Notas de Aula - AED1 – Algoritmos de Ordenação (parte 1)
Prof. Jefferson T. Oliva

A ordenação é uma das operações mais conhecidas dos sistemas de programação e tem como objetivo tornar simples e rápido o acesso e a recuperação de uma determinada informação a partir de um grande conjunto de informações. Na ordenação, cada elemento de uma sequência é posicionado de forma que Xi seja menor que o seu sucessor e assim por diante.

Um exemplo de como a ordenação é importante é a lista telefônica (provavelmente, vcs nunca usaram isso): onde é mais fácil encontrar um contato quando os nomes são mantidos ordenados. Imaginem o quão seria difícil encontrar um número de telefone caso a lista não fossem ordenado.

Terminologia básica:
- Arquivo de tamanho n é uma sequência de n itens (X1, X2, …, Xn)
- O i-ésimo componente do arquivo é chamado de item. Se voltarmos para o exemplo da lista telefônica, poderíamos observar que para cada item, há um sobrenome, nome, número de telefone e endereço
- Uma chave é associada a cada registro. Qual seria a chave para cada item em um registro telefone?
- Ordenação pela chave

Mais sobre terminologia básica:
- Ordenação interna: os dados estão na memória principal
- Ordenação externa: os dados estão em um meio auxiliar
- Ordenação estável: um algoritmo de ordenação é dito estável, se ele preserva a ordem relativa original dos registros com mesmo valor de chave. Lembram do conceito de chave primária e secundária vistas nas aulas anteriores?
- Ordenação pode ocorrer sobre os próprios registros ou sobre uma tabela auxiliar de ponteiros

Ver figura do slide 4

Devido à relação entre a ordenação e a busca, surge uma pergunta: o aquivo deve ser classificado ou não? Caso a operação de busca (recuperação de dados) seja realizada com frequência, é recomendada ordenação de dados.

Para medir a eficiência dos algoritmos de ordenação, consideramos o tempo para a execução do método e espaço de memória.

O tempo gasto pelo algoritmo de ordenação é comumente medido pelo número de operações críticas, como a comparação de chaves, em que é verificado se um registro é maior ou menor que o outro.

Outra operação crítica considerada na avaliação de algoritmos de ordenação é o movimento de registros, em que, caso a comparação de chaves seja positiva, é realizada a troca de posição entre os elementos comparados.

Em outras palavras, o desempenho de um algoritmo de ordenação é medido geralmente em função do tamanho do conjunto no pior caso, ou seja é considerada a notação O.

Por mais que nessa e nas próximas aulas serão apresentados diferentes tipos de algoritmos de ordenação, deve ser enfatizado que não existe um algoritmo que seja melhor que todos os outros, pois isso varia de acordo com diversos fatores, como o tamanho do conjunto de dados, pré-ordenação, repetição de chaves, entre outros.

Existem vários tipos de métodos de ordenação (exemplo: troca, seleção, inserção), mas não existe um método de ordenação considerado superior aos outros.

Nessa aula serão abordados apenas algoritmos de ordenação interna por troca e por seleção.

## Ordenação por Troca

Em cada comparação pode haver troca de posições entre itens comparados. Caso os itens comparados não estejam posicionados na ordem correta, ambos são trocados de posição.

Os algoritmos bubble sort e quick sort são dois dos algoritmos de ordenação por troca mais conhecidos.

## Bubblesort

O algoritmo percorre o arquivo várias vezes, onde cada elemento é comparado com o seu sucessor. Caso as posições dos elementos estejam no lugar errado, são trocados de posição. Assim, a movimentação dos itens lembra bolhas.

o Bubble sort é um dos algoritmos de ordenação mais simples, pois é de fácil compreensão e implementação, mas é um dos métodos menos ineficientes, pois pode fazer até $n^2$ trocas. Se para 10 registros, podem ocorrer na ordem de 100 trocas, Imagine fazer ordenação em de um arquivo grande, com aproximadamente 1 milhão de registros.

Implementação

```
void bubblesort(int v[], int n){
        int i, j, x;

        for (i = 0; i < n - 1; i++)
                for (j = 0; j < n - i - 1; j++)
                        if (v[j] > v[j + 1]){
                                x = v[j];
                                v[j] = v[j + 1];
                                v[j + 1] = x;
                        }
}
```

Implementação com melhoria:

```
void bubblesort(int v[], int n){
        int i, j, x, troca = 1;

        for (i = 0; i < n - 1; i++){
                troca = 0;

                for (j = 0; j < n - i - 1; j++)
                        if (v[j] > v[j + 1]){
                                x = v[j];
                                v[j] = v[j + 1];
                                v[j + 1] = x;
                                troca = 1;
                        }
        }
}
```

**Ver o restante dos slides referentes ao bubblesort.**

**Quicksort**

Outro algoritmo de ordenação por troca bastante conhecido é o quicksort, que é baseado na estratégia de divisão e conquista:
- na parte da divisão, o conjunto X[p...q] é dividido em dois subconjuntos $X[p ... r - 1]$ e $X[r + 1 ... q]$, tais que $X[p ... r - 1] <= X[r] <= X[r + 1 ... q]$
- A conquista ocorre a partir da ordenação dos dois subconjuntos por meio de chamadas recursivas do algoritmo de ordenação

Procedimento quicksort(X, p, q)
1 - definir o pivô r e as posições i = p e j = q
2 - enquanto i ≤ j, trocar de posição os elementos maiores (lado esquerdo do arranjo) com os itens menores (lado direito) que o pivô
3 - quicksort(X, p, j)
4 - quicksort(X, i, q)

Implementação:

```
void quicksort(int x[], int esq, int dir){
        int i = esq, j = dir, pivo = x[(i + j) / 2], aux;

        do{
                while (x[i] < pivo)
                        i++;
                while (x[j] > pivo)
                        j--;
                if (i <= j){
                        aux = x[i];
                        x[i] = x[j];
                        x[j] = aux;
                        i++;
                        j--;
                }
        }while (i <= j);
        if (j > esq)
                quicksort(x, esq, j);
        if (i < dir)
                quicksort(x, i, dir);
}
```

Neste código, o valor do pivô é definido pela posição do meio entre i e j. Em seguida é executado o laço do-while, enquanto i for menor igual a j

**Ver slides de 20 a 22**

O algoritmo quicksort possui a complexidade na ordem de n^2 no pior caso, mas no melhor e no caso médio, a complexidade é $n \log_{2} n$.

O pior caso de particionamento ocorre quando o elemento pivô divide a lista de forma desbalanceada, ou seja, divide a lista em duas sub listas: uma com tamanho 1 e outra com tamanho n - 1 (no qual n se refere ao tamanho da lista original).

**Referências**

Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. Introduction to Algorithms. Third edition, The MIT Press, 2009.

Horowitz, E., Sahni, S. Rajasekaran, S. Computer Algorithms. Computer Science Press, 1998.

Rosa, J. L. G. Métodos de Busca. SCE-181 - Introdução à Ciência da Computação II. Slides. Ciência de Computação. ICMC/USP, 2018.

Ziviani, N. Projeto de Algoritmos - com implementações em Java e C++. Thomson, 2007.