Semana 2: Ordenação – Merge Sort e Quick Sort

Prof. Dr. Juliano Henrique Foleis

Estude com atenção os vídeos e as leituras sugeridas abaixo. Os exercícios servem para ajudar na fixação do conteúdo e foram escolhidos para complementar o material básico apresentado nos vídeos e nas leituras. Quando o exercício pede que crie ou modifique algum algoritmo, sugiro que implemente-o em linguagem C para ver funcionando na prática. O único exercício que é necessário entregar está descrito na Seção "Atividade Para Entregar".

Vídeos

Merge Sort (Ordenação Por Intercalação) - Parte 1: Intercalação

Merge Sort (Ordenação Por Intercalação) - Parte 2: Merge Sort

Merge Sort (Ordenação Por Intercalação) - Exemplo

Quick Sort (Ordenação Por Particionamento) - Parte 1: Particionamento

Quick Sort (Ordenação Por Particionamento) - Parte 2: Quick Sort

Leitura Sugerida

PEREIRA, Silvio Lago. Estruturas de Dados em C - Uma Abordagem Didática. [Minha Biblioteca]. Capítulo 8 (Ordenação e Busca), Seção 8.2.4 (Link)

FEOFILOFF, Paulo. Projeto de Algoritmos em C. Mergesort: Ordenação por Intercalação (Link) – (Pode pular a seção "Intercalação de vetores ordenados".)

FEOFILOFF, Paulo. Projeto de Algoritmos em C. Quicksort (Link)

SZWARCFITER, Jayme Luiz e MARKENZON, Lilian. Estruturas de Dados e Seus Algoritmos. [Minha Biblioteca]. Capítulo 7 (Algoritmos de Ordenação), Seções 7.4 e 7.5 - (Link) – (Não precisa se preocupar com as subseções de análise. Vamos estudar esse assunto em outro momento.)

Exercícios

Exercícios dos materiais de leitura sugerida

Exercícios 7.5 e 7.11 do livro de Szwarcfiter e Markenzon (Link)

Exercícios 1.1, 2.1, 2.2, 2.3, 2.6, 2.7, 3.1, 3.3, 3.4, 4.2, 5.2 da página do Prof. Feofiloff (Quicksort) (Link)

Exercícios Complementares

Merge Sort

1) O número de inversões de um vetor $v[0 \ldots n-1]$ é o número de pares ordenados (i, j) tais que $0 \leq i < j < n$ e $v[i] > v[j]$. Modifique o algoritmo Merge Sort (e Merge) para calcular o número de inversões em v.

2) O algoritmo Merge estudado é eficiente no sentido que faz a intercalação de dois vetores em tempo linear. Entretanto, o custo dele para intercalar dois vetores unitários é maior que simplesmente trocar os elementos caso não estejam em ordem. Modifique o algoritmo de Merge Sort (do vídeo) para que intercale dois vetores unitários usando uma troca simples caso necessário. Compare o tempo para ordenar 100000 elementos aleatórios usando o algoritmo original (do vídeo) e o algoritmo modificado.

3) Modifique o algoritmo Merge apresentado no vídeo para que a cópia dos subvetores de V para os vetores E e D seja feita usando *memcpy*. Compare o tempo para ordenar 100000 elementos aleatórios usando o algoritmo original e o algoritmo modificado.

4) Modifique o algoritmo Merge Sort e Merge de forma que os vetores E e D sejam pré-alocados no início da execução e seja reusado em todas as chamadas a Merge. Compare o tempo para ordenar 100000 elementos aleatórios usando o algoritmo original e o algoritmo modificado. **DICA:** Você pode criar uma função *mergesort(int *v, int n)* que funciona como um *wrapper* para a implementação de MergeSort (apresentada no vídeo). A idéia é que *mergesort* invoque MergeSort (ou seja um *wrapper*). Tanto a alocação, quando a liberação da memória pode ser feita nessa função. Os endereços dos vetores E e D pré-alocados devem ser passados como parâmetros para o algoritmo MergeSort e Merge, respectivamente.

5) Implemente uma versão iterativa do algoritmo Merge Sort.

Quick Sort

1) Algoritmos "simplórios", como o *Insertion Sort*, normalmente são mais rápidos para ordenar vetores pequenos do que algoritmos mais sofisticados, como o Quick Sort. Por essa razão é comum invocar *Insertion Sort* ao invés de uma chamada recursiva para *Quick Sort* quando o subvetor se torna menor que um certo M, que varia de 10 a 20. Implemente essa variação e experimente com M entre 10 e 20. Compare o tempo de execução dessa versão com a versão do *Quick Sort* discutida no vídeo. Avalie a ordenação com vetores de 100000 posições contendo números aleatórios.

2) Uma variação da abordagem anterior simplesmente não ordena os subvetores menores que M. Ao invés de chamar *Insertion Sort* para ordenar cada subvetor menor que M, somente uma chamada a *Insertion Sort* é realizada após a execução completa do *Quick Sort*. Compare o tempo de execução dessa versão com os resultados obtidos no exercício anterior. Avalie a ordenação com vetores de 100000 posições contendo número aleatórios.

