Técnicas de Busca e Ordenação
Roteiro de Laboratório 2 – Métodos Elementares de Ordenação

1 Introdução

Para a nossa primeira incursão na área de algoritmos de ordenação, vamos estudar vários métodos elementares que são apropriados para uma pequena quantidade de dados, ou quando os dados seguem uma estrutura especial. Há várias razões para estudarmos os algoritmos simples de ordenação em detalhes. Primeiro, eles fornecem um contexto aonde podemos aprender a terminologia e os mecanismos básicos de algoritmos de ordenação, permitindo desenvolver uma base adequada para o estudo dos algoritmos mais avançados. Segundo, os métodos simples são perfeitamente adequados para uma série de aplicações de ordenação. Terceiro, vários dos métodos simples podem ser modificados para gerar métodos mais elaborados ou podem ser empregados como rotinas auxiliares em outros algoritmos de ordenação.

Neste laboratório vamos trabalhar com quatro métodos elementares de ordenação: *selection sort*, *insertion sort*, *bubble sort* e *shaker sort*. Via de regra, todos esses métodos levam tempo proporcional a N^2 para ordenar N itens organizados em uma sequência aleatória. Se N é pequeno, esse tempo de execução pode ser perfeitamente adequado. No entanto, quando N cresce, é necessário empregar outros métodos de ordenação mais eficientes que serão estudados adiante.

2 As regras do jogo

Antes de considerar algoritmos específicos, é útil discutir a terminologia geral e as suposições básicas para ordenação. Vamos considerar métodos para ordenar *coleções* (arquivos) de *itens* contendo *chaves*. Todos esses conceitos são abstrações naturais em ambientes de programação modernos. As chaves, que são apenas uma parte (geralmente pequena) dos itens, são usadas para controlar a ordenação. O objetivo da ordenação é rearranjar os itens de forma que as suas chaves fiquem ordenadas segundo alguma regra bem definida (geralmente ordem numérica ou alfabética). Caraterísticas específicas das chaves e dos itens podem variar dramaticamente entre aplicações. Na maioria dos casos vamos abstrair da maioria dessas características.

Se o arquivo a ser ordenado cabe na memória primária, então o método de ordenação é chamado *interno*. Por outro lado, ordenação em memória secundária (disco) é chamada ordenação *externa*. O foco do nosso curso é ordenação interna. Os dados a serem ordenados podem estar armazenados na memória em *arrays* e listas encadeadas. Neste laboratório vamos considerar somente o primeiro caso.

Para desacoplar a implementação dos algoritmos de ordenação da estrutura dos itens a serem ordenados, é conveniente criar uma interface padronizada com as operações sobre itens necessárias para os algoritmos. Fazendo uma simplificação usual de que os itens são números inteiros, podemos ter uma interface como abaixo.

```
typedef int Item;
#define key(A)          (A)
#define less(A, B)      (key(A) < key(B))
#define exch(A, B)      { Item t = A; A = B; B = t; }
#define compexch(A, B)  if (less(B, A)) exch(A, B)
```

Exercício 1 – Interface de itens. Crie um arquivo `item.h` e coloque o código acima nele. **Não esqueça o guarda de inclusão!** Todos os seus programas de ordenação devem usar essa interface.

Exercício 2 – Cliente dos algoritmos de ordenação. Crie um programa cliente que utilizará os algoritmos de ordenação que serão desenvolvidos. O seu cliente deve realizar os seguintes passos:

1. Receber como um parâmetro o número N de itens a serem ordenados.
2. Alocar dinamicamente um *array* para guardar os N itens na memória.
3. Ler os N itens (no caso aqui, inteiros) de `stdin` para o *array*.
4. Executar um algoritmo de ordenação e medir o seu tempo, como já feito em laboratórios anteriores.
5. Exibir o *array* ordenado em `stdout`.
6. Liberar a memória do *array*.

Com relação ao passo 4 acima, para podermos utilizar o cliente sem modificação para todos os métodos de ordenação, declare o seguinte cabeçalho de função antes da função `main()`:

```
extern void sort(Item *a, int lo, int hi);
```

Todos os algoritmos devem implementar a sua função de ordenação segundo esse cabeçalho. A ideia é que o programa ordene todas as posições do *array* entre `lo` e `hi`, inclusive. Embora no momento não seja necessário o uso de `lo` (vamos passar sempre 0), em implementações recursivas de métodos de ordenação esse parâmetro será útil.

3 *Selection sort*

Um dos métodos mais simples de ordenação funciona como a seguir. Primeiro, encontre o menor elemento do *array*, trocando esse elemento pelo elemento da primeira posição. A seguir, encontre o segundo menor elemento e troque-o com o elemento da segunda posição. Continue dessa forma até que todo o *array* esteja ordenado. Esse método é chamado de *selection sort*.

