Trabalho Final de Programação Funcional

Davi Felipe Ramos de Oliveira Vilarinho

12011BCC006

Organizei o trabalhando de tal forma que cada seção contém a análise do desempenho separado de cada variação e, ao final de todo o trabalho, uma comparação entre todos.

Bolha

Original

O tempo para executar todos os testes foi de: 68.69 s

Teste	Tempo (s)	Contador
l1	4.27	0
l2	4.71	1999000
l3	3.75	2000
l4	4.78	1990000
l5	17.77	4000000
l6	17.72	4000000
l7	17.93	5999000
x1	0.01	0
x2	0.01	190
x3	0.01	100
x4	0.00	90
x5	0.01	95
x6	0.01	66
x7	0.01	94

Nota-se que o algoritmo cresce exponencialmente

Variação 1

O tempo para executar todos os testes foi de: 55.37 s

Teste	Tempo (s)	Contador
l1	0.04	0
l2	4.76	1999000
l3	3.93	2000
l4	4.50	1990000
l5	18.01	4000000
l6	9.59	4000000
l7	18.82	5999000
x1	0.00	0
x2	0.00	190
x3	0.00	100
x4	0.00	90
x5	0.00	95
x6	0.01	66
x7	0.00	94

Não houve diminuição nas trocas (como era esperado), mas houve uma diminuição no tempo de avaliação de elementos repetidos.

Variação 2

O tempo para executar todos os testes foi de: 36.43 s

Teste	Tempo (s)	Contador
l1	0.04	0
l2	2.82	1999000

Teste	Tempo (s)	Contador
I3	2.24	2000
I4	2.67	1990000
I5	10.34	4000000
I6	8.52	4000000
I7	11.52	5999000
x1	0.00	0
x2	0.00	190
x3	0.00	100
x4	0.00	90
x5	0.00	95
x6	0.00	66
x7	0.00	94

Houve uma diminuição bem relevante no tempo, uma vez que ele não precisa mais percorrer toda a lista, e sim a lista ainda não ordenada, aproveitando do fato de que o maior elemento da lista, em cada iteração, já está no final.

Seleção

Original

Teste	Tempo (s)	Contador
I1	0.82	0
I2	> 11min	indefinido
I3	> 11min	indefinido
I4	> 11min	indefinido
I5	> 11min	indefinido
I6	> 11min	indefinido

Teste	Tempo (s)	Contador
l7	> 11min	indefinido
x1	0.01	0
x2	0.37	19
x3	0.03	10
x4	0.19	18
x5	0.19	10
x6	0.07	13
x7	0.14	19

O tempo é grande demais por causa de sua para os piores cenários possíveis, sendo usável, então, apenas com pequenas listas ou já ordenadas.

Variação 1

Os testes todos rodaram em: 17.10s

Teste	Tempo (s)	Contador
l1	1.02	0
l2	1.39	1999
l3	1.06	1
l4	1.34	1999
l5	5.04	2000
l6	4.74	3999
l7	4.95	3999
x1	0.01	0
x2	0.01	19
x3	0.01	10
x4	0.01	18

Teste	Tempo (s)	Contador
x5	0.00	10
x6	0.00	13
x7	0.01	19

O tempo reduziu drasticamente, graças a diminuição da quantidade de vezes que uma lista era circulada.

Variação 2

Todos rodaram em 18.90 s

Teste	Tempo (s)	Contador
l1	1.04	0
l2	1.31	1999
l3	0.97	1
l4	1.27	1999
l5	4.67	1
l6	4.74	3999
l7	4.83	3999
x1	0.00	0
x2	0.01	19
x3	0.01	10
x4	0.01	18
x5	0.00	10
x6	0.00	13
x7	0.00	19

Similar ao teste passado em questão de tempo, uma vez que é a mesma função remove_minimo. O problema, provavelmente, é o uso do foldr que pode não ser otimizado para esta tarefa por causa do lazy evaluation quando em massa. Mas em testes separados aparentemente é minimamente

superior.

Inserção

Original

O tempo para testar todos foi de: 22.34 s

Teste	Tempo (s)	Contador
l1	0.04	0
l2	2.66	1999000
l3	0.03	2000
l4	2.58	1999000
l5	4.92	4000000
l6	4.93	4000000
l7	7.16	5999000
x1	0.00	0
x2	0.01	190
x3	0.01	100
x4	0.00	90
x5	0.01	95
x6	0.00	66
x7	0.01	94

O algoritmo evita perder tempo quando a lista já está quase ordenada, mas faz muitas comparações, embora mais eficientes que o Bubble Sort, por exemplo, mesmo assim, consegue um resultado satisfatório.

Variação 1

Realiza todos testes em: 20.82s

Teste	Tempo (s)	Contador
l1	0.05	0
l2	2.46	1999000
l3	0.04	2000
l4	2.37	1999000
l5	4.78	4000000
l6	4.87	4000000
l7	7.16	5999000
x1	0.00	0
x2	0.22	190
x3	0.01	100
x4	0.00	90
x5	0.01	95
x6	0.00	66
x7	0.00	94

Melhorias mínimas para os casos grandes, e para os pequenos teve uma anomalia. O foldr não fez tanta diferença relativamente à diferença que demais variações em outros algoritmos fizeram

Quicksort

Original

Realiza todos testes em: 41.28s

Teste	Tempo (s)	Contador
l1	4.52	3998000
l2	4.29	3998000
l3	4.37	3999002

Teste	Tempo (s)	Contador
I4	4.25	4002000
I5	8.56	8004000
I6	8.22	8004000
I7	8.28	8004000
x1	0.00	380
x2	0.00	380
x3	0.01	200
x4	0.00	200
x5	0.00	160
x6	0.00	168
x7	0.01	162

O Quicksort não é ruim em sua forma "padrão", mas também realiza muitas operações desnecessárias se perceorre duas vezes a lista identificando maiores e menores.

