# Algoritmo de Ordenação

# Merge Sort

Integrantes:   
Lucas Antonetti Pereira – RA: 2024203287  
Diego Rodrigo – RA: 2024204348  
Rodrigo – RA: 2024204270

**1. Problema de ordenação em memória primária**

A questão da ordenação em memória primária (Random Access Memory ou RAM) contempla vários segmentos dentro do escopo da computação, alguns deles sendo: bancos de dados, sistemas embarcados, processamento e análise de grandes quantidades de dados, arquivamento e compressão rápida de dados. Foi observado que para as inúmeras utilidades da ordenação em memória primária, há diferentes tipos de demandas, e estas exigem algoritmos distintos para atendê-las, conforme dito pelo cientista da computação Donald E. Knuth em *The Art of Computer Programming Vol. 3:*

*“Por que existem tantos métodos de ordenação? Na programação de computadores, isso é um caso específico da pergunta: ‘Por que existem tantos métodos para x?’, onde x representa um conjunto de problemas. E a resposta é que cada método tem suas próprias vantagens e desvantagens, de forma que ele supera os outros em determinadas configurações de dados e hardware. Infelizmente, não existe um ‘melhor’ método conhecido para ordenação; existem vários métodos considerados os melhores, dependendo do que será ordenado, em qual máquina e para qual finalidade. ”*

Dentre as muitas possibilidades de algoritmos de ordenação há implementações mais populares tendo seus pros e contras bem definidos, como por exemplo Quick Sort, notável por sua velocidade e por não necessitar de memória adicional, diferente do Merge Sort, protagonista deste documento, este requer uso de memória equivalente para manipular todos os níveis de recursão do algoritmo, mas desempenha bem em grandes volumes de dados em qualquer ordem ou tipo de número (números negativos, muitas casas decimais, etc…), seu desempenho é previsível e constante, do melhor ao pior caso. Nota-se que em comparação a algoritmos mais simplórios, o Merge Sort demonstra desempenho consideravelmente superior, se comparado com Insertion Sort, por exemplo, considerando o grande volume de 100 milhões de numeros, no tempo em que Insertion Sort ordena uma lista, Merge sort já terá a ordenado a muito tempo, conforme exemplifica *Thomas H.Cormen e Charles E. Leiserson em Introduction to Algorithms.*

*“A vantagem do merge sort se torna ainda mais evidente quando ordenamos 100 milhões de números: enquanto o insertion sort leva mais de 23 dias, o merge sort leva menos de quatro horas. De forma geral, à medida que o tamanho do problema aumenta, também aumenta a vantagem relativa do merge sort.”*

Entretanto, vale ressaltar que em um contexto real, a velocidade dos algoritmos nunca será, este fator sempre será alterado por algumas variáveis, como o modelo da máquina utilizada, ou haverem outras aplicações funcionando em segundo plano. Mesmo assim, o exemplo vale para mostrar que quando se trata de BIG DATA, Merge sort terá sempre uma performance relativamente rápida e constante, tendo como limitação a quantidade de memória RAM da máquina utilizada.

**2. Algoritmo Merge Sort**

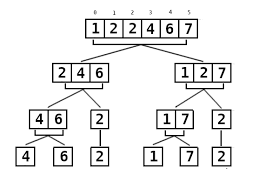
O Mergesort foi inventado por John Von Neumann e 1945 como parte de sua pesquisa de algoritmos, é importante destacar que Neumann é considerado “pai” do computador que conhecemos hoje, ele trabalhou no primeiro esboço teórico do EDVAC (Eletronic Discrete Variable Automatic Computer), ou seja, a ideia conceitual do Mergesort surgiu junto do primeiro rascunho de uma máquina moderna, tudo isso logo após o termino da segunda guerra mundial. Uma descrição detalhada e uma análise do algoritmo na versão bottom-up apareceram no relatório de Goldstine e Neumann em 1948, ambos buscavam formas eficientes de processar dados para o EDVAC, assim, foi desenvolvida uma maneira de ordenar listas de números utilizando mesclagem de sublistas (ou subarrays) ordenadas.



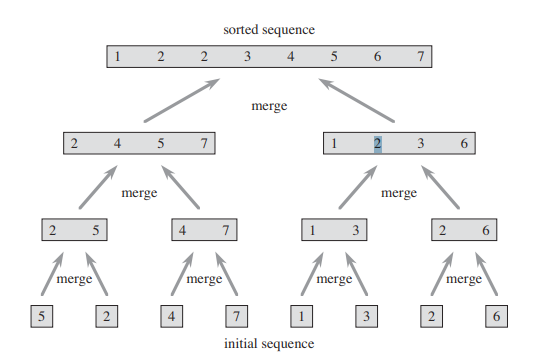
Goldstine a esquerda e Neumann a direita

Para aprofundar os estudos no algoritmo de ordenação Merge Sort, é necessário primeiro entender teoricamente seu funcionamento, esta explicação é essencial para que se compreenda sua complexidade, estabilidade e implementação.

