

Insertion e Merge Sort

Frank Coelho de Alcantara

Insertion Sort

Todos os principais livros de algoritmos começam a descrição do algoritmo Insertion Sort pelo jogo de Bridge. No Brasil, jogamos Buraco. No jogo de Buraco você recebe 11 cartas viradas para baixo. Pega a cartas uma a uma e vai *inserindo* em ordem na sua mão. Primeiro por naipe e depois por valor. Este *inserindo* da sentença a anterior, força um pouco a amizade mas, cai como uma luva para metáfora do funcionamento do **Insertion Sort**. Agora, depois deste parágrafo, eu vou ali arder no mármore do inferno e já volto.

Este processo que usamos naturalmente para ordenar as cartas de baralho, seja em que jogo for, é intuitivo e descreve muito bem os conceitos que suportam o **Insertion Sort**. Você pega a primeira carta e coloque onde colocar, ela estará na posição certa. A partir da segunda, já é necessário observar a posição da carta e, a partir da terceira, pode ser que a posição correta seja entre as cartas que já estão na sua mão. O único algoritmo de ordenação mais simples que este é o **Selection Sort** Que deve ser estudado em cursos e livros básicos de algoritmos e estruturas de dados e está fora das nossas ambições.

O **Insertion Sort** interage por uma lista de itens e cada um destes itens é inserido em uma lista de itens na posição correta.

Este é um daqueles algoritmos que aprendemos apenas para fixar o conceito.

Use o **Insertion Sort** apenas para conjuntos de dados pequenos e com baixa entropia.

Do ponto de vista do algoritmo, começamos por assumir que a primeira posição do conjunto de dados representa uma lista de elementos que já está ordenada. Em cada passagem do algoritmo percorremos do item 1 até o item $n - 1$, o item da passagem será comparado com aqueles que já estão ordenados, na primeira passagem, apenas o primeiro elemento está ordenado. Na primeira passagem vamos comparar o segundo item com a lista já ordenada, que é composta de apenas um item e inserir este segundo item no lugar correto.

Na terceira passagem vamos comparar o terceiro item com os dois primeiros e inseri-lo no lugar correto e assim, sucessivamente. Ou seja, com o passar do tempo a lista de itens ordenados aumenta e a lista de itens não ordenados

diminui.

Este é um algoritmo com complexidade $O(n^2)$ no pior caso e na média. No melhor caso possível é $O(n)$, como é possível ver no Pseudocode 1.

Pseudocode 1: Insertion Sort PseudoCode

\\entrada conjunto A com n elementos
\\saída conjunto A de n elementos ordenados

```
ordene(A)
    for i=1 até n-1
        inserte(A, i, A[i])

inserte(A, posição, valor)
    i = posição - 1
    while (i >= 0 E A[i] > valor ) do
        A[i+1] = A[i]
        i=i-1
```

Fonte: o autor (2020)

Exemplo de código em C++

```
/*
AUTHOR: Frank de Alcantara (frank.alcantara@gmail.com)
DATA: 07 ago. 2020
Programa de demonstração do uso do Insertion Sort.
*/

#include <iostream>
#include <ctime>
#include <chrono>

using namespace std;

//protótipo de função
void display(int *, string);

int main()
{
    //iniciando um gerador de números randômicos
    srand((unsigned)time(0));

    int array_teste[100000];
    string string_1;
```

```

// preenchendo nosso conjunto com números aleatórios
for (int i = 0; i < 100000; i++)
{
    array_teste[i] = (rand() % 100) + 1;
}

//variáveis usadas para medir o tempo de execução
clock_t clock1, clock2;
//display(array_teste,"lista não ordenada: ");
//função clock() do ctime
clock1 = clock();
//insertion sort
for (int i = 1; i < 100000; i++)
{

    int temp = array_teste[i];
    int j = i - 1;
    while (j >= 0 && temp <= array_teste[j])
    {
        array_teste[j + 1] = array_teste[j];
        j = j - 1;
    }
    array_teste[j + 1] = temp;
}

clock2 = clock();

cout << (float)(clock2 - clock1) / CLOCKS_PER_SEC << endl;
//display(array_teste,"lista ordenada: ");

return 1;
} // fim do main

//implementação da função display para mostrar os arrays
void display(int m[100000], string s)
{

    cout << s << endl;

    for (int i = 0; i < 100000; i++)
    {
        cout << m[i] << "\t";
    }

    cout << endl;
}

```

}

Merge Sort

O algoritmo **Merge Sort** é uma aplicação do padrão de construção de algoritmos **dividir & conquistar**.

No caso do **Merge Sort** esta técnica será aplicada em três fases:

- **Dividir**: divida o conjunto de entrada em dois. Exceto que este conjunto for muito pequeno;
- **Recorrência**: recursivamente ordene os dois conjuntos de dados resultantes;
- **Conquistar**: una os dois conjuntos.

Quando falamos em muito pequeno, a leitora deve entender menor que um tamanho pré-determinado. Em inglês chamamos este valor mínimo para o qual o algoritmo será aplicado de *threshold*.

No caso do **Merge Sort** podemos detalhar um pouco mais o padrão **dividir & conquistar**.

