

INF 213 - Roteiro da Aula Prática

Arquivos disponibilizados para a prática:

https://drive.google.com/file/d/1l_QO_maS5AFy7h8Dc-2DJ78RHr1xd8jM/view?usp=sharing

O programa geraEntrada.cpp recebe como argumento (via linha de comando) numérico N e, então, imprime N e uma sequência de N números pseudo-aleatórios (na verdade, a semente nunca muda). Use-o para gerar 100 mil números aleatórios e salve-os em um arquivo (`./a.out 100000 > entrada100000.txt`). Nesta prática vamos utilizar arquivos de teste nesse formato para os experimentos.

Etapa 1

Considere a implementação do QuickSort em quicksort.cpp e a do InsertionSort disponível em insertionsort.cpp. Compile esses dois programas e meça o tempo de execução para essa entrada de tamanho 100 mil.

QuickSort:23396

InsertionSort:6233324

Para entradas muito pequenas o InsertionSort costuma ser mais eficiente do que o QuickSort. Repita os testes utilizando entradas pequenas: descubra tamanhos de entrada onde o InsertionSort é mais eficiente. (como as entradas são pequenas, pode ser que medir a diferença entre os dois métodos não seja algo tão simples -- idealmente em vez de ordenar uma vez deveríamos ter um “for” que ordena, por exemplo, 1000 cópias do mesmo array com o objetivo de fazer com que todos algoritmos estudados gastem mais tempo de execução).

Note que algumas vezes, para entradas pequenas, um algoritmo $O(n^2)$ é mais eficiente do que, por exemplo, um algoritmo $O(n \log n)$ ou mesmo um $O(n)$. Isso ocorre porque a notação “ O ” esconde algumas constantes relacionadas ao custo dos algoritmos. Por exemplo, **para $n < 100$** , um algoritmo que faz n^2 ($O(n^2)$) operações *provavelmente* é mais eficiente do que um que faz $100n$ ($O(n)$) operações.

Ideia: assim como o MergeSort, o QuickSort divide os arrays em arrays menores e os ordenam recursivamente. E se ordenarmos esses subarrays com o InsertionSort quando eles chegarem a um tamanho onde o InsertionSort é mais rápido? Muitas das implementações mais rápidas de ordenação (por exemplo, a disponível no C++) utilizam essas abordagens híbridas combinando bons algoritmos.

Crie um programa com nome quicksort2.cpp. Nessa versão, modifique a parte recursiva do quicksort para que o InsertionSort seja chamado para ordenar os subarrays quando eles chegarem a tamanhos menores do que uma constante K . Teste sua implementação ordenando um array com 100 mil elementos e descubra um bom valor para K (fazendo alguns testes).

Etapa 2

Considere o código que você criou na etapa anterior. Meça o tempo de execução desse algoritmo para os arquivos entrada30k.txt e entrada30k2.txt. Ambos arquivos possuem 30 mil elementos.

30k = 11441

30k2 = 2819674

Essa diferença de performance ocorre porque nessa implementação (e também na original disponível em quicksort.cpp) sempre pegamos o primeiro elemento de cada subarray como pivô. (descubra o motivo dessa escolha ser ruim nesse caso de teste).

Se pegarmos, por exemplo, o elemento do meio de cada subarray esse problema seria eliminado **NESSSES CASOS DE TESTE**. Porém, seria possível fazer um caso de teste onde esse quicksort não funcionaria bem.

Uma ideia melhor seria pegar um elemento qualquer do subarray de forma aleatória. Ainda assim poderia haver um caso de teste onde o quicksort não funciona bem, mas isso seria muito raro de acontecer (além disso, seria difícil alguém intencionalmente criar uma situação onde isso ocorre). Crie uma cópia de quicksort2.cpp (da etapa anterior) em quicksort3.cpp. A seguir, modifique quicksort3.cpp para que ele escolha o pivô de forma aleatória. Teste seu programa e veja como o tempo de execução melhora com essa nova versão.

Etapa 3

Considere a implementação do MergeSort disponível em mergesort.cpp. Um problema dela é que em cada chamada ao método de partição é preciso alocar um array temporário e desalocá-lo. Crie um programa mergesort2.cpp, onde apenas um array é alocado no início do MergeSort (ele deverá ser reutilizado em toda chamada recursiva).

