**SCC0220 - Laboratório Introdução à Ciência da Computação II**

**Relatório - Aula 6**

**Alunos NUSP**

Pedro Henrique de Sousa Prestes 15507819

Pedro Lunkes Villela 15484287

**Trabalho 5 - A outra pilha**

**Heap Sort x Merge Sort**

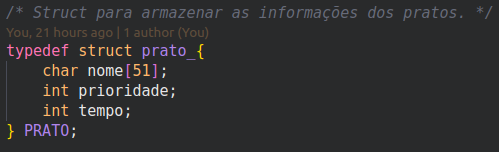
**🡺 Comentário**

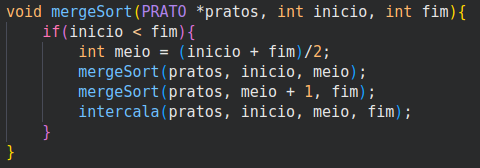
❖ **Merge Sort:** Consiste em dividir o array em duas metades, ordena cada metade separadamente e, em seguida, mescla as duas metades ordenadas para formar um array final ordenado. Ele tem complexidade $$O(n log n)$$. Durante o processo de divisão recursiva, novas cópias dos subarrays são criadas, o que consome espaço adicional. Portanto, além da pilha de chamadas recursivas, o MergeSort requer memória extra para realizar a mesclagem dos subarrays, tendo complexidade espacial auxilair de $$O(n)$$.

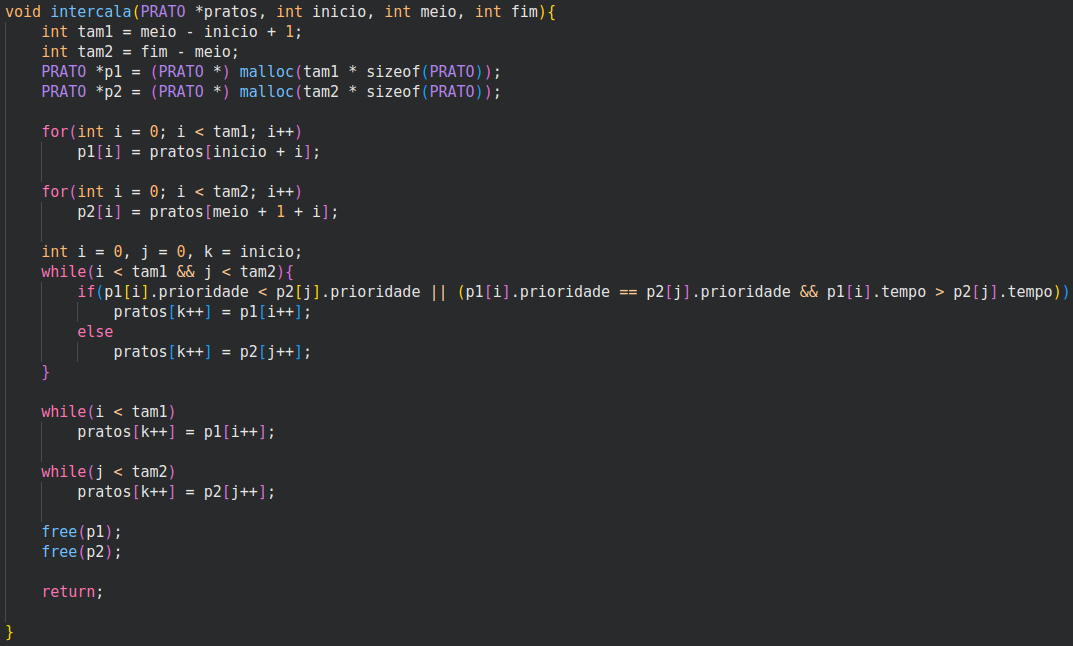
❖ **Heap Sort:** Consiste em utilizar uma estrutura de heap\* para ordenar os elementos. Ele primeiro constrói um heap máximo e, em seguida, troca o maior elemento com o último, ajustando o heap restante. Sua complexidade é $$O(n log n)$$ e sua complexidade espacial auxiliar é $$O(1)$$.