3) Conforme discutido no vídeo, o tempo de execução do *Quick Sort* no caso médio e no melhor caso é $\Theta(n \lg n)$. Além disto, o custo de *partition*, é significativamente menor que *merge*, fazendo com que o Quick Sort seja mais rápido que o *Merge Sort*, outro algoritmo de ordenação eficiente. Entretanto, no pior caso, o desempenho do *Quick Sort* pode degenerar para $\Theta(n^2)$. O pior caso (que é a permutação inicial do vetor a ser ordenado que leva ao tempo $\Theta(N^2)$) varia de acordo com a forma que o pivô é escolhido. No vídeo eu discuti que a versão de *partition* implementada sempre usa o último elemento do vetor como pivô e que, nessa implementação, o pior caso ocorre quando o vetor de entrada está ordenado em ordem crescente ou decrescente.

a. Ordene o vetor $v = [1, 2, 3, 4, 5]$ usando as implementações de *Quick Sort* e *partition* apresentadas nos vídeos. Qual a peculiaridade que você notou?

b. ***QuickSort randomizado:*** Para evitar que o algoritmo caia no pior caso discutido no enunciado, uma modificação pode ser realizada na função *partition*. Ao invés de usar o elemento que já está na última posição do subvetor ($v[r]$) no início da execução de *partition*, um elemento aleatório do subvetor $v[p \dots r]$ é trocado com o elemento $v[r]$ antes da linha $x = v[r]$. Assim, mesmo que o subvetor esteja inicialmente ordenado, há uma chance significativa que um elemento intermediário seja escolhido como pivô, evitando o pior caso. Implemente essa modificação na função *partition* apresentada no vídeo. Compare o tempo de execução do Quick Sort com essa versão modificada do *partition* com a versão discutida no vídeo em: **i.** um vetor já ordenado de 50000 posições; e **ii.** um vetor aleatório com 50000 posições.

c. Embora a modificação apresentada no item **b** acima evite o pior caso do *Quick Sort*, o custo de gerar o número aleatório pode atrapalhar o desempenho do *Quick Sort* no caso médio. Uma abordagem alternativa bastante conhecida consiste em escolher o número mediano entre três elementos em posições fixas do subvetor. As posições comumente utilizadas são p, r e $(p + r)/2$. Esta abordagem é conhecida como *mediana de três*. Note que as três posições escolhidas não são aleatórias, uma vez que estamos evitando chamar a função de geração de números aleatórios. Implemente essa modificação na função *partition* apresentada no vídeo. Execute os mesmos testes realizados no experimento proposto no item **b** e compare os resultados.

d. Combine a estratégia apresentada no exercício **1** com o exercício **3b** e **3c** (*1 + 3b*, *1 + 3c*) e compare com os resultados obtidos nos exercícios **1**, **3b** e **3c**. O que você notou?

Atividade Para Entregar

A atividade a seguir é para ser feita individualmente e entregue via Moodle no tópico da Semana 2. A data-limite para entrega é dia 16/3/2021 às 23:55. Em caso de cópia as atividades dos participantes serão desconsideradas.

Descrição da Atividade

Nesta atividade vamos comparar o desempenho dos algoritmos de ordenação desenvolvidos até o momento. Em todos os experimentos abaixo utilize sua função *int* random_vector(int n, int max, int seed)* desenvolvida na atividade da semana passada para gerar os vetores aleatórios. Utilize *seed = 0*.

a) Seja N o número de elementos em um vetor. Gere vetores aleatórios com $N = 1000$, $N = 10000$, $N = 100000$ e $N = 500000$ elementos. O valor dos elementos deve ser entre 0 e N*100 (por exemplo, no vetor com $N = 1000$ elementos, os números devem ser sorteados entre 0 e 100000). Ordene esses vetores com os 5 algoritmos de ordenação estudados até o momento. Anote apenas o tempo de execução de cada algoritmo na tabela a seguir:

	1000	**10000**	**100000**	**500000**
BubbleSort				
InsertionSort				
SelectionSort				
MergeSort				
QuickSort				

b) Execute os mesmos testes que no item anterior, mas dessa vez, use vetores já ordenados como entrada. Anote apenas o tempo de execução de cada algoritmo em uma tabela como a apresentada acima.

c) De acordo com os resultados obtidos nos itens **a** e **b**, responda as perguntas a seguir:

i) Qual algoritmo você usaria em um vetor que está praticamente ordenado, ou seja, tem apenas alguns elementos fora do lugar? Por quê?

ii) Qual algoritmo você usaria para ordenar um vetor que está ordenado em ordem decrescente? Por quê? (vamos fazer de conta que essa é a melhor forma de ordenar um vetor ordenado em ordem decrescente)

iii) Qual algoritmo você usaria em um vetor que você não faz idéia sobre a ordem dos elementos? Por quê?

iv) Você usaria os algoritmos ineficientes em alguma situação? Se sim, qual deles você usaria? Em que situação?

Você deve Entregar

Entregue em formato .zip os arquivos a seguir:

- Um arquivo *pdf* com as tabelas dos itens **a** e **b** e as respostas do item **c**;
- Um arquivo .c com o programa principal utilizado para gerar os resultados no item **a** e **b**.

Por favor entregue como especificado acima!

BONS ESTUDOS!