Trabalho 2 - Programação Funcional

Trabalho de Programação Funcional

Gustavo Vinícius Alba - 11911BCC016

Otávio Almeida Leite - 11911BCC010

- Obs: os valores correspondentes às entradas de "tempo" nas tabelas a seguir foram obtidos através do comando `:set +s`. Eles estão sujeitos a condições específicas de cada computador e contexto, e serão analisados apenas como um valor aparente.

Exercício 1 - Bubble Sort

Original:

Lista	Trocas	Tempo (s)
l1	0	6.66
l2	1999000	8.54
l3	2000	6.19
l4	1999000	7.76
l5	4000000	30.93
l6	4000000	31.57
l7	5999000	33.44
x1	0	0.01
x2	190	0.01
x3	100	0.01
x4	90	0.00
x5	95	0.00
x6	66	0.01
x7	94	0.01

Variação 1:

Lista	Trocas	Tempo (s)
l1	0	0.31
l2	1999000	6.3
l3	2000	5.05
l4	1999000	6.17
l5	4000000	22.39
l6	4000000	12.67
l7	5999000	23.17
x1	0	0.01
x2	190	0.01
x3	100	0.00
x4	90	0.01
x5	95	0.01
x6	66	0.01
x7	94	0.01

Variação 2:

Lista	Trocas	Tempo (s)
l1	0	0.3
l2	1999000	4.86
l3	2000	3.73
l4	1999000	4.91
l5	4000000	17.18
l6	4000000	14.35
l7	5999000	18.99
x1	0	0.01
x2	190	0.00

Lista	Trocas	Tempo (s)
x3	100	0.01
x4	90	0.01
x5	95	0.01
x6	66	0.00
x7	94	0.01

Análise Comparativa das Execuções

Em geral, não há mudança de quantidade de trocas a serem realizadas para ordenar as listas de acordo com o Bubble Sort, isso porque as trocas seguem sempre o mesmo raciocínio. O que melhora progressivamente do original para a variação 1 e depois para a variação 2 é que, por ocorrerem paradas antecipadas nos algoritmos, ocorrem menos comparações entre elementos da lista e, portanto, a execução ocorre em menos tempo. A última variação se aproveita que após cada chamada de "troca", o último elemento da lista (e nas chamadas seguintes, os imediatos antes dele), já se encontra na posição correta de ordenação da lista. Ou seja, a cada nova chamada de "troca", há menos uma comparação para ser feita na próxima chamada recursiva. A primeira variação apenas confere se, em algum momento anterior ao "final" do algoritmo original a lista já está ordenada, o que pode acontecer em alguns casos, e mais comparações se tornam desnecessárias.

Exercício 2 - Selection Sort

Original:

Lista	Trocas	Tempo (s)
l1	0	2.06
l2	Não determinado	Muito grande
l3	Não determinado	Muito grande
l4	Não determinado	Muito grande
l5	Não determinado	Muito grande
l6	Não determinado	Muito grande
l7	Não determinado	Muito grande
x1	0	0.00
x2	19	1.02

Lista	Trocas	Tempo (s)
x3	10	0.04
x4	18	0.49
x5	10	0.51
x6	13	0.16
x7	19	0.35

Variação 1:

Lista	Trocas	Tempo (s)
l1	0	2.18
l2	1999	3.02
l3	1	2.12
l4	1999	2.82
l5	2000	9.78
l6	3999	9.64
l7	3999	10.25
x1	0	0.0
x2	19	0.0
x3	10	0.0
x4	18	0.0
x5	10	0.0
x6	13	0.01
x7	19	0.0

Análise Comparativa das Execuções

Nos casos de listas muito grandes, o algoritmo original se mostrou extremamente ineficiente, ao ponto de não conseguirmos completar as chamadas de execução de forma prática, em tempo hábil. Nas listas menores, não houve variação da quantidade de trocas, isso porque a lógica do algoritmo permanesce a mesma: sempre haverá a necessidade de mover o menor valor para a "frente" da lista,

ordenando-a dessa forma. A diferença é que na segunda variação o mínimo é encontrado de forma muito mais eficiente usando a estratégia do 'foldr1' de operações binárias feitas repetidamente com um acumulador. No algoritmo original, a estratégia para encontrar o mínimo é a recursiva, que pesa muito mais para o aumento do tempo de execução, especialmente em listas grandes.

