ACH2002 – Introdução a Análise de Algoritmos – Profa Ariane Machado Lima

Exercício Programa 2 – INDIVIDUAL - Prazo de entrega: 17/11/2024 às 23:59h

Implementação e comparação empírica de algoritmos de ordenação - v.3

Objetivo: Implementar e comparar o desempenho de sete algoritmos de ordenação: Insertion Sort, Selection Sort, Bubble Sort melhorado, Shell Sort, Merge Sort, Heap Sort e Quick Sort aleatório. A comparação deve incluir medições de tempo de execução, número de comparações e número de movimentações realizadas em conjuntos de dados de tamanhos diferentes.

Instruções:

1. **Implementação**: Você deve implementar os sete algoritmos sem o uso de bibliotecas externas que realizem a ordenação e sem o uso do ChatGPT ou similares. Cada algoritmo deve ser implementado em uma função separada. Cada algoritmo deve ordenar um vetor de tamanho ***n*** (sendo *n* um parâmetro do algoritmo) de estruturas do tipo ***Registro***. A ordenação deve ser realizada de forma crescente pelo campo ***chave*** dessa estrutura.
2. **Entrada**: Serão fornecidos nove arquivos de dados com diferentes tamanhos, contendo na primeira linha o número de elementos da segunda linha, e na segunda linha uma lista de números inteiros, separados por espaço, que serão chaves de uma estrutura contendo, adicionalmente, um array (`campoDaEstrutura`) de determinado tamanho cujo conteúdo não importa (nem precisa inicializá-lo). Ex:

```
typedef struct {
      int chave;
      int campoDaEstrutura[1];
} Registro1;
typedef Registro1 Registro;
```

   Logo, para cada arquivo de entrada, seu programa deve:

   i. criar dinamicamente o vetor de tamanho n contendo essas estruturas, com as chaves lidas do arquivo, na ordem em que foram lidas:

```
Registro* vetor;
```

   ii. ordenar esse vetor (pelo campo chave) usando cada um dos algoritmos;

   iii. repetir os passos i) e ii) para tamanho = 1000 do vetor `campoDaEstrutura`:

```
typedef struct {
   int chave;
   int campoDaEstrutura[1000];
} Registro1000;
typedef Registro1000 Registro;
```

Cada um dos arquivos de entrada terá um número de chaves diferente (100, 1.000 e 10.000), podendo elas já estar ordenadas crescentemente, decrescentemente ou em ordem aleatória (totalizando os nove arquivos de entrada).

3. **Análise de Desempenho**: você deve medir o tempo de execução de cada algoritmo para os nove conjuntos de dados fornecidos, para os tamanhos 1 e 1000 do vetor `campoDaEstrutura`, totalizando portanto 18 execuções de cada algoritmo de ordenação. Em cada execução você deve contar o número de comparações e o número de movimentações realizadas. Para isso, você deve usar um contador no código que registra cada comparação de elementos e outro para cada movimentação de estrutura. A medição de tempo deve ser realizada com alguma rotina ou comando que conte o tempo de CPU. Todos os testes devem ser executados em uma mesma máquina, e sem nenhum outro processo estritamente necessário executando em paralelo (fechar todas as demais janelas, desligar antivírus, etc).

   Exemplo utilizando a função clock[1]:

```
#include <time.h>
void testaAlg1(/* parâmetros */)
{
    clock_t hora_inicio, hora_fim;
    hora_inicio = clock();
    /* código do algoritmo a ter seu tempo de execução medido */
    hora_fim = clock();
    double tempo = (hora_fim - hora_inicio)  / CLOCKS_PER_SEC;
 }
```

4. **O que deve ser entregue:** O trabalho a ser entregue é um relatório em formato PDF contendo:

   i. cabeçalho contendo o nome e o nr USP do aluno;

   ii. arquitetura da máquina utilizada (processador, memória RAM);

   iii. 12 gráficos e 12 tabelas comparando os tempos de execução, o número de comparações e o número de movimentações para cada conjunto de dados (listados a seguir);

   iv. uma discussão explicando/relacionando todos os gráficos.

   Os gráficos (de barras, uma barra para cada algoritmo de ordenação) a serem gerados são:

   a) 3 gráficos de tempo de execução (eixo y) por nr de chaves (“eixo x”: valores 100, 1.000 e 10.000): um para chaves em ordem ascendente, outro para chaves em ordem descendente, outro para chaves em ordem aleatória;

1 https://petbcc.ufscar.br/time/

b) 3 gráficos de tempo de execução (eixo y) por tamanho de `campoDaEstrutura` (“eixo x”: valores 1 e 1.000): um para chaves em ordem ascendente, outro para chaves em ordem descendente, outro para chaves em ordem aleatória;

c) 3 gráficos de número de comparações (eixo y) por nr de chaves (“eixo x”: valores 100, 1.000 e 10.000): um para chaves em ordem ascendente, outro para chaves em ordem descendente, outro para chaves em ordem aleatória;

d) 3 gráficos de número de movimentações (eixo y) por nr de chaves (“eixo x”: valores 100, 1.000 e 10.000): um para chaves em ordem ascendente, outro para chaves em ordem descendente, outro para chaves em ordem aleatória;

Juntamente com cada gráfico, a tabela correspondente deve ser também incorporada ao relatório (uma linha para cada algoritmo e uma coluna para cada valor do que foi variado (número de chaves ou tamanho de `campoDaEstrutura`), contendo em cada célula o tempo de execução, número de movimentações ou número de comparações, a depender da tabela.

Utilize como unidade de tempo a que for mais adequada para que os tempos não fiquem zerados (ex: mseg).

Observações:

- **Sobre as contagens de movimentações.** Lembrem-se que, na vida real, normalmente não ordenamos números, mas sim ESTRUTURAS (no nosso caso do tipo `Registro`) pelo campo chave. Logo, nos algoritmos vistos, sempre que você precisa trocar de lugar dois números no vetor sendo ordenado, na vida real movimentamos a estrutura inteira. Em primeiro lugar, é isso que você deve fazer neste EP, para poder ver a diferença de tempo na cópia de estruturas de diferentes tamanhos. Em segundo lugar, sempre que uma cópia de uma estrutura for feita (comando de atribuição), você deve incrementar o contador de movimentações. Por exemplo, quando no vetor você troca de lugar duas estruturas, isso envolve a cópia de uma delas para uma variável (do tipo `Registro`) auxiliar, logo há três incrementos no número de movimentações.
- **BubbleSort modificado.** Quando fala-se aqui em BubbleSort modificado, quer-se dizer uma versão que precisa ser modificada em relação à que está nos slides de aula. Essa modificação significa, a cada iteração (do laço mais externo) de passar o maior elemento para o fim do vetor, verificar quantas trocas foram realizadas naquela iteração. Se nenhuma troca foi realizada, o algoritmo pode encerrar, pois o vetor já ficou ordenado.
- **InsertionSort com sentinela.** Na hora de implementar o InsertionSort, recomenda-se a versão vista em aula com sentinela.
- **Pequenas alterações ou adições.** Se você desejar acrescentar testes adicionais (ex: QuickSort Ziviani, QuickSort Cormen não aleatório, QuickSort Cormen aleatório) pode fazer, e conta ponto que pode melhorar a nota. Mas não esqueça de mencionar isso no relatório. Por isso eu recomendo que, antes da apresentação dos resultados, você escreva uma seção denominada “Métodos” em que você descreverá quais métodos você implementou a mais ou de forma diferente do que exposto em aula.