Universidade Federal do Espírito Santo
Departamento de Informática

André Barreto Silveira

Algoritmos de Ordenação

Trabalho 4 de Estrutura de Dados II

Vitória
2015

1 Introdução

Ordenação de dados é um processo sistemático de separação de informações em uma determinada sequência. Este é um processo comum à muitas aplicações em diversas áreas, e muito relevante na Computação. Por isto, foram desenvolvidos diversos algoritmos para ordenar elementos de forma eficiente [1].

Dentre outros, sequências ordenadas possibilitam:

- Buscar por informações específicas de forma eficiente;
- Unir dados de forma eficiente.

Devido à importância da ordenação, este trabalho tem como objetivo implementar e analisar algoritmos de ordenação, realizando comparações de eficiência entre os métodos avaliados.

2 Algoritmos Considerados

Os algoritmos de ordenação selecionados para a comparação são os seguintes: *bubblesort, shakesort, insertionsort, shellsort, selectionsort, ranksort, quicksort, mergesort, heapsort, radixsort* e *radixsort binário*, implementados em linguagem C.

Estes foram implementados com referência, principalmente, o *site Rosseta Code* [2]. A seguir seguem considerações sobre cada um dos algoritmos mencionados.

2.1 Bubblesort

O *bubble sort*, ou ordenação por flutuação, é um algoritmo de ordenação dos mais simples. A ideia é percorrer o vetor diversas vezes, a cada passagem fazendo flutuar para o topo o maior elemento da sequência [3].

Considerações sobre o algoritmo:

- Complexidade: $O(n^2)$
- O melhor caso ocorre caso a sequência já esteja ordenada, neste caso, realizado em complexidade linear.
- Algoritmo fácil de implementar porém não é recomendado devido ao grande número de movimentações se o conjunto de entrada é grande.
- O algoritmo é estável.

2.2 Shakesort

Shake sort ou *Cocktail sort*, é uma variação do *bubble sort* que é tanto um algoritmo de ordenação estável quanto uma ordenação por comparação. O algoritmo difere do *bubble sort* pelo fato de ordenar em ambas as direções em cada passagem através da lista [4].

Considerações sobre o algoritmo:

- Complexidade: $O(n^2)$
- Não melhora significantemente o algoritmo da bolha. A complexidade assintótica permanece quadrática.
- Algoritmo mais complicado se comparado com *bubble*.
- O algoritmo é estável.

2.3 Selectionsort

A ordenação por seleção é um algoritmo de ordenação baseado em se passar sempre o menor valor do vetor para a primeira posição (ou o maior dependendo da ordem requerida), depois o de segundo menor valor para a segunda posição, e assim é feito sucessivamente com os (n-1) elementos restantes, até os últimos dois elementos. [5]

Considerações sobre o algoritmo:

- Complexidade: $O(n^2)$
- Custo linear para o número de movimentos de registros.
- É um algoritmo interessante a ser utilizado para arquivos com registros muito grandes devido à propriedade anterior.
- O fato de o arquivo já estar ordenado não ajuda em nada, pois o custo continua quadrático.
- O algoritmo não é estável.

2.4 Insertionsort

Insertion sort, ou ordenação por inserção, é um simples algoritmo de ordenação, eficiente quando aplicado a um pequeno número de elementos. Em termos gerais, ele percorre um vetor de elementos da esquerda para a direita e à medida que avança vai deixando os elementos mais à esquerda ordenados. [6]

Considerações sobre o algoritmo:

- Complexidade: $O(n^2)$
- O número mínimo de comparações e movimentos ocorre quando os itens estão originalmente em ordem.
- O número máximo ocorre quando os itens estão originalmente na ordem reversa.
- É o método a ser utilizado quando o arquivo está "quase" ordenado.
- É um bom método quando se deseja adicionar uns poucos itens a um arquivo ordenado, pois o custo é linear.
- O algoritmo de ordenação por inserção é estável.

2.5 Shellsort

Criado por Donald Shell em 1959, publicado pela Universidade de Cincinnati, *Shell sort* é o mais eficiente algoritmo de classificação dentre os de complexidade quadrática. É um refinamento do método de inserção direta. O algoritmo difere do método de inserção direta pelo fato de no lugar de considerar o *array* a ser ordenado como um único segmento, ele considera vários segmentos sendo aplicado o método de inserção direta em cada um deles. [7]

- A razão da eficiência do algoritmo ainda não é conhecida.
- Shellsort é uma ótima opção para arquivos de tamanho moderado.
- Sua implementação é simples e requer uma quantidade de código pequena.
- O tempo de execução do algoritmo é sensível à ordem inicial do arquivo.
- O método não é estável.

