ALGORITIMOS DE ORDENAÇÃO INTERNA

1 O PROBLEMA DA ORDENAÇÃO 2

1.1 Conceito 4

1.2 Análise de eficiência 7

2 BUBBLE SORT 11

3 INSERTION SORT 16

4 MERGE SORT.....................................21

5 REFERÊNCIAS 29

1 O PROBLEMA DA ORDENAÇÃO

Entre os algoritmos mais importantes destacam-se os algoritmos de ordenação, em inglês *sort*. A a maior parte das tarefas diárias que resolvemos envolve alguma forma de ordenação. Ordenamos registros em bancos de dados, as tarefas do dia, conhecimento em parágrafos, parágrafos em páginas e páginas em sites. Poderíamos estender esta lista para lembrar que nós, como espécie, ordenamos desde muito cedo, antes mesmo de aprender a contar. Esta tarefa, a ordenação está emaranhada com o desenvolvimento da humanidade desde que percebemos que existem vantagens e desvantagens na ordem em que eventos acontecem. A própria civilização, como conhecemos, deve seu início à agricultura e a percepção da ordem dos ciclos solares. Ao longo da história os processos de ordenação tiveram impacto na ascensão e queda de impérios e empresas. Neste capítulo vamos estudar a ordenação do ponto de vista da computação e vamos estudar os algoritmos mais eficientes para ordenar grandes conjuntos de dados.

O *Google* deve seu sucesso a uma lista ordenada criada a partir de um algoritmo de classificação, o *PageRank*. Foi este algoritmo desenvolvido por ***Larry Page*** e ***Sergey Brin*** que permitiu que o *Google* retornasse uma lista de páginas na ordem que melhor atendesse as demandas de busca de um usuário específico e se tornasse a empresa quase onipresente que é hoje. A missão do Google era, talvez ainda seja, indexar todo o conhecimento da humanidade, mas seu sucesso comercial se deve a ordem que este conhecimento foi apresentado aos seus usuários nos primeiros anos da empresa. A leitora há de concordar comigo que o uso da palavra clientes seria mais adequado na última sentença. Não a uso apenas

para manter o *status quo*[1] da internet quem usa um serviço é o usuário. Com tal envolvimento na vida diária, uma parte importante das tarefas computacionais incluem alguma forma de ordenação. Esta adoção faz com que os algoritmos de *sort* estejam também entre os algoritmos mais estudados.

Estamos sempre em busca de mais eficiência, para os algoritmos já conhecidos, sem abandonar a esperança de encontrar novos algoritmos para solucionar problemas novos ou antigos.

Uma das primeiras regras de otimização, tarefa diária de todos os envolvidos com eficiência, diz que é melhor começar pelos algoritmos de ordenação. Não bastasse isso como justificativa para o estudo dos algoritmos de ordenação. Precisamos ressaltar que os algoritmos de ordenação são excelentes exemplos do uso das técnicas de criação de algoritmos como, por exemplo: *dividir-e-conquistar*, *estruturas de dados compostas* e uso de aleatoriedade na criação de algoritmos.

Fora do mundo acadêmico é raro utilizarmos um dos algoritmos que vamos estudar para ordenar uma lista de números, palavras ou frases. O comum é que o dado que precisa ser ordenado esteja representado por uma estrutura composta de vários campos, um registro. Em inglês os registros são chamados de *record*. Estes *records* são estruturas de dados compostas de um ou mais campos chaves, a partir dos quais realizaremos a ordenação, e outros tantos campos de informação. Nas linguagens de programação C e C++, os *records* podem ser representados por *structs* e objetos. O uso típico destas estruturas para armazenar informações se estende aos bancos de dados e suas tabelas de registros. Em todos os casos, a ordenação deverá ser realizada em conjuntos contendo itens compostos e resolver um problema específico.

> ***Problema:*** *dado um conjunto de dados qualquer, de que forma podemos ordenar estes dados em uma sequência determinada.*

[1] Sempre quis usar *status quo* em uma frase.

1.1 Conceito

A escolha do algoritmo para a solução de um determinado problema de ordenação irá depender do problema, dos recursos disponíveis e da característica dos dados. A leitora precisa ter em mente que nem sempre o algoritmo mais rápido será a melhor escolha. Pode ser que o custo envolvido na implementação de um algoritmo mais rápido e os recursos necessários para garantir que ele realmente seja o mais rápido não compense o benefício adquirido. Esta restrição na escolha dos algoritmos é mais perceptível quando a lista de itens que precisa ser ordenada é pequena ou quando esta lista tem um nível de entropia baixo.

Algoritmos diferentes terão eficiência diferente em dados diferentes. Isso quer dizer que um algoritmo pode ser muito rápido em casos onde os dados estão arranjados em um conjunto com pouca entropia e não ter utilidade nenhuma em casos onde os dados estão na ordem inversa daquela que desejamos.

A palavra entropia, importada da matemática e da física, neste conceito se refere ao percentual de dados que está fora da ordem desejada. Sendo assim, vamos considerar que dados que já se encontrem na ordem desejada terão entropia zero dados totalmente ordenados na ordem invertida, terão entropia 1, ou 100%. A entropia também diz respeito a diferença que existe entre os itens do conjunto que desejamos ordenar, se tivermos pouca diferença entre estes itens, a entropia é baixa, se os itens forem muito diferentes, a entropia é alta.

Do ponto de vista dos algoritmos dividimos o problema em duas classes. Os problemas de ordenação interna e externa. Quando nos referimos a ordenação interna, nos referimos a conjuntos de dados que estão residentes na memória do sistema que está rodando o algoritmo. Especificamente a memória de alto desempenho. Nestes problemas não existe a necessidade de recorrer a dispositivos como discos rígidos, ou armazenamento remoto. Todos os outros problemas de ordenamento serão considerados com externos e, serão abordados quando for conveniente.