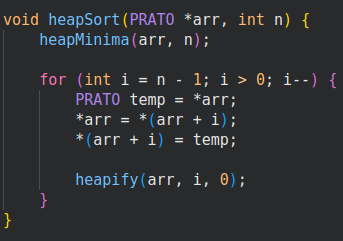
\*Heap é uma estrutura de dados baseada em árvore binária completa, onde cada nó segue a propriedade do heap. Em um heap máximo, o valor de cada nó é maior ou igual aos seus filhos, enquanto em um heap mínimo, o valor de cada nó é menor ou igual aos seus filhos. Essa estrutura é usada em algoritmos como HeapSort e em filas de prioridade, permitindo acesso eficiente ao maior ou menor elemento em tempo $$O(1)$$ e remoções/inserções em $$O(log n)$$.

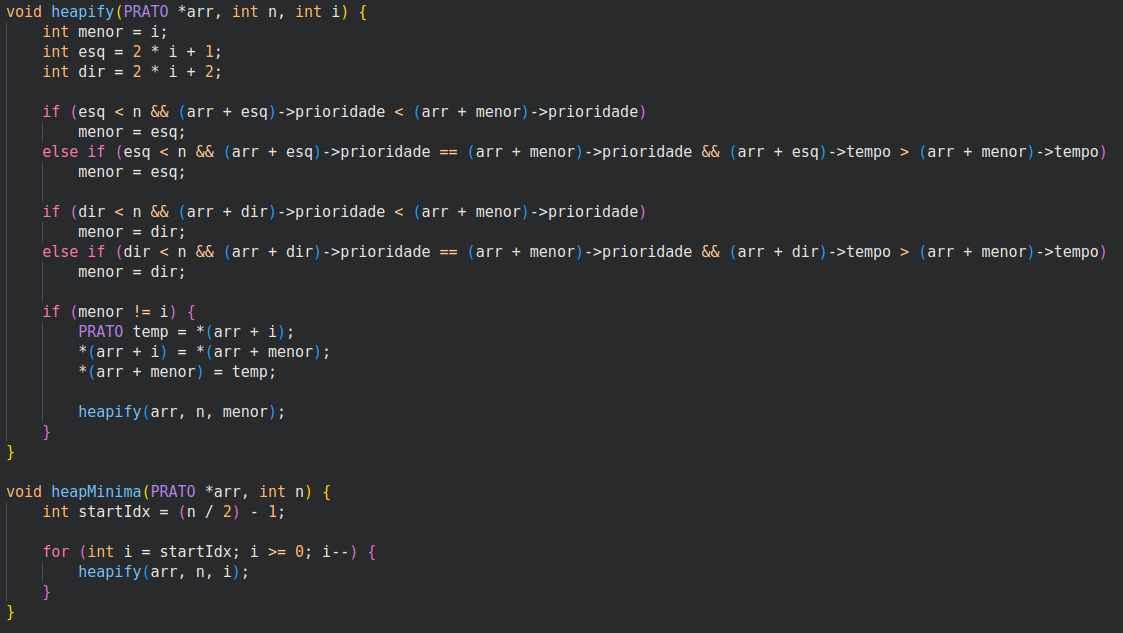
Para comparar os dois sorts, foram utilizados casos de pratos de comida em um restaurante que devem ser ordenados a partir de seus valores de prioridade, caso dois pratos tenham prioridades iguais, o tempo de preparo deverá ser considerado como critério para ordenação.

**🡺 Código**

****

****

****

****

**🡺 Saída**

Seguem alguns casos testados comparando o desempenho de ambos algoritmos.

