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 $((n-1) + (n-2) + (n-3)... + 1)$ 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, consequentemente, a complexidade de tempo permanece n^2 para os casos pior, médio e melhor. Podemos verificar isso calculando o limite de de $T(n)/n^2$, para n tendendo ao infinito:

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

Portanto,

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

1.1.2 VERSÃO RECURSIVA

Neste algoritmo de ordenação, são efetuadas comparações entre os dados amarzenados em um array de tamanho n. Cada elemento de posição i será comparado com o elemento de posição $i + 1$, se o primeiro for maior (no caso de ordenação crescente), os elemento trocam de posições. Então o algoritmo chama a si mesmo até a coleção estar completamente ordenada.

```
function bubbleSort(array, tamanho) {
    se tamanho <= 1 {                      // Θ(1)
    retornar
    }
    para i de 0 até tamanho-1 {            // Θ(n - 1)
        se array[i] > array[i+1] {
          trocar array[i] com array[i+1]
        }
    }
    bubbleSort( array, tamanho-1)         // T(n - 1)
}
```

Analisando a função verifique que há um laço que faz n - 1 comparações. Além da chamada recursiva com tamanho de entrada decrementado em 1, logo com tempo de execução representado por T(n - 1). Assim, temos a seguinte expressão de recorrência:

$$T(n) = \begin{cases} O(1), \text{ se } n \leq 1 \\ T(n-1) + (n-1), \text{ se } n > 1 \end{cases}$$

1.1.2.1 MÉTODO DA SUBSTITUIÇÃO Considerando a recorrência acima, mostraremos que o algoritmo é limitado por O(n^2).

Temos $T(n) \leq T(n-1)+n$, queremos mostrar que T(n) é limitada superiormente por uma função F(n) = cn^2, para algum c. Para isso usaremos indução.

Caso base:

$n = 1, T(1) = 1$

$$T(n) \leq cn^2 \implies T(1) = 1 \leq c(1^2) \implies 1 \leq c$$

Passo indutivo:

Hipotese: $T(k) \leq ck^2, (\forall k)[1 \leq k \leq n]$

$$\begin{aligned} T(n) \leq cn^2 &\implies T(n-1) + n \leq c(n-1)^2 + n \leq cn^2 \\ &\implies c(n^2 - 2n + 1) + n \leq cn^2 \\ &\implies cn^2 - 2nc + c + n \leq cn^2 \\ &\implies -2nc + c + n \leq 0 \\ &\implies c + n(1 - 2c) \leq 0 \end{aligned}$$

Como `n` é sempre um valor positivo e tende ao infinito, para que $c+n(1-2c) < 0$ seja verdade, precisamos que $1 - 2c < 0$.

$$\begin{aligned} 1 - 2c < 0 &\implies 1 < 2c \\ &\implies c > 1/2 \end{aligned}$$

Logo, $T(n)$ é $O(n^2)$.

1.1.2.2 MÉTODO DA ITERAÇÃO Considerando que o recorrência acima. Vamos expandir-la até encontrar o caso base.

Aplica-se n -1 sobre a fórmula de T(n). E assim por diante.

$$\begin{aligned} T(n) &= T(n-1) + (n-1), \\ &= (T(n-2) + (n-2)) + (n-1) \\ &= (T(n-3) + (n-3) + (n-2)) + (n-1) \\ &= ... \\ T(n) &= T(n-k) + \sum_{i=1}^{k} n - i \end{aligned}$$

Quando k = n -1, temos:

$$T(n) = T(1) + (n-1) + (n-2) + ... + 1$$

Que é igual a:

$$T(n) = T(1) + \frac{n(n-1)}{2}$$

Aqui, T(1) representa o custo da função no caso base. Podemos assumir que T(1) = O(1), já que não há comparações necessárias quando temos apenas um elemento.