De uma forma um pouco mais formal, o problema de ordenação pode ser definido da seguinte forma:

> ***Problema:*** *dado um conjunto de dados qualquer, representado por* $R = \{r_1, r_2, r_3, ..\}$ *contendo, no mínimo, um campo chave. De tal forma que, o conjunto de chaves será representado por* $K = \{k_1, k_2, k_3, \dots\}$ *. Precisamos ordenar o conjunto* R *de forma que ao final do algoritmo seus itens estejam em uma sequência* $\{k_1 \leq k_2 \leq k_3 \dots .\}$.

Claramente a descrição matemática do problema de ordenação, anteriormente descrita, está diretamente relacionada com a ordenação crescente do conjunto R utilizando como referência o conjunto de chaves K em **ordem crescente**. Poderíamos ter definido o problema em ordem decrescente, sem prejuízo para a definição do problema. Para isso tudo que precisaríamos mudar seria a definição da sequência final para que esta fosse determinada por $\{\mathrm{k}_1 \geq \mathrm{k}_2 \geq \mathrm{k}_3 \dots .\}$. Ou seja, a ordem, seja ela crescente, ou decrescente, não afeta a caracterização do problema de ordenação apenas o resultado do processo. Chegaremos à solução do problema de ordenação, crescente ou decrescente, permutando itens dentro do conjunto.

> *A **permutação** de um conjunto finito de elementos é a organização, arranjo, destes elementos em uma linha (KNUTH, 1998). Segundo uma ordem qualquer. A ordenação é fazer a permutação destes elementos em uma ordem definida.*

A definição que usamos é suficientemente ampla para permitir a existência de itens no conjunto a ser ordenados que possuam o mesmo valor k_n. Dois ou mais registros podem ter chaves iguais. Aqui, é preciso que a leitora considere que esta possibilidade de repetição de chaves de ordenação não pode ser aplicada a todos os problemas que encontraremos. Bancos de dados, por exemplo. Não é raro que algumas tabelas, tenham registros com chaves únicas, não permitindo qualquer tipo de repetição. No universo dos bancos de dados, a tabela, representa o conjunto de itens a serem ordenados.

O problema de ordenação é relativamente simples se considerarmos apenas chaves numéricas. Chaves alfabéticas requerem um conjunto de regras mais complexos devido a existência de letras maiúsculas, minúsculas, pontuação e nomes compostos. Estas regras determinarão, por exemplo, se Silvia deve ser colocado antes ou depois de silvia na lista ordenada. A forma de resolver o problema da ordenação alfabética é a definição de métodos, ou funções, especificamente criados para aplicar as regras de ordenação no momento da permutação de itens. Neste momento de permutação faremos as comparações necessárias a ordenação sejam as chaves numéricas, alfabéticas, ou qualquer outra que seja interessante para resolução do problema.

Caso existam valores duplicados no conjunto de chaves, chamaremos de estáveis aos algoritmos de ordenação que, ao final do processo, tenham mantido a ordem inicial dos itens com chaves duplicadas. Infelizmente poucos algoritmos de ordenação rápidos, são também estáveis.

Os algoritmos de ordenação estáveis, quando aplicados sobre conjuntos com chaves duplicadas, irão colocar os itens duplicados na ordem em que eles aparecem, um em relação ao outro, antes da ordenação. Se não ficou claro, a Figura 1 apresenta um pequeno exemplo de ordenação estável em um conjunto de cartas ordenadas de forma estável por valor e não por naipe.

Figura 1 - Exemplo de ordenação estável com cartas de baralho.

A leitora pode observar que no conjunto não ordenando a Dama de Copas aparece antes da Dama de Espadas e, consequentemente, acabou ficando antes ao término do processo de ordenação.

Nossa preocupação, neste capítulo, será entender os algoritmos de ordenação, notadamente os que são mais eficientes para a manipulação de conjuntos de dados muito grandes. Todos estes algoritmos serão aplicados sobre um *array* de itens numéricos e, sempre que for interessante para a eficiência de um determinado algoritmo vamos expandir este limite de simplicidade para incluir estruturas de dados mais complexas. A leitora, curiosa como é, deve estar se perguntando e como determinamos a eficiência de um algoritmo? Neste caso, usamos a análise assintótica.

1.2 Análise Assintótica

Analisamos algoritmos para ter uma ideia da sua eficiência independentemente da plataforma escolhida para a sua execução. Queremos saber se um determinado algoritmo será o mais eficiente possível, independente da máquina que o executará. Um algoritmo ruim em uma máquina excelente, ainda será um algoritmo ruim, apenas rodará mais rápido do que se estivesse em uma máquina ruim. Um algoritmo eficiente será eficiente, não importa a arquitetura. Observe que eu não falei em tempo de execução, este dependerá da máquina, do conjunto, da entropia dos itens e do algoritmo. Em geral, não temos como mudar nenhum destes itens, exceto o algoritmo. Assim, analisamos os algoritmos, com uma visão matemática do problema, justamente para garantir que, caso nada mais seja alterado, tenhamos o resultado no menor tempo possível.

Vamos analisar os algoritmos para caracterizar sua eficiência utilizando uma variação do artigo de Donald Knuth que, de forma prática, estabeleceu as regras para este tipo de classificação (KNUTH, 1976). Esta teoria da análise dos algoritmos se baseia na definição de três pontos relevantes entre todas as possibilidades de uso de um algoritmo: o pior caso possível, o melhor caso possível

e o caso médio. Para definir estes casos definimos três funções: $O(f(n))$; $\Omega(f(n)$ e $\Theta(f(\mathrm{n}))$ para representar o pior, o melhor e o caso médio de uso dos algoritmos. E chamaremos este processo de análise assintótica.

