Análise e complexidade de algoritmos de ordenação

Igor Lúcio Rocha Alves - Matrícula: 3902

8 de Outubro de 2018

1 Introdução e Objetivos

A ordenação ou classificação de dados consiste em receber um montante de dados e organizá-los em ordem crescente e/ou decrescente, a fim de utilizá-los posteriormente. Essa ordenação tem o objetivo de facilitar as buscas e pesquisas por um elemento em um conjunto ordenado. Na computação existe uma série de algoritmos de ordenação, sendo eles de diferentes comportamentos e complexidades, sendo assim, devemos nos atentar ao problema e só depois escolher o método mais adequado para usar e, consequentemente, solucioná-lo.

Neste relatório são reportadas análises de tempo de execução dos algoritmos: *Selection Sort*, *Insertion Sort*, *Shell Sort*, *Merge Sort*, *Quick Sort*, *Heap Sort*, *Bubble Sort* a fim de, compará-los computacionalmente, e através do tempo de execução determinar sua eficiência. Todos tem como finalidade a tarefa de ordenar o vetor de entrada N de diferentes ordens.

2 Materiais e Métodos

Os algoritmos estudados neste trabalho foram implementados sobre a linguagem de programação C e avaliados em um *hardware* composto com um processador Intel Core^{TM} i7-8550U, 8GB de RAM e 240GB de SSD.

Para avaliar os algoritmos, foram utilizados 5 (cinco) casos de teste de números aleatórios, crescente e decrescente com entradas dos tamanhos: 10, 1000, 10.000, 100.000 e 1.000.000.

3 Resultados

Com base no tempo de execução e na amostragem dos dados através de gráficos e tabelas, podemos inferir e analisar, de forma distinta, cada algoritmo. Os dados nas tabelas são medidos na escala de segundos (s). É mostrado, também, o cálculo da complexidade (se disponível).

3.1 *Selection Sort*

É um dos algoritmos mais simples de ordenação. Seu funcionamento é o seguinte: seleciona o menor item do vetor e a seguir troca com o item que está na primeira posição do vetor. Repita estas duas operações com os n-1 itens restantes, depois com os n-2 itens, até que reste apenas um elemento. Trabalha muito bem com entradas pequenas. Não é estável, pois suas posições podem variar muito no tempo de processamento.

Pior caso

$$t(n)=c1*n+c2*(n-1)+c3*\sum t(n-j)+c4*\sum t(n-j-1)+c5*\sum t(j)+c6*(n-1)+c7*(n-1)$$

$$t(n)= a*n^2+b*n+c$$

$$t(n) = O(n^2)$$

Função Quadrática

Melhor caso

$$t(n)=c1*n+c2*(n-1)+c3*\sum t(n-j)+c4*\sum t(n-j-1)+c5*\sum t(j)+c6*(n-1)$$

$$t(n)= a*n^2+b*n+c$$

$$t(n)= O(n^2)$$

Função Quadrática

Figura 1: Complexidade *Selection Sort*

Tamanho da entrada (n)	Aleatório	Crescente	Decrescente
10	0,000002	0,000002	0,000001
1000	0,001294	0,001125	0,001176
10000	0,108541	0,111414	0,108682
100000	8,82	8,76	9
1000000	1096	1094	1118

Tabela 1: Tabela de tempo de processamento do algoritmo *Selection Sort*

3.2 *Insertion Sort*

Sua implementação é simples, consiste em cada ciclo, selecionar o segundo elemento, e caso ele for menor que seus elementos à esquerda, reposicionando-o até que ele seja o menor elemento. O algoritmo é mais bem aproveitado se a entrada for pequena, ou se já estiver "quase"ordenada. É considerado um algoritmo estável, um algoritmo que não muda drasticamente suas posições do vetor.