O algoritmo opera sob o conceito de dividir para conquistar (*H.Cormen e Charles E. Leiserson em Introduction to Algorithms. Pag. 51*), ou seja, ele recebe um array de tamanho N, este será divido em sub-arrays recursivamente até que possuam apenas um índice (n = 1 sendo a condição de parada da recorrência).

******

Assim, temos a relação de recorrência T(n) = 2T(n/2) + O(n) quando n > 1 e O(1) quando n = 1, onde T(n) representa o tempo necessário para percorrer todo o array, ou seja, 2T(n/2) representa o tempo que o algoritmo levará para percorrer dois sub-arrays de tamanho n/2, depois 4 sub-arrays de tamanho n/4 e assim sucessivamente. O(n) representa a quantidade de tempo que será necessária para juntar todos os sub-arrays de cada nível de recursão (*H.Cormen e Charles E. Leiserson em Introduction to Algorithms).* Agora, para mesclar, os sub-arrays de forma ordenada, é necessário uso de uma função exclusiva, esta irá se utilizar de arrays auxiliares e temporários para mesclar dois sub-arrays de forma ordenada. Em anexo imagem do livro *Introduction to Algorithms, pag 56.*

**

A ordenação acontece comparando um array auxiliar da esquerda com um da direita assim atribuindo o menor dos dois números comparados no momento e o alocando no sub-array original, não só, o meio utilizado para a mescla de dois sub-arrays revela a natureza estável do algoritmo, pois quando um número for comparado com outro igual, manterá sua posição, como acontece no caso dos arrays A{2, 4, 5, 7} e B{1, 2, 3, 6}, onde após 2 do vetor A ser comparado com 2 do vetor B, percebe-se que 2 A mantem sua posição no vetor original passando a comparar 4 A com 2 B.

**Resolvendo a Recorrência:** Para melhor compreensão teórica da resolução, será considerado n = 2^i, sendo assim n passa a ser dividido por dois até chegar em 1:  
  
Passo 0: T(n) = 2T(n/2) + n

Passo 1: T(n) = 2[2T(n/4) + n/2] + n

4T(n/4) + 2n

Passo 2: T(n) = 4[2T(n/8) + n/4] + 2n

8T(n/8) + 3n

**...**

Passo i: T(n) = 2^iT(n/2^i) + i . n

Assim conclui-se que (n/2^i) = 1 🡪 i = log n, então, substituindo i, temos:

T(n) = (2^log n)T(1) + n . log n = n . T(1) + n . log n = O(n . log n)

Tendo conhecimento da complexidade de tempo do algoritmo, é importante destacar que o Merge Sort não é um algoritmo in-place, pois utiliza memória auxiliar adicional de complexidade O(n). No entanto, essa memória auxiliar NÃO É ALOCADA EM CADA NÍVEL DA RECURSÃO. Como o processo de mescla ocorre na fase de subida da recursão (após todas as divisões), a mesma estrutura de memória pode ser reutilizada entre os diferentes níveis. Isso significa que, embora o algoritmo tenha log n níveis de recursão, a quantidade máxima de memória necessária simultaneamente permanece O(n), e não O(n log n). Uma analogia útil é imaginar uma única caixa sendo usada para organizar diferentes grupos de peças em etapas sucessivas: a caixa é reutilizada a cada etapa de mescla, e não há necessidade de múltiplas caixas para cada nível (*Algorithms in a Nutshell, 2nd Edition, Geoge T. Heineman, Garry Pollice, Stanley Selkow*).

“Cada chamada recursiva da função sort exigirá um espaço equivalente ao tamanho do array, ou seja, O(n), e haverá O(log n) dessas chamadas recursivas. Portanto, o requisito de armazenamento para essa implementação ingênua é O(n log n). Felizmente, existe uma maneira de utilizar apenas O(n) para armazenamento, como discutiremos a seguir.”

Pontos fortes do algoritmo Merge Sort:  
 - Ponto mais forte: Paralelização (Excelente para decompor problemas e resolvê-los por partes simultaneamente)

– Bom uso de memória, o algoritmo não sofre de carregamentos desnecessários entre cache e RAM, sendo bom para grandes vetores.  
- Eficiente quando manipulando listas encadeadas, sua lógica de separação e mescla de sub-arrays de tamanho 1 funciona bem quando manipulando células de listas   
- Por último, sua característica mais proeminente, o Merge sort tem estabilidade absoluta O(n . log n) então se sai bem em situações aonde se precisa de garantia no pior caso.

Pontos fracos do algoritmo Merge Sort:

- Requer uso adicional de memória O(n), não só, quando mal implementado, pode exigir memória equivalente a O(n . log n).