Etapa 4

Faça uma cópia do código da etapa anterior em mergesort3.cpp. A seguir, adapte o código para que ele seja não-recursivo.

Dicas:

1) A parte nova do seu código provavelmente terá em torno de 10 linhas (você poderá aproveitar as funções já implementadas em mergesort2.cpp).

2) Faça um código que que passe várias vezes pelo array juntando blocos (subarrays) de elementos já ordenados. Assim, no início seu array será composto por n subarrays de tamanho 1 já ordenados (onde n é o tamanho do array original). Passe por eles juntando (fazendo merge) pares de subarrays vizinhos (agora, você terá vários

subarrays de tamanho 2 ordenados). Passe novamente juntando subarrays de tamanho 2 (para ter subarrays de tamanho 4 ordenados). Repita isso até ter um único subarray ordenado (de tamanho igual ao do array original).

3) Faça vários rascunhos em papel antes de implementar isso!!!!

Etapa 5

Em C++ há uma implementação pronta de um algoritmo de ordenação (o algoritmo implementado depende do seu compilador, mas costuma ser um híbrido baseado no quicksort). No arquivo `cppsort.cpp` usamos essa implementação (disponível na biblioteca *algorithm*) para ordenar um array de forma similar à das etapas anteriores.

Compare o tempo de execução dos algoritmos (em microsegundos) insertion (InsertionSort) quick (quicksort original), quick2 (QuickSort com insertion sort), quick3 (quick2 com escolha aleatória de pivo), merge2 (MergeSort usando apenas uma alocação), merge3 (MergeSort iterativo e usando apenas uma alocação) e `cppsort` (sort do C++) para arquivos contendo diferentes quantidades de números.

Faça uma cópia deste documento e preencha a planilha abaixo ([envie um PDF desse google doc pelo submitty com nome roteiro.pdf](#)) com os tempos. Use o programa `geraEntradaAleat.cpp` para gerar arquivos de teste com diferentes quantidades de elementos (os N da tabela abaixo).

N	insertion	quick	quick2	quick3	merge2	merge3	cppsort
1000	964	635	149	147	178		185
10000	69720	2712	2138	2076	2183		2116
100000	6217397	31371	18765	17747	18107		18497
1000000	8045723	271148	223317	222184	217011		218771

O g++, por padrao, não otimiza o código compilado. Para ativar a otimização, a flag `-O3` pode ser utilizada (exemplo: `g++ -O3 insertion.cpp -o insertion.exe`). Com isso, o compilador tentará utilizar várias técnicas para otimizar seu código (a ordem de complexidade dele normalmente não é alterada) -- por exemplo, ele elimina algumas chamadas de funções, reorganiza o código para melhor eficiência de uso do processador (sem mudar resultados), evita algumas cópias de dados, etc. Recompile todos programas usando a flag `-O3` e meça os tempos novamente (coloque-os na planilha abaixo)

N	insertion	quick	quick2	quick3	merge2	merge 3	cppsort
1000	188	71	49	50	107		36
10000	15766	849	656	692	1151		465
100000	1117872	6941	5826	6145	9313		5781
1000000	175412637	81169	70455	72378	109717		68905

Quais conclusões você obteve a partir desses experimentos? (eles foram feitos de forma simplista e poderiam ser melhores, mas mesmo assim é possível obter algumas conclusões preliminares).

R: Métodos recursivos tendem a ser mais eficientes.

Algoritmos quadráticos(insertionsort) tendem a ser terríveis para entradas grandes.

Métodos de ordenação diferente quando usados juntos e de maneira correta tendem a ser mais eficientes(exemplo: quicksort com insertionsort).

Quais outros experimentos seriam interessantes de serem realizados para avaliar melhor o comportamento desses algoritmos?

R: Poderiam ser feitos testes com arrays já ordenados ou parcialmente ordenados a fim de verificar quais algoritmos se saem melhor nesses casos.

Casos em que o array seja notavelmente grande(acima de 1.000.000).

Dica: no bash do Linux você pode rodar todos os casos de teste, por exemplo, com quick3.exe usando o comando: “for f in entrada1000*; do echo \$f; ./quick3.exe < \$f; done” (é possível automatizar ainda mais...)

Submissao da aula pratica:

A solucao deve ser submetida ate as 18 horas da proxima Segunda-Feira utilizando o sistema submittty (submittty.dpi.ufv.br). Envie todos seus arquivos .cpp (e o PDF)