2.6 Quicksort

O Quicksort adota a estratégia de divisão e conquista. A estratégia consiste em rearranjar as chaves de modo que as chaves "menores" precedam as chaves "maiores". Em seguida é ordenado as duas sublistas de chaves menores e maiores recursivamente até que a lista completa se encontre ordenada [8].

Considerações sobre o algoritmo:

- Complexidade caso médio e melhor caso: $\theta(nlogn)$

- Complexidade no pior caso: $O(n^2)$
- O pior caso ocorre quando, sistematicamente, o pivô é escolhido como sendo um dos extremos de um vetor já ordenado.
- É extremamente eficiente para ordenar arquivos de dados.
- Necessita de apenas uma pequena pilha como memória auxiliar.
- O algoritmo não é estável.

2.7 Mergesort

O *merge sort*, ou ordenação por mistura, é um exemplo de algoritmo de ordenação do tipo dividir-para-conquistar. Sua ideia básica consiste em Dividir(o problema em vários sub-problemas e resolver esses sub-problemas através da recursividade) e Conquistar(após todos os sub-problemas terem sido resolvidos ocorre a conquista que é a união das resoluções dos sub-problemas). Como o algoritmo do *Merge Sort* usa a recursividade em alguns problemas esta técnica não é muito eficiente devido ao alto consumo de memória e tempo de execução [9].

Considerações sobre o algoritmo:

- Complexidade caso médio: $\theta(nlogn)$
- Não possui um pior caso como o Quicksort.
- É um algoritmo muito eficiente para ordenação.
- Necessita o dobro de memória para realizar a ordenação.
- O algoritmo é estável.

2.8 Heapsort

É um algoritmo de ordenação que pode ser considerado como um refinamento do *selection sort*. Utiliza da estrutura de uma *Heap* para realizar a ordenação, reduzindo significantemente a quantidade de repetições do algoritmo [10].

Considerações sobre o algoritmo:

- Complexidade: $O(nlogn)$ para todos os casos
- O anel interno do algoritmo é bastante complexo se comparado com o do *Quicksort*.
- O algoritmo não é estável.

- Não é recomendado para arquivos com poucos registros, por causa do tempo necessário para construir o *heap*.

2.9 Ranksort

O *Rank sort* ou *Counting sort* é um algoritmo de ordenação de acordo com chaves representadas por números inteiros pequenos. Ou seja, é um algoritmo de ordenação de inteiros. Ele opera contando o número de objetos que possuem chaves diferentes, e usa aritmética para determinar as posições de cada valor na saída. É frequentemente usado como subrotina para algoritmos como o *radix sort* [11].

Considerações sobre o algoritmo:

- Complexidade: $O(n + k)$
- O algoritmo não é estável.

2.10 Radixsort

Radix sort é um algoritmo de ordenação que ordena inteiros processando dígitos individuais. Como os inteiros podem representar strings compostas de caracteres (como nomes ou datas) e pontos flutuantes especialmente formatados, *radix sort* não é limitado somente a inteiros [12].

Considerações sobre o algoritmo:

- Complexidade: $O(nw)$
- Utiliza o *bucket sort* no algoritmo.
- Possui 10 filas (0-9) para armazenar os números no processo.
- O algoritmo é estável.

2.11 Radixsort binário

Possui a mesma ideia do *Radix sort*, porém adaptado especialmente para número binários. Isto é, não serão necessárias 10 filas e alguns cálculos são feitos de forma mais prática para o tipo de dado alvo.

Considerações sobre o algoritmo:

- Complexidade: $O(nw)$
- Utiliza o *bucket sort* no algoritmo.
- Possui 2 filas (0 e 1) para armazenar os números no processo.
- O algoritmo é estável.

3 Comparação entre algoritmos

Nesta seção serão feitas comparações entre os algoritmos e uma análise dos mesmos a fim de estabelecer parâmetros de eficiência e usabilidade dos métodos considerados.

Para isto, os algoritmos serão testados com registros em três ordens diferentes (10^3, 10^5 e 10^6), cada qual com três características desiguais: registros gerados *aleatoriamente*, em ordem *crescente* e em ordem *decrescente*.

Nas comparações de velocidade de processamento, foi estabelecido um tempo limite para os algoritmos. Caso este exceda *1 minuto* para completar a ordenação, o processo é interrompido e será dito que ele levou tempo ∞ para ser finalizado.

