Universidade Federal do Rio Grande do Sul
Instituto de Informática
INF01124 - Classificação e Pesquisa de Dados - Turma B
Prof. Renan Maffei

Entrega: 23/09/2017

Entregue via Moodle um arquivo compactado com os códigos desenvolvidos e com um relatório contendo as respostas das questões (ex. plots ou análises, quando requisitados).

1. Complexidade de Funções

Usando propriedades matemáticas ou comparando as funções experimentalmente através de um gráfico, ordene as funções abaixo em ordem crescente de complexidade:

- $n,\ n^2,\ \lg n,\ n!,\ n^3,\ \sqrt{n},\ \lg^2 n,\ e^n,\ \sqrt{\lg n},\ \ln\ln n$

OBS: $\lg n = \log_2 n$ e $\ln n = \log_e n$

Em anexo fornecemos um exemplo de código em python usando a biblioteca matplotlib para plotar gráficos. Para rodar digite em um terminal `python plotResults.py`

2. BubbleSort e InsertionSort

2.1. Implemente os métodos BubbleSort e InsertionSort, conforme vistos em aula.

Para auxiliar nessa tarefa, é fornecido um pequeno framework de exemplo em C, organizado da seguinte forma:

- `main.c` - cria os arrays a serem ordenados, faz as chamadas dos métodos de ordenação e mede o tempo de cada execução
- `utils.c` e `utils.h` - métodos auxiliares utilizados para geração de instancias de arrays, e medição de tempo
- `sorting.c` e `sorting.h` - **local onde deverão ser feitas as implementações dos métodos de ordenação**

Os cabeçalhos dos dois métodos são:

```
int bubbleSort(int* C, int n);
int insertionSort(int* C, int n);
```

Cada método deve retornar um inteiro indicando o número de trocas feitas durante o processo de ordenação.

Adicionalmente, deverá ser implementada a variação do método InsertionSort usando busca binária:

```
int binaryInsertionSort(int* C, int n);
```

Para facilitar, utilize a função `binarySearch` já implementada em `sorting.c`.

(Note que o número de trocas obtido na execução do `binaryInsertionSort` deve ser o mesmo obtido na execução do `insertionSort`, visto que a otimização feita com a busca binária reduz a quantidade de comparações, mas não a quantidade de trocas.)

2.2. Teste os métodos implementados com diferentes entradas

Varie o tamanho do array a ser ordenado, começando em 1.000 e dobrando até chegar em 128.000 (ou seja: 1.000, 2.000, 4.000, ..., 64.000, 128.000)

Compare os algoritmos com três diferentes tipos de configurações de entrada:

- *ordem correta* - usando a função `createSortedArray` para gerar o array
- *ordem reversa* - usando a função `createReversedArray` para gerar o array
- *ordem aleatória* - usando a função `createRandomArray` para gerar o array

Faça uma tabela com a quantidade de trocas executadas por cada algoritmo nos tamanhos de 1.000, 8.000 e 64.000 em cada uma das configurações.

Plote um gráfico para cada um dos três diferentes tipos de configurações, descrevendo o tempo gasto por cada algoritmo em função do tamanho da entrada (ou seja, um plot para ordem correta, um para ordem reversa e um para ordem aleatória).

Dica: redirecione a saída do programa para um arquivo (ex: `./program > log.txt`) e modifique o programa em python anexado para plotar os dados do arquivo gerado.

3. ShellSort

3.1. Nesta questão iremos comparar implementações do ShellSort que usam diferentes sequências de pulos:

- (Shell,1959): h = 1, 2, 4, 8, 16, 32, ...
- (Knuth,1971): h = 1, 4, 13, 40, 121, 364, ...
- (Tokuda,1992): h = 1, 4, 9, 20, 46, 103, ...

No framework de exemplo, em anexo, já está implementada uma função chamada `generateGapSequence` que gera essas três sequências. Também está implementada a função ShellSort que para cada valor h da sequência escolhida, invoca a inserção direta pulando h passos. Cabe nesse exercício, implementar essa última função:

```
int insertionShellSort(int* C, int n, int h, int f);
```

3.2. Para arrays de tamanho 8.000 e 64.000 com números gerados aleatoriamente, ordene usando ShellSort cada uma das sequências.

Informe para cada um dos testes, as seguintes informações:

- Quantas iterações houveram sobre o array, isto é, quantos valores diferentes de h foram usados em cada caso?
- Qual o total de trocas em cada iteração, isto é, com cada valor diferente de h?
- Qual o total de trocas final?
- Qual o tempo de processamento final?

3.3. Compare o tempo de execução das três variações do ShellSort, juntamente com a melhor técnica observada no exercício anterior.

Varie o tamanho do array a ser ordenado, começando em 1.000 e dobrando até chegar em 128.000 (ou seja: 1.000, 2.000, 4.000, ..., 64.000, 128.000). Plote um gráfico descrevendo o tempo gasto por cada algoritmo em função do tamanho da entrada.

Depois repita o teste só entre as sequencias de (Knuth,1971) e (Tokuda,1992) para tamanhos começando em 1.000 e dobrando até chegar em 8.192.000, e plote o gráfico. Em qual intervalo de valores a sequência de Knuth é mais eficiente do que a sequência de Tokuda?

4. Desafio Bonus - 25% extra

Resolva o problema "DNA Sorting"

https://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=8&page=show_problem&problem=553

Este é um problema de maratona de programação. O problema será considerado completo se for aceito no site da https://uva.onlinejudge.org