→ Sempre faça vários exemplos e diagramas em papel ←

INF 213 - Roteiro da Aula Prática

Arquivos disponibilizados para a prática:
https://drive.google.com/file/d/1ur2prtLcMiqoXeo4vx-Ct0oguIpmH6oY/view?usp=share_link

O programa geraEntrada.cpp recebe como argumento (via linha de comando) numérico N e, então, imprime (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.

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 (que pode ser uma constante global). 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.

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 **NESSES 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

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. As implementações merge2 e merge3 devem ser as que você fez na aula anterior.

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	1009	999	0	0	0	0	0
10000	50561	2175	1000	1011	1000	1659	2000
100000	4957405	18352	19164	18731	14512	13240	15079
1000000	- -	206475	216526	216912	153700	-	166517

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	merge3	cppsort
1000	1037	0	0	0	0	0	0
10000	9517	1000	0	996	996	0	0
100000	1086128	6640	6004	7592	6999	7336	5001
1000000	93001477	63127	65411	80998	77559	-	51508

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).

Pode-se perceber que o cppsort é o algoritmo mais rápido de ordenação no geral, o insertion sem a otimização não dá uma resposta em tempo hábil para 1bi de elementos e que o quicksort3 e mergesort2 tem tempo de execução bastante diferentes para os arquivos de maior quantidade de elementos, mas tem tempos parecidos para depois de otimizados.

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

Seria interessante testar com arquivos que estejam na ordem decrescente de ordenação, ou que já estejam ordenados e com arquivos com elementos repetidos.

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…)

Etapa 4

Na aula passada, vimos que uma forma de permitir que arrays com tipos quaisquer possam ser ordenados consiste em utilizar um template para esse tipo. Assim, podemos ordenar arrays de int, float, Pessoa, Aluno, etc. Porém, ainda temos uma dificuldade: a ordenação sempre é realizada com base no operador “<”. E se quisermos ordenar elementos com base em diferentes critérios? E se quisermos uma ordenação em ordem decrescente?

Por exemplo, podemos ter uma classe Pessoa com nome e CPF e fazer a ordenação por nome ou por CPF. Uma forma de fazer isso seria ter duas funções de ordenação: uma que ordena por nome e outra por CPF. O problema é que temos, novamente, código duplicado e “reinvenção de roda” (estamos implementando um algoritmo de ordenação novamente).

Uma forma melhor seria passar uma função de comparação customizável para o método de ordenação.

O desafio é: como passar uma função como parâmetro? Isso pode ser feito em C/C++, porém, uma alternativa que costuma ser mais simples é utilizar um functor, que é basicamente um objeto que se comporta como função.

Veja o arquivo functorExemplos.cpp . Estude esse exemplo e leia os comentários com atenção para entender um pouco sobre functors.

A seguir, veja o arquivo insertionSortTemplate2.cpp. Nele, o insertionSort foi adaptado para receber um functor como terceiro argumento (como o functor é uma classe que não conhecemos de antemão, precisamos utilizá-lo com tipo de template, ou seja, a função terá dois parâmetros genéricos: o tipo dos elementos armazenados no array e o tipo do functor de comparação).

Observe que, para facilitar o aprendizado, chamamos a variável do functor de "antesDe". Assim, o array será ordenado com base nesse functor. Dados dois objetos *a,b* no array, se antesDe(a,b) retornar true → *a* deverá ficar antes de *b* no array.

Note que a função pronta de ordenação sort() (do C++) também suporta functors! Para praticar isso, comente o trecho "insertionSort(pessoas,5, ComparadorNome());" e descomente "//sort(pessoas,pessoas+5,ComparadorNome()); ".

A sintaxe do sort é muito parecida com a sintaxe da nossa função com templates insertionSort: a diferença básica é que o primeiro argumento é um ponteiro para o início do array a ser ordenado e o segundo é um ponteiro para uma posição que está 1 elemento após o último (ou seja, é um intervalo semiaberto à direita). Assim, se formos ordenar os 10 primeiros elementos de um array v → os dois primeiros argumentos de sort() deverão ser *v* (ponteiro para o início do array) e *v+10* (ponteiro para posição 10 elementos depois da primeira, ou seja, a 11a posição).

O terceiro argumento do sort() é um functor (pode também ser uma função de comparação ou um lambda → isso será estudado posteriormente). Porém, esse argumento é opcional (se não for passado, o array será ordenado com o uso do operador <, que deverá ser implementado na classe sendo ordenada).

Dessa forma, na maior parte das vezes um programador pode simplesmente utilizar o sort para ordenar qualquer tipo usando qualquer critério de ordenação customizado (basta implementar um functor!). Isso facilita bastante a programação, reduz as chances de erros de implementação, deixa o código mais legível, etc.

Por fim, o exercício que você deverá entregar nesta etapa é o seguinte: modifique o arquivo insertionSortTemplate2.cpp, criando um functor comparador que, se utilizado em um método de ordenação, ordena as pessoas por CPF em ordem decrescente. Caso haja empate de CPF (isso não ocorre na vida real…), as pessoas devem estar ordenadas por nome (em ordem crescente). Use esse functor para ordenar o array de pessoas com o método sort() do C++ (chamando-o no lugar indicado, no final do programa).

Submissao da aula pratica:
A solucao deve ser submetida utilizando o sistema submitty (submitty.dpi.ufv.br). Envie todos seus arquivos .cpp (e o PDF)