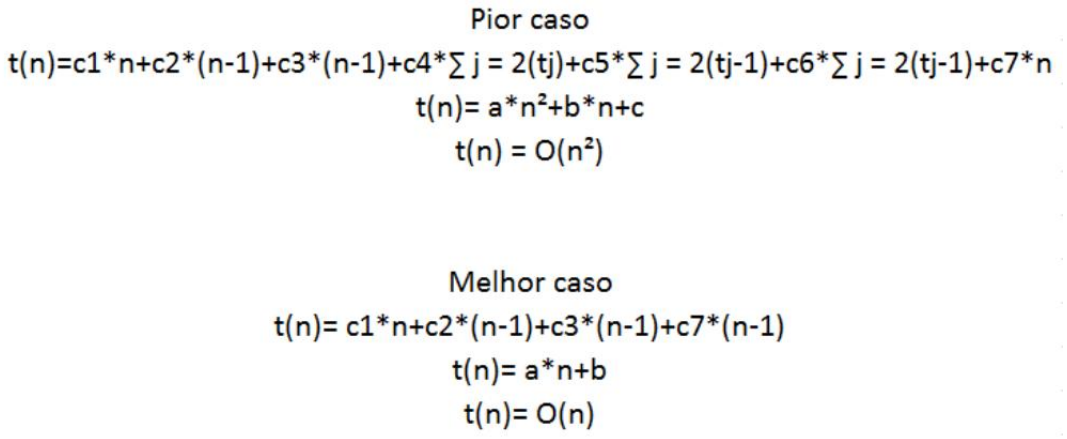

Figura 2: Complexidade *Insertion Sort*

Tamanho da entrada (n)	Aleatório	Crescente	Decrescente
10	0,000002	0,000001	0,000002
1000	0,0007	0,000005	0,0017
10000	0,07	0,000028	0,14
100000	5,5	0,00035	11
1000000	711	0,002	1445

Tabela 2: Tabela de tempo de processamento do algoritmo *Insertion Sort*

3.3 *Shell Sort*

O algoritmo Shell Sort, também é um algoritmo de ordenação, porém ele é uma melhoria do algoritmo Insertion Sort, ele também trabalha a cada ciclo comparando 2 elementos, mas não são elementos subsequentes, são elementos separados por uma variável que é incrementada a cada ciclo, existem várias formas de implementar esse incrementos, o algoritmo mais eficiente encontrado até o momento é o algoritmo de Knuth, por esse motivo, de existir diversas maneiras de ser implementado, a seu custo para melhor e pior caso ainda é incerto.

Por ser um algoritmo com infinitas formas de ser implementado, para melhor e pior caso, não existe uma complexidade definitiva, porém, pelos testes apresentados até hoje, o menor custo já calculado foi $O(n)$.

Tamanho da entrada (n)	Aleatório	Crescente	Decrescente
10	0,000001	0,000001	0,000001
1000	0,000222	0,000027	0,00006
10000	0,004	0,000395	0,000790
100000	0,08	0,003	0,009
1000000	0,93	0,04	0,10

Tabela 3: Tabela de tempo de processamento do algoritmo *Shell Sort*

3.4 *Merge Sort*

O algoritmo Merge Sort é muito bem aproveitado ao ordenar grande quantidade de dados, por´em para qualquer que seja a organização de entrada de dados (crescente, decrescente, aleatória), ele produz um tempo de processamento muito similar. Ele é dividido em 2 etapas de processamento, a primeira divide o vetor de entrada em trechos da metade do tamanho inicial, enquanto a segunda mescla esses trechos. Na etapa de mescla, assume-se que os trechos já estão previamente ordenados.

Linha	Código	Custo
1.	int i, j, k, *w;	1
2.	w = (int*) malloc ((r-p) * sizeof (int));	1
3.	i = p; j = q;	1
4.	k = 0;	1
5.	while (i < q && j < r)	n
6.	if (v[i] <= v[j])	n-1
7.	w[k++] = v[i++];	n-1
8.	else w[k++] = v[j++];	n-1
9.	while (i < q)	n
10.	w[k++] = v[i++];	n-1
11.	while (j < r)	n
12.	w[k++] = v[j++];	n-1
13.	for (i = p; i < r; ++i)	n
14.	v[i] = w[i-p];	n-1
15.	free (w);	1

Linha	Código	Custo
1.	if (p < r-1)	1
2.	int q = (p + r)/2;	1
3.	merge_sort(p, q, v);	t(n/2)
4.	merge_sort(q, r, v);	t(n/2)
5.	intercala(p, q, r, v);	t(n)

Parte interativa

c5*n+c6*(n-1)+c7*(n-1)+c8*(n-1)+c9*n+c10*(n-1)+c11*n+c12*(n-1)+c13*n+c14*(n-1)

t(n) = n*(c5+c6+c7+c8+c9+c10+c11+c12+c13+c14) - (c6+c7+c8+c10+c12+c14)

t(n) = n*A - B

t(n) = O(n)

Parte recursiva

t(n) = t(n/2) + n

t(n) = n + log2(n) * n

t(n) = O(n log n)