- Inferior a algoritmos como Quick sort quando lidando com arrays pequenos - Data Structures and Algorithms in Java *(Goodrich et al.)*

- Um de seus pontos fortes, a depender do contexto, é também um de seus pontos fracos, sua estabilidade, garante que o algoritmo não seja minimamente adaptativo, ou seja, um vetor ordenado consumira a mesma quantia de tempo de um vetor em ordem decrescente.

**Implementação merge sort**

A implementação do algoritmo depende de duas funções, uma para separar o array inserido e outra para mescla-lo e ordena-lo.  
 A primeira função de maneira muito simples, irá chamar a si mesma recursivamente, redefinindo os parâmetros “inicio” e “fim” a cada chamada. Vale ressaltar que o algoritmo na prática irá separa sempre o lado esquerdo primeiro.

Esquerda e direita no algoritmo são definidos pelo meio que é recalculado a cada chamada usando os valores dos índices dos arrays. (ex: inicio = 0; fim = 9 🡪 meio = 4, pois 0 + (9 - 0)/2 = 4,5). Tendo definido os sub-arrays recursivamente, a função junta\_vetor irá receber o atual início e fim (ou esquerda e direita) do nível de recursão em que o algoritmo se encontra.

Primeiro a função define novamente o meio com base nos parâmetros fornecidos, depois define mais uma vez esquerda e direita utilizando “meio”, “inicio” e “fim”. Tendo definido os respectivos limites, declaramos dois vetores auxiliares ocupando o espaço de memória equivalente a quantidade de índices contemplados dentro da metade da esquerda e direita. Por fim usaremos dois loops “for” para atribuir aos vetores auxiliares os valores do vetor original.

A função agora irá utilizar um looping “while” para atribuir os valores ao array original, isso acontece simplesmente verificando se aux\_esquerda[i] é menor **OU IGUAL** que aux\_direita[j], caso não haja o sinal de “=” na verificação o algoritmo perderá sua estabilidade, se a condição da verificação for atingida, atribui ao vetor original (vet[k]) o valor de aux\_esquerda[i] e incrementa k e i, se não atingir a condição, faz o mesmo processo com aux\_direita[j]. Quando um dos dois vetores auxiliares for incrementado até que não existam mais números a frente (no caso do início ao meio para a esquerda e do meio + 1 até fim para a direita), o looping será interrompido.   
 Ao final, através de dois loopings “while”, o algoritmo irá verificar se ainda há algum número que não foi alocado, caso haja, este simplesmente será alocado ao vetor original, incrementando k e i ou j, assim entregando ao nível de cima uma mescla de dois sub-arrays ordenados (e liberando a memória utilizada para os vetores auxialiares dentro da função).  
  
Segue link com algoritmo implementado (consultar a pasta sort):

<https://github.com/unibrasil-programacao-computadores/tp-mergesort.git>

**Comparativo com Quicksort:**

Nota-se que com a implementação utilizada do merge sort, o desempenho foi consideravelmente inferior quando comparado ao Quicksort, isto ocorre pela forma com que o algoritmo Mergesort foi implementado. A constante alocação e liberação de memória na função responsável por juntar e ordenar o vetor é muito custosa computacionalmente, entretanto, vale ressaltar que este é o caso apenas no exemplo implementado, há outras formas mais eficientes para se implementar o algoritmo Mergesort.

Conclui-se então, que o algoritmo Merge Sort, caso bem implementado, e relativamente veloz para grandes quantidades de dados, entretanto, a velocidade não é seu ponto mais forte, sua verdadeira força está em seu gigantesco potencial para paralelização e estabilidade.  
  
**Informações** **da máquina utilizada:**

|  |  |
| --- | --- |
| Sistema operacional | Microsoft Windows 10 Pro |
| Processador | Intel(R) Core(TM) 13-4160 CPU @ 3.60GHz |
| Memoria RAM instalada | 12,0 GB |

**Referencias:**   
**Knuth, D. E.** The Art of Computer Programming, Volume 3: Sorting and Searching. Addison-Wesley, 2ª ed., 1998.

**Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Stein, C.** Introduction to Algorithms. MIT Press, 3ª ed., 2009.

**Ziviani, N.** Projeto de Algoritmos: com implementações em Pascal e C. Cengage Learning, 3ª ed., 2011.

***Goodrich et al.*** Data Structures and Algorithms in Java

***Geoge T. Heineman, Garry Pollice, Stanley Selkow.*** *Algorithms in a Nutshell, 2nd Edition,*

<https://thomas.baudel.name/Visualisation/VisuTri/inplacestablesort.html>

Katajainen, Jyrki; Träff, Jesper Larsson (March 1997). "Algorithms and Complexity". *Proceedings of the 3rd Italian Conference on Algorithms and Complexity*. Italian Conference on Algorithms and Complexity. Lecture Notes in Computer Science. Vol. 1203. Rome. pp. 217–228.