

1. ANALISE TEORICA

1.1 BUBBLESORT

1.1.1 VERSÃO ITERATIVA

Pseudocódigo da versão iterativa:

```
function bubbleSort(array) {  
    n = tamanho do array  
    para i de 0 até n-1 {  
        para j de 0 até n-i-2 {  
            se array[j] > array[j+1] {  
                trocar array[j] com array[j+1]  
            }  
        }  
    }  
}
```

Note que o algoritmo foi implementado para um array de tamanho n , onde o loop externo é executado $n - 1$ vezes. O loop interno realiza suas comparações com base no valor de i no loop externo para evitar comparações desnecessárias nas últimas posições do array, que já foram ordenadas em passagens anteriores.

Por exemplo, para um array de tamanho $n = 5$:

- Para $i = 0$, o loop interno é executado 4 vezes, pois ele compara do primeiro até o penúltimo elemento, colocando o maior valor na última posição.
- Para $i = 1$, o loop interno executa 3 vezes, pois o último elemento já está ordenado, e agora o segundo maior será posicionado.
- Para $i = 2$, o loop interno executa 2 vezes, pois os dois últimos elementos estão ordenados.
- Para $i = 3$, o loop interno executa apenas 1 vez, pois os três últimos elementos já estão na posição correta.

Veja que somando o número de iterações para esse array, temos $(4 + 3 + 2 + 1) = 10$. Note que o número de comparações realizadas pelo loop interno em cada passagem do loop externo diminui gradualmente. Assim, em um array de tamanho n , temos $((n - 1) + (n - 2) + (n - 3) \dots + 1)$ comparações. Ou seja, o tempo de execução do algoritmo pode ser representado da seguinte forma:

$$\begin{aligned}
T(n) &= \sum_{i=1}^{n-1} i \\
&= \frac{(1+n-1)(n-1)}{2} \\
&= \frac{n(n-1)}{2} \\
&= \frac{n^2 - n}{2}
\end{aligned}$$

Portanto,

$$T(n) = O(n^2)$$

Como o algoritmo percorre o array com o mesmo número de iterações em todos os casos, o número total de comparações e, conseqüentemente, a complexidade de tempo permanece n^2 para os casos pior, médio e melhor. Podemos verificar isso calculando o limite de $T(n)/n^2$, para n tendendo ao infinito:

$$\begin{aligned}
\lim_{n \rightarrow \infty} \frac{\frac{n^2-n}{2}}{n^2} &= \frac{1}{2} \lim_{n \rightarrow \infty} \frac{n^2 - n}{n^2} \\
&= \frac{1}{2} \lim_{n \rightarrow \infty} 1 - \frac{1}{n} \\
&= \frac{1}{2} \left(\lim_{n \rightarrow \infty} 1 - \lim_{n \rightarrow \infty} \frac{1}{n} \right) \\
&= \frac{1}{2} (1 - 0) \\
&= \frac{1}{2} \in \mathbb{R}_+^*
\end{aligned}$$

Portanto,

$$T(n) = \Theta(n^2)$$

1.2 MERGESORT

1.2.1 VERSÃO ITERATIVA

Pseudocódigo da versão iterativa:

```

function mergeSort(array, tamanho) {
    i=1
    enquanto i é menor que tamanho {
        esquerda = 0
        enquanto esquerda é menor que tamanho-1 {
            meio = o menor entre (esquerda + i-1) e (tamanho-1)
            direita = o menor entre (esquerda + 2*i-1) e (tamanho-1)
            merge(array, esquerda, meio, direita)
            esquerda += 2*i
        }
        i *= 2
    }
}

function merge(array, esquerda, meio, direita) {
    arrayEsquerdo = elementos de array[esquerda] até array[meio]
    arrayDireito = elementos de array[meio+1] até array[direita]

    posiçãoAtual = esquerda

    enquanto esquerda não estiver vazia e direita não estiver vazia {
        se arrayEsquerdo[0] < arrayDireito[0] {
            sobrescrever array[posiçãoAtual] com arrayEsquerdo[0]
            remover arrayEsquerdo[0] de arrayEsquerdo
        } senão {
            sobrescrever array[posiçãoAtual] com arrayDireito[0]
            remover arrayDireito[0] de arrayDireito
        }
        posiçãoAtual += 1
    }

    enquanto arrayEsquerdo não estiver vazio {
        adicionar arrayEsquerdo[0] a array[posiçãoAtual]
        remover arrayEsquerdo[0] de arrayEsquerdo
        posiçãoAtual += 1
    }

    Enquanto arrayDireito não estiver vazio {
        adicionar arrayDireito[0] a array[posiçãoAtual]
        remover arrayDireito[0] de arrayDireito
        posiçãoAtual += 1
    }
}