A análise assintótica de um algoritmo representado por uma função $f(n)$ refere-se a taxa de crescimento de $f(n)$ à medida que n cresce. Ou seja, analisamos a eficiência do algoritmo à medida que a cardinalidade do conjunto de itens aumenta. Estamos interessados em conjuntos com muitos elementos, cardinalidade muito alta, na casa dos milhões de itens. Assim, um algoritmo cuja taxa de crescimento seja zero, será melhor que um algoritmo cuja taxa de crescimento seja logarítmica. Pelo menos, esta é a regra típica que usaremos para analisar nossos algoritmos.

Imagine, que temos dois algoritmos, um que pode ser representado por uma função linear, $f_1(n) = d \times n + k$, ou seja, para cada n, o tempo necessário para seu processamento será dado pela cardinalidade do conjunto multiplicada por uma constante d qualquer e este resultado somado a outra constante k, usando as mesmas constantes poderíamos ter um algoritmo dado pela função exponencial $f_1(n) = d \times n^2 + k$. Para que não reste dúvidas da diferença no crescimento destas duas funções e do seu impacto sobre o tempo. Considere que, nas duas funções,

$d = k = 5$ e que n pode assumir os valores $1, 2, 3, 4, 5, 6, 7$, se assim for, o Gráfico 1 mostra o crescimento destas duas funções.

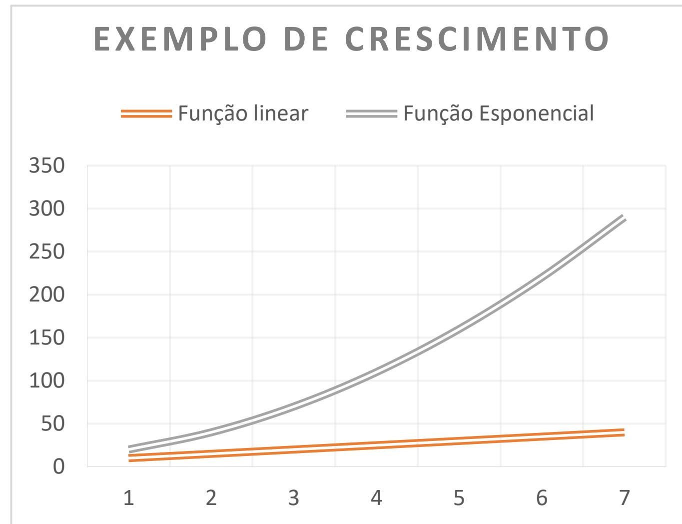

Gráfico 1 - Exemplo de comparação do crescimento de uma função exponencial com uma linear

Se o eixo dos y representar o tempo, com apenas 5 itens, o tempo necessário para execução do algoritmo já é centenas de vezes maior na função exponencial. Eis porque precisamos nos preocupar com a análise de algoritmos.

Funções assintóticas são aquelas que não atingem um determinado valor. Tendem a este valor apenas no infinito. Como não existe o infinito, nunca atingimos este valor. Usamos este nome para este tipo de análise porque vamos analisar o algoritmo no seu pior caso, aquele que irá provocar o maior gasto de tempo. Quando dizermos que um algoritmo roda no tempo $T(n)$ no seu pior caso e, neste caso vamos utilizar a notação $O(f(n))$, chamada, apenas pelos íntimos de notação *Big-O*.

E, usando a notação *Big-O*, ainda usaremos uma técnica de simplificação. Vamos pensar grande, muito grande. Para que a leitora entenda o que é pensar grande, responda, sem pensar muito, quanto custa um automóvel mais um skate? Pode chutar! Não precisa nem procurar na internet. O que queremos é apenas uma aproximação, uma faixa de valores. Que tal R$70.000,00, sem escolher um modelo específico esta é uma faixa válida? Não? Que tal R$100.000,00? Concorda.

Observe que em nenhum momento, nesta consideração, se quer levamos em consideração o preço do skate. Este valor é tão pequeno em relação ao preço de um carro que podemos desprezá-lo completamente e ainda ter uma boa ideia do valor total do investimento. Este é o truque que usaremos na análise assintótica. Vamos nos preocupar apenas com a parte do algoritmo que realmente tem efeito no tempo total de processamento.

Nas funções que usamos como exemplo $f_1(n) = d \times n + k$, e $f_1(n) = d \times n^2 + k$ o impacto das constantes $d\ e\ K$ é tão pequeno que podemos desprezar seu efeito. Em alguns casos, a diferença de tempo será tão grande que não será possível diferenciar o efeito de $d\ e\ K$ no tempo final gasto pelos algoritmos. Assim, recorremos a notação *Big-O,* e estas funções serão respectivamente representadas por: $O(n)$ e $O(n^2)$ que leremos *ordem* n e *ordem* n *ao quadrado*. Sendo assim, o algoritmo representado pela função $f_1(n) = d \times n + k$ é representada por $O(n)$ o que quer dizer que este algoritmo roda em tempo linear. Enquanto, o algoritmo representado por $f_1(n) = d \times n^2 + k$ é representada por $O(n^2)$ o que significa que roda em tempo quadrático. A leitora já deve ter deduzido, mas não custa ressaltar, algoritmos representados por $O(1)$ rodam em tempo constante e independem da cardinalidade do conjunto de itens de entrada.