Portanto, a complexidade total é:

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

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 arrayEsquerdo[0] em array[posiçãoAtual]
            remover arrayEsquerdo[0] de arrayEsquerdo
        }senão{
            sobrescrever arrayDireito[0] em array[posiçãoAtual]
            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, pecorrendo 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

1.2.2 VERSÃO RECURSIVA

Neste algoritmo de ordenação, a sequência de n elementos é dividida em duas subsênqueicias de $n/2$ elementos e não ordenadas recursivamente. Então as subsequência são intercaladas para produzir uma solução.

```
function mergeSort(array, esquerda, direita ) {
    se esquerda >= direita {                          // Θ(1)
        retornar
    }

    meio = esquerda + (direita - esquerda) / 2       // Θ(1)

    mergeSort(array, esquerda, meio)
    mergeSort(array, meio+1, direita)

    merge(array, esquerda, meio, direita)
}
```

Antes de calcular o tempo de execução do mergeSort, devemos analisar a função merge.

```
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 { //O(n)
        se arrayEsquerdo[0]   arrayDireito[0] {
            sobrescrever arrayEsquerdo[0] em array[posiçãoAtual]
            remover arrayEsquerdo[0] de arrayEsquerdo
        }senão{
            sobrescrever arrayDireito[0] em array[posiçãoAtual]
            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
    }
}
```

A função merge faz a intercalação de dois arrays, pecorrendo 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.

Assim, verifica-se que nosso método principal faz duas chamadas recursivas com tamanhos de entrada divididos pela metade, logo com tempo de execução representado por $T(\frac{n}{2})$ cada uma.

Portanto, a complexidade total é:

$$T(n) = \begin{cases} O(1), \text{ se } n \leq 1 \\ 2T(\frac{n}{2}) + n, \text{ se } n > 1 \end{cases}$$

1.2.2.1 MÉTODO DA SUBSTITUIÇÃO

1.2.2.2 MÉTODO DA ITERAÇÃO Considerando que o recorrência acima. Vamos expandir-la até encontrar o caso base.

Aplica-se $n/2$ sobre a fórmula de T(n). E assim por diante.

$$\begin{aligned} T(n) &= 2T(\frac{n}{2}) + n, \\ &= 2(2T(\frac{n}{4}) + \frac{n}{2}) + n \\ &= 2(2(2T(\frac{n}{8}) + \frac{n}{4}) + \frac{n}{2}) + n \\ &= ... \\ T(n) &= 2^k T(\frac{n}{2^k}) + k.n \end{aligned}$$

Vamos encontrar para qual valor de k, $\frac{n}{2^k} = 1$.

$$\begin{aligned} \frac{n}{2^k} = 1 &\implies n = 2^k \\ &\implies k = log_2 n \end{aligned}$$

Aplicando na recorrência:

$$\begin{aligned} T(n) &= 2^{log_2 n} T(\frac{n}{2^{log_2 n}}) + n.log_2 n \\ &= n.T(\frac{n}{n}) + n.log_2 n \\ &= n + n.log_2 n \end{aligned}$$

\

Portanto, a complexidade total é:

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

1.2.2.3 MÉTODO DA ARVORE DE RECURSÃO A árvore de chamadas do MergeSort começa com um nó raiz que representa o problema original de tamanho $\frac{n}{2}$. Em cada nível da árvore, o array é dividido em duas sublistas de tamanhos iguais, resultando em duas chamadas recursivas para problemas de tamanho . Cada uma dessas chamadas recursivas gera mais dois nós, e assim por diante.

Esse processo de divisão continua até atingirmos o caso base, em que o tamanho do array é $n = 1$, como mostrado a seguir:

```
           T(n)------------------n
          /    \
         /      \
  T(n/2)          T(n/2)---------2n/2 = n
  /   \           /   \
 /     \         /     \
T(n/4) T(n/4) T(n/4) T(n/4)-----4n/4 = n
 ...     ...   ...     ...
T(1)
```

A altura da árvore é o número de níveis até chegar ao caso base. Na primeira chamada recursiva, temos o termo $T(\frac{n}{2})$, em seguida $T(\frac{n}{2^2})$, $T(\frac{n}{2^3})$,... até $T(\frac{n}{2^h}) = T(1)$, onde h corresponde a altura da árvore.