Exercício 3 – Implementar *selection sort*. Implemente e teste a sua versão do *selection sort* em C. Você pode ver uma animação do algoritmo, por exemplo, em `https://www.toptal.com/developers/sorting-algorithms/selection-sort`. Coloque o seu código, por exemplo, em um arquivo `select_sort.c` e compile-o juntamente como seu cliente para gerar o executável.

***Observação importante*: existem milhares de implementações desses algoritmos básicos na Internet. Evite copiar um código pronto, tente fazer o seu.**

4 *Insertion sort*

Um método muito utilizado pelas pessoas para ordenar uma mão em um jogo de cartas é considerar uma carta de cada vez, inserindo-a no seu local correto, mantendo a ordenação das cartas que já foram analisadas. Em uma implementação no computador, é necessário criar um espaço para o elemento sendo inserido, movendo os elementos maiores uma posição para a direita. Esse tipo de método de ordenação é chamado de *insertion sort*.

Exercício 4 – Implementar *insertion sort*. Implemente e teste a sua versão do *insertion sort* em C. Você pode ver uma animação do algoritmo, por exemplo, em `https://www.toptal.com/developers/sorting-algorithms/insertion-sort`. Coloque o seu código, por exemplo, em um arquivo `insert_sort.c` e compile-o juntamente como seu cliente para gerar o executável.

5 *Bubble sort*

O *bubble sort* é geralmente o primeiro método de ordenação que as pessoas aprendem, devido a sua simplicidade: fique percorrendo o *array*, trocando os elementos adjacentes que estão fora de ordem, até que toda a sequência esteja ordenada. *Bubble sort* em geral é mais lento que os outros métodos, mas vamos considerá-lo aqui por questões de completude.

Exercício 5 – Implementar *bubble sort*. Implemente e teste a sua versão do *bubble sort* em C. Você pode ver uma animação do algoritmo, por exemplo, em `https://www.toptal.com/developers/sorting-algorithms/bubble-sort`. Coloque o seu código, por exemplo, em um arquivo `bubble_sort.c` e compile-o juntamente como seu cliente para gerar o executável.

6 *Shaker sort*

O *shaker sort* também é chamado de *bubble sort* bidirecional, porque o *array* é percorrido nas duas direções a cada passada. Em alguns casos, o *shaker sort* é mais rápido que o *bubble sort*, em particular por fazer melhor uso da localidade de *cache*.

Exercício 6 – Implementar *shaker sort*. Implemente e teste a sua versão do *shaker sort* em C. Você pode ver uma animação do algoritmo, por exemplo, em `http://www.programming-algorithms.net/article/40270/Shaker-sort#`. Coloque o seu código, por exemplo, em um arquivo `shaker_sort.c` e compile-o juntamente como seu cliente para gerar o executável.

7 Entradas de teste

"Qual é o melhor algoritmo de ordenação?" Essa pergunta já foi feita inúmeras vezes, e a melhor resposta é sempre: "Depende do que você está ordenando!" Essa dependência ocorre não somente com relação ao tamanho da entrada, mas também com relação ao quanto a entrada já está inicialmente ordenada. Por exemplo, *insertion sort* é quadrático no pior caso, mas é muito rápido para entradas que já estão quase ordenadas. A escolha do melhor algoritmo de ordenação requer não só o conhecimento do desempenho relativo dos algoritmos, bem como o conhecimento das características dos arquivos que se quer ordenar.

Qual seria, então, bons casos de teste? Muitos dos dados no "mundo real" já estão parcialmente ordenados, então os testes devem incluir esses casos. Dados com uma ordenação inversa também são importantes, pois levam ao desempenho de pior caso dos algoritmos. O arquivo com as entradas de teste disponibilizado no AVA está dividido da seguinte forma:

- **Chaves aleatórias** (diretório `in/unif_rand`): sequências de inteiros gerados aleatoriamente segundo uma distribuição uniforme de probabilidade.
- **Chaves ordenadas** (diretório `in/sorted`): sequências de inteiros que já estão completamente ordenadas.

- **Chaves ordenadas ao contrário** (diretório `in/reverse_sorted`): sequências de inteiros que já estão completamente ordenadas mas na sequência inversa.
- **Chaves quase ordenadas** (diretório `in/nearly_sorted`): sequências de inteiros que já estão quase ordenadas, salvo algumas posições.

Exercício 7 – Análise empírica. Para cada um dos grupos de testes descritos acima, execute as entradas com tamanho 1K, 10K e 100K para todos os quatro algoritmos de ordenação implementados. Crie quatro tabelas para agregar os resultados de tempo medidos. Quais conclusões você consegue tirar desses resultados?