```
/*
Aluno: Francisco Vinícius Lopes Costa
Matrícula: 2021022958
*/
```

Questão 8 - Prova

O algoritmo mergesort divide-se em três passos:

- O primeiro é a divisão do vetor ao meio, esse é o passo da divisão do problema, de fato aqui só é calculado o ponto médio entre as extremidades do vetor;
- O segundo passo é o de conquista, onde os dois vetores resultante do passo 1 são recursivamente chamados;
- O terceiro passo é o da combinação dos vetores, feito através da intercalação dos vetores do passo 2.

O primeiro passo leva um tempo constante, então dizemos que tem uma complexidade constante O(1).

O terceiro passo apresenta um tempo linear, pois depende linearmente da quantidade de elementos do vetor, dizemos que tem uma complexidade O(n).

Se pensarmos no passo da divisão e da combinação juntos temos que a complexidade obedece a ordem do passo que tem maior complexidade, logo a unicidade desses dois passos apresenta complexidade O(n).

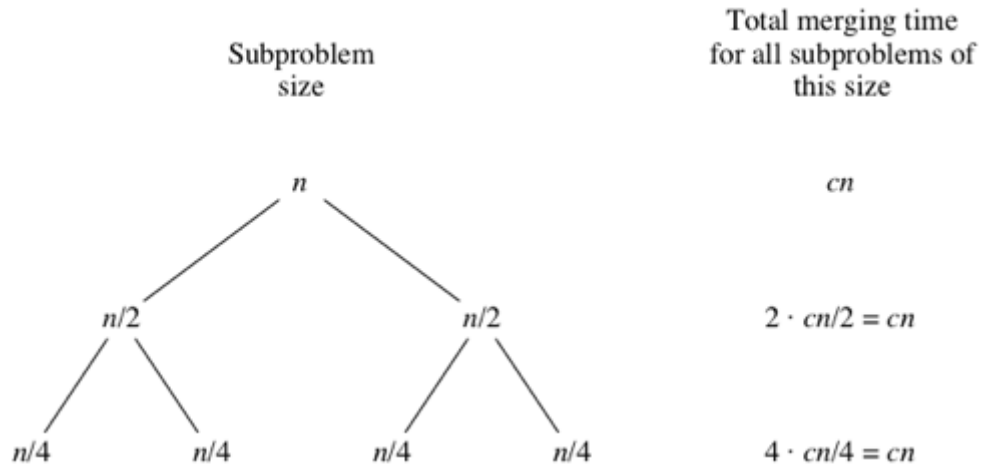
De forma mais concreta vamos dizer que o passo da divisão e da combinação levam um tempo *cn* de execução, onde c é uma constante e n é o número de elementos do vetor.

Considerando ainda que n>1, podemos assumir que o n/2 será sempre um inteiro (*cast* feito de forma automática). Assim podemos pensar no tempo total de execução do *mergesort* como a soma do tempo de execução dos subvetores (n/2) (passo da conquista) com o tempo *cn* (que é tempo de execução do passo da divisão e da combinação).

Agora para achar o tempo total de execução do *mergesort* é necessário verificar o tempo gasto pelas chamadas recursivas do *mergesort* para subarrays de n/2 elementos. O tempo de divisão e combinação do array com n/2 elementos é dado pela expressão cn (c uma contante e n o número de elementos), mas como o número de elementos do subarray é n/2, então o tempo de intercalação é dado por c*(n/2), que resulta em cn/2. Como temos 2 subarrays de tempo cn/2, o tempo total de intercalação é dado por 2*cn/2, que é o mesmo que cn.

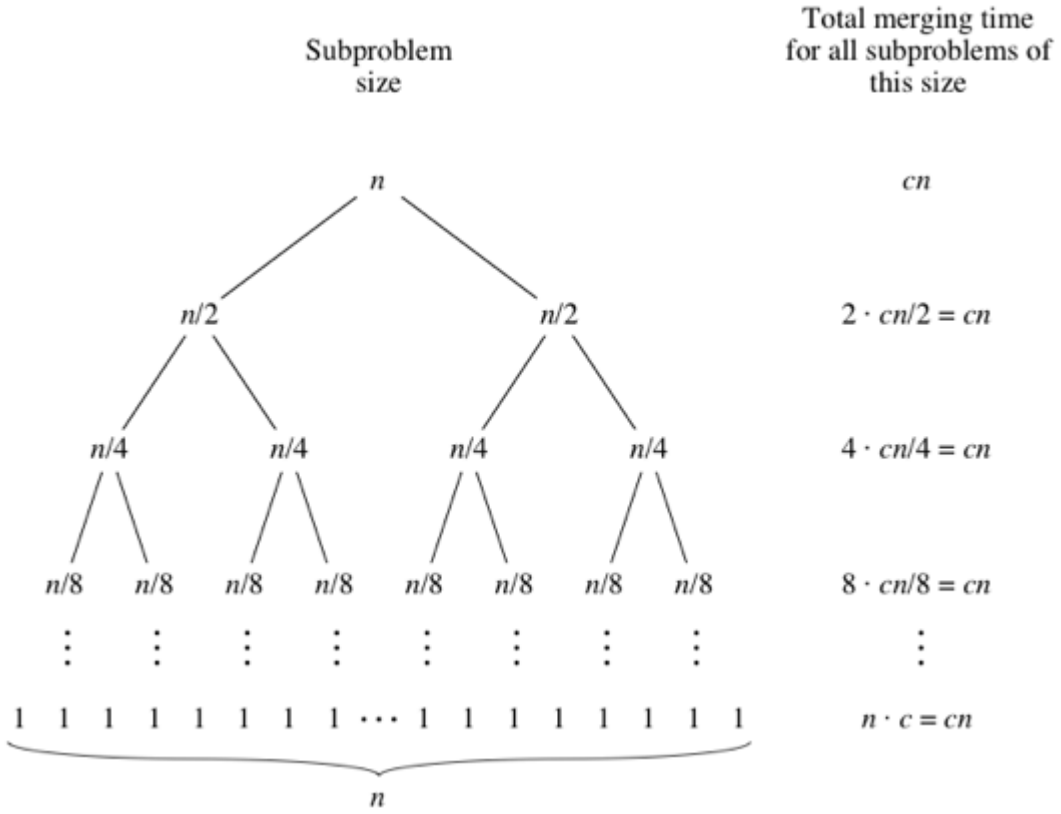
Algo similar ocorre quando chamamos recursivamente o *mergesort* para o subarray com n/2 elementos; ou seja, esse é quebrado em mais dois sub arrays com n/4 elementos; como essa chamada recursiva também acontece no outro subarray de n/2 elementos, teremos um total de 4 subarrays de n/4 elementos. Para o tempo de intercalação de cada um desses subarrays temos a expressão cn/4 (onde c é uma constante e n/4 é o número de elementos do vetor), como existem 4 subarrays de n/4 elementos, temos o tempo total de intercalação é 4*cn/4, ou seja, é o mesmo que *cn*.