MELHOR E PIOR CASO O(nlogn)

Figura 3: Complexidade *Merge Sort*

Tamanho da entrada (n)	Aleatório	Crescente	Decrescente
10	0,000003	0,000001	0,000002
1000	0,000117	0,000075	0,000072
10000	0,001408	0,000860	0,0010
100000	0,016	0,011	0,012
1000000	0,17	0,11	0,11

Tabela 4: Tabela de tempo de processamento do algoritmo *Merge Sort*

3.5 *Quick Sort*

O algoritmo *Quick Sort*, possui várias implementações, com valores diferentes para o seu pivô. São eles: **pivô como final do vetor**, **pivô como mediana de três valores aleatórios**, **pivô como valor do meio**, **pivô como valor randômico**. De forma geral, funciona da seguinte maneira: o primeiro passo é determinar um pivô, que tem de ser um valor presente na sequencia a ser ordenada. Então ele ordena o vetor deixando os valores a esquerda sendo menores que o pivô, e a direita maiores que o pivô, a após feito isso, o vetor é dividido entre dois, o primeiro com os valores menores que o pivô, e o segundo com os valores maiores que o pivô. Esse procedimento é realizado até que cada vetor tenha apenas um elemento.

3.5.1 Pivô como final do vetor

O valor do pivô foi calculado a partir do final da sequência.

Tamanho da entrada (n)	Aleatório	Crescente	Decrescente
10	0,000001	0,000001	0,000002
1000	0,000075	0,001	0,0011
10000	0,00077	0,10	0,11
100000	0,008	8,4	8,5
1000000	0,08	1919	0,2

Tabela 5: Tabela de tempo de processamento do algoritmo *Quick Sort* pivô final

Os tempos de processamento maiores foram das entradas: crescente e aleatório. Isso se deu pelo fato, que quando a entrada já estava ordenada de alguma maneira, e o pivô se encontrava em uma extremidade, os vetores foram divididos em apenas uma parte, sendo que a outra parte ficou vazia, fazendo com que nestes casos eles se encontrassem perto do pior caso.

3.5.2 Pivô como mediana de três valores aleatórios

O valor do pivô foi calculado a partir da mediana de valore aleatórios da sequência.

Tamanho da entrada (n)	Aleatório	Crescente	Decrescente
10	0,000014	0,000011	0,000013
1000	0,00066	0,00049	0,00068
10000	0,008	0,008	0,0089
100000	0,07	0,06	0,06
1000000	0,53	0,54	0,51

Tabela 6: Tabela de tempo de processamento do algoritmo *Quick Sort* pivô mediana

Com o pivô encontrando-se pela mediana de três valores aleatórios, todos os tempos de processamento foram relativamente equivalentes, pois o pivô provavelmente foi encontrado como um valor mediano da sequência.

3.5.3 Pivô como valor do meio

O valor do pivô foi calculado a partir do meio da sequencia pela fórmula; pivô = $\frac{(incio+fim)}{2}$.

Tamanho da entrada (n)	Aleatório	Crescente	Decrescente
10	0,000001	0,000001	0,000001
1000	0,00009	0,000048	0,00003
10000	0,0010	0,000388	0,0004
100000	0,010	0,004	0,005
1000000	0,12	0,04	0,05

Tabela 7: Tabela de tempo de processamento do algoritmo *Quick Sort* pivô valor do meio

Assim como a mediana, o valor do meio do vetor, se deu como um dos melhores casos de complexidade, pois com os valores crescente e decrescentes, o vetor foi divido exatamente na metade, fazendo o mínimo de comparações possíveis.

3.5.4 Pivô como valor randômico

O valor do pivô foi calculado a partir de um valor aleatório presente na sequencia.

Tamanho da entrada (n)	Aleatório	Crescente	Decrescente
10	0,000015	0,000013	0,000012
1000	0,0006	0,0006	0,0007
10000	0,006	0,005	0,006
100000	0,06	0,05	0,05
1000000	0,52	0,41	0,43

Tabela 8: Tabela de tempo de processamento do algoritmo *Quick Sort* pivô valor randômico

Os tempos de processamento maiores foram das entradas crescente a aleatório, isso se deu pelo fato, que quando a entrada já estava ordenada de alguma maneira, e o pivô se encontrava em uma extremidade, os vetores foram divididos em apenas uma parte, sendo que a outra parte ficou vazia, fazendo com que nestes casos eles se encontrassem perto do pior caso.

