Tópico 3: Ordenação Eficiente – Parte 3: Counting Sort e Radix Sort

Prof. Dr. Juliano Henrique Foleis

Estude com atenção os vídeos abaixo. Os exercícios servem para ajudar na fixação do conteúdo e foram escolhidos para complementar o material básico apresentado nos vídeos. Quando o exercício pede que crie ou modifique algum algoritmo, sugiro que implemente-o em linguagem C para ver funcionando na prática.

Vídeos

Counting Sort (Ordenação Por Contagem)

Radix Sort (Ordenação por Dígitos)

Exercícios

1. Cronometragem do CountingSort e RadixSort. Neste exercício serão ordenados vetores da seguinte estrutura:

```
typedef struct Info {
    int chave;
    int dado;
} Info;
```

a. Escreva uma função *Info* random_info_vector(int n, int max, int seed)* que retorne um vetor com n elementos com chaves e dados aleatórios entre *0* e *max*. Seed é a semente usada para iniciar o gerador de números aleatórios.

b. Execute CountingSort e Radixsort em um vetor aleatório gerado com a função *Info* random_info_vector(int n, int max, int seed)* com n = *1000, 10000, 100000, 500000 e 1000000*, *max* = n * *100* e *seed = 42*. Anote o tempo de execução para cada n. Ordene os mesmos vetores com MergeSort, QuickSort e HeapSort e compare os tempos de execução.

2. Altere o CountingSort apresentado no vídeo para operar ordenar vetores que também contenham chaves negativas. **DICA:** você pode transformar as chaves todas em inteiros positivos! Só não esqueça de voltar as chaves para os valores corretos após a ordenação!

3. No vídeo do Counting Sort (em 9:34) eu mencionei que se o vetor sendo ordenado fosse apenas um vetor de inteiros, a ordenação já estaria concluída naquele momento. Altere o algoritmo apresentado no vídeo para ordenar um vetor de inteiros sem que seja necessário computar a soma de prefixos. Compare o tempo desta versão do Counting Sort com RadixSort, MergeSort, QuickSort e RadixSort para ordenar vetores aleatórios gerados usando *int* random_vector(int n, int max, int seed)* com n = *1000, 10000, 100000, 500000, 1000000*, *max* = n * *100* e *seed = 0*.

4. Internamente os computadores representam números inteiros em formato binário. Em outras palavras, os números são armazenados em base 2. Por causa disso, a divisão por potências de 2 pode ser realizada com instruções de deslocamento de bits (*bit shifting*). Desta forma, a fórmula de obteção dos dígitos pode

	Radix Base 2	**Radix Base 10**
$n = 10^3$		
$n = 10^4$		
$n = 10^5$		
$n = 5 \cdot 10^5$		
$n = 10^6$		

Table 1: Resultados

ser reescrita em C na base 2 como: ***(N >> pos) & 1***, tal que ***N*** é o número, ***pos*** é a posição do bit a ser copiado (0 é a posição do bit menos significativo, à direita) e ***&*** é a operação binária AND.

a. Na maioria das arquiteturas de computadores modernas, as instruções de deslocamento de bits são muito mais rápidas que as instruções de divisão. Para verificar se a ordenação na base 2 pode ser mais rápida que na base 10 no Radix Sort, reimplemente o Radix Sort em base 2 utilizando essa nova função para obter os dígitos durante o Counting Sort. Use inteiros de 32-bits para armazenar cada elemento.

b. Compare o tempo de execução do Radix Sort na base 2 e do Radix Sort na base 10 ordenando vetores aleatórios gerados usando *int* random_vector(int n, int max, int seed)* com *n = 1000, 10000, 100000, 500000, 1000000*, *max = n * 100* e *seed = 0*. Anote os resultados na Tabela abaixo em milisegundos (*ms*).

c. Analisando os resultados obtidos, qual versão foi mais rápida? Por quê?

BONS ESTUDOS!