**3° Caso**

Tempo de execução MergeSort 0.000084

Tempo de execução HeapSort 0.000157

Tempo de execução MergeSort 0.000163

Tempo de execução HeapSort 0.000157

Tempo de execução MergeSort 0.000172

Tempo de execução HeapSort 0.000216

**6° Caso**

Tempo de execução MergeSort 0.000226

Tempo de execução HeapSort 0.000351

Tempo de execução MergeSort 0.000340

Tempo de execução HeapSort 0.000351

Tempo de execução MergeSort 0.000154

Tempo de execução HeapSort 0.000350

**9° Caso**

Tempo de execução MergeSort 0.050297

Tempo de execução HeapSort 0.057139

Tempo de execução MergeSort 0.051503

Tempo de execução HeapSort 0.058397

Tempo de execução MergeSort 0.050720

Tempo de execução HeapSort 0.056100

**10° Caso**

Tempo de execução MergeSort 0.053169

Tempo de execução HeapSort 0.055588

Tempo de execução MergeSort 0.049765

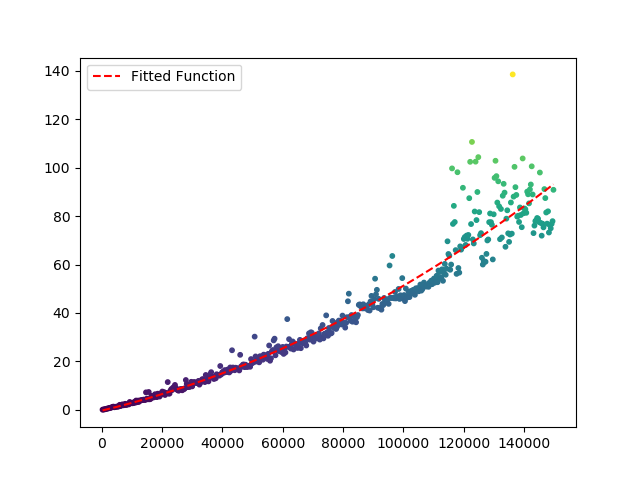
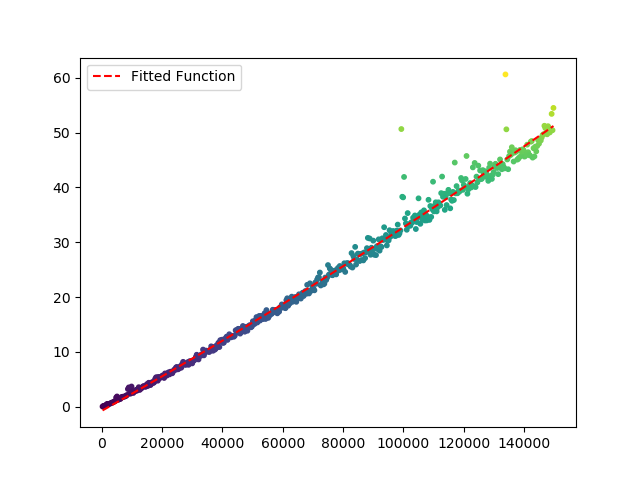
Tempo de execução HeapSort 0.053954

Tempo de execução MergeSort 0.055818

Tempo de execução HeapSort 0.062255

**🡺 Conclusão**

Apesar de ambos terem complexidade $$O(n log n)$$, o Merge Sort mostrou-se ligeiramente mais eficaz em lidar com grande número de dados em relação ao Heap Sort. Isso ocorre por conta de sua melhor localidade no cache, acessando a memória de maneira sequencial, enquanto o Heap faz acessos dispersos na memória. Entretanto, apesar de mais rápido, o Merge Sort apresenta uma complexidade espacial consideravelmente maior, por ter que criar vetores auxiliares durante a intercalação.



(Gráfico Merge e Heap, Eixo X representa o número de entradas e o Eixo Y milissegundos, feito a partir de casos teste a parte do Runcodes)