Para entender melhor o que foi dito, segue uma árvore explicativa.



Percebemos que o problema vai ficando cada vez menor, porém, sua quantidade vai dobrando. Temos então que em cada nível de recursão a quantidade de problemas (vetores para ordenar) dobra, mas tem-se também que o tempo gasto para intercalar esses vetores é menor, pois são vetores menores, o tempo de intercalação vai reduzindo pela metade, já que cada subarray vai recebendo a metade de elementos do array pai.

A recursão se estenderá até chegarmos em arrays de tamanho 1, que é o caso base, e por definição já estão ordenados e tem tempo de execução O(1). Então o problema se resume em somar o tempo de execução desses subarrays, como tínhamos de início n elementos, teremos ao fim n subarrays de tamanho 1. Segue abaixo árvore explicando a lógica.



Agora que já sabe-se o tempo de intercalação de cada subarray, podemos calcular o tempo total do *mergesort* somando o tempo de intercalação de todos os níveis de recurssão. No nível 1

temos tempo cn, no nível 2 temos tempo cn, no nível 3 temos tempo cn... veja que o tempo total é a multiplicação do total de níveis por cn; sendo *l* o total de nível temos então que o tempo total é dado por *l*cn*.

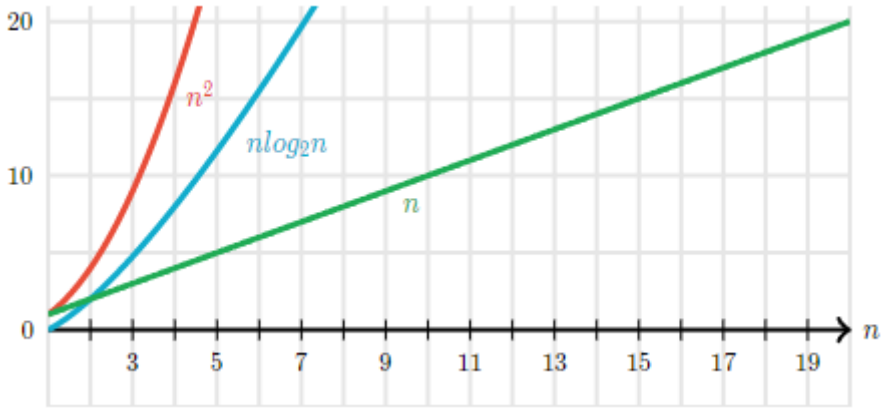
Mas o que seria *l*? Sendo que começa-se com n problemas, e vai dividindo e subdividindo até problemas menores de tamanho 1. Tal característica é logatímica, e *l* pode ser enxergado como *l = log n + 1.* De fato se temos n=8 e fizermos log 8(base 2) + 1 teremos *l=4*, que é justamente o total de níveis de recursão de um problema com 8 elementos.

Por fim temos que o tempo total é então ***cn*(log n +1)***. Quando utiliza-se a notação big O pode-se descartar o termo de menor ordem, nesse caso é o +1, e pode-se também descartar o coeficiente constante (c). Então em notação big O temos **O (n*log n)**.

Em termos de comparação com outros algoritmos, podemos utilizar a tabela auxiliar abaixo.

Algoritmo	Tempo de execução no pior caso	Tempo de execução no melhor caso	Tempo de execução médio
ordenação por seleção (selection sort)	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Insertion sort	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$
Ordenação por combinação (merge sort)	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$

Além da tabela, podemos entender o comportamento de cada notação através do gráfico abaixo.



Pelo gráfico e tabela acima, vemos que o *InsertionSort* cresce de forma quadrática em seu pior caso (ordem inversa), e no melhor caso (quando já está ordenado) o tempo cresce de forma linear. Ou seja, em termos práticos ordenar 8 elementos no pior caso levaria 64 unidades de tempo, e em seu melhor caso 8 unidades de tempo.