A principal vantagem da notação *Big-O* é que podemos desconsiderar todos os termos da equação polinomial que represente o algoritmo, por exemplo nas funções $f_1(n) = d \times n + k$, e $f_1(n) = d \times n^2 + k$, podemos desprezar o k antes mesmo de começar. Isto é possível porque a definição original de Knuth explicita que:

$$O\big(g(n)\big) = \{f(n) : constantes\ positivas\ c\ e\ n_0\ tal\ que\colon 0 \leq f(n) \leq cg(n)\ para\ todos\ n \geq n_0\}$$

A função não negativa $f(n)$ pertence ao conjunto de funções $O(g(n))$ se existir uma constante positiva c que faça $f(n) \leq cg(n)$ para um valor de n suficientemente grande. Sempre é possível escrever $f(n) \in O(g(n))$ porque $O(g(n))$ é um conjunto. Observe, que na notação assintótica, o sinal de igualdade

indica pertencimento e não igualdade. Sendo assim, considere a função $f_3 = 4n^4 + 3n^2 + n + 1237$ usando a notação *Big-O*, podemos desprezar todos os termos cuja ordem seja 4 assim sendo ordem deste algoritmo será dado por $O(n^4)$. Vamos deixar a matemática pesada para um curso de análise de algoritmos. Ainda assim, é necessário destacar os casos mais comuns na notação assintótica e a Tabela 1 assume esta função.

Big-O	**Tempo**
$\boldsymbol{O(1)}$	Constante
$\boldsymbol{O(\log n)}$	Logarítmico
$\boldsymbol{O(n)}$	Linear
$\boldsymbol{O(n \log n)}$	Linearitimico
$\boldsymbol{O(n^2)}$	Quadrático
$\boldsymbol{O(n^3)}$	Cúbico
$\boldsymbol{2^{O(N)}}$	Exponencial
$\boldsymbol{O(n!)}$	Fatorial

Tabela 1 - Notação Assintótica (Big-O), mais comuns.

Veremos como fazer a análise assintótica dos algoritmos de ordenação a cada um dos algoritmos que analisarmos. Começando pelo mais simples de todos os algoritmos de ordenação.

2 *BUBBLE SORT*

Bubble sort, ordenação em bolha, é um dos mais antigos algoritmos de ordenação, em 1962 Kenneth Iverson utiliza este nome em um livro chamado *A Programming Language*[2] e, a partir deste ponto, parece que todos passam a usar este nome para se referir a um processo de ordenação por inversão. Trata-se de um algoritmo simples cuja única utilidade real é demonstrar o processo de

[2] Em tradução livre: uma linguagem de programação. Foi reconfortante encontrar uma versão de *A Programming Language* de Iverson preservada na internet em: ***http://www.softwarepreservation.org/projects/apl/Books/APROGRAMMING%20LANGUAGE***

ordenação e explicar como podemos fazer a análise assintótica de um algoritmo. Ainda assim, insistentemente, este algoritmo continua aparecendo nos livros de algoritmos e programação.

Trata-se de um algoritmo de ordenação baseado em comparação. Em cada ciclo, o algoritmo compara dois elementos adjacentes no conjunto que está sendo ordenado e troca a posição daqueles elementos que não estiverem ordenados. Basicamente, a cada passagem completa pelo conjunto, o item com o maior índice é posto no lugar correto. Lendo o livro do Iverson (1962), o temo *bubbling*, borbulhando, é usado como metáfora para indicar o que ocorre com cada item do conjunto, borbulhando até a sua posição. O Pseudocode 1, apresenta uma versão do *Bubble sort*.

Pseudocode 1: *Bubble sort*

```
for i = 1 to n - 1
    for j = 1 to n - i
        if (a[j] > a[j + 1]) then
            swap(a[j], a[j+1]
```

Consideraremos a letra i para indicar cada passada do algoritmo pelo conjunto que está sendo ordenado. Vamos considerar o melhor e o pior caso. No caso do *Bubble sort*e, independente da entrada, o número de passagens completas será dado por $n-1$. E para cada passagem o algoritmo irá realizar $n-i$ comparações. No melhor caso, o conjunto de entrada já está na ordem certa, nenhuma troca é realizada, ainda assim, passamos por todo conjunto $n-1$ vezes e verificamos a necessidade de troca, ou não, $n-i$ vezes. Ou seja, $n-1$ comparações serão feitas na primeira passagem, $n-2$ comparações na segunda passagem, $n-3$ comparações na terceira passagem e assim, sucessivamente até o fim do conjunto. Ou seja, o número total de passagens será:

$$(n-1)+(n-2)+(n-3)\ldots+3+2+1$$

Resolvendo esta soma temos:

$$\frac{n(n-1)}{2} \therefore \frac{n^2}{2} - \frac{n}{2}$$

Como podemos nos concentrar apenas no termo de maior ordem, temos a função assintótica dada por $O(n^2)$, para o *Bubble sort*, nos três pontos de análise: pior cenário, os itens perfeitamente ordenados na ordem inversa da desejada; melhor cenário, o itens perfeitamente ordenados na ordem desejada e cenário médio, itens aleatoriamente distribuídos. A Implementação 1 apresenta uma versão do *Bubble sort* em C++.