Obs.: os tempos foram registrados desconsiderando o tempo de impressão da resposta. Ou seja, apenas o tempo utilizado pelo algoritmo de ordenação foi contado.

3.1 Comparação de complexidade

Veja na tabela 1 a comparação de complexidade entre os algoritmos.

Algoritmo	**Complexidade**
Bubblesort	$O(n^2)$
Shakesort	$O(n^2)$
Selectionsort	$O(n^2)$
Insertionsort	$O(n^2)$
Shellsort	-
Quicksort	$O(nlogn)$
Mergesort	$O(nlogn)$
Heapsort	$O(nlogn)$
Ranksort	$O(n+k)$
Radixsort	$O(nw)$
Radixsort Bin	$O(nw)$

Tabela 1: Complexidade dos algoritmos

3.2 Registros gerados aleatoriamente

Veja na tabela 2 os tempos de processamento dos algoritmos submetidos à diferentes ordens de registros gerados aleatoriamente.

Vemos que para registros aleatórios, os algoritmos mantém um padrão esperado. Os quadráticos são significantemente mais lentos que os de complexidade menor.

Algoritmo	**1 000**	**100 000**	**1 000 000**
Bubblesort	0.007s	36.595s	∞
Shakesort	0.005s	21.579s	∞
Selectionsort	0.004s	11.088s	∞
Insertionsort	0.002s	5.861s	∞
Shellsort	0.001s	0.031s	0.475s
QuickPrim	0.001s	0.045s	0.302s
QuickMedia3	0.001s	0.044s	0.290s
QuickCentral	0.001s	0.046s	0.291s
QuickRandom	0.001s	0.047s	0.310s
Mergesort	0.002s	0.027s	0.358s
Heapsort	0.001s	0.026s	0.370s
Ranksort	0.006s	0.012s	0.151s
Radixsort	0.002s	0.079s	1.097s
RadixsortBin	0.003s	0.111s	3.288s

Tabela 2: Testes em ordem aleatória

Os primeiros quatro algoritmos, os de complexidade $O(n^2)$ não são recomendados para estes casos de ordenação, enquanto todos os outros são aplicáveis.

3.3 Registros gerados em ordem crescente

Veja na tabela 3 os tempos de processamento dos algoritmos submetidos à diferentes ordens de registros gerados crescentemente.

Nesta leva de testes podemos ver casos especiais dos algoritmos. O *Bubblesort* e o *shakesort* apresentam um desempenho inicialmente e talvez naturalmente não esperado, enquanto o *quicksort* com pivô primeiro demonstra um processamento bastante lento. O *insertionsort* também possui este caso como seu melhor, similar ao *bubble*.

Isto ocorre devido às características dos algoritmos. A ordem de dados crescente, ou já ordenada, é o melhor caso do algoritmo *bubblesort* e, consequentemente, o *shakesort*, resultando em uma complexidade resultante $\theta(n)$. Já o *quicksort* com o pivô selecionando o primeiro elemento é seu pior caso se os elementos estão em ordem crescente ou decrescente. O que resulta em uma complexidade $\theta(n^2)$.

Os outros algoritmos de *quicksort* contornam este pior caso com uma seleção mais inteligente do pivô.

Algoritmo	**1 000**	**100 000**	**1 000 000**
Bubblesort	0.001s	0.020s	0.156s
Shakesort	0.001s	0.009s	0.087s
Selectionsort	0.002s	11.146s	∞
Insertionsort	0.001s	0.019s	0.154s
Shellsort	0.001s	0.013s	0.143s
QuickPrim	0.001s	8.898s	∞
QuickMedia3	0.001s	0.030s	0.212s
QuickCentral	0.001s	0.030s	0.214s
QuickRandom	0.001s	0.047s	0.312s
Mergesort	0.001s	0.019s	0.207s
Heapsort	0.001s	0.022s	0.239s
Ranksort	0.003s	0.012s	0.144s
Radixsort	0.002s	0.083s	1.104s
RadixsortBin	0.001s	0.111s	3.277s

Tabela 3: Testes em ordem crescente

3.4 Registros gerados em ordem decrescente

Veja na tabela 4 os tempos de processamento dos algoritmos submetidos à diferentes ordens de registros gerados de forma decrescente.

Como nos casos de teste em ordem crescente, neste é notável que o *quicksortprimeiro* também tem seu pior caso. Ou seja, este algoritmo é recomendável apenas quando sabe-se que a entrada de dados difere de uma crescente ou decrescente.