```

Vamos começar analisando a função merge, que faz a intercalação de dois arrays, percorrendo todas as posições dos vetores, com custo de $n = m1 + m2$, onde $m1$ e $m2$ são os tamanhos do vetor 1 e vetor 2, respectivamente. Logo a complexidade de tempo da função merge é $T(n) = O(n)$.

No mergeSort, há dois loops, o externo controla a divisão do array em sublistas de tamanho crescente, que vão sendo mescladas à medida que o valor de i dobra a cada iteração. O loop interno percorre o array, criando pares de sublistas para serem mescladas em cada iteração do loop externo. Como i dobra a cada iteração, o número de vezes que o loop externo executa é $O(\log_2 n)$, pois a cada iteração o tamanho das sublistas dobra até que elas cubram o array inteiro. Portanto, para cada valor de i , o loop interno executa $O(n)$ operações, pois ele percorre o array inteiro dividindo-o em sublistas de tamanho i e realizando uma chamada para merge para cada par de sublistas.

Portanto, a complexidade total é

$$T(n) = O(n \log n)$$

Mesmo que o array esteja ordenado, o algoritmo ainda precisa percorrer todos os níveis de divisão e realizar operações de mesclagem, logo todos os casos (melhor, médio e pior) têm a mesma complexidade

1.3 QUICKSORT

1.3.1 VERSÃO ITERATIVA

Pseudocódigo da versão iterativa:

```
function quickSort(array, tamanho){
    pilha = pilha vazia

    empilhar(pilha,0)
    empilhar(pilha, tamanho-1)
    enquanto pilha não estiver vazia {
        direita = topo da pilha
        retirar topo da pilha
        esquerda = topo da pilha
        retirar topo da pilha

        pivoIndex = particionar(array, esquerda, direita)

        se pivoIndex - 1 > esquerda {
            empilhar(pilha, esquerda)
            empilhar(pilha, pivoIndex-1)
        }
        se pivoIndex + 1 < direita {
```

```

        empilhar(pilha, pivoIndex+1)
        empilhar(pilha, direita)
    }
}

function particionar(array, esquerda, direita) {
    pivo = array[direita]
    i = esquerda - 1
    para j de esquerda até direita - 1 {
        se array[j] <= pivo {
            i += 1
            trocar array[i] com array[j]
        }
    }
    trocar array[i + 1] com array[direita]
    retornar i + 1
}

```

Pior caso Quando o array está ordenado e o pivô escolhido é sempre o maior ou menor elemento, a função particionar divide o array em uma sublista vazia e uma sublista com $n-1$ elementos. Assim, Cada chamada para particionar tem um custo de $\Theta(n)$ no pior caso, pois é necessário percorrer o array para posicionar o pivô corretamente.

A função quickSort usa uma pilha iterativa que armazena os limites das partições ainda não processadas. No pior caso, essa pilha armazena até $\Theta(n)$ elementos ao longo da execução, como ele precisa ser realizado n vezes, a complexidade de tempo total é:

$$T(n) = O(n^2)$$

Melhor caso No melhor caso, o pivô divide o array em duas partes de tamanho $n/2$, na primeira chamada, $n/4$, na segunda chamada, $n/8$, na terceira chamada,..., $n/2^i$, para i chamadas. Isso resulta em custo $\log n$. Como partionamento tem custo $\Theta(n)$, então o custo total é:

$$T(n) = \Omega(n \log n)$$

2. IMPLEMENTAÇÕES

As implementações dos códigos foram feitas na maior quantidade de linguagens possível a tarefa: 3.