Dividir

Se o conjunto A tem zero, ou um elemento, retornamos o conjunto A já que ele já está ordenado. Você não consegue ordenar conjuntos vazios ou unitários. Caso a cardinalidade do conjunto A seja maior que 1, dividimos este conjunto em dois, A_1 e A_2 com cardinalidade próxima da metade da cardinalidade de A .

Ou, em outras palavras, se o comprimento n de A é maior que 1 então o comprimento de A_1 deve ser $n/2$.

Recursividade e Conquista

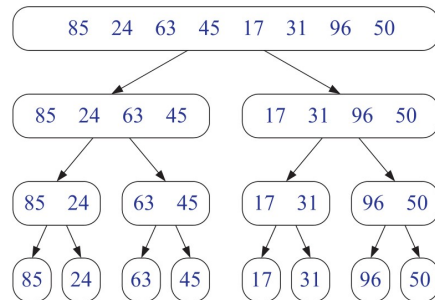
Recursivamente ordene os itens de A_1 e A_2 .

Junte, *merge*, A_1 e A_2 em A .

Visualização

Podemos ver o algoritmo **Merge Sort** em uma árvore binária chamada de **Merge-Sort Tree**. Nesta árvore cada nó representa uma chamada recursiva a função de ordenação do **Merge Sort**. A **Merge-Sort Tree** pode ser vista na Figura 1.

Figura 1: árvore binária do Merge Sort.



Fonte: (GOODRICH; TAMASSIA; MOUNT, 2011)

O **Merge Sort** é um algoritmo relativamente simples, conceitualmente falando. Ainda assim, não está entre os algoritmos mais fáceis do ponto de vista da implementação. O seu pseudocódigo pode ser visto em Pseudocódigo 2.

Pseudocódigo 2: Algoritmo Merge Sort

```
mergeSort(S)
  \entrada conjunto A com n elementos
  \saída conjunto A de n elementos ordenados

  if A.size() > 1
    (A1, A2) ← partition(A, n/2)

  mergeSort(A1)
  mergeSort(A2)

  S ← merge(S1, S2)
```

Fonte: o autor (2020)

Exemplo de Código em C++

```
/*
AUTHOR: Frank de Alcantara (frank.alcantara@gmail.com)
DATA: 07 ago. 2020
Programa de demonstração do uso do Merge Sort.
*/

#include <iostream>
#include <ctime>
#include <chrono>
```

```

using namespace std;

//protótipo de função
void display(int *, string);
void merge(int *, int, int, int);
void merge_sort(int *, int, int);

int main()
{
    //iniciando um gerador de números randômicos
    srand((unsigned)time(0));
    int num = 100000;
    int array_teste[100000];
    //string string_1;

    // preenchendo nosso conjunto com números aleatórios
    for (int i = 0; i < 100000; i++)
    {
        array_teste[i] = (rand() % 100) + 1;
    }

    //variáveis usadas para medir o tempo de execução
    clock_t clock1, clock2;

    //display(array_teste, "\nNão ordenado:\n");

    clock1 = clock();
    merge_sort(array_teste, 0, num - 1);
    clock2 = clock();
    //display(array_teste, "\nNão ordenado:\n");

    //O(nlogn) 100 x log 100
    //O(n^2) 100^2

    cout << (float)(clock2 - clock1) / CLOCKS_PER_SEC << endl;

} //fim do main

//funções implementação
void merge_sort(int *arr, int baixo, int alto)
{
    int meio;

    if (baixo < alto)

```

```

    {
        meio = (baixo + alto) / 2; //typecasting ....
        //dividiu o array em dois, usando os limites passados
        //para identificar o array inferior e o superior
        merge_sort(arr, baixo, meio);
        merge_sort(arr, meio + 1, alto);
        // juntando os arrays já ordenados.
        merge(arr, alto, baixo, meio);
    }
}

void merge(int *arr, int alto, int baixo, int meio)
{
    int temp[200000], i, j, k;
    i = baixo;
    k = baixo; //array final
    j = meio + 1;

    while (i <= meio && j <= alto)
    {
        if (arr[i] < arr[j])
        {
            temp[k] = arr[i];
            k++;
            i++;
        }
        else
        {
            temp[k] = arr[j];
            k++;
            j++;
        } // fim do if
    } // fim while

    while (i <= meio)
    {
        temp[k] = arr[i];
        k++;
        i++;
    }

    while (j <= alto)
    {
        temp[k] = arr[j];
        k++;
        j++;
    }
}

```

```

    }

    for (i = baixo; i < k; i++)
    {
        arr[i] = temp[i];
    }
}

void display(int m[100000], string s)
{
    cout << s << endl;

    for (int i = 0; i < 100000; i++)
    {
        cout << m[i] << "\t";
    }

    cout << endl;
}

```

Material de apoio

Você pode baixar a versão em pdf desta aula clicando [aqui](#)

Obras Citadas

GOODRICH, Michael T.; TAMASSIA, Roberto; MOUNT, David M.. Data Structures and Algorithms in C++. 2. ed. Mariland: John Wiley & Sons, 2011.