Análise Empírica (simples) do Algoritmo InsertionSort

Daniele P. Santiago, daniele@estudante.ufscar.br, RA: 792175

Universidade Federal de São Carlos

1. INTRODUÇÃO

O algoritmo InsertionSort ou ordenação por inserção tem como ideia base, dada uma lista, ordenar seus itens inserindo-os na posição adequada, um de cada vez, e por fim, retornar os elementos de maneira ordenada. É considerado de simples implementação, leitura e manutenção. Essa estratégia tem o potencial de obter melhores resultados quando precisa fazer menos trocas de posição, ou seja, quando a lista está relativamente ordenada, e seus piores resultados ocorrem quando a matriz está na ordem inversa daquela que se deseja ordenar. Nessa análise, será mostrado o código de implementação e uma análise simples relacionando a quantidade de comparações feitas, trocas de posição e o tempo de execução, para por fim chegarmos a uma resolução aproximada sobre a eficiência do algoritmo.

Entrada: V[1..n] de inteiros
Resultado: V ordenado

```
for (i=1; i< n ; i++){
        for(j = i; j>0 && v[j-1] > v[j]; j--){
                aux = v[j];
                v[j] = v[j-1];
                v[j-1] = aux;
            }
        }
```

Algoritmo 1: Pseudocódigo de InsertionSort na linguagem C.

2. DESENVOLVIMENTO

Para realizar a análise, fora implementado no código duas variáveis contadoras, sendo elas respectivamente “comparacao” e “trocaPosicao”. O primeiro contador foi responsável por armazenar a quantidade de vezes que alguma comparação é feita durante a execução do algoritmo. Segue o exemplo de como ele foi disposto.

Entrada: V[1..n] de inteiros, comparacao = 0.
Resultado: V ordenado, quantidade de comparações feitas.

```
for(i=1; i<n ; i++){
    comparacao++;
        for(j = i; j>0 && v[j-1] > v[j]; j--){
          comparacao++;
          aux = v[j];
          v[j] = v[j-1];
          v[j-1] = aux;
        }
}
```

Algoritmo 2: Pseudocódigo de InsertionSort com implementação da variável contadora "comparacao".

Se analisarmos a estrutura do código é possível fazer algumas observações. Notemos que a comparação dentro do primeiro loop "for", terá como resultado $(n-1)$ iterações, nos dando a projeção de uma função linear. Porém se observamos o segundo loop "for", que será o responsável por fazer a troca de posições dentro da lista, não podemos inferir um valor constante, pois neste caso está submetido a mudanças de acordo com a quantidade de elementos e a sua disposição.

Podemos, de qualquer modo projetar o que aconteceria no pior caso, onde os valores estariam inversamente ordenados, nos dando a equação $\frac{n*(n-1)}{2}$. Sendo assim, no pior caso possível, a quantidade de comparações feitas no algoritmo seria equivalente à soma das equações previamente ditas, ou seja $(n-1) + \frac{n*(n-1)}{2}$ *(1)*. Podemos analisar ainda que no melhor caso possível, quando os elementos já estão ordenados, a segunda comparação não chegaria a acontecer, visto que não seria necessário troca alguma. O valor seria equivalente a constante 0, e ficaríamos então com 0 + (n-1). Sendo assim, os resultados pela variável "comparacao" ficam restringidos a (n-1) ≤ comparacao ≤ $(n-1) + \frac{n*(n-1)}{2}$.

De modo a observar os resultados obtidos com diferentes quantidades de elementos, foram feitos sucessivos testes utilizando o algoritmo Insertion Sort. Os elementos variaram de dez a dez mil, sendo os mesmos gerados aleatoriamente durante a execução do código. Tivemos como resultado a seguinte tabela:

Quantidade de Elementos	Comparações (Média de 5 execuções)	Melhor cenário	Pior cenário
10	34,4	9	54
100	2.581	99	5.049
1000	254.738,6	999	500.499
2000	987.746,4	1.999	2.000.999
3000	2.273.501	2.999	4.501.499
4000	3.998.911	3.999	8.001.999
5000	6.232.328	4.999	12.502.499
6000	9.034.007	5.999	18.002.999
7000	12.298.769	6.999	24.503.499
8000	15.965.771	7.999	32.003.999
9000	20.177.379	8.999	40.504.499
10000	25.026.626	9.999	50.004.999