3.6 *Heap Sort*

O *Heapsort* é um dos algoritmos mais utilizados nos dias de hoje, pois seu pior caso computacional sempre será $O(n \log n)$. O seu uso em filas de prioridade, apresenta uma abordagem mais eficiente,

que o modelo unidirecional que e comumente usado. Sempre que necessitar inserir um item, por exemplo, devido ao fato de o arranjo estar abstraído num sistema de árvore B, quando for necessário andar por todo o arranjo para encontrar um determinado local, se o arranjo tiver tamanho N, em uma fila comum, deveria andar por no máximo N posições, mas em uma árvore, andaria por, no máximo, $\log n$.

Linha	Código	Custo
1.	void heapify(int arr[], int n, int i)	
2.	int largest = i;	1
3.	int l = 2*i + 1;	1
4.	int r = 2*i + 2;	1
5.	if (l < n && arr[l] > arr[largest])	1
6.	largest = l;	1
7.	if (r < n && arr[r] > arr[largest])	1
8.	largest = r;	1
9.	if (largest != i){	1
10.	int aux = arr[i];	1
11.	arr[i] = arr[largest];	1
12.	arr[largest] = aux;	1
13.	heapify(arr, n, largest);	n

Melhor Caso
O(NlogN)

Pior Caso
O(NlogN)

1.	void heap_sort(int arr[], int n)	
2.	for (int i = n / 2 - 1; i >= 0; i--)	n
3.	heapify(arr, n, i);	t(n/2)
4.	for (int i=n-1; i>=0; i--)	n
5.	int aux = arr[0];	n-1
6.	arr[0] = arr[i];	n-1
7.	arr[i] = aux;	n-1
8.	heapify(arr, i, 0);	n-t(n/2)

Figura 4: Complexidade *Heap Sort*

Tamanho da entrada (n)	Aleatório	Crescente	Decrescente
10	0,000007	0,000003	0,000007
1000	0,000642	0,000521	0,000567
10000	0,007	0,006	0,006653
100000	0,07	0,05	0,06
1000000	0,39	0,32	0,32

Tabela 9: Tabela de tempo de processamento do algoritmo *Heap Sort*

3.7 *Bubble Sort*

O método de ordenação *Bubble Sort* ou Bolha, é um método de implementação simples, e que pode ser facilmente compreendido, consiste em cada ciclo, comparar dois elementos, e se o segundo for menor que o primeiro eles trocam de posição. É um algoritmo estável, que não possui mudanças significativas em relação as suas posições a cada ciclo de troca, mas que também é um pouco lento em comparação com outros algoritmos de ordenação, como o *Insertion Sort*. O algoritmo é melhor aproveitado com entradas menores.

		Custo	Vezes
1.	for(i=0 ; i<=tam-1 ; i++)	c1	n
2.	for(j=0 ; j<=tam-(i+1) ; j++)	c2	∑j=1 até-n1 de t(j)
3.	if(vet[j] > vet[j+1])	c3	∑j=1 até-n1 de t(j-1)
4.	troca(vet[j] , vet[j+1])	c4	∑j=1 até-n1 de t(j-1)

Melhor Caso

$c1*(n) + c2*(\sum j=1 \text{ até-n1 de } t(j)) + c3*(\sum j=1 \text{ até-n1 de } t(j-1))$

$t(n) = n^2*A + n*B + C$

$t(n) = O(n^2)$

Função Quadrática

Pior Caso

$c1*(n) + c2*(\sum j=1 \text{ até-n1 de } t(j)) + c3*(\sum j=1 \text{ até-n1 de } t(j-1)) + c4*(\sum j=1 \text{ até-n1 de } t(j-1))$

$t(n) = n^2*A + n*B + C$

$t(n) = O(n^2)$

Função Quadrática

Figura 5: Complexidade *Bubble Sort*

Tamanho da entrada (n)	Aleatório	Crescente	Decrescente
10	0,000002	0,000002	0,000001
1000	0,00184	0,001307	0,001953
10000	0,25	0,12	0,18
100000	27	11	16
1000000	3118	1258	1972

Tabela 10: Tabela de tempo de processamento do algoritmo *Bubble Sort*

4 Conclusões

4.1 *Selection Sort*

Apresentou grande desempenho em entradas pequenas, e um desempenho médio em entradas maiores. Com a entrada crescente, o maior tempo entre os vetores de entrada foi 1094 segundos, e apresentou o melhor caso $O(n^2)$; com entradas decrescentes e aleatórias, o maior tempo entre os vetores de entrada foi 1118 e 1096 segundos respectivamente, ambos com $O(n^2)$, o pior caso foi o vetor de entrada aleatória, e o decrescente foi um caso médio.