Porém, distinto do caso crescente, registros gerados em ordem decrescente acarretam nos casos esperados dos algoritmos *bubblesort/shakesort* e *insertionsort*. Ou seja, estes são ineficientes para casos de ordem elevadas.

3.5 Considerações gerais

A respeito dos testes realizados, podemos chegar à algumas afirmativas. Dentre elas, vale ressaltar que:

- Para casos pequenos, todos os algoritmos considerados possuem resultados em tempos muito próximos. Implicando que não é necessário desenvolver um procedimento complexo de ordenação caso sua entrada de dados for pequena.
- Os algoritmos quadráticos (*bubble, shake, selection e insertion*) possuem características diferentes que permitem aplicações desiguais mesmo sendo eles todos da mesma complexidade.

Algoritmo	**1 000**	**100 000**	**1 000 000**
Bubblesort	0.004s	30.458s	∞
Shakesort	0.004s	30.562s	∞
Selectionsort	0.002s	17.144s	∞
Insertionsort	0.002s	11.740s	∞
Shellsort	0.001s	0.016s	0.168s
QuickPrim	0.001s	7.413s	∞
QuickMedia3	0.001s	0.033s	0.219s
QuickCentral	0.001s	0.033s	0.214s
QuickRandom	0.001s	0.036s	0.245s
Mergesort	0.001s	0.020s	0.208s
Heapsort	0.001s	0.022s	0.237s
Ranksort	0.003s	0.012s	0.096s
Radixsort	0.006s	0.087s	1.090s
RadixsortBin	0.001s	0.112s	3.200s

Tabela 4: Testes em ordem decrescente

- Os *quicksorts* podem ser considerados melhores que os anteriores, porém deve-se analisar com atenção a forma de se escolher um pivô, pois isto pode levar a resultados ruins, tanto quanto o pior caso do algoritmo *bubble*. Isto ocorreu claramente com o *quicksort* que seleciona o primeiro elemento como pivô. Por isto, é recomendável sempre utilizar outra técnica de seleção do pivô.

- Dentre os algoritmos de *quicksort* avaliados, podemos dizer que o método de seleção do pivô como primeiro elemento é ruim. Enquanto os outros possuem resultados melhores. Dentre estes, vemos que a seleção aleatória não é tão boa quanto o central e a mediana de três. E entre estes dois últimos, pode-se dizer que a mediana de três é mais interessante, pois sempre apresenta um resultado bom e, na maioria dos testes realizados, mais rápido.

- O *mergesort*, diferente do *quick*, não possui um pior caso. Porém requisita o dobro de armazenamento de memória, o que limita sua aplicação em casos de ordens muito elevadas de registros.

- *Heapsort, shellsort, ranksort, radixsort* são algoritmos muito bons de ordenação, apresentando resultados similares ao *quicksort* e o *mergesort*.

4 Conclusão

Existem diversos algoritmos de ordenação já conhecidos com variações significativas de complexidade. Além da análise de complexidade, deve-se

conhecer a entrada a qual deve ser ordenada, pois isto influencia diretamente na escolha do algoritmo.

Como visto nos testes apresentados, alguns algoritmos que são considerados "ruins" podem ser tão bons quanto os chamados "ótimos", dependendo da entrada de dados que se trabalha. Neste casos, é mais vantajoso utilizar estes mais simples, justamente por sua simplicidade de implementação e manutenção.

Em casos mais gerais, porém, é seguro escolher algoritmos como o *quicksort*, *heapsort* e *radixsort*, que são muito eficientes na maioria dos casos.

Referências

[1] Wikipedia, *Sorting*, Oct. 2015. 2

[2] M. Mol, *Rosetta Code*, Jan. 2007. 2

[3] Wikipedia, *Bubble sort*, Nov. 2015. 2

[4] Wikipedia, *Shake sort*, Nov. 2015. 3

[5] Wikipedia, *Selection sort*, Nov. 2015. 3

[6] Wikipedia, *Insertion sort*, Nov. 2015. 3

[7] Wikipedia, *Shell sort*, Nov. 2015. 4

[8] Wikipedia, *Quick sort*, Nov. 2015. 4

[9] Wikipedia, *Merge sort*, Nov. 2015. 5

[10] Wikipedia, *Heap sort*, Nov. 2015. 5

[11] Wikipedia, *Rank sort*, Nov. 2015. 6

[12] Wikipedia, *Radix sort*, Nov. 2015. 6