Tabela 1: Relação comparação por elementos

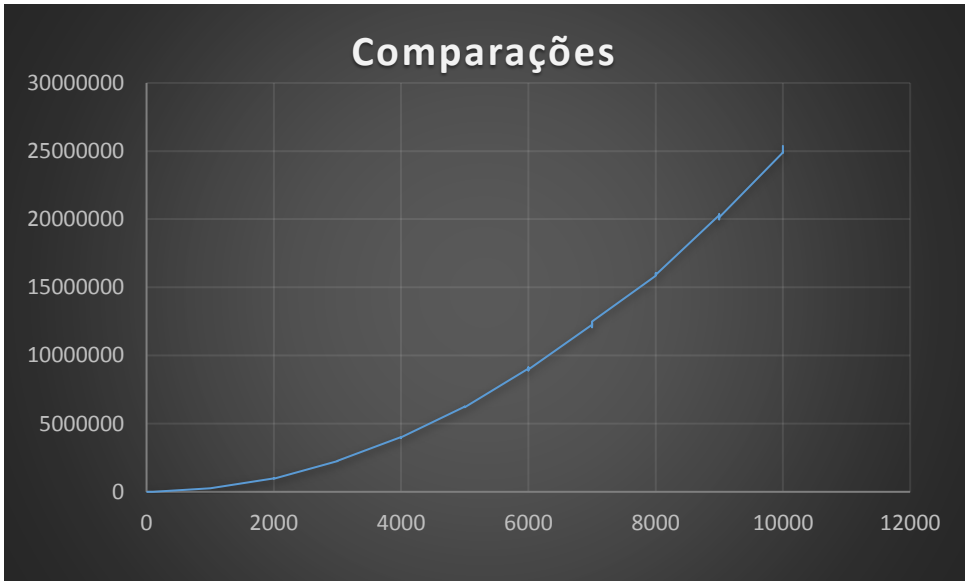

Gráfico 1: Relação Comparação por elementos

Para determinar a quantidade de vezes que foram trocadas as posições do elemento, fora implementado o seguinte algoritmo:

Entrada: V[1..n] de inteiros, trocaPosicao = 0.
Resultado: V ordenado, quantidade de vezes que as posições foram trocadas.

```
for(i=1; i<n ; i++){
        for(j = i; j>0 && v[j-1] > v[j]; j--){
          trocaPosicao++;
          aux = v[j];
          v[j] = v[j-1];
          v[j-1] = aux;
        }
}
```

Algoritmo 3: Pseudocódigo de InsertionSort com implementação da variável contadora "trocaPosicao".

Como observado previamente, a quantidade de vezes que ocorrerá a troca de posições entre os elementos dependerá de sua quantidade e da ordenação do vetor. Podemos, de qualquer maneira inferir o pior e o melhor caso. No melhor caso, o vetor já estará ordenado, então teremos 0 trocas de posição. No pior, ocorrerá $\frac{n*(n-1)}{2}$ *(2)* trocas de posição. Nesta situação também fora feita a análise de doze elementos variando de dez a dez mil, onde obtivemos os seguintes resultados[2]:

Quantidade de Elementos	**Troca de Posição (Média de 5 execuções)**	**Melhor cenário**	**Pior cenário**
10	25,4	0	45
100	2.482	0	4.950
1000	253.739,6	0	499.500
2000	986.946,8	0	1.999.000
3000	2.270.502	0	4.498.500
4000	3.994.912	0	7.998.000
5000	6.227.329	0	12.497.500
6000	9.028.008	0	17.997.000
7000	12.291.770	0	24.496.500
8000	15.957.772	0	31.996.000
9000	20.168.380	0	40.495.500
10000	25.016.627	0	49.995.000

Tabela 2: Relação troca de posição por quantidade de elementos

Gráfico 2: Relação troca de posição por quantidade de elementos

De modo a observar a relação entre a quantidade de elementos e o tempo de execução, tais medições foram tomadas, onde tivemos os seguintes resultados:

Gráfico 3: Relação entre o tempo por quantidade de elementos

Sendo o menor valor 0.000006s referente a 10 quantidades de elementos e o maior 0.202186s referentes a 10 mil quantidades de elementos.