- Uma linguagem para idadeRep e idadeRep2: Lua

- Uma para a busca binária: C
- Uma para os algoritmos de ordenação: Rust

A escolha foi feita pela familiaridade dos integrantes do grupo com tais linguagens, e porque parecia mais divertido utilizar mais linguagens visto que a escolha é livre. Isso de forma que a escolha das linguagens não prejudique os resultados, obviamente.

2.1 AMBIENTE COMPUTACIONAL UTILIZADO

Todas as implementações foram executadas e cronometradas no mesmo ambiente computacional, para uma comparação justa e adequada entre as diferentes formas de implementação das funções citadas no roteiro avaliativo.

O ambiente computacional consiste em uma máquina virtual no servidor pessoal de um dos integrantes do grupo. A máquina virtual possui dois núcleos do processador, 2GB de memória RAM e 32GB de disco rígido à sua disposição, rodando uma distribuição Linux conhecida como Arch Linux, com uma instalação mínima, contendo somente os serviços necessários para funcionamento do sistema operacional e o serviço de servidor SSH, para conexão remota com a máquina.

Todas as implementações foram cronometradas utilizando os métodos apropriados para cada linguagem de programação e seus respectivos tempos de execução foram armazenados em um arquivo final, para serem analisados no presente relatório.

2.2 FUNÇÕES ITERATIVAS - IDADEREP e IDADEREP2

Essas funções foram implementadas na linguagem de programação lua, devido a um integrante que gosta muito de lua!

2.2.1 IdadeRep

A função idadeRep percorre a lista duas vezes:

O primeiro for percorre a lista para encontrar o menor valor (caso o menor valor seja menor que 200), com complexidade de $O(n)$, onde n é o tamanho da lista. O segundo for verifica se o menor valor está presente na lista, também com complexidade $O(n)$, com a verificação podendo ser interrompida antecipadamente caso encontrá-lo. Portanto, a complexidade de idadeRep é $O(2n)$, o que é linear.

2.2.2 IdadeRep2

Por outro lado, a função idadeRep2 ordena a lista e faz uma comparação:

A chamada a `table.sort()` ordena a lista com complexidade $O(n \log n)$. Em seguida, há uma comparação entre os dois primeiros elementos, uma operação de custo $O(1)$. Assim, a complexidade de idadeRep2 é $O(n \log n)$.

2.2.3 Tempos de execução

A tabela a seguir mostra como as funções `idadeRep` e `idadeRep2` se comportaram com entrada de cem, mil e um bilhão.

Tamanho	IdadeRep	IdadeRep2
100	0.017 ms	0.071 ms
1,000	0.063 ms	0.891 ms
1,000,000	48.32 ms	1688.003 ms

Com 100 elementos de entrada, a função `idadeRep2` já apresentou um tempo de execução maior em relação a `idadeRep`, apesar da diferença minúscula de apenas 0,054 milissegundos. Algo interessante de se apontar, é que a função de ordenação em lua, é feita com base em C, que é uma linguagem objetivamente mais rápida. Porém, devido a complexidade da função de ordenação ser maior, `Idaderep` ainda é mais rápida que `Idaderep2`.

À medida que a lista cresce, a diferença fica mais evidente. A `idadeRep2` começa a ser notavelmente mais lenta porque a ordenação envolve mais operações do que o simples percurso linear de `idadeRep`.

Com 1.000.000 de elementos, a diferença é alarmante. A função `idadeRep` (linear) tem um tempo de execução muito menor que `idadeRep2`, que é $O(n \log n)$. Isso reflete a diferença assintótica entre $O(n)$ e $O(n \log n)$. A ordenação vai se tornando muito mais custosa à medida que o tamanho da entrada aumenta.

Portanto, `idadeRep` é mais eficiente assintoticamente, como evidenciado pelos tempos de execução menores.