Implementação 1 *Bubble sort* Tradicional.

```
/*
AUTHOR: Frank de Alcantara
DATA: 31 jul. 2020
Programa de demonstração do uso do Bubble sort.
*/
#include <iostream>
#include <ctime>

using namespace std; // usando a biblioteca padrão

int main(){
    //iniciando o gerador randômico
    srand((unsigned)time(0));

    int temp = 0, tam = 10, passo = 0, conjunto[10];
    //preenchendo o conjunto (array a) com números randômicos.
    for (int i = 0; i < tam; i++){
        conjunto[i] = (rand() % 100) + 1; //randômicos < que cem
    }

    //imprime a lista criada que será ordenada só para testes
    cout << "\n\nLista original: {";
    for (int k = 0; k < tam; k++){
        cout << conjunto[k] << ",";
```

```
    }
    cout << "}" << endl;

    //Aqui está o Bubble sort
    for (int i = 0; i < tam; i++){
        for (int j = i + 1; j < tam; j++){
            if (conjunto[j] < conjunto[i]){
                temp = conjunto[i]; //para fazer a troca
                conjunto[i] = conjunto[j];
                conjunto[j] = temp;
            }
        }
        passo++; //não faz parte do algoritmo. Conta as passagens
    }
    //imprimindo a lista ordenada
    cout << "\n\nLista ordenada: {";
    for (int k = 0; k < tam; k++){
        cout << conjunto[k] << ",";
    }
    //imprimindo o total de passagens
    cout << "\nPassos: " << passo << endl;
}
```

Nesta implementação geramos um conjunto de itens aleatórios, populamos o *array* $conjunto[10]$ com estes números aleatórios, mostramos esta lista na tela, aplicamos o *Bubble sort* sobre este *array*, mostramos no terminal o conjunto ordenado e, com uma pequena alteração, a inclusão da variável $passo$ podemos contar quantos passos são necessários para ordenar todo o conjunto.

O acréscimo de uma variável de monitoramento pode otimizar o algoritmo. Podemos usar esta variável no laço interno, dentro do *if* para indicar se em cada passagem alguma troca de posição foi realizada ou não. Caso uma passagem termine sem essa troca, encerramos o laço for externo. Este encerramento é feito com o comando *brake,* colocado em um if no fim do laço externo. A Implementação 2 mostra uma versão do *Bubble sort* com essa alteração.

Implementação 2 – *Bubble sort* Otimizado

```
/*
AUTHOR: Frank de Alcantara
DATA: 31 jul. 2020
Programa de demonstração do uso do Bubble sort.
*/
#include <iostream>
#include <ctime>

using namespace std; // usando a biblioteca padrão

int main(){
    //iniciando o gerador randômicos
    srand((unsigned)time(0));

    int temp = 0, tam = 10, passo = 0, conjunto[10], monitora = 0;
    //preenchendo o conjunto (array a) com números randômicos.
    for (int i = 0; i < tam; i++){
        conjunto[i] = (rand() % 100) + 1; //randômicos < que cem
    }

    //imprime a lista criada que será ordenada só para testes
    cout << "\n\nLista original: {";
    for (int k = 0; k < tam; k++){
        cout << conjunto[k] << ",";
    }
    cout << "}" << endl;

    //Aqui está o Bubble sort
    for (int i = 0; i < tam; i++){
        for (int j = i + 1; j < tam; j++){
            if (conjunto[j] < conjunto[i]){
                temp = conjunto[i]; //para fazer a troca
                conjunto[i] = conjunto[j];
                conjunto[j] = temp;
                monitora = 1;
            }
```

```
        }
        passo++; //não faz parte do algoritmo. Conta as passagens
       if (monitora == 0){
     break; //encerra o laço principal se não haver trocas
    }
    //imprimindo a lista ordenada
    cout << "\n\nLista ordenada: {";
    for (int k = 0; k < tam; k++){
        cout << conjunto[k] << ",";
    }
    //imprimindo o total de passagens
    cout << "\nPassos: " << passo << endl;
}
```

Esta otimização economiza alguns ciclos de máquina já que o algoritmo irá parar assim que todos os itens estejam em posição e traz o melhor caso para a complexidade $O(n)$, mas não é de grande ajuda já que, no melhor caso, os itens já estão na posição desejada.

3 INSERTION SORT

Todos os principais livros de algoritmos começam a descrição do algoritmo *Insertion Sort* usando como metáfora o 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.

A palavra inserindo do parágrafo anterior, força um pouco a amizade, mas serve como uma luva como metáfora do funcionamento do *Insertion Sort*.

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

O *Insertion sort* é o único algoritmo da classe dos algoritmos de ordenação que resolvem este problema usando o próprio conjunto de itens desordenados. Nesta classe estão os algoritmos que ordenam por seleção, inserção e troca. Como todos estes algoritmos sofrem por falta de eficiência sempre que o conjunto de itens cresce, vamos usar como exemplo desta classe de algoritmos o *Insertion Sort.* A amável leitora precisará se lembrar que só pode usar o *Insertion Sort* em conjuntos de dados pequenos e com baixa entropia. Antes de começarmos a analisar este algoritmo, algumas considerações precisam ser feitas:

1. O caso mais simples consiste em um conjunto unitário. Se o conjunto de itens desordenado tiver apenas um item este conjunto já está ordenado;
2. Vamos assumir que no pior caso, os $n-1$ elementos estarão ordenados após $n-1$ interações.

Do ponto de vista da execução do algoritmo, começamos por assumir que a primeira posição do conjunto de dados representa uma lista de elementos que já está ordenada. Em cada ciclo do algoritmo percorremos do item 1 até o item $n-\ 1$, uma conclusão baseada na consideração 2.

Em cada ciclo, o item correspondente a esta passagem será comparado com aqueles que já estão ordenados:

- **Na primeira passagem**, apenas o primeiro elemento está ordenado.
- **Na segunda 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. A cada ciclo do algoritmo a lista de itens ordenados aumenta e a lista de itens não ordenados diminui.

Este é um algoritmo com complexidade $O(n^2)$ tanto no pior caso quanto no caso médio. Já no melhor caso possível o *Insertion sort* é da família dos algoritmos cuja análise assintótica é expressa por $O(n)$. Veremos como é possível chegar a esta conclusão, logo depois que analisarmos o Pseudocode 2.