Variação 1

Realiza todos testes (sem extra) em: 27.50s

Teste	Tempo (s)	Contador
I1	3.02	1999000
I2	2.91	1999000
I3	3.02	1999001
I4	2.77	2001000
I5	5.93	4002000
I6	5.55	4002000
I7	5.61	4002000
x1	0.00	190

Teste	Tempo (s)	Contador
x2	0.01	190
x3	0.01	100
x4	0.01	100
x5	0.01	80
x6	0.01	84
x7	0.00	81
EXTRA: `[700..2000]++[4000,3999..200]`	9.56	3924201

Como se pode ver, a redução de comparações à metade ajuda muito o quicksort, conseguindo uma diminuição de aproximadamente 33% do tempo necessário para execução dele. O quicksort tem dificuldade em listas que já estão parcialmente ordenadas (crescentemente ou decrescentemente), por isso, não obteve um resultado perfeito. Ao final, um teste extra que mostra que numa lista ainda maior (5102 de tamanho), mas levemente desordenada, o quicksort tem um resultado excelente, com até menos troca que em listas menores.

Variação 2

Realiza todos testes (sem extra) em: 15.57s

Teste	Tempo (s)	Contador
l1	1.51	1002997
l2	1.43	1002997
l3	1.52	1002999
l4	1.40	1004000
l5	2.98	2009000
l6	2.88	2009000
l7	2.87	2009000
x1	0.00	127
x2	0.00	127

Teste	Tempo (s)	Contador
x3	0.01	92
x4	0.00	92
x5	0.00	82
x6	0.00	86
x7	0.00	77

O Quicksort com a mediana teve um ótimo desempenho, reduzindo novamente à metade o número de comparações e praticamente também o tempo. Isso acontece porque o pior cenário do Quicksort (quando a lista já está ordenada) é contornado e só acontece quando a lista é muito pequena (na recursão inclusive).

Mergesort

Realiza todos testes (sem extra) em: 0.62s

Teste	Tempo (s)	Contador
l1	0.07	10864
l2	0.05	11088
l3	0.06	10880
l4	0.05	11102
l5	0.15	25966
l6	0.13	25958
l7	0.14	26190
x1	0.00	40
x2	0.00	48
x3	0.01	40
x4	0.00	40
x5	0.00	48

Teste	Tempo (s)	Contador
x6	0.00	60
x7	0.00	65

O Mergesort é disparadamente o que realizou menos comparações e teve resultados exorbitantemente superiores. Tempo reduzindo à menos de 1s em todos os casos e contador diminuindo em 100 vezes. Isso se deve ao fato de que ele consegue reduzir o tamanho das listas recursivamente e ordená-las *ao mesmo tempo* em que as diminui com a função "intercala".

Conclusão

É possível observar que os resultados tendem a ser melhor na medida em que reduz a quantidade de vezes em que circula pelos elementos da lista e o quanto consegue reduzir o tamanho da lista. Isso é basicamente um princípio da recursividade: reduzir o tamanho do problema para casos mais fáceis de resolver (chamados casos base).

Este fato constatado é provado pela eficiência de tempo e comparações existententes no Mergesort, que aplica (e bem) ambos, sendo uma escala de grandeza menor que todos os outros.

Comparando todos, é possível identificar que:

Algoritmo	Aleatória	Ordenada	Ordenada Reversamente	Lista Pequena	Lista Grande
Bolha (original)	Ruim	Ruim	Ruim	Ruim	Ruim
Bolha (variação 1)	Ok	Ok	Ruim	Ruim	Ruim
Bolha (variação 2)	Ok	Ok	Ruim	Ruim	Ruim
Seleção (Original)	Péssimo	Ok	Péssimo	Excelente	Péssimo
Seleção (Variação 1)	Ok	Excelente	Bom	Excelente	Bom
Seleção (Variação 2)	Ok	Excelente	Bom	Excelente	Bom

Algoritmo	Aleatória	Ordenada	Ordenada Reversamente	Lista Pequena	Lista Grande
Inserção (Original)	Ok	Excelente	Ok	Ok	Ok
Inserção (Variação 1)	Ok	Excelente	Ok	Ruim	Ok
Quicksort (Original)	Bom	Ruim	Ruim	Ruim	Ruim
Quicksort (Variação 1)	Bom	Ruim	Ruim	Ruim	Ok
Quicksort (Variação 2)	Bom	Ruim	Ruim	Ok	Bom
Mergesort	Excelente	Excelente	Excelente	Ok	Excelente

Sendo assim é entendível que o mergesort é praticamente a melhor opção, embora tenha uma maior complexidade de implementação. Em suas melhores implementações, o Quicksort seria o ideal para listas bem embaralhadas e grandes, Seleção é ideal se precisar de implementações simples, assim como o Inserção, embora um pouco menos, e o Bolha sendo o menos otimizado de todos na média dos cenários.