Calculando h:

$$\begin{aligned} T(\frac{n}{2^h}) = T(1) &\implies \frac{n}{2^h} = 1 \\ &\implies n = 2^h \\ &\implies \log_2 n = \log_2 2^h \\ &\implies h = \log_2 n \end{aligned}$$

Como o tempo de execução do algoritmo corresponde corresponde a soma dos passos de todos os níveis, temos:

$$\begin{aligned} T(n) &= \sum_{i=0}^{h} n \\ &= n \sum_{i=0}^{h} 1 \\ &= n(\log_2 n + 1) \\ &= O(n \log n) \end{aligned}$$

1.2.2.4 MÉTODO DO TEOREMA MESTRE !TODO

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 Θ(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 Θ(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:

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

1.3.2 VERSÃO RECURSIVA

O QuickSort é um algoritmo de ordenação que funciona dividindo repetidamente o array em duas partes menores até que cada subarray tenha no máximo um elemento.

```
function quickSort(array, esquerda, direita) {
    se esquerda >= direita {
    retornar
    }

    pivoIndex = particionar(array, esquerda, direita)


    quickSort(array, esquerda, pivoIndex - 1)
    quickSort(array, pivoIndex + 1, direita)
}
```

A função *particionar* percorre o array usando o índice j, comparando cada elemento array[j] com o pivô. Se o valor de array[j] é menor ou igual ao pivô, ele é trocado com o elemento na posição i, que mantém a posição de divisão entre os elementos menores e maiores que o pivô. Ao final do loop, todos os elementos à esquerda de i são menores ou iguais ao pivô, e todos os elementos à direita são maiores.

```
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
}
```

No pior caso do QuickSort, a função particionar divide o array de forma altamente desbalanceada, resultando em uma sublista vazia e uma sublista com n−1 elementos. Isso acontece, por exemplo, quando o array já está ordenado e o pivô escolhido é o maior ou menor elemento. Como a função particionar tem complexidade $\Theta(1)$, a recorrencia pode ser expressa como:

$$T(n) = T(n-1) + n$$

No melhor caso do QuickSort, cada chamada de particionamento divide o array em duas sublistas de tamanho aproximadamente n/2. Isso resulta na seguinte recorrência:

$$T(n) = 2T(\frac{n}{2}) + n$$

Onde $2T(\frac{n}{2})$ representa o custo das duas chamadas recursivas em subarrays de tamanho $\frac{n}{2}$.

Como visto na analise do Merge Sort, tal recorrência tem custo:

$$T(n) = O(n * \log 2)$$

1.3.2.1 MÉTODO DA SUBSTITUIÇÃO

1.3.2.2 MÉTODO DA ITERAÇÃO Considerando que o recorrência acima. Vamos expandir-la até encontrar o caso base.

Aplica-se n -1 sobre a fórmula de T(n). E assim por diante.

$$
\begin{aligned}
T(n) &= T(n-1) + n, \\
&= (T(n-2) + n) + n \\
&= ((T(n-3) + n) + n) + n \\
&= ... \\
T(n) &= T(n-k) + k.n
\end{aligned}
$$

Note que,

$$n - k = 1 \implies k = n - 1$$

Aplicando na recorrência:

$$
\begin{aligned}
T(n) &= T(n-k) + k.n, \\
&= T(n-(n-1)) + n(n-1) \\
&= T(1) + n^2 - n \\
&= 1 + n^2 - n
\end{aligned}
$$

Portanto, a complexidade no pior caso é:

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

1.3.2.3 MÉTODO DA ARVORE DE RECURSÃO

1.3.2.4 MÉTODO DO TEOREMA MESTRE

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 minuscula de apenas 0,054 milisegundos. 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.