Pseudocode 2: Insertion sort

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

insira(A, posição, valor):
    i = posição - 1
    while (i >= 0 AND A[i] > valor) do
    A[i+1] = A[i]
    i = i-1
```

Podemos observar, analisando o Peseudocode 2 que no número de comparações C_i que será realizado até a inésima posição é, no pior caso, quando todos os itens do conjunto estiverem perfeitamente ordenados no sentido oposto, igual a $n-1$, logo $C_i = n-1$. No melhor caso, quando o conjunto de itens estiver perfeitamente ordenado no sentido desejado, $C_i = 1$. Se consideramos que probabilidade de permutação de todos os itens é igual, no caso médio teremos $C_i = 1/2$. O número de movimentos que precisaremos fazer será dado por $Mi = C_1 + 1$. Assim, podemos explicitar matematicamente tanto o número de comparações quanto o número de movimentos e teremos (WIRTH, 1986):

$$C_{min} = n - 1 \qquad M_{min} = 2 \times (n-1)$$

$$C_{med} = \frac{n^2 - n}{4} \qquad M_{med} = \frac{n^2 + 3n - 4}{4}$$

$$C_{pior} = \frac{n^2 - 3n + 2}{2} \qquad M_{pior} = \frac{n^2 - n}{2}$$

Se lembrarmos que o objetivo da análise assintótica é remover os fatores constantes e os fatores de menor ordem chegamos no pior caso a $O(n^2)$ e no melhor caso $O(n)$. Como esta análise foi um tanto quanto matemática, podemos partir para um raciocínio um pouco mais simples. Comece observando que temos duas condições de parada, `i >= 0 AND A[i] > valor` as duas dentro do laço *while.* A primeira irá impedir o que o algoritmo percorra algum valor fora dos limites do conjunto de dados, no Pseudocode 2 representado por um *array* e a segunda para encontrar o lugar correto para inserir um item i qualquer. Vamos lembrar que na análise de pior cenário estamos procurando o limite superior do tempo gasto quando estivermos executando o algoritmo e ignorar as possíveis interrupções nos laços antes de percorrer todo o conjunto e considerar que o laço *while* n vezes, onde n representa a cardinalidade do conjunto de itens que estamos ordenando. Desculpe, me distraí novamente. O n representa o comprimento do *array*. Como o laço externo, o laço *for* obrigatoriamente rodará *n* vezes então temos que o *Insertion sort* é um algoritmo da família $O(n^2)$. Na Implementação 2 é possível ver o *isertion sort* em C++.

Implementação 2 – *Insertion Sort*

```
/*
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: ");

    //anotando o tempo de início
    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;
    }
```

```
     //anotando o tempo de fim
    clock2 = clock();
     // imprimindo a diferença entre o tempo de início e fim
    cout << (float)(clock2 - clock1) / CLOCKS_PER_SEC << endl;
    // imprimindo o conjunto ordenado
    display(array_teste,"lista ordenada: ");
    return 1;
} // fim do main

//implementação da função display para mostar 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;
}
```

---

4 MERGE SORT

Os algoritmos que vimos até o momento, apesar de simples, não têm uso prático em problemas reais se a cardinalidade do conjunto de itens for superior a 1.000.000 de itens. O *Merge Sort* é o primeiro algoritmo que veremos neste livro que usa recursividade e o paradigma de construção de algoritmos chamado de Dividir e Conquistar. Este algoritmo é atribuído a ***John Von Neumann*** e foi publicado a primeira vez no ano de 1945(KNUTH, 1998). Originalmente o *Merge Sort* foi desenvolvido para resolver o problema de ordenação em máquinas que não

dispunham de memória suficiente para armazenar todo o conjunto de itens em memória. Portanto, formalmente, o Merge Sort não deveria estar listado entre os algoritmos de ordenação interna. Contudo, esta divisão entre ordenação interna e externa sofre alteração a cada novo passo que a tecnologia dá. O Merge Sort divide o conjunto de itens em dois subconjuntos de forma recursiva, ordena estes subconjuntos e une os subconjuntos novamente. O que, em última análise é a aplicação de uma estratégia militar: dividir e conquistar.

4.1 Dividir e conquistar

Este padrão de construção de algoritmos resolvem o problema proposto passando por três passos distintos:

1. dividir o problema em problemas menores, ou mais simples. No nosso caso, a instância de entrada, um conjunto de dados, em partes menores, subconjuntos.
2. conquistar os problemas menores, resolvendo estes subproblemas. No nosso caso, ordenar os subconjuntos.
3. combinar as soluções dos problemas menores criando uma solução para o problema original. Outra vez, no nosso caso, fazendo a união dos subconjuntos em um conjunto ordenado.

A estratégia Dividir e Conquistar, apresenta características diferentes para a solução de cada um dos problemas que queremos resolver com sua utilização. No caso do *Merge Sort* o passo dividir cria dois subconjuntos, que chamaremos de conjunto da esquerda e conjunto da direita, ou para simplificar de esquerda e direita, uma concessão aos momentos conturbados que o planeta atravessa neste começo do Século XXI. O passo conquistar chamará a rotina de ordenar de forma recursiva passando para esta função os conjuntos da esquerda e da direita. Por fim, o passo combinar é implementado por uma rotina que irá fundir os conjuntos

esquerda e direita em um conjunto ordenado. Uma implementação deste algoritmo pode ser vista no Pseudocode 3.

Pseudocode 3 – Merge Sort