Porém adicionando um caso onde há 100 mil elementos, onde o tempo de execução fora 19s, tivemos o seguinte resultado:

Gráfico 4: Relação entre o tempo por quantidade de elementos com adição de um caso de tempo

3. CONCLUSÃO

Observando os dados previamente exibidos podemos fazer algumas aferições. Primeiramente, durante o melhor caso possível, a função que relaciona as comparações feitas no algoritmo com a quantidade de elementos na lista é uma função linear; já a que recebe a quantidade de trocas é uma constante 0. No pior caso possível as funções seriam quadráticas dadas as equações (1) e (2). Todavia nos casos médios, levando-se em consideração as tabelas 1 e 2 e os gráficos 1 e 2, vemos que as funções se aproximam mais de uma função quadrática do que uma função linear ou constante. Além disso, pode-se observar que tanto na relação "Comparação x Elementos", como "Troca de Posição x Elementos", os casos médios se aproximam dos resultados do (pior caso)/2 , mas não podemos afirmar uma relação direta visto que esse depende de algumas variantes que não temos como controlar.

A relação do tempo quantidade de execução se mantém quase numa relação linear (gráfico 3), porém a variação de tempo ocorrida em segundos de 10 a 10 mil elementos não é tão considerável. Todavia, quando adicionamos um caso de 100 mil elementos (gráfico 4), conseguimos observar as considerações anteriores, e em como o gráfico aumenta drasticamente, ainda numa função linear. Podemos concluir, então, que o

InsertionSort é eficiente para casos com poucos elementos, porém muito lento para casos com mais elementos. Temos, como resultado:

Comparações x Quantidade de Elementos

Pior caso: função quadrática.

Médio caso: função quadrática.

Melhor caso: função linear.

Troca de posição x Quantidade de Elementos

Pior caso: função quadrática.

Médio caso: função quadrática.

Melhor caso: constante.

Apêndice A – Dados utilizados

Elementos	Tempo de Execução	Troca de Posição	Comparações
10	0.000006	26	35
10	0.000020	25	34
10	0.000006	27	36
10	0.000005	26	35
10	0.000020	23	32
100	0.000021	2438	2537
100	0.000033	2480	2579
100	0.000039	2705	2804
100	0.000028	2106	2205
100	0.000032	2681	2780
1000	0.001525	254648	255647
1000	0.002133	252781	253780
1000	0.001586	250813	251812
1000	0.001416	254088	255087
1000	0.001826	256368	257367
2000	0.005427	981521	981521
2000	0.006904	1012594	1012594
2000	0.004906	981617	983616
2000	0.007115	985344	987343
2000	0.006992	973658	973658
3000	0.012340	2253585	2256584
3000	0.017119	2280436	2283435
3000	0.019860	2272642	2275641
3000	0.017819	2274376	2277375
3000	0.017891	2271473	2274472
4000	0.031884	4016032	4020031
4000	0.029960	3960525	3964524
4000	0.033549	4014484	4018483
4000	0.029115	3999527	4003526
4000	0.029050	3983991	3987990
5000	0.047125	6244174	6249173
5000	0.049001	6217135	6222134
5000	0.051983	6237826	6242825
5000	0.052941	6232885	6237884
5000	0.048401	6204624	6209623
6000	0.079420	9026497	9032496
6000	0.076794	9113827	9119826
6000	0.079428	9134013	9140012
6000	0.076878	8895801	8901800
6000	0.069548	8969902	8975901
7000	0.096293	12260170	12267169
7000	0.093468	12070306	12077305
7000	0.100439	12388978	12395977

7000	0.108272	12241024	12248023
7000	0.113238	12498374	12505373
8000	0.125554	15842991	15850990
8000	0.123620	16018006	16026005
8000	0.125353	16033100	16041099
8000	0.125871	15973199	15981198
8000	0.122256	15921563	15929562
9000	0.170577	20289536	20298535
9000	0.164492	20359897	20368896
9000	0.155543	19980587	19989586
9000	0.157997	20080640	20089639
9000	0.177614	20131239	20140238
10000	0.202186	24883340	24893339
10000	0.138996	24911366	24921365
10000	0.143963	24890230	24900229
10000	0.145072	25025846	25035845
10000	0.161320	25372352	25382351
100000	19.028948	2502790778	2502690779