**Análise de complexidade do mergeSort**

#include "../headers/merge\_sort.h"

void mergeSort(*bigNum* \**numbers*, *bigNum* *L*, *bigNum* *R*){

*bigNum* M; //O(1)

    if (*L* < *R*){ //O(1)

        M = floor((*L* + *R*) / 2); //O(1)

        //Metade à esquerda do vetor

        mergeSort(*numbers*, *L*, M);

        //Metade à direita do vetor

        mergeSort(*numbers*, M + 1, *R*);

        merge(*numbers*, *L*, M, *R*); //O(n)

    }

}

void merge (*bigNum* \**numbers*, *bigNum* *L*, *bigNum* *M*, *bigNum* *R*){

*bigNum* tamanhovetL = *M* - *L* + 1; //O(1)

*bigNum* tamanhovetR = *R* - *M*; //O(1)

*bigNum* i, j; //O(1)

*bigNum* \*numbersL = (*bigNum* \*) malloc (tamanhovetL \* sizeof(*bigNum*)); //O(1)

    if (numbersL == NULL) { //O(1)

        printf ("Falta de memoria...\n");

        return;

    }

*bigNum* \*numbersR = (*bigNum* \*) malloc (tamanhovetR \* sizeof(*bigNum*)); //O(1)

    if (numbersR == NULL) { //O(1)

        printf ("Falta de memoria...\n");

        return;

    }

    for (i = 0; i < tamanhovetL; i++){ //O(n/2) \* O(1)= 1/2 \* O(n) = O(n)

        numbersL[i] = *numbers*[i + *L*]; //O(1)

    }

    for (j = 0; j < tamanhovetR; j++){ //O(n/2) \* O(1) = 1/2 \* O(n) = O(n)

        numbersR[j] = *numbers*[*M* + j + 1]; //O(1)

    }

    i = 0; //O(1)

    j = 0; //O(1)

*bigNum* k; //O(1)

    for (k = *L*; k <= *R*; k++){  //O(n) \* O(1) = O(n)

        if (i == tamanhovetL){ //O(1)

*numbers*[k] = numbersR[j]; //O(1)

            j++; //O(1)

        }

        else if (j == tamanhovetR) { //O(1)

*numbers*[k] = numbersL[i]; //O(1)

            i++; //O(1)

        }

        else if (numbersL[i] <= numbersR[j]){ //O(1)

*numbers*[k] = numbersL[i]; //O(1)

            i++; //O(1)

        }

        else { //O(1)

*numbers*[k] = numbersR[j]; //O(1)

            j++; //O(1)

        }

    }

    free (numbersR); //O(1)

    free (numbersL); //O(1)

}

1º) Análise da complexidade LOCAL

- Vejam que qualquer linha que possua criação de uma nova variável, comparação ou cálculos é O(1), ou seja, é realizado em tempo constante

- Agora dentro da função merge, temos 3 for´s, onde no

1. primeiro e no segundo, percorremos metade do vetor e o que está dentro desses for´s é feito de forma constante, e, assim, ficamos com O(n/2) \* O(1) = 1/2 \* O(n) = O(n)
2. No terceiro, percorremos todo o vetor e as operações dentro desse for é constante. Logo, ficamos com O(n) \* O(1) = O(n)

Vejam que o mergeSort faz duas chamadas recursivas com cada chamada recursiva usando metade do vetor

Logo, teremos essa função de recursão:

Sendo T(1) = 1 como caso base

T(n) = 2 \* T(n/2) + O(n) -> O(n) vem da ordem de complexidade da função de combinação (merge)

Usando o teorema mestre simplificado nessa função, sabendo que podemos aplica-lo, pois temos uma divisão

nos subproblemas e temos um Polinômio como custo local da função. Assim ficamos:

a = 2

b = 2

d = 1

lg 2 = 1 = d -> LOGO, temos que a ordem do mergeSort é de **O(n log n)**

**SELECTION SORT**

🡪 𝑇(𝑛) = O(𝑛) + ((𝑛−1)(𝑛))/2 🡪 T(n) = O(𝑛) + O( (n²+n)/2) = O(n²)