```
mergesort(var array a):
    if (n == 1) return a
    var l1 array = a[0] ... a[n/2]
    var l2 array = a[n/2+1] ... a[n]

    l1 = mergesort(l1)
    l2 = mergesort(l2)
    return merge(l1, l2)

merge(var array a, var array b):
   var array c
   while (a e b têm elementos)
      if (a[0] > b[0])
         add b[0] no fim de c
          remova b[0] from b
       else
         add a[0] no fim de c
         remova a[0] from a

   while (a tiver elementos)
     add a[0] no fim de c
      remova a[0] from a

   while (b tem elementos)
     add b[0] no fim de c
      remova b[0] from b
    return c
```

Como a recursividade provoca a redução da cardinalidade do conjunto que será ordenado, o problema de ordenação é, na realidade, um problema de

conjuntos muito pequenos. Sendo assim, a eficiência do algoritmo Merge Sort depende da eficiência das rotinas que usaremos para fazer a combinação destes subconjuntos. Vamos ver um exemplo, simples de Merge Sort, a Figura 2 mostra o resultado da divisão de um conjunto de dados.

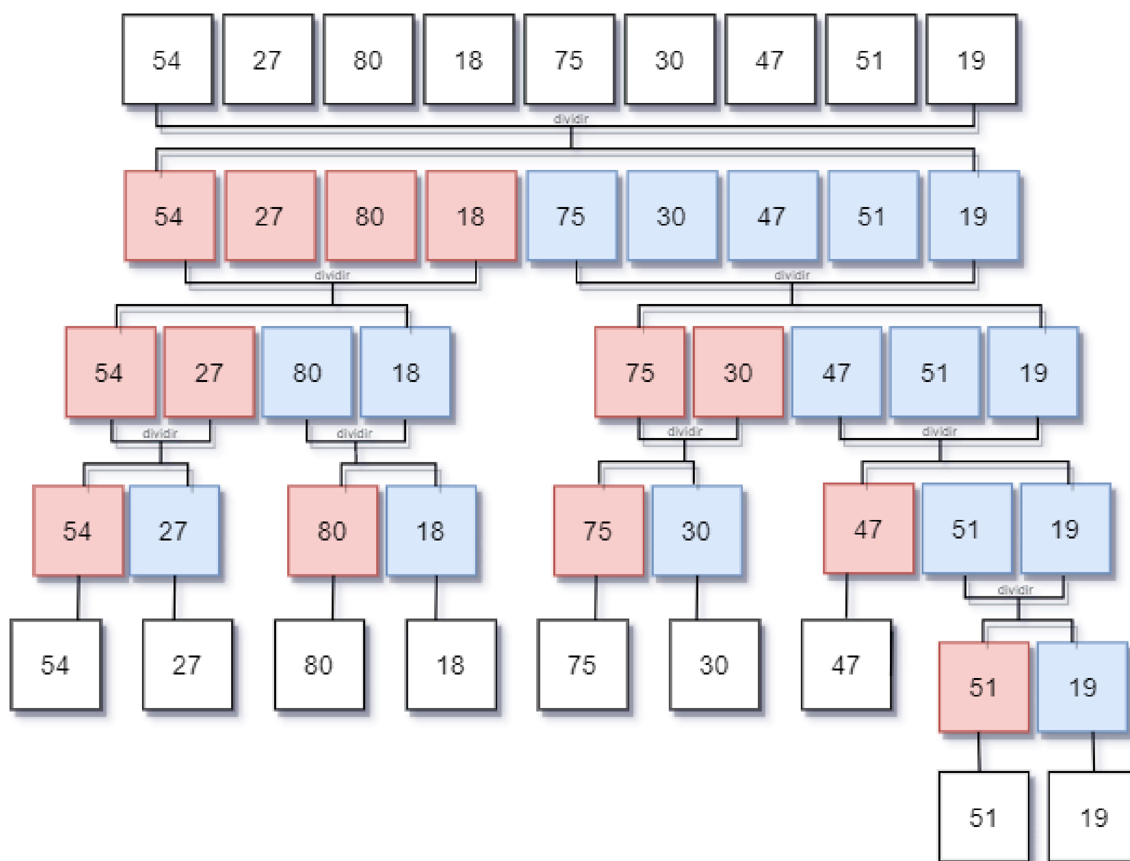

Figura 2 - Exemplo de divisão de um conjunto de dados, para o *Merge Sort*.

Depois que ordenarmos os subconjuntos, o menor item absoluto entre dois subconjuntos deve ser o primeiro item de um destes dois subconjuntos. Este item pode ser separado deixando as duas listas, para trás, uma com um item a menos. Agora, o segundo menor item absoluto, será o primeiro item em uma das duas listas restantes. Podemos retirar este segundo menor item e colocá-lo após o item que removemos anteriormente. Repetimos este processo até que os dois subconjuntos estejam vazios e, desta forma estamos combinando as listas em um conjunto ordenado. No pior dos casos, este processo utilizará $n-1$ comparações.

Assintoticamente podemos dizer que este processo é $O(n)$. O processo de combinação pode ser visto na Figura 3.

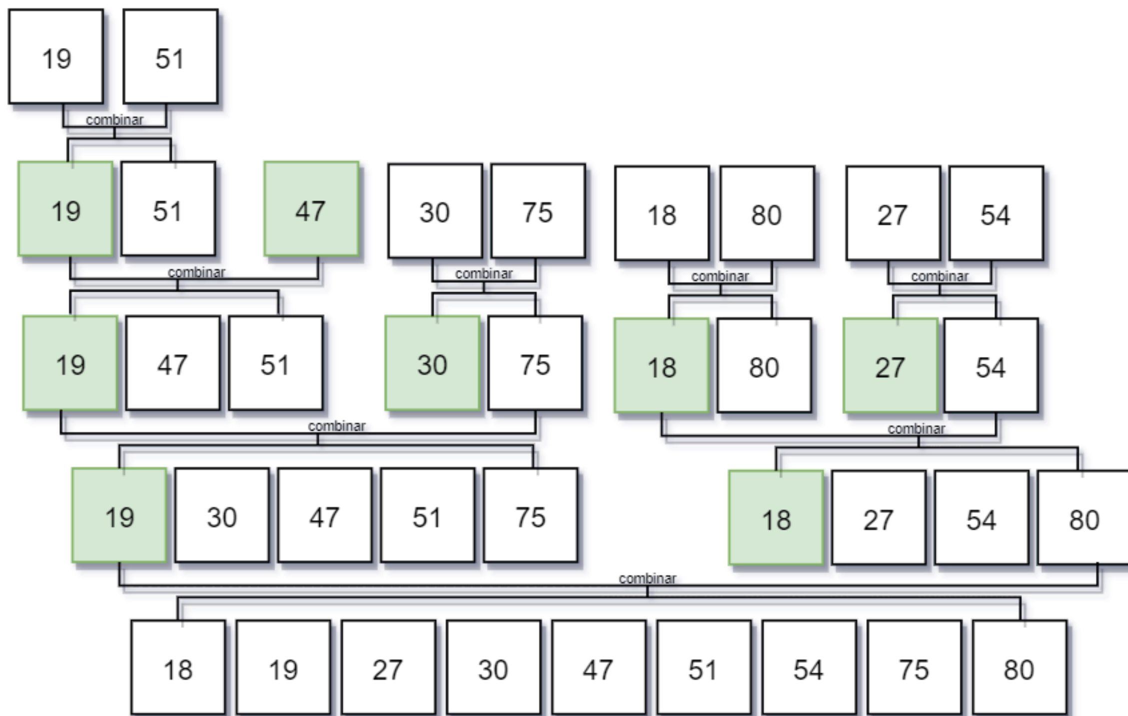

Figura 3 - Exemplo do processo de ordenação do *Merge Sort*

Ainda resta uma dúvida terrível. Qual seria a análise assintótica de todo o processo de ordenação do Merge Sort? Até agora, só vimos a rotina de combinação. Vamos considerar o esforço para trabalhar cada um dos níveis da árvore que criamos. O número de divisões n será uma potência de 2. Sendo assim, a inésima chamada a rotina de combinação (*mergesort*) processará conjuntos com $n/2^i$ elementos. Assim sendo, o trabalho no nível $i = 0$ envolve a combinação de duas listas com $n/2$ elementos que serão comparadas n, no nível $i = 1$, teremos a combinação de $n/4$ elementos com $n - 2$ comparações. Generalizando podemos dizer que o trabalho envolve a combinação de 2^i pares de subconjuntos, cada um com $n/2^{I+1}$ elementos com um total de $n - 2^i$ comparações. Observe que o trabalho realizado para ordenar os itens é cada nível é linear.

Não existem repetições de itens entre subconjuntos, cada item aparece exatamente uma vez em cada conjunto. A não ser que o item já esteja repetido no

conjunto inicial. Assim, em termos de comparações o nível mais alto, será o nível com o maior número de comparações. Dessa forma, o número de comparações é dividido na metade a cada comparação até que o número de comparações n chegue a 1. Esta é a definição de $\log_2 n$. Por outro lado, chamaremos a combinação n vezes por causa da recursão. Assim, a função assintótica do Merge Sort é $O(nlogn)$. Suprimimos a base do logaritmo por ser esta a forma padrão de indicação de logaritmos em notação assintótica. Sempre que ver $\log n$ a leitora deverá entender como $\log_2 n$. Podemos ver um exemplo de implementação do Merge Sort na Implementação 3

. Implementação 3 – *Merge Sort*