4.2 *Insertion Sort*

Constatou-se que as entradas com números em ordem crescente, decrescente e aleatório, com sequências da ordem de 10 até 1.000 unidades, atingem praticamente o mesmo tempo de processamento. A partir de 1.000 até 1.000.000 unidades, a entrada crescente possui um aumento em tempo insignificativo, a sua entrada com 1.000.000 de unidades possui um tempo de apenas 0,002 segundos.

A entrada decrescente aumenta em muito seu tempo de processamento conforme vai progredindo, o tempo de processamento de 1.000.000 unidades é de 1445 segundos. Já a entrada gerada por valores aleatórios também tem seu tempo de processamento drasticamente aumentada conforme o volume de entrada também aumenta, seu tempo de processamento para 1.000.000 de

unidades é de 711 segundos. Assim pode-se concluir que a entrada crescente é o melhor caso, a decrescente é o pior caso, e a entrada aleatória é um caso médio.

4.3 *Shell Sort*

Apresentou grande desempenho em todos os tipos de entrada, com entrada crescente de um milhão de unidades, apresentou o tempo de 0,04 segundos, com a entrada decrescente de um milhão de unidades, apresentou o tempo de 0,10 segundos, e com entrada aleatória de um milhão de unidades, apresentou o tempo de 0,93 segundos.

Seu cálculo de complexidade dependerá do tipo de implementação que foi escolhida. Como já foi publicado por Shell (1959), o algoritmo poderá ter complexidade de $O(n)$ ou até $O(n \log n)$, segundo Pratt (1971).

4.4 *Merge Sort*

Apresentou bom comportamento em relação a tempo, em todos os tipos e tamanhos de entrada. Seus maiores tempos de processamento foram de 0,17, 0,11, e 0,11 segundos para as entradas aleatórias, crescentes e decrescentes respectivamente, com 1.000.000 de unidades. Portanto, o algoritmo teve excelente desempenho em todo tipo de entrada. Sua complexidade para todos os casos é $O(n \log n)$, em outras palavras, logarítmica.

4.5 *Quick Sort*

Através dos testes, constatou-se que o pior tipo de implementação a ser realizado no pivô do algoritmo, seria o valor do final da sequência quando as entradas já estiverem de alguma forma ordenada, pois os números de comparações a ser realizada mesmo que raro, seria n, assemelhando-se aos tempos gastos pelo algoritmo *Insertion Sort*. Isso se dá pelo fato, de que todos os elementos da sequência ficarem sendo maiores ou menores do que o pivô atingindo a complexidade $O(n)$.

Além desse caso específico não foi notada nenhuma alteração significativa nos tempos gastos dos testes realizados, visto que os valores maiores e menores que o valor do pivô seriam mais ou menos equivalentes, atingindo a complexidade de melhor e médio caso $O(n \log n)$, pois se o número de comparações entre as duas extremidades forem equivalentes, o primeiro passo do vetor seria dividi-lo, por partes semelhantes em termos de tamanho.

4.6 *Heap Sort*

A organização de um arranjo por meio de um *heap*, é uma implementação muito mais eficiente que a organização por meio de um arranjo unidirecional, como é comumente estudado. Como ele trabalha com o conceito de árvore de até 2 filhos, o caminho máximo para se buscar, adicionar, remover ou alterar um item, será (na pior das hipóteses) $O(n \log n)$, onde n corresponde ao tamanho do arranjo inicial. Mesmo para casos onde o arranjo a se tornar uma fila, seja muito grande, ele se demonstra muito eficiente.

4.7 *Bubble Sort*

Apresentou bom desempenho em entradas pequenas, e um desempenho ruim em entradas maiores. Com a entrada crescente, o maior tempo entre os vetores de entrada foi 1258 segundos, e apresentou o melhor caso $O(n^2)$, função quadrática; com entradas decrescentes, o maior tempo entre os vetores de entrada foi 1972 segundos, e apresentou o pior caso $O(n^2)$, função quadrática; e com entradas aleatórias, o maior tempo entre os vetores de entrada foi 3118 segundos, e apresentou um caso médio $O(\frac{n^2}{4})$, função quadrática.