Exercício 3 - Insertion Sort

Original:

Lista	Comparações	Tempo (s)
l1	1999	0.32
l2	1999000	3.69
l3	3999	0.31
l4	1999001	3.43
l5	4002001	7.28
l6	4004000	7.29
l7	6001001	11.20
x1	19	0.0
x2	190	0.0
x3	118	0.0
x4	100	0.01
x5	109	0.0
x6	83	0.0
x7	110	0.0

Variação 1:

Lista	Comparações	Tempo (s)
l1	1999	0.32
l2	1999000	3.63
l3	3999	0.33

Lista	Comparações	Tempo (s)
l4	1999001	3.68
l5	4002001	7.46
l6	4004000	7.82
l7	6001001	11.74
x1	19	0.0
x2	190	0.0
x3	118	0.0
x4	100	0.0
x5	109	0.0
x6	83	0.01
x7	110	0.0

Análise Comparativa das Execuções

Nesse caso em específico, ambas as implementações são extremamente parecidas quanto a tempo de execução, e iguais com relação à quantidade de trocas. O motivo disso é que, mesmo que as implementações em código sejam diferentes, as operações realizadas nas duas abordagens são praticamente as mesmas, e dessa forma os algoritmos tem comportamento praticamente igual.

Exercício 4 - Quick Sort

Original:

Lista	Comparações	Tempo (s)
l1	3998000	2.51
l2	3998000	2.03
l3	3998002	2.47
l4	4002000	2.05
l5	8004000	5.03
l6	8004000	4.5

Lista	Comparações	Tempo (s)
l7	8004000	4.28
x1	380	0.0
x2	380	0.0
x3	200	0.0
x4	200	0.0
x5	160	0.0
x6	168	0.0
x7	162	0.0

Variação 1:

Lista	Comparações	Tempo (s)
l1	1999000	3.52
l2	1999000	3.77
l3	1999001	3.25
l4	2001000	3.56
l5	4002000	6.63
l6	4002000	7.67
l7	4002000	7.16
x1	190	0.0
x2	190	0.18
x3	100	0.0
x4	100	0.0
x5	80	0.0
x6	84	0.0
x7	81	0.0

Variação 2:

Lista	Comparações	Tempo (s)
l1	1002997	1.90
l2	1001998	1.89
l3	1002999	1.89
l4	1004000	2.01
l5	2009001	3.61
l6	2009000	4.35
l7	2008000	4.14
x1	127	0.0
x2	118	0.0
x3	92	0.0
x4	88	0.0
x5	78	0.0
x6	86	0.0
x7	74	0.0

Análise Comparativa das Execuções

Do algoritmo original para a primeira variação, é fácil entender o motivo de, em todos os casos, o número de comparações cair pela metade. O algoritmo original filtra os menores e maiores que o pivô percorrendo a lista duas vezes, uma para cada lista derivada. Já a primeira variação separa as listas derivadas percorrendo a original apenas uma vez, e assim, metade das comparações são necessárias. Da primeira para a segunda variação, o processo de escolha do pivô é melhorado, levando em conta as três primeiras posições, e em alguns casos isso promove uma melhoria considerável na quantidade de comparações que são necessárias. Contudo, também é visto um caso (da lista de teste `x6`) em que essa escolha acaba aumentando o número de comparações: a lista está disposta de uma forma que o algoritmo não a divide em partes vantajosas, dada a escolha de um pivô entre os primeiros três elementos.

Exercício 5 - Merge Sort

Lista	Comparações	Tempo (s)
l1	10864	0.34

Lista	Comparações	Tempo (s)
l2	11088	0.35
l3	10880	0.43
l4	11102	0.36
l5	25966	0.68
l6	25958	0.68
l7	26190	0.71
x1	40	0.01
x2	48	0.0
x3	40	0.0
x4	48	0.0
x5	49	0.0
x6	60	0.0
x7	65	0.0

Análise Comparativa das Execuções

Comparado com as melhores versões dos algoritmos Selection Sort e Quicksort, podemos observar algumas diferenças interessantes. Entre os três, o Selection Sort é o que tem os menores valores de trocas realizadas, comparadas à quantidade de comparações realizadas nos outros dois. Além disso, o Selection Sort se mostra eficiente (com relação a quantidade de trocas) com listas que estão quase ordenadas inicialmente, o que não é levado em conta pelos outros dois. Contudo, os outros dois algoritmos executam em tempos muito menores, sendo o Merge Sort o mais rápido, e com uma diferença considerável com relação ao número de comparações e tempo de execução do Quicksort. Assim, o Merge Sort se mostra consideravelmente mais eficiente que os algoritmos Selection Sort e Quicksort, tanto em tempo de execução quanto em comparações.