoritmos Algoritmo-3 e Algoritmo-4 com as modificações apropriadas.
4. Implemente uma versão recursiva para a ordenação por seleção.
5. Na ordenação por seleção de um vetor L o subvetor das chaves selecionadas cresce a partir da extremidade esquerda de L até cobrir todo o vetor. Uma variação deste procedimento consiste em tratar simultaneamente dois sub-vetores: o das chaves mínimas cuja primeira posição está enraizada sobre a posição inicial de L (exatamente como ocorre na ordenação por seleção clássica) e o das chaves máximas cuja última posição está enraizada sobre a última posição de L. Proponha um algoritmo para esta modificação da ordenação por seleção e calcule sua nova complexidade.
6. Insira contadores de trocas (ou testes) nos métodos de ordenação elementares (implementados em C) e realize medições de seus valores em situações variadas (isso significa testar vetores de comprimentos diversos). Em seguida construa gráficos para cada método (total de iterações vs. comprimento do vetor) e faça uma análise comparativa entre os métodos a partir dos gráficos obtidos.
7. Um método de ordenação é dito estável se durante o processamento os elementos repetidos não sofrem mudança de suas posições relativas. (a) Determinar que algoritmos elementares de ordenação (inserção, seleção e borbulhamento) são estáveis. (b) Por que a estabilidade é importante?
8. Mostre que a complexidade do particionamento de um vetor L de comprimento n, Algoritmo-5, vale $O(n)$.
9. Mostre que a complexidade de melhor caso da ordenação rápida de um vetor L de comprimento n, Algoritmo-6, vale $O(n \log n)$.
10. Reescrever o procedimento de partição do QuickSort tomando como referência o primeiro elemento.
11. Determine a complexidade de pior caso da ordenação rápida.

12. O paradigma da ordenação rápida (QuickSort) é conhecido como *divisão e conquista* porque divide progressivamente o vetor em sub-faixas (*divisão*) e particiona cada uma delas (*conquista*) para, por fim, obter uma ordenação global. Considere um algoritmo de ordenação de divisão e conquista que divida o vetor em faixas sucessivamente menores (recursivamente) e aplique nelas a ordenação por inserção. De que forma pode-se fazer isso de forma a gerar um algoritmo mais eficiente que a ordenação por inserção? Construa o algoritmo e calcule sua complexidade.
13. Reconstrua MERGE e MERGESORT de forma que em vez de duas sejam três o número de subdivisões em cada etapa recursiva.
14. Escreva a lei de recorrência do algoritmo MergeSort e utilize-a para demonstrar que o algoritmo é de fato $O(n \log n)$.
15. Uma versão iterativa do MergeSort é descrita a seguir. O vetor L é dividido em blocos de m células que são fundidos aos pares (adjacentes). Inicialmente m vale 1 e em cada etapa é dobrado de forma que os blocos adjacentes a serem fundidos estejam sempre ordenados. Note que o valor de m não pode extrapolar n (comprimento de L). Construa algoritmo da versão iterativa do MergeSort descrita.
16. Seja L um vetor de comprimento n ordenado por fusão. Determine no melhor e pior caso quanta memória auxiliar (vetores A e B, Algoritmo-7), medida em número de células, é utilizada no pior e no melhor caso.