```
/*
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;

//protótipos 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];

    // preenchendo nosso conjunto com números aleatórios
    // geramos números entre 1 e 10000 para conseguir uma boa
    // entropia no array gerado
    for (int i = 0; i < 100000; i++){
        array_teste[i] = (rand() % 10000) + 1;
    }

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

    //registrando o momento do início da ordenação
    clock1 = clock();

    merge_sort(array_teste, 0, num - 1);

    //registrando o fim da ordenação
    clock2 = clock();

    //imprimindo o tempo gasto na ordenação
    cout << (float)(clock2 - clock1) / CLOCKS_PER_SEC << endl;

} //fim do main

//funções implementação
//baixo e alto são os limites de divisão no conjunto de
//ordenação
void merge_sort(int *arr, int baixo, int alto){
    int meio;
    if (baixo < alto){
        meio = (baixo + alto) / 2;
        //encontrar o ponto central do array
```

```
        //para identificar o array da esquerda e da direita
        //chamada recursiva a função merge_sort passando os
        //pontos limite de cada divisão.
        merge_sort(arr, baixo, meio);
        merge_sort(arr, meio + 1, alto);
        // juntando os arrays ordenamos nesta função.
        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];
    }
}
```

5 REFERÊNCIAS

CORMEN, T. H. **Algorithms Unlocked**. Cambridege, MA. USA: Massachusetts Institute of Technology, 2013.

IVERSON, K. E. **A Programming Language**. [S.l.]: John Wiley, 1962.

KNUTH, D. E. Big Omicron and Big Omega and Big Theta. **SIGACT News**, p. 18-24, 1976.

SEDGEWICK, R.; FLAJOLET, P. **An Introduction to the Analysis of Algorithms**. Boston, MA. USA: Pearson Education, 2013.

SHAFFER, C. A. **Data Structures and Algorithm Analysis**. 3º. ed. Blacsburg, VA. USA: Dover Publications, 2011.

SKIENA, S. S. **The Algorithm Design Manual**. 2º. ed. London, UK: Springer-Verlag, 2008.

WIRTH, N. **Algorithms and Data Structures**. [S.l.]: Prentice-Hall , 1986.